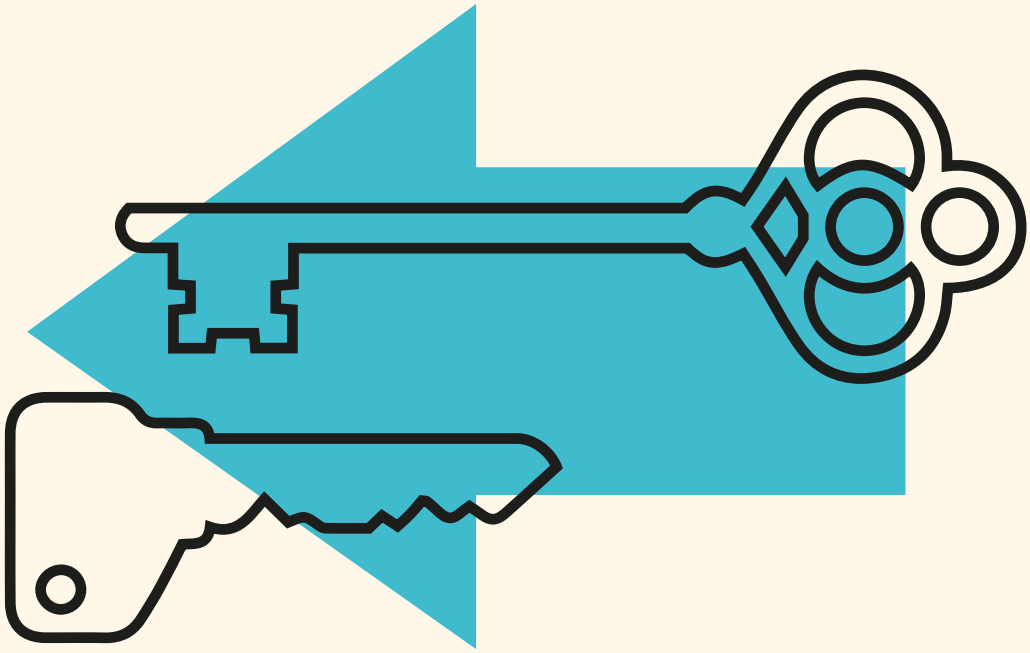
POST-QUANTUM TLS



THOM WIGGERS

Post-Quantum TLS

Thom Wiggers

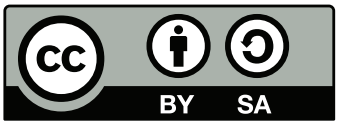
https://orcid.org/0000-0001-8967-8456

ISBN 978-94-6473-330-3

Printing by Ipskamp Printing.
Cover design by Judith van Stegeren.

Digital version available through thomwiggers.nl/publication/thesis/.

This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE).



Post-Quantum TLS

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J. M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

dinsdag 9 januari 2024
om 14.30 uur precies

door

Thomas Vincent Wiggers

geboren op 2 maart 1993
te Vleuten–De Meern

Promotor

Prof. dr. Peter Schwabe

Copromotor

Dr. Douglas Stebila
University of Waterloo, Canada

Manuscriptcommissie

Prof. dr. Lejla Batina

Prof. dr. Cas Cremers
CISPA Helmholtz Center for Information Security, Duitsland

Dr. Britta Hale
Naval Postgraduate School, Verenigde Staten

Prof. dr. Kenneth Paterson
ETH Zürich, Zwitserland

Dr. Paul Rösler
Friedrich-Alexander-Universität Erlangen-Nürnberg, Duitsland

Thanks

The book that lies before you is the culmination of a few years of work, and the end of a long time at Radboud University. I have started out as a first-year Bachelor's student, and am now leaving with a finished Ph.D. thesis. I look back on my time very fondly, which in no small part is thanks to the many people I have had the pleasure of working and interacting with throughout the years I spent at Radboud University.

To start, my primary supervisor and promotor, Peter, has been a large factor in getting to this point. Our introduction was in the course "Cryptographic Engineering", a Master's course I was taking as a Bachelor's student because why not, and Peters enthusiasm for cryptography drew me in to my Bachelor's thesis project. Peter was not discouraged when I later told him that I did not see a future for myself in low-level cryptography implementations, his main focus area at that time. Instead, he ended up offering me a spot in the research project that has culminated in this book, for which I am still grateful. Douglas Stebila later joined as second supervisor, but long before that we had very productive discussions on how to prove KEMTLS and beyond; I also still appreciate hosting me when I visited the University of Waterloo. The cupcakes on my birthday ware a nice surprise.

I would also like to thank the manuscript committee for their reviews; I am sure that they had not expected a thesis of this length when they graciously agreed to join the committee. My apologies.

The work in this thesis could not have been done without the discussions with and insights from many people. Some of them, I had the privilege of working with as co-authors: Andreas, Bas, Bo-Yin, Christopher, Dimitri, Douglas, Elisabeth, Fabio, Felix, Francisco, Goutam, Jesús-Javier, Jonathan, Jorge, Juliane, Krijn, Luke, Marc, Matthias, Michael, Nick, Patrick, Peter, Ruben, Simon, Simona, Sofía, Tanja, and Veelasha: thank you. Listed in alphabetical order, as is tradition.

The Digital Security department of Radboud University is also full of friendly faces. I owe a great deal to those who started and paved part of

the way before me, and hope to have been welcoming and pass on some tips and tricks to those who came after me. Everyone definitely made the environment better. And if things were not going so well, at least you could always take a spanner and bang against the espresso machine until it would turn on again. In no particular order, I would like to shout out Denisa, Amber, Matthias, Joost, Benoït, and Krijn for staffing the PQHQ with me; Anna, Pedro, and Łukasz for being good friends; I could probably mention a lot more people, but I am afraid of missing someone and this book is long enough as it is. I would also be remiss if I did not mention the friendly people at the Max Planck Institute for Security and Privacy in Bochum for hosting me, espresso, and getting distracted from doing actual work while I was visiting.

It was a great privilege to be able to do an internship with Cloudflare, even if everything had to happen remotely. Fortunately, we were able to catch up on things at Real World Crypto in Amsterdam, even if we could not find Dropshot. Armando, Bas, ChrisP, ChrisW, Jonathan, Luke, Nick, Peter, Sofía, Tanya, Watson, I hope we can do some more internet cryptography research some time. Wesley, thanks for teaching me about submarine politics.

Though I have been very privileged in having a fairly smooth Ph.D. trajectory, it has been nice to have a group of friends with joined experiences to fall back on, either for tips, or just to vent. Our IRC server's `#academia` channel (formerly `#promovendi`, until too many people defended) has, and I hope will continue to be, amazing. But also for `#sport` (Amber, Bas, Gerdriaan, Joost, Margot, PP, Rik), stock market analyses, or just memes.

Thank you, Judith, for helping me out with the design of the cover of this thesis, being unfazed when faced with a stressed candidate; and also of course for being a good friend.

Last, but not least, this all would not have been possible without the support of my family, and in particular Leonie. It must have been difficult to carry around a baby while your husband was working on his own (with pretty much the same due dates!). Thanks for your patience and love.

En dan tot slot, dit boek is voor Olivier, die me aanmoedigt terwijl ik dit schrijf. Ik hoop dat je nieuwsgierigheid nooit grenzen leert kennen.

Thom Wiggers
Nijmegen, November 2023

Contents

1 Introduction

Probably soon after people started talking to each other, the desire to communicate in secret came up. Whispering and walking out of anyone's earshot are probably the oldest ways to make sure that what you intend to say stays between you and your conversation partner. As societies started to form, people needed to communicate over greater distances. This necessitated the use of couriers, which would carry letters or communicate the message verbally. To keep the couriers from reading the messages, the art of *cryptography* was invented, ancient Greek for "secret writing". Ciphers and codes were used to make sure only the intended recipient would be able to decipher or *decrypt* the meaning of a particular message. One of the most famous encryption methods is perhaps the Caesar cipher, used by and named after Julius Caesar [347], which substitutes each letter by shifting it by a specified number of spaces in the alphabet. This way, A might become E, B might become F, and so on. The ancient Greek scytale is an early example of using a "device" to protect a message. A scytale is a rod around which you wind a strip of parchment, on which you then write a message. The strip is then unwound and passed to the recipient. Only if they wrap the parchment around a rod with the same diameter, would the message line up again and be legible.

These primitive ciphers already use a foundational principle in cryptography. Both use some additional information to hide and recover the message. In the Caesar cipher, this is the number of spaces by which the letters are shifted; for the scytale, it is the width of the rod. We call this information the *secret key*. Both the sender and the recipient in these schemes need to know this *shared* secret key to be able to decipher the message.

Because it is very inconvenient to have to first agree on a shared secret key, cryptographers have long sought a system where the sender of a message only needs some *public* information to hide the message. In the 1970s this idea, called *public-key cryptography*, finally became reality as the first schemes based on "mathematical trapdoors" were invented [119]. They rely on mathematical problems that are hard to solve if you leave out some extra information. For

example, RSA [303] relies on the difficulty of finding the values of the large prime numbers p and q given the product $n = p \cdot q$ (using computers based on zeros and ones). Encrypting a message m (represented as a number) is as simple as computing $m^e \bmod n$, where e and n are the public encryption parameters called the *public key*; but recovering the message m is very hard if you do not know which p and q were chosen by the recipient when they set up the scheme. We call the extra information that the recipient uses to decrypt the *private key*.

1.1 Cryptography in the internet age

The most significant development in communication after the written word is probably the invention of the internet. Ever since its invention, usage of the internet has only ever gone up and up. In 2022, 96.8 % of people over the age of 12 in The Netherlands had internet access at home; and 89.5 % used the internet (almost) every day [96]. They use it to interact not just with their friends and play games: today, you might go online to talk to your doctor, file a tax return, or order anything ranging from weekly groceries to niche computer equipment [95, 272]. Indeed, one of the most valuable companies in the world is the online retailer Amazon [355].

None of these applications would have been possible without a way to communicate securely over the normally unprotected internet. The Transport Layer Security (TLS) protocol, invented with the name SSL by browser developer Netscape in 1995 [138], has perhaps been one of the most important enablers for the digital economy by allowing secure processing of credit card information on webshops. An ever-increasing number of websites are also just using TLS to provide their visitors with more privacy, by protecting browsing traffic from eavesdroppers on the network. TLS is not just used for websites; it is used for many applications, ranging from VPNs [280], to secure file transfer [146] and email [177, 275]. Even offline, TLS can be used, for example, to secure networks inside of cars [360].

TLS has seen a lot of development since its initial version. Its latest version is TLS 1.3, which was released in 2018 by the standardization organization Internet Engineering Task Force (IETF) as RFC 8446 [298]. Many extensions and additional features were developed, but the principle behind the main protocol has stayed the same.

TLS allows a client, such as a web browser, to connect to a server, such as the one hosting a website. An important twist is that the client may not know the server's (cryptographic) identity before the connection is initiated. This is an important requirement because it means that TLS cannot just rely on having the keys of anyone the client might talk to beforehand. Instead, TLS relies on a mechanism called *certificates*. A certificate states that a certain public key belongs to a certain name. The certificate is signed by a trusted third party, the certificate authority. We use certificates in TLS to transfer the identity of the server to the client during the connection setup. The client thus receives the identity of the server, and because the client can verify the certificate authority's signature on the certificate it knows that it can be trusted.

1.2 Quantum Computers

As already alluded to above, there are different kinds of computers. All the computing devices we rely on today are based on discrete numbers; famously the binary zeros and ones.[1] Representing information and algorithms using these "off" and "on" states directly translates to the electronic circuits based on transistors that we use to build modern microchips and computer memory. Physicists, however, figured out long ago that the rules of the universe are not so discrete. Richard Feynman already theorized in the 1980s about simulating quantum physics using computers based on quantum systems rather than switches [142]. The quantum bit or qubit encodes information not as just a 0 or a 1, but as a *probability* of being 0 or 1. Oversimplifying this, a qubit can be both values at the same time, which we call superposition. This and other mechanisms like quantum interference and entanglement allow accurate simulation of quantum systems, which cannot be represented on a "classical" discrete computer.

For a very long time, quantum computers have been a theoretical concept. Quantum computers are not even suitable for all computation problems. However, quantum algorithms have been developed for various problems that we do not know how to solve efficiently on a classical computer [199]. This includes quantum simulation and approximation problems, as well as various number-theoretic problems.

[1] Binary representation is not a strict requirement: some of the earliest computers used decimal representations instead of binary, including the ENIAC [160].

For this thesis, we are mainly concerned with the algorithms invented by Shor. His quantum algorithms for discrete logarithms and factoring suddenly make the mathematical trapdoors on which we built the currently widely-used public-key cryptographic schemes trivial to solve [324]. Meanwhile, more people than ever are working to make the quantum computer a reality. Each year, new records get broken. At the end of 2022, IBM announced the largest quantum processor yet. This processor, to be made available in 2023, features 433 physical qubits [190], a more than 3 times increase over their 127-qubit processor from the year before. Although we are a long way from what is sometimes referred to as a "cryptographically relevant" quantum computer, these advances and advances in, for example, quantum error correction show that the field of quantum computing is rapidly moving forward [259].

1.3 Post-quantum cryptography

Although cryptographically-relevant quantum computers are not yet available, it is important to start looking for alternatives to the public-key cryptography algorithms that are threatened by them. The first reason is the simple fact that the development of new algorithms takes time; the second reason is the fact that any information that is protected using algorithms that are vulnerable to cryptanalytical attacks using quantum computers might be stored now and decrypted once the cryptographically-relevant quantum computer exists. This also leads to what is known as Mosca's inequality [258] (figure 1.1): if it takes X year to deploy new algorithms, and information needs to be secure for Y years, and we have Z years until a quantum computer is available, then, if $X + Y > Z$, there is a window of time in which data is vulnerable. Because Z is unknown, we have a strong incentive to make X as small as possible.

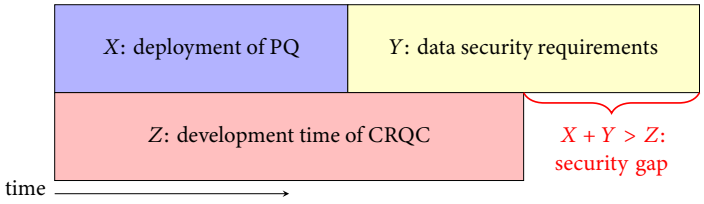


Figure 1.1: Mosca's inequality [258]

Post-quantum cryptography describes the field of cryptography that aims to resist attacks using quantum computers. Post-quantum algorithms are based on different mathematical problems than their pre-quantum equivalents. These problems are conjectured to be hard to solve even if you have a cryptographically-relevant quantum computer. We do not need a quantum computer for these new algorithms; we can still run them on any phone, tablet, laptop, or smart fridge.

The proposed algorithms do behave differently from the cryptography that we are used to today. Pre-quantum RSA [303], Diffie–Hellman [119], and elliptic-curve algorithms [212, 256] are fast to execute *and* the sizes of the keys and digital signatures are quite small. Many of the proposed new algorithms present new trade-offs, however. Lattice-based schemes Kyber [319] and Dilithium [241] are very fast, but the keys, ciphertexts, and signatures are more that an order of magnitude larger than Diffie–Hellman or RSA. CSIDH [91], a scheme based on isogenies between supersingular elliptic curves, has much more modest key sizes but requires a lot of heavy computation. Some schemes have very large public keys, but very small signatures, or vice-versa. Additionally, key exchange and signing operations in pre-quantum schemes are very similar in performance and message size. However, in the post-quantum world, it appears that there is a new gap between these two operations to account for, and this makes deploying them more challenging.

1.4 We

> *You and me could be we*
> *— Unconditional II (Race and Religion) by Arcade Fire*

The work done in this thesis would have been much harder without the support of my co-authors. This is one of the reasons why, throughout this thesis, I often write 'we'. 'We' can also refer to the cryptographic community, or to the large amount of work on which the results in this thesis build. When for example discussing desired properties or security proofs, 'we' may also refer to not just the author, but also the reader, for example when the author takes the reader through the reasoning in the proof as Vergil guided Dante Alighieri (through similarly dangerous and scary environments). Finally, 'we' just refers to me, especially when *I* made any mistakes.

1.5 Organization

These new post-quantum schemes and their trade-offs present us with the main research theme in this thesis: what trade-offs make sense? We focus on TLS, perhaps the most important cryptographic protocol on the internet, and investigate the impact of post-quantum cryptography on its performance.

After this introduction and the discussion of notation, definitions, and common notions in the preliminaries in chapter 2, this book has three parts. Part I first discusses existing proposals to transition the TLS 1.3 handshake to post-quantum cryptography, and re-examines an early proposal for the TLS 1.3 handshake, OPTLS, in light of the new post-quantum primitives. Examining the trade-offs between key exchange and signing operations leads to the proposal of KEMTLS, an alternative TLS handshake protocol that authenticates the participants through a key exchange rather than a signature. Finally, we discuss an extension to KEMTLS, called KEMTLS-PDK.

Part II covers the proofs of security of KEMTLS and KEMTLS-PDK. We first discuss the security properties and two pen-and-paper proofs in the computational model. Finally, we model and present a computer-verified proof of the security of KEMTLS and KEMTLS-PDK in the symbolic model.

In the last part, part III, we collect all experimental results for post-quantum instantiations of TLS 1.3, KEMTLS, and KEMTLS-PDK. We discuss how we have implemented and measured these protocols on simulated networks, but also discuss an experiment that was run over the internet and an experiment that ran KEMTLS on a microcontroller.

Finally, we wrap up with a summary of the results and discuss how post-quantum TLS will keep developing.

1.6 Original works

As customary in cryptography, as a subfield of mathematics, most of the publications in this work list their authors in alphabetical order. This not just side-steps the question of assigning an often difficult-to-determine ranking of individual contributions; this tradition also recognizes that sometimes small contributions can be very influential to the direction work develops in. In this section, I provide an overview of the previously published work that has come together in this thesis. Although I will highlight my contributions to

these works, I again express my gratitude to my co-authors. The remainder of this section is organized by publication, as I split many publications across different chapters of this thesis.

Post-quantum TLS without handshake signatures (ACM CCS 2020)

The basis of this thesis lies in this publication, which was originally presented at ACM CCS 2020. In it, we examine the performance of post-quantum TLS handshakes (chapter 11), and briefly discuss OPTLS (chapter 4). The core contribution lies in the proposal of KEMTLS, an alternative to the TLS 1.3 handshake which eliminates signatures from the handshake. The proposal of KEMTLS, which originates in this publication, is discussed in chapter 5. The original pen-and-paper proof of KEMTLS, the full version of which first appeared in the online version of this publication, appears in a more developed version in chapter 7. Finally, this paper contains the initial implementation of KEMTLS, and we compare the performance of post-quantum TLS 1.3 and KEMTLS. Chapters 10, 11 and 13 are based on these results.

> Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures." In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. Updated version available via url and IACR ePrint. Virtual Event, USA: ACM Press, Nov. 9–13, 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350. IACR ePrint: ia.cr/2020/534. URL: wggrs.nl/p/kemtls

In this work, I was the main author. I contributed to the design of the protocol and the security and performance analysis. I also implemented the protocol and collected the benchmarking results. Discussions with Peter Schwabe and Douglas Stebila, as well as their contributions to the writing of the paper, the protocol design, and the security analysis, were invaluable.

More efficient post-quantum KEMTLS with pre-distributed public keys (ESORICS 2021)

In this follow-up work to "Post-Quantum TLS Without Handshake Signatures," presented at ESORICS 2021, we investigate a variant of KEMTLS. This variant, KEMTLS-PDK, uses the fact that often clients have some prior knowledge

of the server's identity. This is for example true for web browsers, connecting to the same server many times, or mobile apps or IoT devices that have the server information hard-coded. By using a stored server long-term public key, the client can short-circuit the KEMTLS authentication key exchange, submitting the encapsulation to the server's long-term key in the client's initial TLS message. As this initial message is replayable, we require careful analysis of the forward secrecy and authentication properties of the KEMTLS-PDK handshake. We can also use the encapsulated key to submit early an encrypted message containing the client certificate to the server for faster client authentication. Although there exist session ticket- and pre-shared-key-based handshakes in TLS 1.3, those mechanisms rely on symmetric key cryptography, which may require secure storage and has key management concerns.

The proposal of KEMTLS-PDK is the basis of chapter 6. We discuss the pen-and-paper proof of the security of this protocol in chapter 8, but we additionally adopted the more expansive model from this paper in the proof of KEMTLS that we discuss in chapter 7. The improved implementation of post-quantum TLS 1.3, KEMTLS, and KEMTLS-PDK from this paper forms the basis of all the current implementations discussed in chapter 10, and we use this code for the measurements in chapters 11, 13 and 14. This implementation is also used as the server in the results presented in chapter 16. The comparison of the performance of KEMTLS-PDK with TLS 1.3 (with an additional caching mechanism) and KEMTLS is discussed in chapter 14.

> Peter Schwabe, Douglas Stebila, and Thom Wiggers. "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys." In: *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part I*. ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Updated version available via url and IACR ePrint. Darmstadt, Germany: Springer, Heidelberg, Germany, Oct. 4–8, 2021, pp. 3–22. DOI: 10.1007/978-3-030-88418-5_1. IACR ePrint: ia.cr/2021/779. URL: wggrs.nl/p/kemtlspdk

In this work, I was the main author. I contributed to the design of the protocol and the security and performance analysis. I also implemented the protocol, collected the benchmarking results, and contributed to the write-up. Discussions with Peter Schwabe and Douglas Stebila, as well as their contributions to the writing and security analysis, were invaluable.

KEMTLS with delayed forward identity protection in (almost) a single round trip (ACNS 2022)

This work, which was developed concurrently to the KEMTLS-PDK paper "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys," proposes a variant of the KEMTLS-PDK protocol in which additional short-term keys are used to provide some level of forward secrecy to the client certificate message in the KEMTLS-PDK handshake. We prove the security of this variant protocol in a different model from the previous work, and carefully analyze the forward secrecy gained by the short-lived keys. It was originally published in the proceedings of the ACNS 2022 conference. We briefly discuss this work in chapter 6.

> Felix Günther, Simon Rastikian, Patrick Towa, and Thom Wiggers. "KEMTLS with Delayed Forward Identity Protection in (Almost) a Single Round Trip." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Rome, Italy: Springer, Heidelberg, Germany, June 20–23, 2022, pp. 253–272. DOI: 10.1007/978-3-031-09234-3_13. IACR ePrint: ia.cr/2021/725. URL: wggrs.nl/p/kemtls-epoch

This variant of KEMTLS-PDK was originally designed by Patrick Towa, and the proof has been constructed by Patrick Towa and Felix Günther. I helped Simon Rastikian with the implementation of the protocol (which was based on my implementation of KEMTLS), and I did the collection and analysis of the measurement results.

A tale of two models: formal verification of KEMTLS via Tamarin (ESORICS 2022)

Chapter 9 discusses this work, in which we model and prove the security of KEMTLS and KEMTLS-PDK in the Tamarin [27] symbolic analysis tool. It was originally presented at ESORICS 2022. The work presents two different approaches to modeling KEMTLS(-PDK). The first approach adopts the existing model of TLS 1.3 by Cremers, Horvat, Hoyland, Scott, and Van der Merwe [108] and adapts it to KEMTLS. The second sticks close to our pen-and-paper models and states the security properties of KEMTLS(-PDK) in the

same way. It focuses on the core handshake protocol, and mechanisms such as handshake encryption were simplified or left out. The second approach allowed us to validate our pen-and-paper models and found some minor flaws in the security claims. These flaws are corrected in the proofs shown in chapters 7 and 8.

> Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. "A Tale of Two Models: Formal Verification of KEMTLS via Tamarin." In: *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part III*. ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. Lecture Notes in Computer Science. Extended version available via URL and IACR ePrint. Copenhagen, Denmark: Springer, Heidelberg, Germany, Sept. 26–30, 2022, pp. 63–83. DOI: 10.1007/978-3-031-17143-7_4. IACR ePrint: ia.cr/2022/1111. URL: wggrs.nl/p/kemtls-tamarin

In this work, I was one of the two main authors together with Douglas Stebila. I mainly worked on the first approach: modifying and proving the security of KEMTLS in the modified TLS 1.3 model. Douglas Stebila contributed to the second approach. Jonathan Hoyland was indispensable for helping with, and giving insight into Tamarin, and together with Sofía Celi helped with the writing of this paper.

Implementing and measuring KEMTLS (LATINCRYPT 2021)

Chapter 15 discusses the work from this publication, which was originally presented at LATINCRYPT 2021. In it, we again measure and compare the performance of post-quantum TLS with the performance of KEMTLS(-PDK), but instead of an emulated network environment, the experiment uses connections between two data centers, one in Portland and one in Lisbon, and runs a real-world application over the post-quantum TLS and KEMTLS connections. Additionally, this chapter describes how experiments with post-quantum authentication can be done without post-quantum certificates, using a mechanism called delegated credentials.

> Sofía Celi, Armando Faz-Hernández, Nick Sullivan, Goutam Tamvada, Luke Valenta, Thom Wiggers, Bas Westerbaan, and Christopher

A. Wood. "Implementing and Measuring KEMTLS." in: *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America*. Ed. by Patrick Longa and Carla Ràfols. Vol. 12912. Lecture Notes in Computer Science. Bogotá, Colombia: Springer, Heidelberg, Germany, Oct. 6–8, 2021, pp. 88–107. DOI: 10.1007/978-3-030-88238-9_5. IACR ePrint: ia.cr/2021/1019. URL: wggrs.nl/p/measuring-kemtls

In this work, the main contributions were by Sofía Celi. Armando Faz-Hernández contributed optimized implementations of post-quantum primitives. My involvement was limited to the analysis of the obtained results and the writing of the paper. I have also made some minor contributions to the implementation.

KEMTLS vs. post-quantum TLS: performance on embedded systems (SPACE 2022)

Previous chapters discuss the performance of KEMTLS clients that are run on powerful desktop and server-grade CPUs. In chapter 16, which is based on this work that was presented at SPACE 2022, we examine the performance of post-quantum TLS 1.3 and KEMTLS when the client is a small microcontroller connected over a low-bandwidth network.

Ruben Gonzalez and Thom Wiggers. "KEMTLS vs. Post-quantum TLS: Performance on Embedded Systems." In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Lejla Batina, Stjepan Picek, and Mainack Mondal. Jaipur, India: Springer Nature Switzerland, Dec. 9–12, 2022, pp. 99–117. DOI: 10.1007/978-3-031-22829-2. URL: wggrs.nl/p/kemtls-embedded

In this work, Ruben Gonzalez, the main author, implemented KEMTLS on a microcontroller, designed the experiments, and collected the results. I assisted with the selection of parameters, the analysis, and the writing of the paper.

Improving software quality in cryptography standardization projects (SSR 2022)

Chapter 17 discusses the quality of the implementations that were a part of the NIST post-quantum cryptography standardization project. This paper

was originally presented at the Security Standardization Research (SSR) 2022 workshop, co-located with EuroS&P 2022. Many of the reference implementations of submissions in the standardization project had problems, such as a lack of namespacing, which make them hard to integrate with other software for scientific experiments. Furthermore, we found that most of the reference implementations had bugs, ranging from undefined or platform-specific behavior to dead code. We describe in this chapter and the work it was based on how NIST could have made the reference implementations much more useful for the scientific field, for example to our experiments; we also fix many reference implementations by hand and by subjecting them to a testing framework that is part of *PQClean*, a repository of these *cleaned-up* implementations.

> Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Improving Software Quality in Cryptography Standardization Projects." In: *SSR 2022: Security Standardization Research (2022 IEEE S&P Workshops)*. Genoa, Italy: IEEE Computer Society, June 6–10, 2022, pp. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010. URL: wggrs.nl/p/pqclean

All authors contributed equally to this paper. However, I am the main contributor to PQClean. I have designed the majority of the testing and CI framework and integrated many schemes into PQClean. Separately, I maintain a set of Rust crates that package implementations from PQClean for use in Rust and maintain the Rust library for Open Quantum Safe.

Optimizations and practicality of high-security CSIDH (preprint)

Chapter 4 discusses OPTLS, an early proposal for the TLS 1.3 handshake that relies on non-interactive key exchange (NIKE). The performance results discussed in chapter 12 are partially based on this work in which we examine higher-security instantiations of CSIDH, based on conservative security analysis. The implementation of OPTLS used in the experiments was part of this paper, and we report some measurement results for high-security CSIDH parameters from this work.

> Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez,

Peter Schwabe, and Thom Wiggers. "Optimizations and Practicality of High-Security CSIDH." in submission. 2023. IACR ePrint: ia.cr/2023/793

In this manuscript, I contributed the implementation of OPTLS and the integration of the proposed CSIDH parameters into the implementation by providing a Rust wrapper around the optimized implementations. I have measured the performance of OPTLS when instantiated with our CSIDH parameters and analyzed the results.

Verifying post-quantum signatures in 8 kB of RAM (PQCrypto 2021)

This paper is not part of the main contributions of this thesis, but appears in appendix A. It was originally published in the proceedings of the PQCrypto 2021 conference and was also presented at the third National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) standardization conference. In it, we discuss how signatures can be verified on embedded devices with small amounts of memory, by streaming the signature into a verification function and incrementally computing its correctness. This strategy may, for example, be useful for hardware security modules in constrained environments, such as the automotive space: we discuss the automotive use-case of feature activation in this work.

Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. "Verifying Post-Quantum Signatures in 8 kB of RAM." in: *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Daejeon, South Korea: Springer, Heidelberg, Germany, July 20–22, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12. IACR ePrint: ia.cr/2021/662. URL: wggrs.nl/p/verifying-post-quantum-signatures-in-8kb-of-ram

The main contributions in this paper are by Matthias J. Kannwischer and Ruben Gonzalez, who have implemented the schemes and collected the results. I have contributed to the discussions on the use case and constraints, and contributed appendix A.6 which describes a symmetric-cryptography-based key-diversification scheme.

Practically Solving LPN (IEEE ISIT 2021)

This second paper that is not part of the main contributions in this thesis, appears in appendix B. It was originally published at IEEE ISIT 2021. The paper discusses the Learning Parity with Noise (LPN) problem, which is related to the security of some post-quantum schemes, and shows how different cryptanalytical approaches can be combined to incrementally reduce and eventually solve the problem. This approach has been previously proposed, but we show that the search problem finding the most-efficient attack chain for a particular LPN problem instance can be reduced. Furthermore, we show how to find the most efficient attack that fits within a given memory limit, as we argue that memory is often more expensive than time for realistic attacks. As a result we were able to mount practical attacks on the largest parameters reported as of publication of this work, using only 2^{39} bits of memory.

> Thom Wiggers and Simona Samardjiska. "Practically Solving LPN." in: *2021 IEEE International Symposium on Information Theory (ISIT)*. Melbourne, Australia: IEEE Information Theory Society, July 12–20, 2021, pp. 2399–2404. DOI: 10.1109/ISIT45174.2021.9518109. IACR ePrint: ia.cr/2021/962. URL: wggrs.nl/p/lpn

This paper is in large part based on my Master's thesis. I have written large parts of this paper and wrote the software that was used for the execution of our found attack chains. Simona Samardjiska contributed the analysis and proofs that reduce the search space for attack chains, and wrote the software that finds the most efficient combinations of algorithms.

1.7 The NIST post-quantum cryptography standardization project

The work in this thesis did not take place in a vacuum but against the backdrop of the NIST PQC standardization project and the overall discussion around transitioning to post-quantum cryptography. While this certainly ensured that working in this area never got stale, it does mean that some of the original results have been caught up by the passing of time. The oldest work in this thesis was done while the standardization project was in its second phase. At the time, there were 17 key encapsulation mechanisms (KEMs) and 9

signature schemes still in the running. At the end of round 3, in July 2022, NIST announced that the KEM Kyber [319] and signature schemes Dilithium [241], Falcon [293] and SPHINCS$^+$ [184] would be the first PQC standards. Of the other KEMs, only BIKE [17], Classic McEliece [7], and HQC [3] are still under consideration for standardization; for signature schemes, NIST has opened up submissions [267] for new proposals that our work never considered.

In this thesis, we update some of the results to the latest state of the submissions. We will make note of the versions of the schemes when discussing them. However, we will still discuss eliminated schemes in some chapters. Rainbow [122], which got significantly wounded by cryptanalysis by Beullens during the third round of the standardization project [50, 51], still features in results in chapter 16. This signature scheme remains relevant in the context of our results, as it is representative of schemes that have very small signatures but very large public keys. (Indeed, UOV [53], a submission to the NIST call for new signature schemes that came in too late for inclusion in our experiments, fits in this category). SIKE [197], which was catastrophically broken just after it was named a round-4 candidate [90, 242, 304], still features in experimental results in chapter 15. This KEM remains illustrative of the performance of schemes with small sizes, but large computation times.

2 Preliminaries

In this chapter, we will provide definitions and background information on which the remainder of this thesis will build. Though it often feels comfortable and sometimes even more intuitive to provide descriptions of protocols and primitives in prose, making concrete statements about the security of our protocols requires rigid notation and formal definitions. We will describe the primitives below, as well as formal definitions of some security properties and security experiments that we will need for the proofs in part II of this thesis.

2.1 Notation

Let $\mathbb{N}$ denote the set of natural numbers $(0, 1, \ldots)$. We write $a = b$ to denote equality, and $x \leftarrow 1$ denotes assignment of the value 1 to a variable x. Arbitrary sets are written as $\{a, b, c\}$ where a, b, and c are elements of the set. We denote that a variable is part of a set by $a \in X$ and denote the Cartesian product of sets X and Y by $X \times Y$. The notation X^n defines the set of n-length sequences of elements in X. For a set X, the notation $x \leftarrow\!\!\$\ X$ denotes sampling an element uniformly at random from X and storing it in x. If $\mathcal{A}$ is a deterministic algorithm, then $y \leftarrow \mathcal{A}(x)$ denotes running $\mathcal{A}$ with input x and storing the output in y. If $\mathcal{A}$ is a probabilistic algorithm, then $y \leftarrow\!\!\$\ \mathcal{A}(x)$ denotes running $\mathcal{A}$ with input x and uniformly random coins, and storing the output in y. The notation $[\![x = y]\!]$ resolves to 1 if $x = y$, and 0 otherwise. We write vectors as $V = (a, b, c)$, and V_i denotes the value at position i in the vector, starting from position 1. Sometimes we write $(v^{\times n})$ as shorthand for a vector where the value v repeats n times. Similarly to vector indexing, for matrices we write $M_{i,j}$ to denote the value in row i, column j, both starting from 1.

We will make use of many protocol diagrams throughout this thesis. In figure 2.1, we show the diagram style used for sketches of protocols. In these sketches, we often simplify the protocol, for example, leaving out message syntax or inputs to key-derivation function. State and computations described

on the left-hand side of the figure are executed by the party on the left-hand side; those defined on the right-hand side are executed by the other party. State and computations centered in the diagram describe both parties in the protocol. The arrows describe the sending of information to the other party. On top of the arrows, we write the information sent.

Inevitably, we will have to more carefully define the protocols that we propose. We will have to exactly state the computations of keys, messages sent, and encryption used. In figure 2.2, we give an example of the notation used in the detailed protocol descriptions. We still have two parties, one on each side of the diagram. Text aligned to the left- or right-hand sides again indicates the state held or actions executed by that party. We write the messages sent colored in green as message. Usually, we give some of the relevant information included in that message. Multiple messages may be sent in one transmission to the server. To emphasize this, we often batch the messages in our figures before we draw a transmission arrow. Unlike in our protocol sketches, we do not write what is sent on top of the arrows: simply all prior unsent messages are included in the transmission. For the multi-stage security proofs in part II, we will denote when particular keys, for which we will make security claims, are *accepted*. This is denoted by **accept** key and a dotted line stating the numbered stage. Messages that follow a stage may be sent encrypted using that stage's (or any prior stage's) key: we denote this by $\{\text{Message}\}_{\text{stage}_i}$. Finally, we denote when we allow first transmission of application traffic by a dotted, dark-gray arrow.

Even our most detailed figures will still have to leave out information that is necessary for implementation. This includes, for example, precise message syntax and code points; for these and other details, we defer to the referenced (draft) TLS standards and our implementations.

As we will frequently be discussing details of the TLS handshake, we include a list of abbreviations used for TLS messages and handshake keys on page 405. Usually, our notation for messages, for example `ServerCertificate`, indicates which party to the handshake is the intended sender. Sometimes, when this information is not relevant or is clear from context, we may simply write `Certificate` instead. In our figures we will often use the abbreviations of TLS messages to indicate message transcripts: we denote this by `CH` … `CF`. This example denotes all transmitted messages starting from the moment `ClientHello` was sent up to and including the `ClientFinished` message.

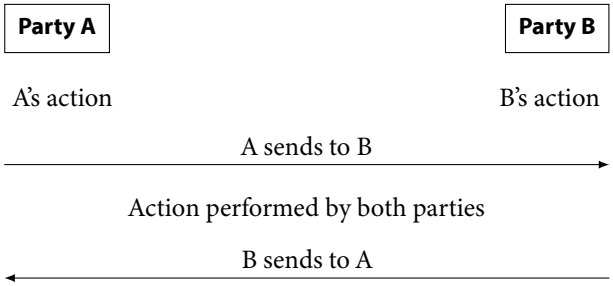


Figure 2.1: Example of the diagrams used to sketch protocol flows

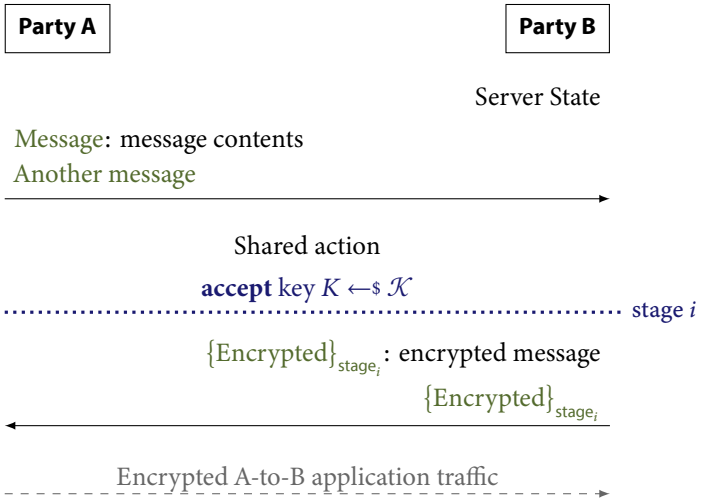


Figure 2.2: Example of the diagrams used for detailed protocol definitions

2.2 Cryptography

Cryptography is the art of securing information and communication from adversaries. We call hiding the information *encrypting* and recovering it *decrypting*. This is usually done with a *key*, some additional information that is the basis of the security. The key should additionally be the only secret thing on which the security of the scheme relies; even if the adversary otherwise knows all the details of the scheme, it should still be secure (*Kerckhoff's principle*) [209].

The information security properties fall into three pillars:

Authentication Ensuring that are we communicating with the intended party;

Confidentiality Keeping the communication unreadable to outside parties, and;

Integrity Ensuring that the contents of the communication cannot be tampered with.

These properties are closely related: for example, a lack of authentication often allows an attacker, which we refer to as the adversary, to break confidentiality by posing as the intended recipient.

Cryptographic schemes may rely on many assumptions and be vulnerable to various forms of cryptanalysis. There are also some tweakable "knobs" when schemes are designed, such as the sizes of keys, that relate to the provided security. Usually, more security implies larger keys and slower computations. To be able to compare and choose appropriate instances of cryptographic primitives, the security level of an algorithm is often expressed in bits.

Definition 2.1 (Security level in bits)**.** Given a *security level of n bits* for a given cryptographic system against a particular attack, a polynomial-time probabilistic adversary $\mathcal{A}$ has at most 2^{-n} chance of performing the attack successfully, i.e.:

$$\Pr[\mathcal{A} \text{ successful}] \leq 2^{-n}.$$

Alternatively, $\mathcal{A}$ requires at least 2^n operations to be likely to succeed.

In proofs, we often model the security properties of a particular system as a security experiment we call a *game*. In such a game, the adversary can *win* if they provide a certain output. This can be a particular input to a function, such as a key, but it is often a bitvalue that was set at the start of the game.

Definition 2.2 (Advantage)**.** To win a security game, the adversary may be able to do better than just guessing. We call this the *advantage* of the adversary in the security game. In the game G for a primitive P, we write the *advantage* of a polynomial-time adversary $\mathcal{A}$ as

$$\mathsf{Adv}^{G}_{P,\mathcal{A}}.$$

We often need the advantage to be *negligible*, by which we mean that it is less than 2^{-n} for some large enough n: for example, the security level of n bits.

2.3 Symmetric cryptography

Although the focus of this thesis is on (asymmetric) key-exchange protocols, we do make use of symmetric cryptographic primitives. The defining characteristic of symmetric primitives, as opposed to asymmetric primitives, is their lack of a "public" operation. To be able to provide confidentiality or authenticity with a symmetric algorithm, both the sender and the recipient of a message will use all the inputs to the algorithms. This usually means both parties share a key that must be kept secret from anyone else; which explains the common alternative name *secret-key cryptography*. This key may be set before communication happens, but often it will be agreed upon by the prior execution of a key-exchange protocol. Using a symmetric encryption algorithm is in practice preferred over asymmetric encryption algorithms, as the latter are typically less performant.

2.3.1 Authenticated Encryption

In modern versions of TLS, application data, and some handshake messages are encrypted using authenticated-encryption algorithms. Authenticated encryption provides both confidentiality and integrity. A subset of these algorithms allows for additionally specified "associated data", which will only be integrity-protected. This may for example be used to specify metadata. Algorithms that allow associated data are referred to as authenticated encryption with associated data (AEAD) algorithms.

Definition 2.3 (Authenticated Encryption)**.** We write the *authenticated encryption with associated data* encryption of message m with algorithm AEAD

under an appropriate key $K \in \mathcal{K}$ as follows:

$$\mathsf{AEAD}_K(m).$$

Algorithm AEAD gives us at least n-bit security guarantees for the integrity and confidentiality of m.

We do not make explicit use of associated data in the descriptions or proofs contained in this thesis, so we leave it out of our notation. In all protocol descriptions, we additionally assume that symmetric decryption and verification are handled implicitly by the recipient, as they should know K to any message that was (honestly) sent to them. If the recipient cannot decrypt or verify, they will abort the protocol.

In TLS 1.3, two AEAD algorithms are currently required for implementations: 128-bit AES-GCM [309] and ChaCha20-Poly1305 [276].

2.3.2 Hash functions

Hash functions map arbitrary inputs to fixed-length outputs. In this thesis all inputs and outputs are bitstrings.

Definition 2.4 (Hash function)**.** A *hash function*

$$\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$$

maps arbitrary-length messages $m \in \{0,1\}^*$ to a hash value $\mathsf{H}(m) \in \{0,1\}^\lambda$ of fixed length $\lambda \in \mathbb{N}$.

We denote the application of hash function H on message m as simply

$$\mathsf{H}(m).$$

Good *cryptographic* hash functions additionally provide:

pre-image resistance: it must be hard to recover the input m from any output $\mathsf{H}(m)$;

second pre-image resistance: it must be hard to recover any other input m', $m' \neq m$ that produces the given output $\mathsf{H}(m)$; and

collision resistance: it must be hard to find two arbitrary inputs m, m' that produce the same output $\mathsf{H}(m) = \mathsf{H}(m')$.

In the security analyses in part II, we will rely on collision resistance:

Definition 2.5 (Collision resistance)**.** The *collision resistance* of a hash function H measures the ability of a polynomial-time adversary $\mathcal{A}$ to find two distinct messages that hash to the same output:

$$\mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{A}} = \Pr\left[(m, m') \leftarrow\!\!\$\ \mathcal{A} : (m \neq m') \wedge (\mathsf{H}(m) = \mathsf{H}(m'))\right].$$

Many mechanisms build on hash functions. This includes key-derivation functions and message-authentication codes which we will describe next. In TLS, hash functions are also used as a state compression trick; it is easier to keep track of the hash of the state rather than keeping all previous messages in memory.

The hash functions currently supported in TLS 1.2 and TLS 1.3 are all from the SHA-2 family: SHA256, SHA384, and SHA512 [298, 300].

2.3.3 Message-authentication codes

Message-authentication codes (MACs) allow computing an *authentication tag* from a shared secret key and a message. The recipient of a tagged message can compute the tag independently and compare it to the received tag, thus ensuring the message is authentic and was not tampered with.

Definition 2.6 (Message-authentication code)**.** A *message-authentication code* (MAC)

$$\mathsf{MAC} : \mathcal{K} \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$$

maps a key $k \in \mathcal{K}$ and a message $m \in \{0,1\}^*$ to an authentication tag of fixed length in $\{0,1\}^\lambda$.

It must be impossible for tags for any messages to be created by anyone other than the legitimate participants with the keys. So in the EUF-CMA security experiment for MACs we specify that even a polynomial adversary that can ask for arbitrary messages and tags cannot create a valid tag for a new message without querying the oracle.

Definition 2.7 (Existential unforgeability under chosen-message attack for MACs)**.** The *existential unforgeability under chosen-message attack* (EUF-CMA) for MACs measures the ability to forge an authentication tag on a new message in polynomial time, given access to a tag-generation oracle, as shown in figure 2.3:

$$\mathsf{Adv}^{\text{EUF-CMA}}_{\text{MAC},\mathcal{A}} = \Pr\left[G^{\text{EUF-CMA}}_{\text{MAC},\mathcal{A}} \Rightarrow 1\right].$$

$G^{\text{EUF-CMA}}_{\text{MAC},\mathcal{A}}$

1: $k \leftarrow\$ \mathcal{K}$
2: $L \leftarrow \varnothing$
3: $(m, t) \leftarrow \mathcal{A}^{\mathcal{O}}$
4: **return** $[\![(t = \mathsf{MAC}(k, m)) \wedge (m \notin L)]\!]$

Oracle $\mathcal{O}(z)$

1: $L \leftarrow L \cup \{z\}$
2: **return** $\mathsf{MAC}(k, z)$

Figure 2.3: Security experiment for existential unforgeability under chosen-message attack (EUF-CMA-security) of a message-authentication code MAC.

TLS uses the HMAC [218] algorithm for message-authentication codes.

2.3.4 Key-derivation functions

Key-exchange protocols produce cryptographic keys, but those keys are often not immediately useful. Often, they are not the right length for the symmetric encryption algorithms or MACs, which usually need 128- or 256-bit keys. We may also need to expand a single secret key into multiple distinct keys. Key-derivation functions (KDFs) serve to compute appropriate keys for symmetric cryptographic algorithms from arbitrary-length inputs. They cannot add entropy to their inputs, so the inputs must be of appropriate quality if the output keys need to be unpredictable.

In TLS 1.3, the HKDF [214, 219] KDF is used. It is a set of two algorithms that follow an "extract-then-expand" paradigm. The HKDF.Extract algorithm takes two inputs, namely a salt, and the input keying material. The salt is an optional, random value that does not need to be secret. It outputs a pseudorandom key *PRK*. This key can then be used multiple times in HKDF.Expand calls. This algorithm takes *PRK*, context information, and the desired output length. In TLS 1.3, the context information is set to a label and optionally a hash of

the handshake transcript. The labels separate the different domains in which the key is used, and the transcript ensures the key is bound to the specific TLS session. In our notation, we will omit the output length, and assume an appropriate length for the output key is chosen implicitly.

In our security analyses, we assume HKDF is a pseudorandom function, and we rely on the (dual-)PRF-security assumption.

Definition 2.8 (Pseudorandom function)**.** A *pseudorandom function*

$$\mathsf{PRF} : \mathcal{K} \times \mathcal{L} \to \{0,1\}^\lambda$$

maps a key $k \in \mathcal{K}$ and a label $\ell \in \mathcal{L}$ to an output of fixed length in $\{0,1\}^\lambda$.

Definition 2.9 ((Dual-)PRF security)**.** The *PRF-security* of a pseudorandom function PRF measures the ability of a polynomial-time adversary $\mathcal{A}$ to distinguish the output of PRF from random:

$$\mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{PRF},\mathcal{A}} = \left|\Pr\left[k \leftarrow\!\!\$\ \mathcal{K} : \mathcal{A}^{\mathsf{PRF}(k,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{R(\cdot)} \Rightarrow 1\right]\right|$$

where $R : \mathcal{L} \to \{0,1\}^\lambda$ is a truly random function.

A pseudorandom function PRF satisfies *dual-PRF security* [29] if it is a pseudorandom function with respect to either of its inputs k or ℓ being the key, i.e., if both PRF and $\mathsf{PRF}'(x,y) = \mathsf{PRF}(y,x)$ have PRF-security.

When we instantiate the PRF-security experiment for HKDF.Extract, the input key material is the PRF key, and the salt is the PRF label. The TLS 1.3 key schedule uses the salt argument to pass through the key schedule state while the input key material argument is used to feed in pre-shared keys and Diffie–Hellman key exchange outputs. In TLS 1.3, the input key material may be a string of zero bytes in, for example, handshakes where no Diffie–Hellman key exchange is performed or in the computation of the Main Secret MS. In those cases, as well as in cases where the input key material may be corrupted by an adversary, we rely on the dual-PRF security of HKDF.Extract in its first (salt) argument. For HKDF.Expand, the pseudorandom key *PRK* is the PRF key and the tuple of TLS label and handshake transcript are the PRF label.

Recent work has pointed out there is no security proof for the dual-PRF security of HKDF [18]. However, their attack on the dual-PRF security of the HMAC function underlying HKDF relies on variable-length input key material and a lack of collision resistance in the underlying hash function.

As collision resistance of the hash inside HKDF is required for the security of TLS, and as all input key materials used in TLS are fixed length, we feel comfortable relying on the dual-PRF assumption in our analyses.

2.4 Asymmetric cryptography

As the name suggests, asymmetric cryptographic algorithms have an *asymmetry* as opposed to symmetric algorithms where both parties share the inputs and outputs to all operations. Asymmetric algorithms define a *public* and *private* operation and split the key into appropriate parts, a *public key* and *private key*, to facilitate those operations. A public key can be freely given to anyone, including adversaries; without the private key, they cannot perform the private operation. This means that in a cryptographic system with n participants, we only need to distribute n key pairs to allow any two participants to communicate securely. This is in contrast to the $n(n-1)$ keys that one would need to distribute and securely store in a system based on symmetric cryptography. The most notable asymmetric primitives are key exchange and digital signature algorithms, which we will describe below. Public-key encryption algorithms are not relevant to this thesis, but are closely related to key-exchange algorithms; we leave constructing a naive public-key encryption scheme from a KEM and AEAD as an exercise to the reader.

We point out that in many applications, the *ability to execute* the private operation can serve to prove the possession of the private key, and the products of the private operation are sometimes of arguably less importance. This means that we can use asymmetric algorithms for authentication purposes if instantiated correctly.

2.4.1 Digital signature schemes

A digital signature scheme allows *signing* of messages using a private key. The signature of a message is publicly verifiable by anyone who knows the public key. Digital signatures can thus serve to authenticate the message signed, which is for example used in TLS certificates: these are essentially signed Statements that certain public keys belong to certain identities. As noted above, signatures can also be used to authenticate the signer, by making them sign a challenge message.

Definition 2.10 (Signature scheme)**.** A *signature scheme* Sig defines the operations Sig.Keygen, Sig.Sign and Sig.Verify. The first operation Sig.Keygen will probabilistically generate a keypair consisting of a public key and a private key. Sig.Sign produces a signature given the private key and a message. Finally, Sig.Verify allows verifying this signature given the signed message and public key, Returning whether verification succeeded. For a given message m, public key pk, and private key sk, we write signing of m and verification of the signature as below. We require that the following equality holds for correctness:

$$\mathsf{Sig.Verify}\,(\mathsf{pk}, \mathsf{Sig}(\mathsf{sk}, m)) = \mathsf{true}.$$

As for message-authentication codes, a polynomial-time adversary must not be able to forge a signature on a message. We define the EUF-CMA security experiment for signature schemes similar to the experiment for message-authentication codes in definition 2.7.

Definition 2.11 (Existential unforgeability under chosen-message attack for signature schemes)**.** The *existential unforgeability under chosen-message attack* (EUF-CMA) for a signature scheme Sig measures the ability to forge a signature on a new message in polynomial time, given access to a signing oracle, as shown in figure 2.4:

$$\mathsf{Adv}^{\text{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}} = \Pr\left[G^{\text{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}} \Rightarrow 1\right].$$

$G^{\text{EUF-CMA}}_{\mathsf{Sig},\mathcal{A}}$

1: $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{Sig.Keygen}()$
2: $L \leftarrow \varnothing$
3: $(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pk})$
4: **return** $[\![\mathsf{Sig.Verify}(\mathsf{sk}, \sigma) \wedge (m \notin L)]\!]$

Oracle $\mathcal{O}(z)$

1: $L \leftarrow L \cup \{z\}$
2: **return** $\mathsf{Sig}(\mathsf{sk}, z)$

Figure 2.4: Security experiment for existential unforgeability under chosen-message attack (EUF-CMA-security) of a signature scheme Sig.

The TLS 1.3 handshake supports RSA [303] and both the ECDSA [13] and EdDSA [44, 201] elliptic-curve signature algorithms.

2.4.2 Diffie–Hellman key exchange

Diffie–Hellman (DH) key exchange [119] based on finite fields or elliptic curves is not resistant to attacks by quantum computers. However, it is used in current versions of TLS for ephemeral key exchange.

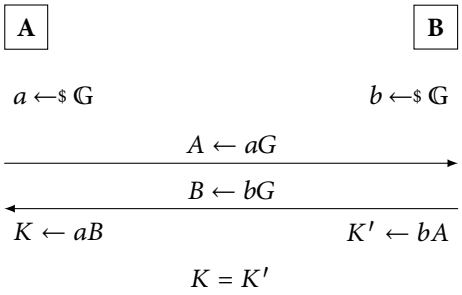


Figure 2.5: Unauthenticated Diffie–Hellman key exchange

In DH key exchange, sketched in figure 2.5, the two participants fix a finite cyclic group $\mathbb{G}$ and a generator $G \in \mathbb{G}$. In practice, DH is often instantiated over elliptic curves, which is generally more efficient than the finite-field-based instantiations of DH. This is also referred to as elliptic-curve Diffie–Hellman (ECDH). Both finite-field DH and ECDH support non-interactive key exchange, which we define next.

2.4.3 Non-interactive key exchange

NIKE schemes go back to the original Diffie–Hellman paper, but have not been studied much until recent years. We will give a slightly simplified version of the definition given by Freire, Hofheinz, Kiltz, and Paterson [147].

Definition 2.12 (Non-interactive key exchange)**.** A *non-interactive key exchange* algorithm NIKE defines operations NIKE.Keygen and NIKE.SharedSecret. The operation NIKE.Keygen probabilistically produces a public key pk and a private key sk. The NIKE.SharedSecret operation takes a public key pk along with a private key sk, and deterministically computes a shared secret ss in key space $\mathcal{K}$. For correctness, we require that for any two sets of public and private keys $(\mathsf{pk}_1, \mathsf{sk}_1)$ and $(\mathsf{pk}_2, \mathsf{sk}_2)$,

$$\mathsf{NIKE.SharedSecret}(\mathsf{pk}_1, \mathsf{sk}_2) = \mathsf{NIKE.SharedSecret}(\mathsf{pk}_2, \mathsf{sk}_1).$$

Non-interactive key exchange only has two operations and requires no exchange of information except for the public keys. As you can pre-generate and publish the public keys anywhere, this allows computing the shared secret completely without interaction. Keys often correspond with identities. A NIKE allows combining a sender's secret key with a recipient's authenticated public key; thus establishing an immediately mutually authenticated key. This property is for example used in Signal's X3DH handshake [245].

It is easy to see that Diffie–Hellman can be used as a (pre-quantum) NIKE, as $b(aG) = a(bG)$. However, as of writing, the only known and somewhat efficient post-quantum NIKEs are CSIDH [91] and its variants, and Swoosh [153]. CSIDH, which is based on isogenies, is much slower than most KEMs, and its security is hotly debated [42, 49, 58, 71, 289]. Swoosh, which is based on Module-LWE, was only proposed in 2023; its public key size exceeds 120 kB for the actively-secure variant, and no parameters have yet been put forward for a passively secure variant. CSIDH and Swoosh are also not part of the NIST PQC standardization project.

TLS 1.3 and prior versions of TLS do not use the non-interactive properties of DH key exchange. However, the original OPTLS proposal for TLS 1.3, which we will discuss in chapter 4, did make use of DH's non-interactive key exchange properties.

2.4.4 Key Encapsulation Mechanisms

A key encapsulation mechanism (KEM) is an asymmetric cryptographic primitive that allows two parties to establish a shared secret key. Although TLS 1.3 is currently not specified to use a KEM, all the key exchange algorithms considered for standardization in the NIST PQC standardization project are KEMs. The ephemeral key exchange in TLS 1.3 can be straightforwardly replaced by a KEM, as we will show in chapter 3. KEMs also form the basis of KEMTLS, which we will discuss in chapter 5.

Definition 2.13 (Key Encapsulation Mechanism)**.** A *key encapsulation mechanism* KEM defines three operations. These are KEM.Keygen, KEM.Encapsulate and KEM.Decapsulate. The KEM.Keygen operation probabilistically generates a public and private keypair (pk, sk). Public operation KEM.Encapsulate takes a pk and probabilistically generates a shared secret ss in key space $\mathcal{K}$ and a ciphertext (encapsulation) ct against a given public key. Finally, private operation KEM.Decapsulate takes sk with the encapsulation ct and decapsulates

the shared secret $\mathsf{ss}' \in \mathcal{K}$ from ct. In a δ-correct scheme (definition 2.14), ss' is equal to ss with probability at least $1 - \delta$.

Figure 2.6 shows a schematic example of unauthenticated key exchange via KEM. Note that KEMs cannot be used to instantiate a NIKE scheme. KEMs require exchanging the ciphertext; they do not allow combining a secret key with a public key in the same way that the NIKE.SharedSecret operation does.

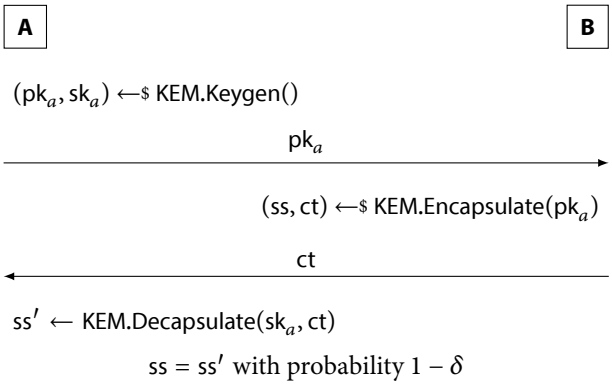


Figure 2.6: Simple key exchange via KEM

Some schemes have a failure probability, in which the two parties in a key exchange disagree on the computed shared secret. As these failures may allow adversaries in our security proofs to distinguish keys from random, we define the delta-correctness of KEMs.

Definition 2.14 (Delta-correctness of KEMs)**.** A key encapsulation mechanism KEM is δ-*correct* [178] if

$$\Pr\left[\mathsf{KEM.Decapsulate}(\mathsf{sk}, \mathsf{ct}) \neq \mathsf{ss} \mid (\mathsf{ss}, \mathsf{ct}) \leftarrow\!\!\$\ \mathsf{KEM.Encapsulate}(\mathsf{pk})\right] \leq \delta,$$

taking the probability over $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KEM.Keygen}()$ and the random coins that are used in KEM.Encapsulate.

The standard security definitions for a KEM require that the shared secret be indistinguishable from random (IND), given just the public key (chosen plaintext attack (CPA)) or additionally given access to a decapsulation oracle (chosen ciphertext attack (CCA)). We make use of a restricted form of IND-CCA

$G^{\text{IND-}atk}_{\text{KEM},\mathcal{A}}$

1: $(\mathsf{pk}^\star, \mathsf{sk}^\star) \leftarrow\$ \text{KEM.Keygen}()$
2: $b \leftarrow\$ \{0,1\}$
3: $(\mathsf{ss}_0^\star, \mathsf{ct}^\star) \leftarrow\$ \text{KEM.Encapsulate}(\mathsf{pk}^\star)$
4: $\mathsf{ss}_1^\star \leftarrow\$ \mathcal{K}$
5: $b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(\mathsf{pk}^\star, \mathsf{ct}^\star, \mathsf{ss}_b^\star)$
6: **return** $[\![b' = b]\!]$

Oracle $\mathcal{O}(\mathsf{ct})$ for IND-CPA

1: **return** $\bot$

Oracle $\mathcal{O}(\mathsf{ct})$ for IND-1CCA and IND-CCA

1: **if** $\mathsf{ct} \neq \mathsf{ct}^\star$ (for IND-1CCA: **and** this is first $\mathcal{O}$ query) **then**
2: **return** KEM.Decapsulate($\mathsf{sk}^\star$, ct)
3: **else**
4: **return** $\bot$
5: **end if**

Figure 2.7: Security experiments for indistinguishability (IND) of KEMs under chosen plaintext (atk = CPA), single chosen ciphertext (atk = 1CCA), and (multiple) chosen ciphertext (atk = CCA) attacks.

security where the adversary can make only a single query to its decapsulation oracle; we denote this IND-1CCA. Although no IND-1CCA KEMs are expected to be standardized in the NIST PQC standardization project, Huguenin-Dumittan and Vaudenay have shown that IND-1CCA KEMs may be constructed that are more efficient than IND-CCA KEMs that require the Fujisaki–Okamoto transform [182].

Definition 2.15 (Indistinguishability security of KEMs)**.** The *indistinguishability* experiments IND-CPA, IND-1CCA, and IND-CCA for KEMs are shown in figure 2.7. The advantage of $\mathcal{A}$ in breaking IND-atk security of KEM, for $atk \in \{\text{CPA}, \text{1CCA}, \text{CCA}\}$, is

$$\mathsf{Adv}^{\text{IND-}atk}_{\text{KEM},\mathcal{A}} = \left| \Pr\left[G^{\text{IND-}atk}_{\text{KEM},\mathcal{A}} \Rightarrow 1 \right] - \frac{1}{2} \right|.$$

2.5 Secure key-exchange protocols

In the above, we have hinted towards key-exchange protocols, especially as we discussed DH key exchange, NIKE, and KEMs. Indeed, all of these primitives immediately define protocols that exchange keys as we have seen in e.g. figures 2.5 and 2.6. A key-exchange protocol is a sequence of operations where two parties at the end agree on a shared secret key.

Note that protocols constructed naively from primitives, as shown above for DH and KEMs, are often not secure against trivial impersonation attacks. To construct secure protocols, we can use the primitives described in this chapter as building blocks to define larger protocols. To overcome the impersonation attacks mentioned, we, for example, could remove the generation of new keys in the first steps of figures 2.5 and 2.6 and use static keys associated with the identities of A and B. We would then add the distribution of the keys to the protocol as an assumption before executing the steps given in the diagrams. Alternatively, we might layer one or more separate protocols on top of a given "naive" key-exchange protocol. For example, TLS uses signatures in the protocol to provide the required authentication properties. To verify these signatures we of course need to provide the parties in the protocol with the public keys in some verifiable way; we give more details on TLS's solution to this problem in the next chapter.

2.5.1 Forward secrecy

A problem of the static-keys solution for secure key exchange described above is the fact that the shared secret will only remain secure as long as the static private keys remain secure. We can assume that any attackers to our key-exchange protocols are patient: they might record key exchanges and any transmitted information that was encrypted with the resulting shared secret keys. This means that the value of the static private key in such a protocol keeps increasing, as it will allow an attacker to recover more and more previously transmitted information. Meanwhile, the secrecy of static keys is constantly being eroded: for example, a new software vulnerability that allows an attacker access to the machine to pull out the keys might be discovered at any time.

Forward secure protocols sever this relationship between previous executions of the protocol and the static private keys [120]. To achieve *forward secrecy*, each execution of the protocol computes brand-new key exchange

keys. The static keys of the peers are only used to authenticate the exchange. After the shared secret has been computed in the protocol, the key-exchange keys are thrown away: they are *ephemeral*.

Note that there exist many (weaker) variants of forward secrecy. Indeed, in section 7.1.7, to describe the nuanced authentication and downgrade security properties of KEMTLS, we will introduce variants where the shared secret keys only remain forward secure if the adversary was not active during the execution of the protocol.

Implementations also may choose to weaken the forward secrecy by using an ephemeral key for a certain number of sessions or a certain amount of time. This may for example be done for performance reasons if key generation is (computationally) expensive.[1] We should note that this behavior may require stronger security properties from the ephemeral key exchange algorithm (for example, it may require IND-CCA rather than IND-CPA security). Additionally, the more sessions such a reused "ephemeral" secret key is used in, the more valuable the key becomes to attackers. We refer to [2, 134, 252, 331, 348] for some further discussion on the caveats of ephemeral (DH) key reuse in TLS.

2.6 Post-quantum cryptography

Though not strictly necessary for understanding the work described in the next chapters, we will give a brief overview of the quantum threats to what we refer to as "classical" cryptography. We hope that this will provide a bit of context and set the background for our work. We will also briefly go into the main post-quantum asymmetric primitives.

2.6.1 Quantum attacks on classic cryptographic

As we touched upon in the introduction, many of the cryptographic schemes that we rely upon today are vulnerable to attacks by quantum computers. This is not because quantum computers are vastly superior in general computing power. A quantum computer does not break those cryptographic systems through brute force, it cannot simply exhaustively attempt decryption with all possible keys in minutes. Rather, quantum computers enable *quantum*

[1] Indeed, server-side ephemeral (finite-field) DH key reuse was the default in OpenSSL (until 2016) and Microsoft SChannel.

algorithms, which employ the characteristics of the quantum mechanisms present in a quantum computer to approach certain computational problems from a completely different angle.

The currently most used asymmetric cryptographic algorithms are based on one of two mathematical difficulties: the discrete-logarithm problem (recovering a given $aG \in \mathbb{G}$, relevant to, e.g., DH and elliptic-curve-based cryptography), or the integer factorization problem (recovering prime numbers p, q given $p \cdot q$, relied on by, most notably, RSA). Both of these problems fall prey to algorithms designed by Shor, who proposed what became known as *Shor's algorithm* for integer factorization and for solving discrete logarithms [324]. Shor's algorithm solves either of these problems in polynomial time, given a large enough quantum computer—this completely wipes out the cryptographic security of these schemes.

For symmetric algorithms, the story is a bit more nuanced. The best known quantum attack on most symmetric schemes, such as AES, is a quantum-computer-accelerated brute-force search. The relevant quantum algorithm here is *Grover's algorithm*, which reduces a search problem that would, on a regular computer, take 2^n steps to solve, to $\sqrt{2^n} = 2^{\frac{n}{2}}$ steps [165]. As a result, we can say that Grover *halves* the bit-security level (definition 2.1) of an algorithm. Grover is optimal for the black-box search problem [39]. This means that any other quantum attacks on symmetric algorithms that aim to do better will have to "open the black box", i.e., use algorithm particulars.

To counter Grover's algorithm, we can simply switch to symmetric primitives that have double the classic bit-security level. Fortunately, many schemes of which we currently use 128-bit (classically) secure instances, also have variants at 256-bit (classically) security levels. Although these variants take slightly more computation time, they are already widely deployed and used, suggesting there are no significant practical objections to this move. Some have however pointed to the large size of the quantum computer that is required to run Grover's algorithm. Following this argument, smaller increases than doubling pre-quantum security levels have been suggested [145].

None of the currently announced quantum computers can run Shor's or Grover's algorithms, which need large quantities of stable, error-corrected quantum bits. There is no way to tell when a quantum computer that can run them will be available, but considering the vast commercial, personal privacy, and state security risks tied to the algorithms threatened by such a quantum computer, it is better to be prepared.

2.6.2 Post-quantum cryptography

Post-quantum cryptography describes cryptographic schemes that hold up to attacks using quantum computers. However, crucially, post-quantum cryptography does not require quantum computers to run: these schemes are intended to protect *today's* computers and information. As described above, symmetric algorithms can be easily protected from Grover's algorithm within currently specified schemes. Consequently, whenever we refer to post-quantum cryptography we generally refer to asymmetric schemes meant to replace RSA, Diffie–Hellman, and elliptic-curve cryptography.

Most of the proposed algorithms can be related to one of a small set of mathematical constructions: lattices, error-correcting codes, isogenies, or multivariate equations. There are also schemes whose security is based on symmetric hash functions. We briefly describe these categories below.

Lattice-based cryptography

In the NIST PQC standardization project, lattice-based schemes represented the majority of submissions. Three of the four algorithms selected for standardization are based on lattices, namely KEM Kyber [319], and signature schemes Dilithium [241] and Falcon [293]. Lattice-based schemes typically have moderately sized keys, ciphertexts, or signatures; their sizes range from hundreds of bytes to a few kilobytes. Many lattice-based schemes are also very computationally efficient: even though the keys, ciphertexts, or signatures are much larger than their RSA and especially elliptic-curve-based equivalents, lattice-based schemes often rival them in performance. As these characteristics are generally favorable, most submissions to the NIST PQC standardization project were lattice-based, and the schemes mentioned above were three of the four first selected for standardization in July 2022.

Code-based cryptography

Already in 1978, Robert McEliece proposed a public-key encryption scheme based on error-correcting codes [248]. Error-correcting codes are used in communication to correct bit-flips and other transmission errors. However, decoding a random linear code is NP-hard, so most transmission applications use specific codes with efficient decoding algorithms. McEliece's cryptosystem however disguises the specific code so that the efficient decoding algorithm is hidden. The Classic McEliece [7] KEM submission in the NIST PQC standardization project is a direct descendant of the original proposal: as such,

it is considered a very conservative choice. Encapsulation and decapsulation are quite fast; however, Classic McEliece's key generation operation is slow. The most important disadvantage of code-based cryptography is the large size of the public keys. A Classic McEliece public key ranges in size from about 260 kB to over 1.3 MB. The HQC [3] and BIKE [17] KEMs are two other KEM submissions in the NIST PQC standardization projects that are based on codes, though they use a different design than Classic McEliece.

Isogeny-based cryptography

Isogeny-based cryptography is a very recent development; the first schemes were proposed in the late nineties and early 2000s [107, 341], and only much more recently have they gained much attention. They rely on mappings, called isogenies, between elliptic curves. The two most notable examples of isogeny-based schemes in recent years are SIKE [197] and CSIDH[2] [91]. SIKE was a round-4 KEM submission in the NIST PQC standardization project before it fell to a devastating new attack in July 2022 [90, 242, 304]. CSIDH is not part of the NIST post-quantum standardization project but deserves a mention as it currently is one of the few known post-quantum NIKEs. The isogeny schemes have small public keys and ciphertexts; however, the computational complexity is much larger than the lattice-based schemes. Also, isogeny-based schemes have only been studied for a few years at the time of writing, so many have concerns new attacks may be discovered.

Multivariate-based cryptography

Multivariate-based cryptography is based on systems of equations. The most prominent example is the unbalanced oil and vinegar (UOV) digital signature scheme. It is also one of the few multivariate-based systems that have held up against cryptanalysis: the security of the LUOV [54], GeMSS [89], MQDSS [310], and Rainbow [122] submissions to the NIST PQC standardization project was all reduced by cryptanalysis. It was announced that UOV will be submitted to NIST's on-ramp for digital signatures [52]. A drawback of UOV is the very large public key sizes; in the smallest instance, the public key is 330 kB. However, the signatures are very small.

Hash-based cryptography

Finally, hash-based cryptographic schemes are based on cryptographic hash functions; The most notable examples are hash-based signature schemes,

[2] Pronounced *sea-side*, as the authors came up with it on a beach.

dating back to Lamport's one-time signature scheme [230]. The XMSS [84, 181], LMS [249], and SPHINCS$^+$ [184] signature schemes construct a Merkle tree [254] of keys, which can each be used in a one- or few-time signature. XMSS and LMS are *stateful*, which means the user needs to keep track of which one-time signature keys have been used; SPHINCS$^+$, which is *stateless*, uses a larger number of few-time signature keys to make sure keys are overwhelmingly unlikely to be used too many times when picked at random. A stateless scheme requires much less careful consideration of the state, which needs to be very carefully kept track of in XMSS and LMS: any rollback allows signature forgery. Such rollbacks may happen in unexpected ways due to for example implementation details of virtual machines hypervisors or filesystems [250]. Although the public keys of these schemes are quite small, signatures are quite large: the smallest SPHINCS$^+$ signature is 7.8 kB. Also, as the schemes rely on constructing large trees of hashes, they require significant amounts of computation. However, as these schemes only rely on the cryptographic security of the hash function, they are very conservative. Parameters for XMSS and LMS have been selected by the Internet Research Task Force (IRTF) in Requests for Comments (RFCs) 8391 and 8554 [181, 249], and standardized by NIST [106], while SPHINCS$^+$ has been selected for standardization in the NIST PQC standardization project.

2.6.3 NIST's security levels

In the NIST PQC standardization project, NIST requested submitters to categorize the instances of their submitted schemes into one of five security levels. These security levels relate to the difficulty of breaking the symmetric schemes AES and SHA-2. How these security levels are to be exactly interpreted has been the subject of much discussion (see, e.g., [318]), but we provide the definitions given by NIST in table 2.1 [270].

NIST asked submitters to focus on categories I–III, leaving levels IV and V for high-security instances. As the project progressed, most submissions settled on providing parameter sets for security levels I, III, and V.

Table 2.1: NIST's categorization of security levels

Level	**Security description**
I	At least as hard to break as AES128 (exhaustive key search)
II	At least as hard to break as SHA256 (collision search)
III	At least as hard to break as AES192 (exhaustive key search)
IV	At least as hard to break as SHA384 (collision search)
V	At least as hard to break as AES256 (exhaustive key search)

Part I

Post-Quantum TLS

3 The TLS protocol

The Transport Layer Security (TLS) protocol is possibly the most-used secure-channel protocol. It provides not only a secure way to transfer web pages [295], but it is also used to secure communications to mail servers [177, 275] or to set up VPN connections [280]. The most recent iteration is TLS 1.3, standardized in August 2018 [298]. TLS 1.3 introduced many enhancements to the TLS protocol, notably a 1-round-trip time (RTT) handshake, encryption of the handshake protocol, the possibility to send data in the first message in resumed handshakes, and a new key-derivation scheme. During the design of TLS 1.3, there was an intense collaboration with the academic community [287]. Many proofs, both handwritten and using computer-assisted methods, of the security of TLS 1.3 have validated the proposed draft protocols and the final standard [55, 108, 109, 127, 128, 129, 213, 221]. This has increased confidence in the protocol.

The main goal of TLS is to transmit application data between two peers: a client and a server. This data is encrypted using a symmetric authenticated-encryption algorithm. The particular algorithm and the keys used for encryption are however not generally agreed upon beforehand. The server and client may also support different versions of TLS, or support different key exchange or authentication algorithms. So when setting up a TLS 1.3 connection, the protocol allows the client and server to negotiate the exact parameters that they are going to use in that particular session. The output of the *handshake protocol* that covers this negotiation is the keys that are then used in the record layer for application data. This means we do not have to make changes to the record layer or handling of application data in the later chapters where we propose alternative handshakes; we only need to ensure that the resulting keys have the right security properties. Thus, we will focus on the handshake protocol in this thesis and not further discuss the record layer.

The main TLS 1.3 handshake, in which a client is connecting to a server, assumes the client has no prior knowledge of the server's preferences or keys. It allows the peers to agree on a shared secret and authenticates the server to

the client, unilaterally, by a certificate. Optionally, the server may require the client to present a client certificate for mutual authentication.

In practice, clients often connect many times to the same server. For this reason, TLS 1.3 also allows connection setup via a pre-shared symmetric key. This symmetric key might be obtained out-of-band, but more commonly it is obtained from a so-called session ticket. Such a ticket can be issued by a server to a client after completing a full TLS 1.3 handshake. With the ticket, the client can *resume* the TLS session when it needs to set up a new connection, avoiding the full handshake. Note that TLS 1.3's pre-shared key (PSK) mode is not fit for low-entropy keys (i.e., passwords) [298, Sec. 2.2]. Prior versions of TLS supported password-based authenticated key exchange [173, 340], but this relied on messages that were removed in TLS 1.3. Some designs for password-based authenticated key exchange have been proposed, but have so far not gained much traction [22]. However, a new proposal for password-based authentication in TLS 1.3, based on the newer OPAQUE scheme [198], had just been put forward at the time of writing [176].

Contributions

This chapter surveys the existing work on TLS and gives context for the remainder of this thesis. We will first discuss the details of the different handshakes of TLS 1.3. Next, we will cover the existing proposals for how the handshake may be protected against post-quantum attackers. In chapter 11, we provide an extensive examination of the performance of post-quantum TLS 1.3. This examination will be contrasted against our proposals for alternative TLS handshakes, KEMTLS (chapter 5) and KEMTLS-PDK (chapter 6), in the benchmark results reported in chapters 13 and 14.

3.1 The TLS 1.3 handshake

We first cover the most common way that TLS connections are set up: a full handshake where only the server is authenticated.

The client and server both have preferences for what kind of kinds of asymmetric and symmetric cryptography they would like to use. For example, mobile devices without AES acceleration might prefer to use ChaCha20 for symmetric encryption. Regulatory requirements, most notably FIPS 140 certification [262], may also affect the options under consideration. The server is additionally set up with a certificate issued by a certificate authority,

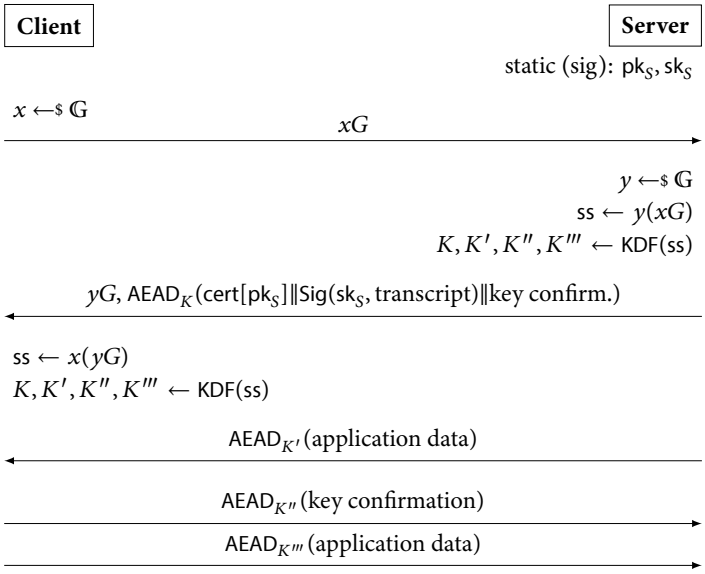


Figure 3.1: High-level overview of the TLS 1.3 handshake.

containing its identity and its long-term authentication public key. Most often, this is an RSA key, though elliptic-curve digital signatures are also supported. In figure 3.1 we give a high-level overview of the server-authenticated TLS 1.3 handshake protocol, focusing on the signed-Diffie–Hellman (DH) aspect of the handshake. Figure 3.2 shows a detailed overview of the messages in the TLS 1.3 handshake, including all the cryptographic computations.

The first message in a TLS handshake is sent by the client. In TLS 1.3, the client picks its favorite DH key-exchange algorithm and generates a public key xG. It submits this to the server along with its other preferences in the `ClientHello` message. Note that the client may not know if the server even supports the chosen key-exchange algorithm. So the server may send a `HelloRetryRequest` message to the client in response to the first choice, telling it to pick a new public key using an algorithm from the list of algorithms supported by the server and send a new `ClientHello`. If the server can use the xG sent by the client, it generates its own public key yG. It submits this public key along with the list of algorithms it supports back to the client in the `ServerHello` message.

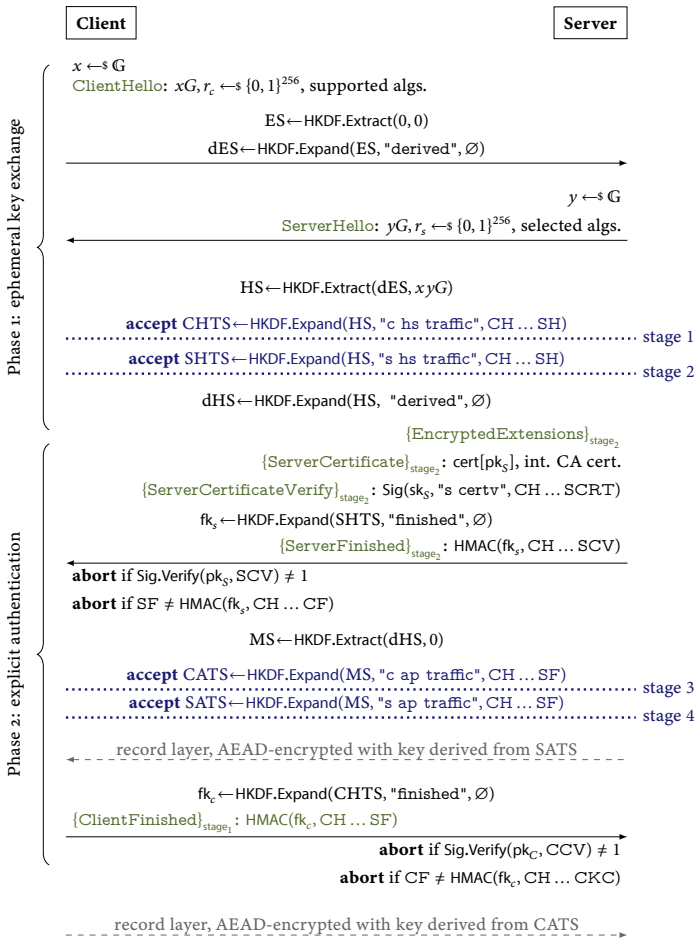


Figure 3.2: Detailed overview of the TLS 1.3 handshake with server-only authentication. Note that we have abbreviated labels in key computations.

Meanwhile, the server can immediately continue with the remainder of the handshake. The server computes xyG and feeds it into the key schedule. It derives the Handshake Secret (HS) and derives the handshake traffic keys necessary to encrypt the remainder of the handshake messages. In the next message from the server, `EncryptedExtensions`, it may advertise any TLS extensions that were not necessary for the cryptographic computations. Next, the server sends its `ServerCertificate` message. This is followed by the `ServerCertificateVerify` message which contains a signature over all the messages sent so far. The last handshake message that the server sends is the `ServerFinished` message. This last message contains a MAC over the complete transcript. Now, the server has completed its part of the handshake.

The client, after also computing xyG using the yG received in the `ServerHello` message, computes HS and the handshake traffic keys. It then receives the messages `ServerCertificate`, `ServerCertificateVerify`, and `ServerFinished` from the server, and verifies them. If verification fails, the connection is aborted. Otherwise, the client sends `ClientFinished`, which contains another MAC over the transcript.

Both the client and the server then proceed with computing the Main Secret (MS).[1] This key is computed immediately from HS. From MS the application traffic keys are computed, and the handshake is complete.

Note that the server already has all the information necessary to compute MS before receiving the `ClientFinished` message. The server may choose to already send data to the client before `ClientFinished` is received. If it does, the server has to accept that it has not yet authenticated the client or the preferences indicated in the `ClientHello` message.

Optionally, it is possible to also authenticate the client in this handshake. We show how this modifies the handshake in figure 3.3. The server sends a special `CertificateRequest` message after sending `EncryptedExtensions` to request the client's certificate. If the client receives this message, it needs to send back `ClientCertificate` and `ClientCertificateVerify` messages to the server.[2] It does this after it received the `ServerFinished` message, before sending `ClientFinished` with a MAC over the transcript.

There are additional handshake modes, such as the PSK based handshakes.

[1] This key was previously called Master Secret but will be renamed to Main Secret as RFC 8446 is updated by RFC 8446bis [299]. We will use the new name.

[2] If the client has no appropriate certificate, it sends back an empty `ClientCertificate` message and omits `ClientCertificateVerify`.

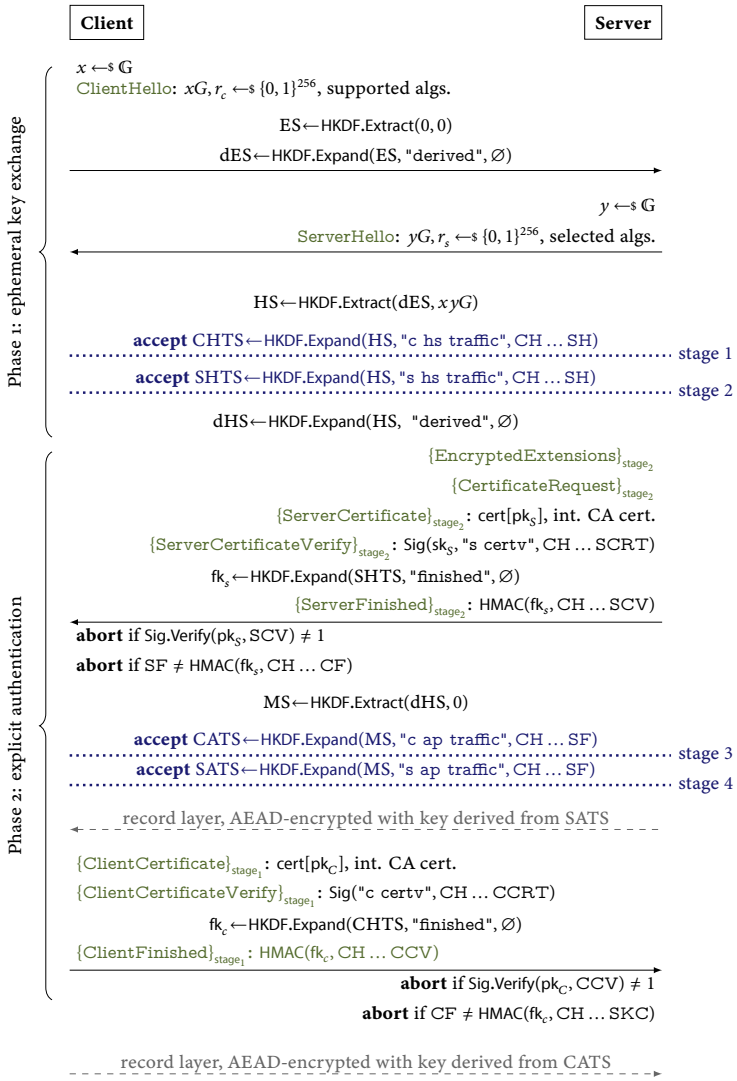


Figure 3.3: Detailed overview of the TLS 1.3 handshake with mutual authentication.

A PSK may for example be obtained from session tickets. In PSK-based handshakes, no certificates are even exchanged and the ephemeral DH key exchange is optional. The server and the client use it to derive the early handshake secret (ES). (In handshake where PSKs are not used, ES is derived from strings of zero bytes.) The client may use this secret key to send 0-RTT data along the `ClientHello` message. As we will not discuss PSK mode in the remainder of this thesis, we will not elaborate further than this informal description. We refer to the standard, and security analyses such as the work by Dowling, Fischlin, Günther, and Stebila [127], for more details.

3.2 The TLS 1.3 key schedule

The TLS 1.3 handshake can be described as a layering of two protocols. The initial messages perform an ephemeral key exchange. The resulting key is used to set up a secure channel for the remainder of the handshake. This channel is encrypted using (keys derived from) the stage-1 and stage-2 keys in figures 3.2 and 3.3. Inside this channel, the authentication protocol is executed, namely exchanging the certificates and signatures. Finally, the application traffic keys are derived from the stage-3 and stage-4 keys.

To compute all of these different keys, TLS 1.3 has a carefully designed key schedule. It starts by HKDF.Extract-ing Early Secret (ES) from a pre-shared key, or a string of zero bytes if there is none (we denote this by '0'). From ES we derive secrets using HKDF.Expand. The derived secrets are context-separated by different strings for each secret, for example `"tls13 c e traffic"` for the Early Traffic Secret.[3] They are also bound to the handshake messages by including an incremental context hash of the messages sent so far. The transcript hash of messages $M_1 \dots M_n$ is computed by $\mathsf{H}(M_1 || \cdots || M_n)$. TLS assumes this can be computed by absorbing the message into the hash state one message at a time, such that the messages do not need to be kept in memory for the entire duration of the handshake.

We thus derive the early traffic secret, which is used for optional 0-RTT data in handshakes using pre-shared keys. We also derive dES, the "derived" early traffic secret, which is used in the next key computation as context. The next secret key, the Handshake Secret (HS), is computed by HKDF.Extract from

[3] For presentation reasons, we may abbreviate context labels in our handshake diagrams.

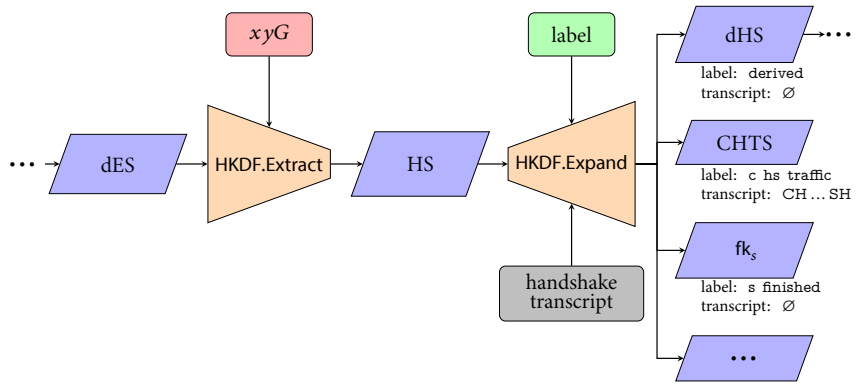


Figure 3.4: Part of the TLS 1.3 key schedule computations.

the DH key exchange and the dES key. Again, the handshake traffic secrets, CHTS and SHTS, and the derived secret dHS are computed from HS using HKDF.Expand and appropriate labels and hashed message transcript. The keys to compute the HMAC in the `Finished` messages are also derived from HS.

Finally, the Main Secret (MS) and application traffic secrets CATS and SATS are computed from dHS using HKDF.Extract. As there is no new secret key, a string of 0 bytes is used as input key. Some other secrets are computed from MS; for example secrets for the resumption and exporter extensions. We leave these secrets out for simplicity.

Much work has been done on the security of TLS 1.3 and its components. For formal analyses of the TLS 1.3 protocol and its key schedule, one can for example refer to [55, 81, 108, 113, 118, 127, 182], as well as many other works cited throughout this thesis.

3.3 Preparations for post-quantum TLS

We can straightforwardly replace each of the pre-quantum public-key cryptographic operations in the TLS 1.3 handshake with post-quantum primitives, at least in theory. The RSA or elliptic-curve signatures on the handshake and in the certificate ecosystem can be straightforwardly replaced by post-quantum signatures, as they provide the same functionality. DH key exchange does not exactly match the interface that post-quantum KEMs provide, but in TLS we can simply replace the server's keygen by the encapsulation operation

and send the ciphertext instead of the server's public key. A sketch of how post-quantum primitives fit in TLS 1.3 is shown in figure 3.5. This was already demonstrated for TLS 1.2 in the 2015 paper by Bos, Costello, Naehrig, and Stebila, including a proof of the security of replacing DH by KEM key exchange [73]. This paper also suggested the use of so-called *hybrid* primitives. These combine the post-quantum algorithm with a traditional, pre-quantum algorithm like ECDH. As breaking a hybrid scheme requires breaking its components, this allows the confidence in the implementation and the security of the pre-quantum scheme against attacks *without* quantum computers to carry over to the hybrid scheme.

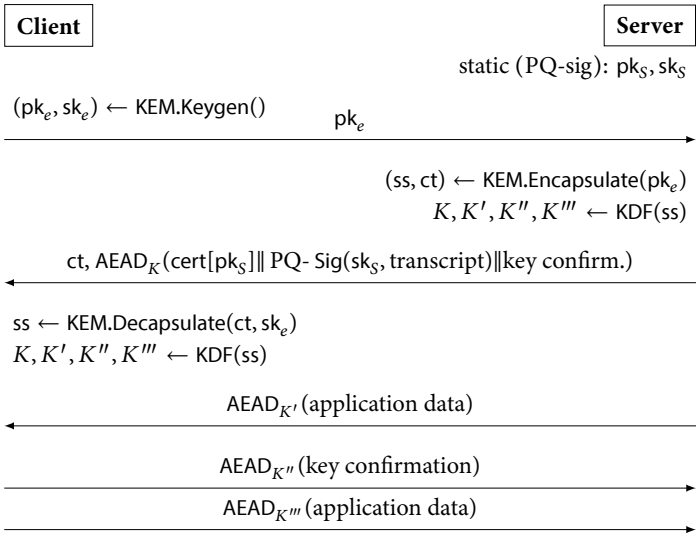


Figure 3.5: A high-level overview of TLS 1.3 with post-quantum primitives. The DH ephemeral key exchange is replaced by KEM operations, and the signature uses a post-quantum signature scheme.

Public experiments by industry parties started in 2016 with the CECPQ1 ("combined elliptic-curve and post-quantum 1") experiment in the Google Chrome browser [232]. It combined X25519 ECDH [41] with NewHope lattice-based KEM key exchange [8] in the TLS 1.2 handshake. A follow-up CECPQ2 experiment by Google together with Cloudflare using TLS 1.3 was announced in late 2018 [226, 233], using a combination of X25519 and the lattice-based

scheme NTRU-HRSS [186, 314],[4] and X25519 together with the isogeny-based scheme SIKE [197]. The first results from this experiment are presented in [227]. In late 2019, Amazon announced support for two elliptic-curve with post-quantum hybrid modes was added in AWS KMS [179]; one also using SIKE, the other one using the code-based scheme BIKE [17].[5]

On the academic side, the Open Quantum Safe (OQS) initiative [333] provides prototype integrations of post-quantum and hybrid key exchange in TLS 1.2 and TLS 1.3 to the OpenSSL library [279]. First results in terms of feasibility of migration and performance using OQS were presented in [110]; more detailed benchmarks are presented in [281]. Meanwhile, draft specifications for hybrid key exchange have started the discussion on standardization of post-quantum cryptography for TLS [87, 180, 210, 315, 316, 332, 352].

Previous works have mainly focused on post-quantum confidentiality; there have been fewer experiments deploying post-quantum authentication. Sikeridis, Kampanakis, and Devetsikiotis [326] have measured the performance of post-quantum signature schemes between servers in two data centers. They concluded that out of the (round-2) schemes they tested, only two (Falcon [293] and Dilithium [241]) seem viable for deployment in TLS 1.3. Experiments by Cloudflare [351] that added dummy data to TLS connections to measure the impact of the larger sizes of post-quantum signature schemes, seem to support these results. Still, when using Dilithium as a drop-in replacement for all signatures in TLS (which adds 17 kB to the handshake), Cloudflare reports an expected 60–80 % slowdown for the Linux default congestion window of 10 maximum segment size (MSS). Falcon has more favorable public key and signature sizes but requires hardware support for 64-bit floating-point operations. Without this, Sikeridis, Kampanakis, and Devetsikiotis report that signing handshakes with Falcon is not viable [326].

We will restrict our focus to public-key authenticated TLS, rather than pre-shared key (which uses symmetric algorithms for authentication and can readily have its ephemeral key exchange replaced with a post-quantum KEM) or password-authenticated TLS (for which there has been some exploration of post-quantum algorithms [155]).

While post-quantum algorithms generally have larger public keys, cipher-

[4] NTRU-HRSS has since merged into the NTRU [98] submission in the NIST PQC standardization project, from which it was later eliminated.

[5] BIKE is a round-4 submission in the NIST standardization project. SIKE was broken shortly after round 4 started [90, 242, 304].

texts, and signatures compared to pre-quantum elliptic-curve schemes, the gap is bigger for post-quantum signatures than post-quantum KEMs. We will return to this in the following chapters. In part III, we report on the performance of post-quantum key exchange and authentication in TLS 1.3.

4 Post-quantum OPTLS

When the IETF TLS working group started the work on TLS 1.3, they set out to build the new handshake on a more solid foundation. TLS 1.2 was built on the same foundations as its wounded predecessors and contains a patchwork of implementation caveats and countermeasures. This was a result of the development process of these earlier versions, which Paterson and Van der Merwe [287] describe as a "design-release-break-patch" cycle: a new handshake would be put on paper and deployed in applications, after which security researchers would find vulnerabilities that would then be patched. The development of TLS 1.3 involved the academic community in a much more significant way and resulted in a significant cleanup of the standard.

The OPTLS proposal by Krawczyk and Wee [221], which we summarize in this chapter, was a significant step in this direction. We provide a sketch of a variant of OPTLS in figure 4.1. It provides, among other things, a signature-free TLS handshake with authentication via long-term DH keys. The server sends a certificate with a DH public key. It then constructs a shared secret by combining its corresponding long-term secret key with the ephemeral public key from the client. This key is then used to generate a MAC, which authenticates the server. OPTLS was the foundation of early drafts of TLS 1.3 but was later dropped in favor of the familiar handshake authentication via signature. One reason for this was that otherwise different certificates would be necessary for different versions of TLS. Other elements of OPTLS, such as the new key schedule we discussed in section 3.2, still live on.

Contributions

This chapter surveys the OPTLS proposal and other proposals for authenticated key exchange without handshake signatures. It provides context for and leads into the development of KEMTLS in chapter 5. Our main contribution to the discussion around OPTLS is found in chapter 12, where we examine the performance of OPTLS in the post-quantum setting.

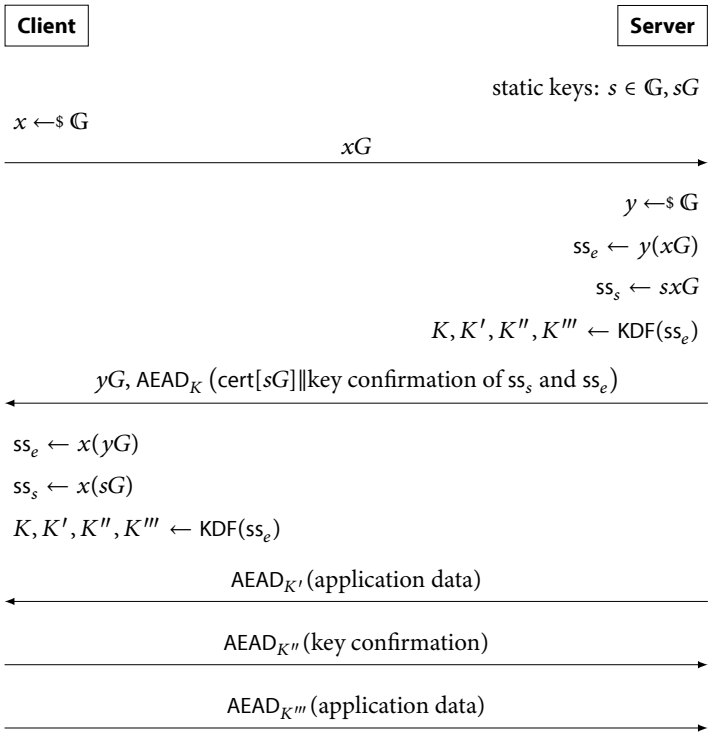


Figure 4.1: Sketch of the OPTLS handshake protocol as proposed as an TLS 1.3 extension in `draft-ietf-tls-semistatic-dh` [302].

4.1 Revisiting OPTLS

OPTLS was at the heart of early designs for TLS 1.3 but was dropped in favor of signed-Diffie–Hellman for the final standard. Starting in 2018, there has been an attempt to revive OPTLS in TLS 1.3 [302], but so far we do not see that these drafts have gained much traction. (The only prior implementation of OPTLS that we are aware of is described in the Master's thesis by Kuhnen [224].)

If a signature-free approach for the TLS handshake has not been very successful in the past, why revisit it now? We see two reasons why OPTLS has not gained much traction; both change with the eventual move to post-quantum cryptography in TLS.

To tap the full potential of OPTLS, servers need to obtain certificates containing DH public keys instead of signature keys. This also means that they need to use different certificates to support different TLS versions, as prior versions of TLS would still use authentication based on signatures. While in theory, this is not a problem it requires certificate authorities to adapt their software and needs other changes to the public-key infrastructure, which would have been an obstacle to TLS 1.3's goals of widespread deployment and fast adoption. However, the move to post-quantum authentication will require rolling out a new generation of certificates regardless of whether signatures or KEMs are used for authentication.

Moreover, when using pre-quantum primitives based on elliptic curves, the advantages of OPTLS compared to the traditional TLS 1.3 handshake are limited. The performance differences between elliptic-curve Diffie–Hellman operations and elliptic-curve signing and verification algorithms are not very large, and the sizes of signatures and signature public keys are small. A TLS implementation with secure and optimized elliptic-curve arithmetic implemented for ECDH already has most of the critical code needed to implement elliptic-curve signatures. This also means that, in protected applications, not much extra effort has to be put into hardening measures against, e.g., side-channel and fault attacks to support both operations.

For current post-quantum key-exchange algorithms and signature schemes, this picture changes. It is possible to choose key-exchange algorithms that offer considerably smaller sizes and much better speed than any of the signature schemes. Also, post-quantum signatures and key-exchange algorithms no longer share large parts of the code base; even though lattice assumptions can be used to construct both key-exchange and signature algorithms, such

schemes need different parameters and thus different optimized routines. Thus, in the post-quantum setting, the signature-free approach to the TLS handshake reduces the size of the trusted code base and potentially the amount of traffic sent.

4.2 Authenticated key exchange without signatures

There is a long history of protocols for authenticated key exchange (AKE) without signatures. *Key transport* uses public-key encryption: authentication is demonstrated by successfully decrypting a challenge value. Examples of key transport include the SKEME protocol by Krawczyk [217] and RSA key-transport cipher suites in all versions of SSL and TLS up to TLS version 1.2 (but RSA key transport did not provide forward secrecy). Bellare, Canetti, and Krawczyk [31] gave a protocol that obtained authentication from DH key exchange: DH keys are used as *long-term* credentials for authentication, and the resulting shared secret is mixed into the session key calculation to derive a key that is *implicitly authenticated*, meaning that no one but the intended parties could compute it. Some of these protocols go on to obtain *explicit* authentication via some form of key confirmation. Many DH-based AKE protocols have been developed in the literature. Other classic examples than the ones mentioned before are the Menezes–Qu–Vanstone (MQV) protocol [236], the HMQV variant by Krawczyk [215], NAXOS [229], and more recently the Noise protocol framework [291]. Protocols using DH-based authentication are used in products such as Signal [290] and WireGuard [126].

There are a few constructions that use generic KEMs for AKE, rather than static DH [76, 149]. A slightly modified version of the [149] KEM AKE has recently been used to upgrade the WireGuard handshake to post-quantum security [185]. One might think that the same approach can be used for KEM-based TLS, but there are two major differences between the WireGuard handshake and a TLS handshake. First, the WireGuard handshake is mutually authenticated, while the TLS handshake typically features server-only authentication. Second, and more importantly, the WireGuard handshake assumes that long-term keys are known to the communicating parties in advance, while the distribution of the server's long-term certified key is part of the handshake in TLS, leading to different constraints on the order of messages and the number of round trips.

4.3 Post-quantum non-interactive key exchange

As pointed out by Kuhnen, OPTLS' combining of the ephemeral public key from the client with the server's long-term key makes use of DH as a NIKE [224]. The inherently interactive character of a KEM creates issues for protocol designers relying on the properties of NIKE. When used with long-term keys (and a suitable PKI), NIKE allows a user Alice to send an authenticated ciphertext to an *offline* user Bob. Signal's X3DH handshake [245] is a notable example using this feature of NIKEs. Indeed, [79] shows that a naive replacement of the DH operations by KEMs does not work. A straightforward adaptation of OPTLS to a post-quantum setting would also require a post-quantum NIKE.

No post-quantum NIKEs are part of the NIST PQC standardization project. For a long time, CSIDH was the only known, somewhat efficient construction for a post-quantum NIKE. In early 2023, a new proposal was put forward, called Swoosh [153]. There do not yet exist implementations of its passively-secure variant, however, and it has very large public keys: the actively secure parameter set for Swoosh has 120 kB public keys. In chapter 12, we will measure the performance of OPTLS when instantiated with CSIDH, and further discuss post-quantum non-interactive key exchange.

4.4 The OPTLS handshake protocol

In figure 4.2, we give a detailed overview of the OPTLS handshake protocol and its key schedule for server-only authentication. We will refer to the version of the protocol as it was proposed by `draft-ietf-tls-semistatic-dh` as an extension to the TLS 1.3 handshake protocol [302]. This extension has some minor differences from the protocol as proposed by Krawczyk and Wee, to minimize the number of changes to the handshake protocol. Notably, it transmits the authentication message through the `ServerCertificateVerify` message and does not include the shared secret that was computed from the server's static key into the key-derivation scheme of the main handshake.

The TLS 1.3 client that wishes to authenticate through the OPTLS mechanism indicates support for OPTLS certificates in its `ClientHello` message. It otherwise does not change anything from the TLS 1.3 `ClientHello` message and the server replies to the ephemeral key exchange in the `ServerHello`

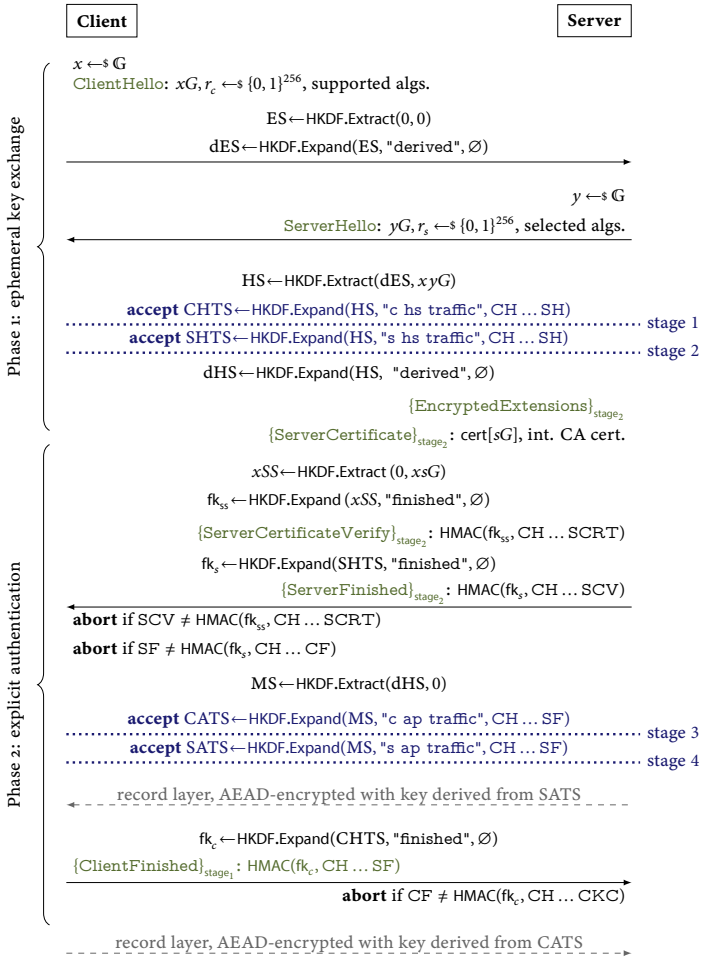


Figure 4.2: Detailed overview of the OPTLS handshake with server-only authentication, as proposed as an TLS 1.3 extension in `draft-ietf-tls-semistatic-dh` [302].

message as it would in TLS 1.3. The OPTLS server has a certificate that contains a long-term DH public key sG. It transmits this key to the client in the `ServerCertificate` message. This static key is combined with the client's ephemeral key share. The client and the server compute the MAC shared secret fk_{ss} from the resulting static-ephemeral key through an HKDF.Extract and HKDF.Expand sequence. As part of the effort to maximize code reuse, the extension proposal reuses the computation for the `Finished` messages to compute the authentication code. The server then transmits the resulting MAC in the `ServerCertificateVerify` message. This completes the authentication of the server to the client; as mentioned above, the authentication shared secrets are not included in the rest of the TLS 1.3 key schedule (though the public keys and MAC are still included as part of the transcript).

4.5 Conclusions

In the original proposal, the security properties of OPTLS were shown by a brief pen-and-paper multi-stage proof. The parts of the proposal that survived into TLS 1.3 have been considered by those who studied the security of TLS 1.3, including [55, 81, 108, 109, 127, 128, 129, 213]. In this thesis, we restrict ourselves to the question: is OPTLS, instantiated with post-quantum NIKE, a suitable candidate for securing the post-quantum internet?

Additionally, we briefly investigate what it means for the performance and practicality of CSIDH if we choose parameters to resist quantum attacks in the cost models used by [71, 97, 289]. Using highly-optimized implementations for CSIDH at higher security levels, we can show the (im)practicality of the scheme in a protocol designed for NIKE, like OPTLS. In turn, if using CSIDH results in poor performance, OPTLS does not seem a viable candidate for post-quantum TLS. We will discuss how we implemented OPTLS with CSIDH in chapter 10. The results of the experiment will be shown in chapter 12.

5 Post-quantum KEMTLS

Although the TLS 1.3 handshake can be made post-quantum straightforwardly, as we discussed in section 3.3, this is not without challenges. The post-quantum KEMs and signature schemes are very different from pre-quantum RSA and DH primitives; especially in that the key exchange and signing operations are not similar in runtime performance or sizes of public keys, ciphertexts, and signatures. Generally, the post-quantum signature schemes considered in the NIST PQC standardization project are larger, slower, and/or harder to implement securely than post-quantum KEMs. The transition to post-quantum cryptography thus gives us new trade-offs, but it also allows us to examine the status quo.

Contributions

In this chapter, we examine an alternative post-quantum TLS handshake. We build on the OPTLS proposal and present a signature-less handshake protocol. Our goal is to achieve a TLS handshake that provides full post-quantum security—including confidentiality and authentication—optimizing for the number of round trips, communication bandwidth, and computational costs. Our main technique is to rely on KEMs for authentication, rather than signatures. We call our proposal KEMTLS. In chapter 7, we examine the security of KEMTLS, and in chapter 13 we show how it is more efficient than TLS 1.3 in most scenarios, in bandwidth, computation, and handshake time.

5.1 Authenticated key exchange from KEMs

Authenticated key exchange using KEMs for authentication is not new, with several examples of mutually authenticated [72, 76, 116, 149] and unilaterally authenticated [72] protocols. The typical pattern among these, restricted to the case of unilaterally authenticated key exchange, is as follows (c.f. [72, Fig. 2]). The server has a static KEM public key, which the client is assumed to (somehow) have a copy of in advance. In the first flight of the protocol, the

client sends a ciphertext encapsulated to this static key, along with the client's own ephemeral KEM public key; the server responds with an encapsulation against the client's ephemeral KEM public key. The session key is the hash of the ephemeral-static and ephemeral-ephemeral shared secrets.

This is a problem for TLS: typically, a client does *not* know the server's static key in advance, but learns it when it is transmitted (inside a certificate) during the TLS handshake. One obvious solution to address this issue is for the client to first request the key from the server and then proceed through the typical protocol flow. However, this increases the number of round trips and thus comes at a steep performance cost.

The other trivial approach to avoid additional round trips is to simply assume a change in the internet's key distribution and caching architecture that distributes the servers' static key to the client before the handshake. For example, in embedded applications of TLS, a client may only ever communicate with very few different servers that are known in advance; in that case, the client can just deploy with the server's static keys pre-installed. Another option would be to distribute certificates through DNS as described in [200]. Neither is a satisfactory general solution, as the former limits the number of servers a client can contact (since certificates must be preinstalled), and the latter requires changes to the DNS infrastructure and precludes connections to servers identified solely by IP address.

5.2 KEMTLS

KEMTLS uses key encapsulation mechanisms as primary asymmetric building blocks, for both forward-secure ephemeral key exchange and authentication. (We unavoidably still rely on signatures by certificate authorities to authenticate long-term KEM keys.) We focus on the most common use case for web browsing, namely key agreement with server-only authentication, but our techniques can be extended to client authentication as shown in section 5.5. Note that the scenario we are considering is orthogonal to resumption mechanisms such as 0-RTT introduced by TLS 1.3.

With KEMTLS, we can retain the same number of round trips until the client can start sending encrypted application data as in TLS 1.3 while reducing the communication bandwidth. We discuss the performance characteristics of KEMTLS in chapter 13. Compared to TLS 1.3, application data transmitted

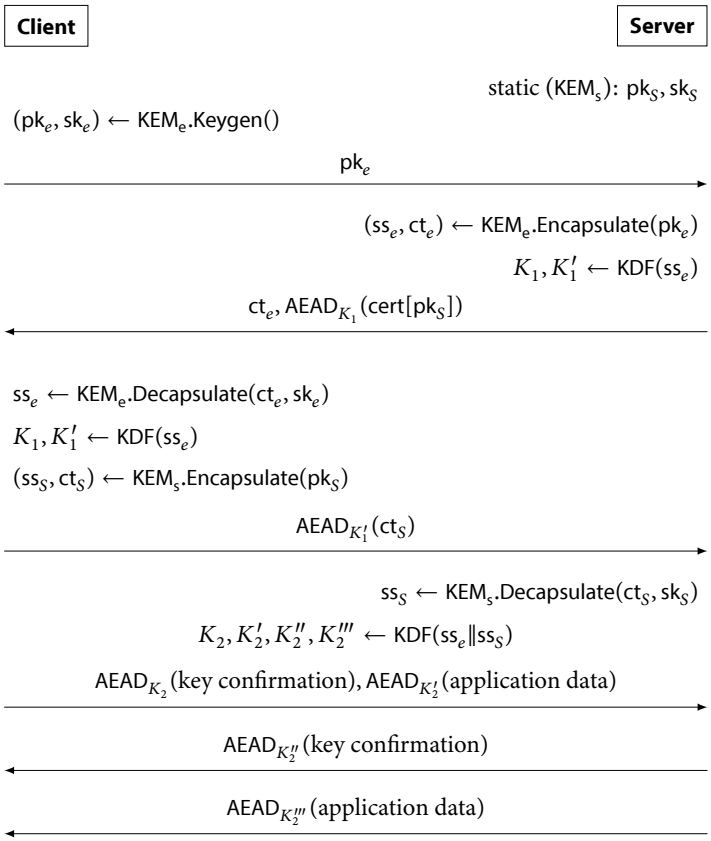


Figure 5.1: A high-level overview of KEMTLS. Instead of using signatures, the server authenticates via a KEM key exchange.

during the handshake is implicitly, rather than explicitly authenticated, and has slightly weaker downgrade resilience and forward secrecy than when signatures are used; but full downgrade resilience and forward secrecy is achieved once the KEMTLS handshake completes; see chapter 7 for details.

5.3 The KEMTLS protocol

KEMTLS achieves unilaterally authenticated key exchange using solely KEMs for both key establishment and authentication, without requiring extra round trips and without requiring caching or external pre-distribution of server public keys: the client can send its first encrypted application data after just as many handshake round trips as in TLS 1.3.

KEMTLS is to a large extent modelled after TLS 1.3. A high-level overview of the handshake is shown in figure 5.1, and a detailed protocol flow is given in figure 5.2. Note that figure 5.2 omits various aspects of the TLS 1.3 protocol that are not relevant to our presentation and cryptographic analysis but which would still be essential if KEMTLS was used in practice. KEMTLS is phrased in terms of two KEMs: KEM_e for ephemeral key exchange, and KEM_s for implicit authentication; one could instantiate KEMTLS using the same algorithm for both KEM_e and KEM_s (as we do in all our instantiations), or different algorithms for different efficiency trade-offs, such as an algorithm with slow key generation but fast encapsulation for the long-term KEM. Either or both could also be a "hybrid" KEM combining post-quantum and traditional assumptions [60].

There are conceptually three phases to KEMTLS, each of which establishes one or more "stage" keys.

Phase 1: Ephemeral key exchange using KEMs

After establishing the TCP connection,[1] the KEMTLS handshake begins with the client sending one or more ephemeral KEM public keys pk_e in its ClientHello message, as well as the list of public-key authentication, key exchange, and authenticated-encryption methods it supports. The server responds

[1] Our exposition and experiments deal with KEMTLS running over TCP, analogously to TLS 1.3. As with TLS 1.3, the overhead from the TCP handshake may be reduced by a variety of techniques as discussed in [100], e.g., TCP Fast Open [101] or QUIC [193].

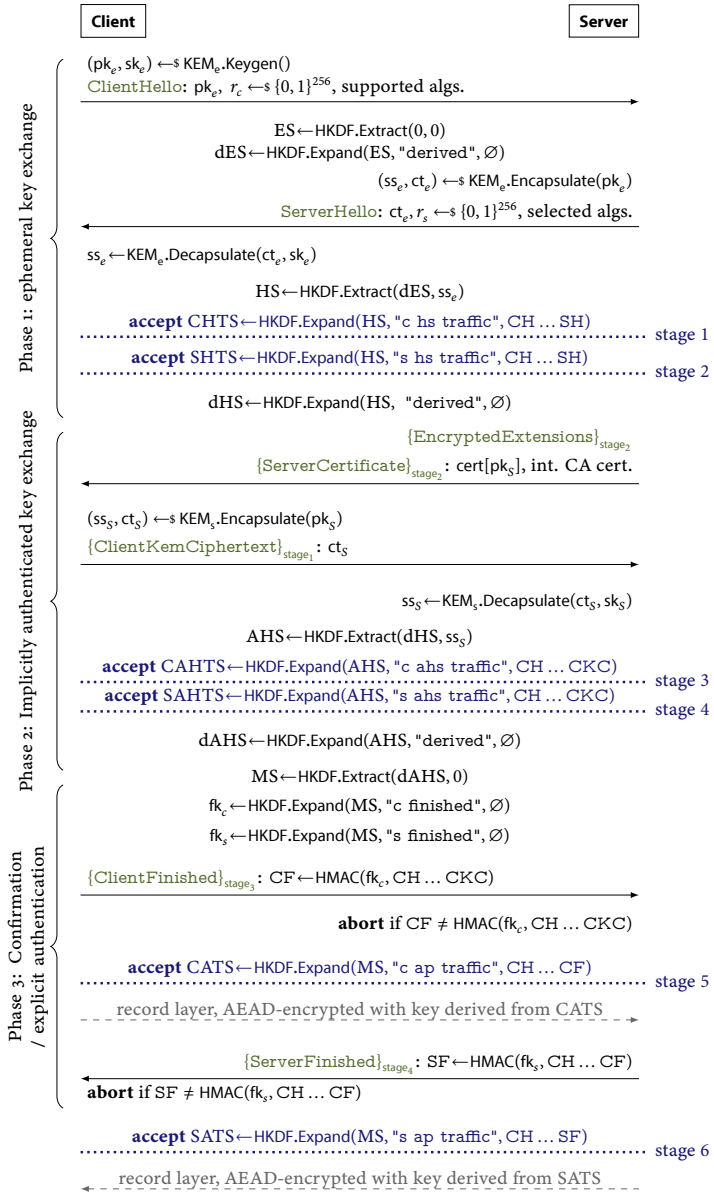


Figure 5.2: The unilaterally authenticated KEMTLS handshake

in the ServerHello message with an encapsulation ct_e against pk_e and the algorithms it selected from the client's proposal; note that if the server does not want to use (any of) the pk_e the client sent, a special HelloRetryRequest message is sent, prompting a new ClientHello message. Nonces r_c and r_s are also transmitted for freshness. At this point, the client and server have an unauthenticated shared secret ss_e. KEMTLS follows the TLS 1.3 key schedule, which applies a sequence of HKDF operations to the shared secret ss_e and the transcript to derive (a) the client and server handshake traffic secrets CHTS and SHTS which are used to encrypt subsequent flows in the handshake, and (b) a "derived handshake secret" (dHS) which is kept as the current secret state of the key schedule.

The key schedule in figure 5.2 starts with a seemingly unnecessary calculation of ES and dES. These values play a role in TLS 1.3 handshakes using pre-shared keys; we retain them to keep the state machine of KEMTLS aligned with TLS 1.3 as much as possible. They are also used in the key schedule of KEMTLS-PDK, which we will discuss in chapter 6.

Phase 2: Implicitly authenticated key exchange using KEMs

In the same server-to-client flight as ServerHello, the server also sends a certificate containing its long-term KEM public key pk_S. The client encapsulates against pk_S and sends the resulting ciphertext in its ClientKemCiphertext message. This yields an implicitly authenticated shared secret ss_S. The key schedule's secret state dHS from phase 1 is combined with ss_S using HKDF to give an "authenticated handshake secret" (AHS) from which are derived (c) the client and server authenticated handshake traffic secrets CAHTS and SAHTS which are used to encrypt subsequent flows in the handshake,[2] and (d) an updated secret state dAHS of the key schedule. The Main Secret (MS) can now be derived from the key schedule's secret state dAHS. From MS, several more keys are derived: (e) "finished keys" fk_c and fk_s which will be used to authenticate the handshake and (f) client and server application transport secrets CATS and SATS from which are derived application encryption keys.[3]

[2] CAHTS and SAHTS are implicitly authenticated: subsequent handshake traffic can only be read by the intended peer server. This is particularly useful in the client-authenticated version of KEMTLS in section 5.5 when the client sends its certificate.

[3] TLS 1.3 also derives exporter and resumption main secrets EMS and RMS from the main secret MS. We have omitted these from our presentation of KEMTLS in figure 5.2,

The client now sends a confirmation message ClientFinished to the server which uses a message-authentication code with key fk_c to authenticate the handshake transcript. In the same flight of messages, the client is also able to start sending application data encrypted under keys derived from CATS; this is implicitly authenticated.

Phase 3: Confirmation / explicit authentication

The server responds with its confirmation in the ServerFinished message, authenticating the handshake transcript using MAC key fk_s. In the same flight, the server sends application data encrypted under keys derived from SATS. Once the client receives and verifies ServerFinished, the server is explicitly authenticated to the client.

5.4 Comparison with TLS 1.3

There are a few subtle differences in the properties offered by KEMTLS compared to TLS 1.3. We will touch on some of them in this section.

5.4.1 Who first sends application data

In TLS 1.3, the *server* can immediately send the first application data after receiving the initial client messages, i.e., in parallel with its first handshake message to the client and before having received an application-level request from the client. This feature is used, for example, in SMTPS to send a server banner to the client. In KEMTLS, it is the *client* that is ready to send application data first. This does incur a small overhead in protocols that require a client to receive, for example, a server banner.

To our knowledge, most applications do not have much, if any application data to transmit to the client before they have received the client's request. This includes HTTPS, probably the most prominent application of TLS. However, recent variants of HTTPS, starting with HTTP/2 [34], as well as variant protocols such as QUIC [193], use the server's first message to advertise connection parameters. Switching to KEMTLS might thus affect the performance

but extending KEMTLS's key schedule to include these keys is straightforward, and security of EMS and RMS follows analogously.

of those protocols. However, as the information contained in such advertisements of connection parameters is most likely public, we can consider transmitting such information during the TLS handshake in an extension. There is precedent for doing so: the Application-Layer Protocol Negotiation (ALPN) extension allows clients to indicate their preferences for the protocol used after completing the TLS handshake [148]. There has also been some discussion on a server-side variant of ALPN, in the context of HTTP/2 and QUIC settings frames, on the TLS working group mailing list [350].

5.4.2 Implicit authentication

KEMTLS provides *implicit* server-to-client authentication at the time the client sends its first application data; explicit server-to-client authentication comes one round trip later when a key confirmation message is received in the server's response. We still retain confidentiality: no one other than the intended server will be able to read data sent by the client. One consequence is that the choice of algorithms used is not authenticated by the time the client sends its first application data. The client cannot be tricked into using algorithms that it does not trust and thus did not advertise, but an adversary might be able to trick the client into using one that the server would have rejected. By the time the handshake fully completes, however, the client is assured that the algorithms used are indeed the ones both parties preferred. We discuss the subtleties of the forward secrecy and downgrade resilience properties of KEMTLS at different stages further in chapter 7.

5.4.3 Presence of the server

Per the previous section, any data the client sends before receiving the server's authentication message is securely encrypted. However, in principle, an attacker could delay or block that authentication message for as long as they desire. A client might submit as much information as they have in that time. The attacker will not be able to generate the implicitly authenticated handshake keys and any key derived from those, so the confidentiality of that data is guaranteed. However, a client should not assume that the handshake was completed, or that the data was received by an honest server until they have received explicit confirmation.

Imagine, for example, a client submitting logging information to a server.

The client submits logging data before the full handshake has been completed. However, the client should be prepared to re-submit the data in case the handshake fails. Otherwise, an active attacker could suppress logs.

The same behavior is also known from TLS 1.3 0-RTT resumption. Also in that protocol mode, the client submits data to a server that has not yet been confirmed to be present. It also closely relates to truncation attacks, where an attacker simply stops the transmission of messages to the server. For this reason, the "connection closed" alerts in TLS must be checked by the server and client, c.f. [298, Sec. 6.1]. This remains still the case in KEMTLS sessions.

5.4.4 Anonymity

Neither TLS 1.3 nor KEMTLS offer server anonymity against passive adversaries, due to the `ServerNameIndicator` extension in the `ClientHello` message. The TLS working group is investigating techniques such as Encrypted `ClientHello` [301] which rely on out-of-band distribution[4] of server keying material to hide this information. As both certificate messages are sent encrypted with ephemeral keys in both TLS 1.3 and KEMTLS, the identity of the server is otherwise protected against passive adversaries. As the identity of the client contained in `ClientCertificate` is encrypted using (explicitly in TLS 1.3, implicitly in KEMTLS) authenticated keys, the identity of the client is additionally protected against active adversaries. This meets the requirements for the protection of the endpoint identities as specified in the TLS 1.3 standard [298, Sec. E.1].

5.4.5 Deniability

Krawczyk pointed out [217, Sec. 2.3.2] that using signatures for explicit authentication in key-agreement protocols adds an unnecessary and undesirable property: non-repudiation, the signer cannot deny that they were an active participant in the protocol. Protocols that provide deniability, a notion first introduced by Dwork, Naor, and Sahai in [135], actively avoid this property. There are many flavors and variations of deniability; see e.g. [189] for a classification. A protocol has *offline deniability* [117] if a judge, when given a protocol transcript and all keys involved, cannot tell whether the transcript is genuine or forged. The KEM-authenticated handshake of KEMTLS, unlike the

[4] Namely, through publishing keys in DNS records.

signature-authenticated handshake of TLS 1.3, has offline deniability. Specifically, following the terminology of [189], KEMTLS provides offline deniability in the *universal deniability* setting (meaning the simulator only has access to parties' long-term public keys) against an unbounded judge with full corruption powers (meaning the judge gets the parties' long-term secret keys as well as any per-session coins). *Online deniability* [124] is harder to achieve. In the online setting, the judge may coerce a party to send certain malicious messages to the target. KEMTLS does not achieve online deniability, and we would likely have to make significant changes to the protocol to achieve it.

5.4.6 Comparison with OPTLS

Our proposal for a signature-free handshake protocol in TLS shares a lot of similarities with the OPTLS protocol [221], discussed in chapter 4. The same arguments for revisiting key exchange for authentication as discussed in section 4.1 apply to KEMTLS. However, post-quantum KEMs are much more mature than CSIDH and much more computationally efficient. Thus, the KEMTLS signature-free approach to the TLS handshake offers major advantages over TLS 1.3 and OPTLS.

As we will show in chapter 13, KEMTLS simultaneously reduces the amount of data transmitted during a handshake, reduces the number of CPU cycles spent on asymmetric crypto, reduces the total handshake time until the client can send application data, and reduces the trusted code base.

5.5 Client-authentication in KEMTLS

Although perhaps not used much for web browsing, client authentication is an important optional feature of the TLS handshake. In TLS 1.3, a server can send the client a `CertificateRequest` message. The client replies with its certificate in a `ClientCertificate` message and a `ClientCertificateVerify` message containing a signature. This allows mutual authentication.

In this section, we show how to extend KEMTLS to provide client authentication. Figure 5.3 adds a client authentication message flow to KEMTLS.

Recall that we assume that the client does not have the server's certificate when initiating the handshake, and similarly, the server does not have the client's certificate in advance. There may be more efficient message flows possible if this is the case. We examine one such possibility in chapter 6.

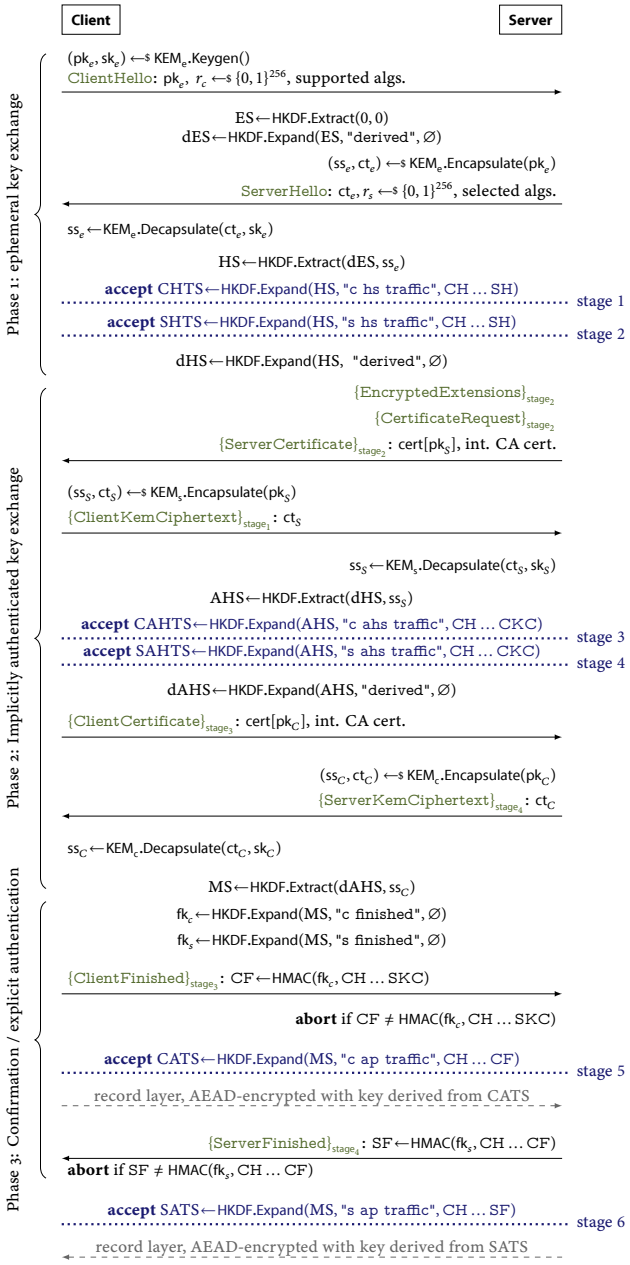


Figure 5.3: The KEMTLS handshake with client authentication

5.5.1 Extending KEMTLS with client authentication

We permit the client and server to use different KEM algorithms (KEM_c and KEM_s) as that may be desirable for functionality or efficiency purposes.

In TLS 1.3, a server is only allowed to send a `CertificateRequest` message if it has been authenticated with a certificate [298, Sec. 4.3.2]. This restriction ensures that the certificate containing the identity of the client is only revealed to the intended server. Transferring this property to KEMTLS requires a careful modification of the key schedule. In the KEMTLS key schedule, we derive the CAHTS and SAHTS "authenticated" handshake traffic secrets from the shared secret ss_S encapsulated against the public key in the server's certificate. This allows the client to encrypt its certificate such that it can only be decrypted by someone holding the server certificate's private key.

After that, the server encapsulates against the public key contained in the client certificate to compute another shared secret ss_C. We mix this shared secret ss_C into the derivation of MS (in a straightforward extension of the key schedule of KEMTLS). Combining ss_C and ss_S ensures that all application traffic encrypted under keys derived from MS (stages 5 and 6) will only be legible to the authenticated server and client; the ephemeral shared secret ss_e further provides forward secrecy. Additionally, by sending the `ClientFinished` message containing a MAC under a key derived from MS, the client explicitly authenticates itself to the server at stage 5.

5.5.2 Certificate request messages

Note that an attacker can insert the `CertificateRequest` message into a handshake: it can construct all messages up to and including the `ServerCertificate` message with public data. The forged handshake will fail (unless the attacker has the server's long-term secret keys), and only the intended server will be able to read the `ClientCertificate` message. However, in applications that prompt the client to select a certificate (such as web browsers), this prompt will pop up, which may be undesirable. Such applications may need to consider post-handshake authentication flows and disallow in-handshake authentication. Post-handshake authentication is supported in TLS 1.3; we leave specifying it for KEMTLS as future work.

We note that large-scale usage statistics, collected by Birghan and Van der Merwe in the Firefox browser, suggest that already the majority of mutually

authenticated connections use post-handshake authentication [61]. Only three connections, out of hundreds of billions of connections, were found that used in-handshake client authentication. Post-handshake authentication was still only used less than 6000 times per day, out of 32–66 billion TLS 1.3 connections. This suggests that client authentication does not see significant use in browsers.

5.5.3 Alternative protocol flows for client authentication

The method for client authentication proposed in this section introduces an extra RTT. This is a consequence of staying close to the existing key schedule and state machine for KEMTLS.

Allowing the `ServerFinished` message to be transmitted immediately after `ServerKemCiphertext` and deriving SATS then would allow the server to initiate transmitting data sooner. This would reduce the overhead to an extra half RTT but relies on implicit authentication. However, this complicates the key schedule, as `ServerFinished` would no longer be sent last.

We might also allow the client to send `ClientFinished` immediately after `ClientCertificate` . The client would then derive CATS without mixing in ss_C. This would not introduce extra RTTs before the client can send data, but the data that the client sent can then not be straightforwardly authenticated.

5.6 KEM public keys in certificates

Although KEMTLS removes the signatures from the TLS handshake, the protocol still relies on signatures to make the public-key infrastructure work. The identity and public key in the client or server "leaf" certificates are still signed by a certificate authority (CA) and the recipient of a certificate message still verifies the certificate's signature using preinstalled CA public keys.

Issuing of certificates is currently typically done by submitting a PKCS#10 certificate-signing request (CSR) to a CA. This submission was often done manually, typically once a year (which was a common validity period). Nowadays, many certificates on the internet are issued by Letsencrypt.org through an automated, interactive certificate issuance protocol called ACME [23]. Although it is interactive, ACME still relies on exchanging CSRs.

CSRs are specified in RFC 5967 [277] and contain the to-be-signed public key as well as information about the entity requesting the certificate. This

information may include hostnames, IP addresses, or even postal addresses. A CSR is signed by the party requesting the certificate before submission. This proves to the CA that the requester has possession of the private key corresponding to the public key. In prior versions of TLS, the algorithms used in certificates could always be used to sign CSRs. This is no longer the case for the post-quantum KEMs that are part of the NIST PQC standardization project and presents a problem for the adoption of KEMTLS.

A naive alternative to signing CSRs is for the CA to take the public key from the CSR and encapsulate to it. The CA then obtains a shared secret ss and ciphertext ct. It would then generate a certificate and simply symmetrically encrypt it using ss; implicitly authenticating the requester. The encrypted certificate and ct would then be returned to the requester, who proves possession of the private key by decapsulating ct and decrypting the certificate.

However, this foregoes other parts of the TLS public-key infrastructure used on the internet. All certificates issued are carefully accounted for on certificate transparency (CT) logs [235]. By requiring all certificates to be publicly posted, one can track which CAs issue certificates for particular websites. This system was developed after CA DigiNotar was breached in 2011, and fraudulent certificates were issued for several websites, including Google's gmail.com [12]. With certificate transparency, Google can set up monitoring for newly issued certificates for gmail.com and the submission of fraudulent certificates to the public logs would trigger an alert. The Chrome and Safari browsers require certificates to have been submitted to CT logs before they consider them valid [15, 103].

Our naive approach requires that the certificate would be signed, and thus necessarily published to CT logs, before the authenticity of the requester was made explicit. As invalid alerts would undermine the reliability of certificate transparency, this presents a problem.

A generic approach to certificate issuance would thus likely have to be interactive. Significant numbers of certificates on the internet are now being issued interactively through ACME every day [192], suggesting this approach may be fruitful. Alternative certificate issuance protocols, such as the Certificate Management Protocol (CMP) [257] or Certificate Request Message Format protocol (CRMF) [313], already define interactive ways of issuing certificates for key exchange keys. We do note that CMP and CRMF's "indirect proof of possession", which sends the encrypted certificate as the challenge message, has the same flaw as our naive approach described above, as the certificate has

to be submitted to the CT logs by the issuer before the issuer received proof of correct decryption.

On the academic side of this problem, Güneysu, Hodges, Land, Ounsworth, Stebila, and Zaverucha proposed a non-interactive proof of possession for the lattice-based KEMs Kyber and FrodoKEM [166]. In their approach, they rely on the multi-party-computation-in-the-head paradigm to generate the proof during key generation. This furthermore allows them to bind attribute data, such as identities, to the proof. This approach could be used to construct a CSR-compatible signature for lattice KEMs. They additionally use the attribute-binding feature to provide a solution to the not-yet-mentioned problem of revocation, which currently also heavily relies on signatures.

How certificates will be obtained for KEM public keys in the real world remains to be seen. The non-interactive proof of possession may be a viable path forward for issuance through CSRs, but it is not generically compatible with all KEMs. This means that we may see more interactive issuance protocols in the future.

5.7 Conclusion

We have now shown how we can avoid handshake signatures and construct a KEM-authenticated TLS handshake, without suffering the penalty of a full additional round trip of a naive solution. In chapter 7 we will discuss the security properties of KEMTLS, as well as provide a proof of its security. We will also model the protocol in Tamarin in chapter 9. Finally, in part III, we will discuss how we implemented KEMTLS and compare the performance with TLS 1.3, OPTLS, and the KEMTLS-PDK proposal that we will discuss in the next chapter.

6 More efficient KEMTLS with pre-distributed keys

In the previous chapter, we explained that TLS and KEMTLS are efficient key-exchange protocols that do not assume that the client already has the server's static key before starting a handshake. In particular, KEMTLS very carefully avoids the extra round trip that would be necessary if the static public key would be fetched beforehand. However, clients often connect to a particular TLS server many times: e.g., because they are fetching many web pages from the same website, or maybe because the client is an Internet-of-Things device that is only set up to connect to a single service.

Contributions

In this chapter, we propose and examine a variant of KEMTLS that makes use of the assumption that the client already knows the server's long-term public key. This variant is more efficient, as less data needs to be transmitted. The server is additionally immediately authenticated, just like in TLS 1.3. Finally, using this variant we can do client authentication without KEMTLS' additional round-trip. In chapter 8, we prove the security of this protocol, and in chapter 14 we examine its performance.

6.1 Introduction

Both TLS 1.3 and KEMTLS assume that the client does not know the server's long-term public key when sending the ClientHello message; the certificate is transmitted as part of the handshake, even if the client already knows the public key. We refer to the scenario when a client already knows the server's public key as the *pre-distributed-key* or *cached-key* scenario. This occurs, for example, when web browsers cache certificates of frequently accessed servers; when mobile apps store certificates of the limited number of servers they connect to; when TLS is used by IoT devices that only ever connect

to one or a handful of servers and have those certificates preinstalled; or when certificates have been distributed out of band, for example, through DNS [200]. This scenario has, in fact, already been considered for TLS. The TLS cached information extension [312] allows the client to inform the server that it already knows certain certificates so they need not be transmitted. However, this RFC is not widely implemented (and indeed has not been updated for TLS 1.3), perhaps because (pre-quantum) certificates are fairly short and thus bandwidth savings are limited. Another reason for the lack of adoption is possibly the privacy implications of the client identifying which server identities it is familiar with. In KEMTLS-PDK, this identification may be avoided by using trail decryption. Although the upcoming encrypted ClientHello TLS extension may be better suited for hiding this information. We will briefly discuss this, along with KEMTLS-PDK's other anonymity properties, in section 8.1.1.

In concurrent work, Kampanakis, Bytheway, Westerbaan, and Thomson proposed letting the client indicate that it already has the server's intermediate CA certificates [202, 203]. This is intended to serve as a "compression" mechanism, rather than an abbreviated handshake. They similarly assume networks where the clients are set up with some knowledge about the server's identities, or web browsers that are set up to fetch all currently valid intermediate CA certificates out-of-band. Their mechanism is a 1-bit signal from the client, which the server uses to omit intermediate certificates; server certificates are still transmitted. KEMTLS-PDK avoids transmitting any server certificate information and can gracefully recover from expired long-term keys.

6.1.1 Using pre-distributed keys

In this chapter, we investigate how we can make use of out-of-band distributed keys in TLS in the post-quantum world, both when caching signature-based certificates in the TLS 1.3 handshake, and, more importantly, when using KEM-based authentication as used in KEMTLS. More specifically, we introduce KEMTLS-PDK, a variant of KEMTLS that makes use of pre-distributed keys for earlier authentication in the protocol flow. We also describe KEMTLS-PDK with proactive client authentication and show that the benefits of earlier authentication are even more significant.

We give a sketch of KEMTLS-PDK in figure 6.1. The central property to observe is that, like in TLS 1.3, but unlike in KEMTLS, the first message from the server

serves as key confirmation. This means that in this variant of KEMTLS, like in TLS 1.3, the server is explicitly authenticated after a single round trip. Another similarity with TLS 1.3 is that the server can send data to the client first.

The version of KEMTLS-PDK with proactive client authentication is shown in figure 6.2. In this variant, we use the key that is encapsulated to the server's long-term public key to encrypt the client certificate and send it along with the client's initial message. This allows 1-RTT client authentication, unlike KEMTLS where client authentication results in a full additional round trip over the server-only-authenticated handshake. We refer to this handshake as using *proactive* client authentication because the client sends its certificate unasked: we believe that in many applications of mutual TLS authentication, such as in Internet-of-Things or service-to-service applications, a client can be expected to know that they need to authenticate themselves.

6.2 KEMTLS with pre-distributed long-term keys

Even though one of the strengths of the TLS protocol is its ability to establish a secure channel with a previously unknown party, it is very often not the case that the communicating party is completely unknown. The TLS 1.3 PSK mechanism can be used with session tickets to enable fast resumption after an initial full handshake [298, Fig. 3]. These mechanisms rely mostly on symmetric cryptography, although TLS 1.3 allows an optional additional ephemeral key exchange in resumption for forward secrecy. There is nothing precluding the use of these mechanisms, including the "0-RTT " client-to-server data flow in the resumption message in KEMTLS.

However, because TLS 1.3 resumption relies on symmetric cryptography, it is not very flexible. The security properties of a resumed session are tied to the previous session. This includes, e.g., if the session was mutually authenticated. For these reasons, session tickets expire quickly, after at most 7 days [298, Sec. 4.6.1]. There are also privacy issues as the opaque tickets might contain tracking information. To prevent such tracking, Sy, Burkert, Federrath, and Fischer [337] even suggested limiting session lifetime to only 10 *minutes*.

Because externally distributed PSKs are symmetric, we quickly run into concerns there as well. If clients have a common installation profile and share keys, then any single client compromise results in no remaining security for any client. A client that also acts as a server additionally needs to use

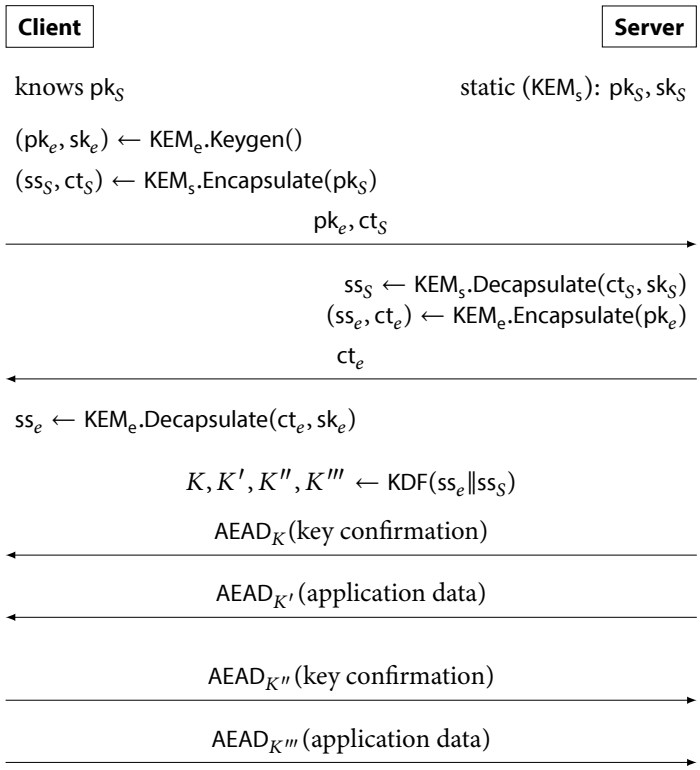


Figure 6.1: Sketch of KEMTLS-PDK with server-only authentication.

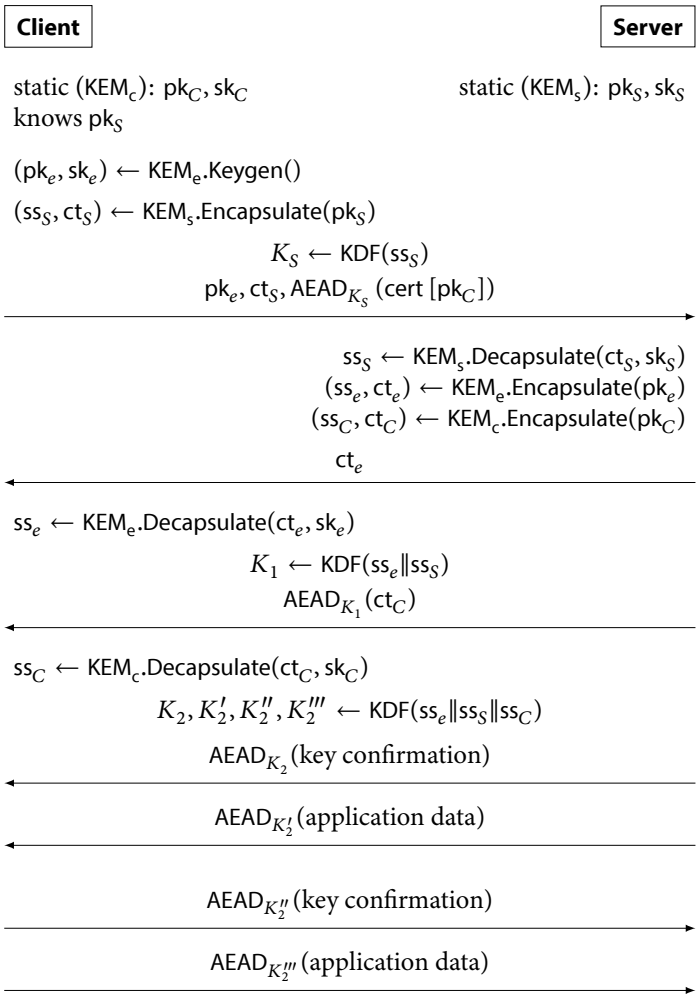


Figure 6.2: Sketch of KEMTLS-PDK with proactive client authentication.

different keys in the two roles role to prevent the Selfie attack [131]. This means we need a key for each client and server pair, quickly turning this into a key-management nightmare.

In our proposed KEMTLS-PDK, we employ a more flexible approach by distributing a server's long-term KEM public key instead of a symmetric key. A detailed protocol flow diagram of KEMTLS-PDK is given in figure 6.3.

Like in KEMTLS, the client encapsulates to the server's long-term KEM public key pk_S, obtaining a ciphertext ct_S and a shared secret ss_S. However, as we assume that the client already has pk_S, it can do this *at the start* of the connection and send ct_S in a `ClientHello` extension. We plug ss_S into the key derivation schedule at the earliest possible stage when deriving the Early Secret (ES). Deriving ES from ss_S avoids changing the key schedule. It also intuitively makes sense, as data encrypted under traffic keys derived from ES has no forward secrecy or replay protection; just as in TLS 1.3 with PSK and 0-RTT data [298, Sec. 2.3]. The only server that can read a message encrypted under a key derived from ES is the server that has access to sk_S; we consider such keys *implicitly authenticated*. For forward secrecy, we also send an ephemeral public key pk_e in the `ClientHello` message.

Except for the additional extension transmitting ct_S, the `ClientHello` message is the same as in KEMTLS. This allows to fall back to the regular KEMTLS handshake protocol, e.g., if the client has an out-of-date server public key.

The server replies with the encapsulation ct_e of ephemeral shared secret ss_e in the `ServerHello` message. It also indicates in an extension that it has accepted ciphertext ct_S and is proceeding with KEMTLS-PDK. Then it proceeds similarly to the original TLS 1.3 handshake. The server derives HS from ES and ss_e and sends the `EncryptedExtensions` message encrypted under a key derived from HS. It then immediately finishes its part of the handshake by sending a MAC over the message transcript in the `ServerFinished` message. This confirms the server's view of the handshake to the client and explicitly authenticates the server. The server can now start sending application data. The client follows up by also confirming its view of the handshake in a `ClientFinished` message. This indicates the client is ready to communicate as well.

6.2.1 Proactive client authentication

In some applications, such as in a VPN, the client may already know that the server will require mutual authentication. This means that a client can

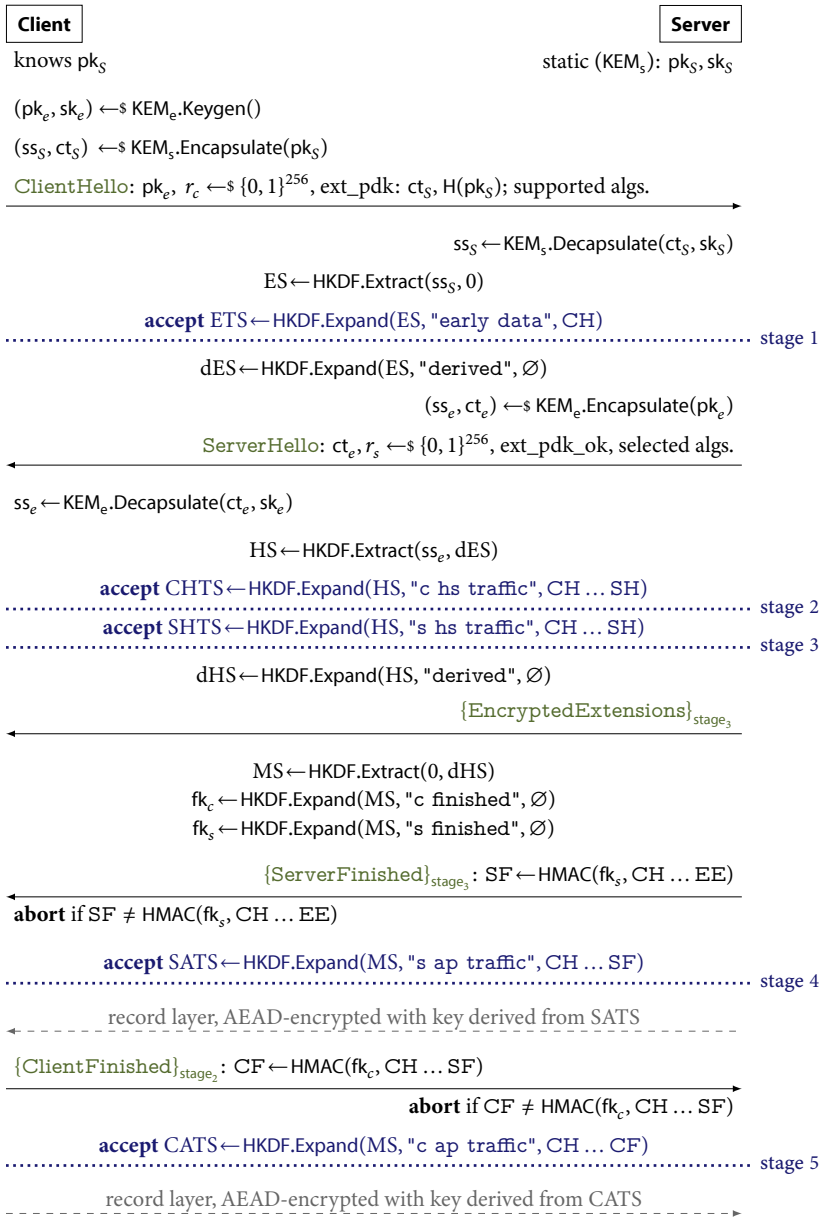


Figure 6.3: The KEMTLS-PDK handshake with unilateral (server-only) authentication using pre-distributed server public keys

proactively authenticate by sending its certificate as early in the handshake as possible, and in particular before the server requests the certificate. For privacy reasons, client authentication in TLS requires that we verify the identity of the server and send the certificate encrypted [298, Sec. E.1.2]. Performing client authentication in KEMTLS thus requires a full additional round trip: we can only send the client certificate after authenticating the server and the server cannot send the ciphertext before it receives pk_C.

In KEMTLS-PDK, the client already possesses the server's long-term public key. We can use the shared secret obtained from encapsulating to the corresponding long-term key to send a client certificate along in the `ClientHello` message. This gives us mutual authentication within a single round trip. The server supplies the challenge ciphertext ct_C to the client and derives the Main Secret MS and, through MS, the confirmation and traffic keys from ss_e, ss_S, and ss_C. At this point, the server can start sending application data to the client. The client is implicitly authenticated, as they have not yet confirmed that they derived the same keys. As the keys are derived from ss_C only the client who possesses sk_C can read these messages. To finish the handshake the client sends its key confirmation message before proceeding to the application traffic. KEMTLS-PDK with mutual authentication is shown in figure 6.4.

Note that we did not include the Authenticated Handshake Secret (AHS) key, which we introduced for KEMTLS, in the key schedule of KEMTLS-PDK. In the KEMTLS-PDK key schedule, we already use the shared secret encapsulated to the server's long-term public key in the very first handshake key ES. Leaving out AHS simplifies our protocol diagrams and avoids some additional analysis in our proofs. If desirable for ease of implementing KEMTLS(-PDK), it is possible to include derivation of AHS in the key schedule.

We submit the complete client certificate in the mutually authenticated KEMTLS-PDK handshake. As there are usually more servers than clients, we thus avoid requiring that the server stores all client certificates if it wants to use mutually authenticated KEMTLS-PDK. Although we do not further discuss this, a variant protocol in which the client certificate is replaced by a small identifier, so that the public key can be retrieved from storage by the server and does not need to be transmitted, can be trivially constructed from the mutually authenticated KEMTLS-PDK handshake.

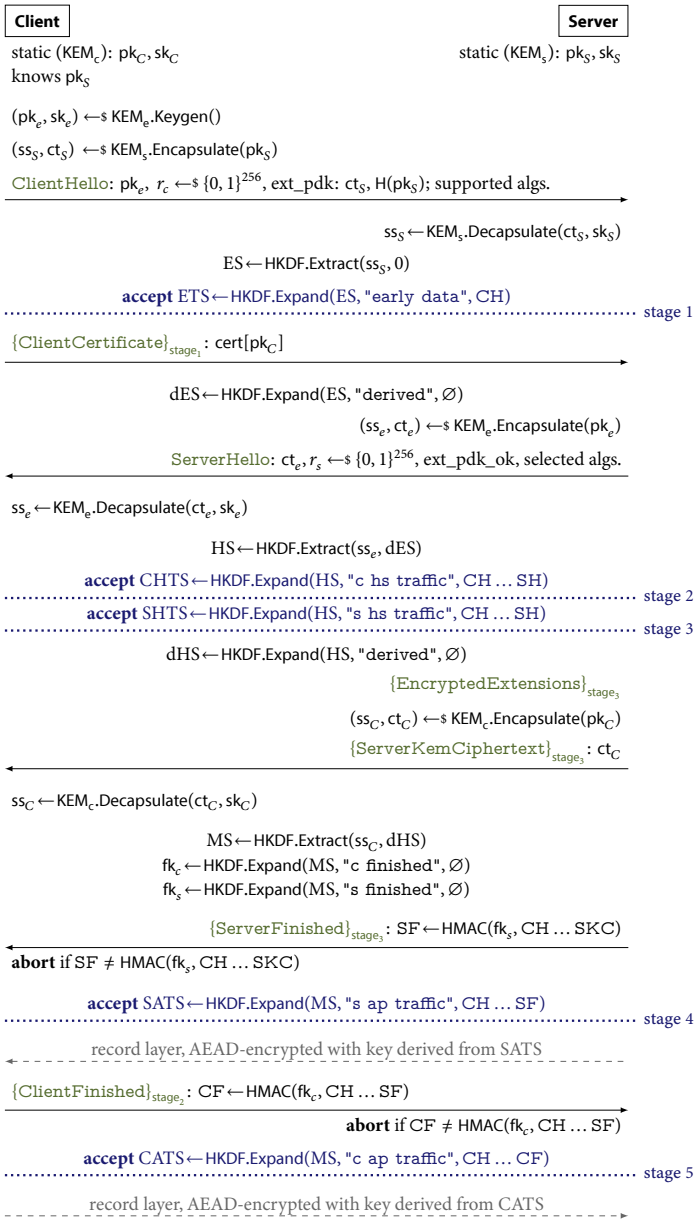


Figure 6.4: The KEMTLS-PDK handshake with proactive client authentication using pre-distributed server public keys

6.2.2 Falling back to KEMTLS

The KEMTLS-PDK handshake may fail if the server no longer has possession of the private key corresponding to the ciphertext ct_S that the client encapsulated to the server's long-term public key. This may for example occur if the long-term public key expired (and perhaps the client had clock drift so did not think it was) or if the key was replaced. In implementations that support both KEMTLS and KEMTLS-PDK, it is possible to add a rejection mechanism for those cases. This can for example be done by including an "accepted KEMTLS-PDK" indicator in an `ServerHello` extension as part of the KEMTLS-PDK handshake, which could then be omitted on rejection. If rejecting is handled in this way, the server does not even need to be aware of KEMTLS-PDK as an unaware server implementation would implicitly omit such an extension. When a KEMTLS-PDK handshake is thus rejected by the server, the server can proceed as if the client sent a `ClientHello` handshake. The client can then simply recompute the ES key as in the KEMTLS key schedule and resume as if it sent a KEMTLS `ClientHello` message: any KEMTLS-PDK handshake extensions that were part of the initial message can simply be assumed to not have been acted upon.

The only problem in this rejection mechanism is how to handle the encrypted `ClientCertificate` message that a proactively authenticating client sends to the server along with the KEMTLS-PDK `ClientHello` message. The server cannot decrypt this message, as it has been encrypted with the shared secret encapsulated to the public key it no longer has the private key to. This prevents it from including this message in the handshake transcripts, which would result in mismatching transcripts in key computations and `Finished` messages. To remedy this, the client drops the proactive `ClientCertificate` message from its transcript upon rejection of KEMTLS-PDK. We will discuss the security implications of this rejection mechanism and the co-existence of KEMTLS and KEMTLS-PDK in section 8.3.

6.3 Restoring some ephemeral secrecy for client authentication

When the client wants to proactively authenticate itself in KEMTLS-PDK, it needs to send its `ClientCertificate` message along with the `ClientHello` message. As the client has not yet completed the ephemeral key exchange

at this point (after all, they have not yet received ct_e from the server), the certificate is encrypted using the ETS key, which is immediately derived from the key encapsulated to the server's static key. This means that the key is not forward-secure and the message can be replayed. While the information in the client certificate is static, the client's identity is part of the certificate and is confidential information.

In Günther, Rastikian, Towa, and Wiggers [168], we proposed a partial remedy to this problem. KEMTLS-EPOCH introduces an additional, *semi-static* server key. This key is periodically rotated, which divides the lifetime of the long-term static key into periods we call "epochs". This means we have a different semi-static key in epoch i than we have in epoch $i - 1$. If the static and semi-static keys are compromised in epoch i, any client certificate messages sent in epochs before the compromise will remain secure. KEMTLS-EPOCH additionally provides a synchronization message, falling back on a full KEMTLS-like handshake when peers are out of sync. We will not further discuss KEMTLS-EPOCH, and we refer to the published work for more details.

6.4 Conclusions

In this chapter, we presented how we can achieve a more efficient KEMTLS handshake if we assume that the client already has the server's public key. In this abbreviated handshake, the server is explicitly authenticated in its first response to the client, and we can do proactive client authentication within a single round-trip. Chapter 8 further discusses and proves the security properties of KEMTLS-PDK. We will also show how we implemented KEMTLS-PDK and compare the performance of our proposal with (post-quantum) TLS 1.3, TLS 1.3 with cached certificates, OPTLS, and KEMTLS in chapter 14.

Part II

Security of KEMTLS

7 Security of KEMTLS

In this chapter, we will give a formal analysis of the security of KEMTLS in the reductionist security model. We will first give a somewhat informal description of the structure and key points of our security claims. We will also give a high-level overview of our proof. Afterward, we will formally define the model that we will use in the proof. We specify and prove both unilaterally (server-to-client) authenticated and mutually authenticated KEMTLS.

The proof below is based on the original proof of KEMTLS as it was presented at ACM CCS 2020, and the updates published in the online version [321]. To support the proof of KEMTLS-PDK in chapter 8, we adopt the syntax of the model originally developed for KEMTLS-PDK [320] in this chapter as well. We also extend the original proofs to cover mutually authenticated KEMTLS.

7.1 Overview of the security analysis

As KEMTLS is an adaptation of TLS 1.3, our security analysis follows previous techniques for proving the security of TLS 1.3. In particular, we base our approach on the reductionist security approach of Dowling, Fischlin, Günther, and Stebila [127, 128]. Briefly, that approach adapts a traditional Bellare–Rogaway-style [32] authenticated-key-exchange security model to accommodate multiple *stages* of session keys established in each session, following the multi-stage AKE security model of Fischlin and Günther [143]. The model used for TLS 1.3 in [127, 128] supports a variety of modes and functionality, such as mutual versus unilateral authentication, full handshake and pre-shared key modes, and other options. We simplify the model for this application, though we also add some other features, such as explicit authentication and granular forward secrecy.

In this section, we give an informal description of the security model, including the adversary interaction (queries) for the model; the specific security properties desired (Match security, which ensures that session identifiers effectively match partnered sessions, and Multi-Stage security, which models

confidentiality and authentication as described below); and a sketch of the proofs showing that KEMTLS satisfies these properties. The full syntax and specification of the security properties as well as the detailed proofs of security for KEMTLS appear in section 7.2.

7.1.1 Security goal

Our main security goal is that keys established in every stage of KEMTLS should be indistinguishable from a random key, in the face of an adversary who sees and controls all communications, can learn other stages' keys, can compromise unrelated secrets (such as long-term keys of parties not involved in the session in question), and may, after-the-fact, learn long-term keys of parties involved in the session ("forward secrecy"). This is the same security goal and threat model for TLS 1.3 [127, 128, 298]. We distinguish between implicit authentication (where a key could only be known by the intended peer), which follows from key indistinguishability and forward secrecy, and explicit authentication (which assures that the intended peer participated). We consider KEMTLS with unilateral and mutual authentication.

7.1.2 Tightness

Theorem 7.7, which states the security of KEMTLS, is *non-tight*, due to hybrid and guessing arguments. While it is certainly desirable to have tight results, only a few authenticated-key-exchange protocols have tight proofs, most of which with specialized designs. Most previous results on TLS 1.3 [127, 128] are similarly non-tight, except for recent works [113, 118] which reduce from multi-user security of the symmetric encryption scheme, MAC, KDF, and signature scheme, and the strong-DH assumption. A tight reduction of the security of TLS 1.3's simpler PSK handshake mode was published at EUROCRYPT 2022 [112]. After our proof was originally published, it was shown that Kyber does have multi-user security [133].

One can view a non-tight result such as theorem 7.7 as providing heuristic justification of the soundness of the protocol design, and one can in principle choose parameters for the cryptographic primitives that yield meaningful advantage bounds based on the non-tight reductions. Proving the security of KEMTLS using multi-user security of the primitives remains as future work.

7.1.3 Quantum adversaries

The proof of theorem 7.7 proceeds in the standard model (without random oracles), and does not rely on techniques such as the forking lemma or rewinding. This means techniques like Song's "lifting lemma" [330] can be applied to show that KEMTLS is secure against quantum adversaries, provided that each of the primitives used is also secure against quantum adversaries.

7.1.4 Negotiation and downgrade resilience

We do not explicitly model algorithm negotiation in KEMTLS, but it merits consideration given the likelihood that any deployment of KEMTLS would support multiple algorithms within KEMTLS, and might also be running in parallel with a TLS 1.3 implementation. We consider adversarial downgrades among each of the following negotiated choices:

- Protocol: KEMTLS versus TLS 1.3.
- Ephemeral key exchange: which KEM within KEMTLS, or which group if downgraded to Diffie–Hellman in TLS 1.3.
- Authenticated-encryption scheme and hash function.
- Public-key authentication: which KEM within KEMTLS, or which signature scheme if downgraded to TLS 1.3.

We consider three levels of downgrade resilience:

1. *Full downgrade resilience*: the adversary cannot cause a party to use any algorithm other than the one that would be used between the two honest parties if the adversary was passive. This is called optimal negotiation by [129] and downgrade security by [56].
2. *No downgrade to unsupported algorithms*: the adversary can cause parties to use a different algorithm than the optimal one that would be used if the adversary was passive, but cannot cause a party to use an algorithm that it disabled in its configuration. This is called negotiation correctness by [56].
3. *No downgrade resilience*: the adversary can cause a party to use any algorithm permitted in the standard (e.g., [2]).

We assume that none of the algorithms supported by the client or server are broken at the time the session is established, and the downgrade adversary's goal is to force the use of an algorithm that the adversary hopes to have a better chance of breaking in the future (e.g., elliptic-curve Diffie–Hellman instead of a post-quantum KEM; AES-128 instead of AES-256).

In KEMTLS, for client sessions, any algorithms used prior to the acceptance of the stage-6 key (i.e., ephemeral KEM, authenticated encryption of handshake and first client-to-server application flow) cannot be downgraded to an unsupported algorithm (barring an implementation flaw), but can still be downgraded to a different client-supported algorithm.[1] The explicit authentication that the client receives for the stage-6 key includes confirmation in the `ServerFinished` message that the client and server have the same transcript including the same negotiation messages, which implies full downgrade resilience once the stage-6 key is accepted, assuming that the hash, MAC, KEM, and KDF used are not broken by the time of acceptance.[2] Since there is no client-to-server authentication in the unilaterally authenticated KEMTLS protocol, servers obtain "no downgrade to unsupported algorithms" for all their stages. In mutually authenticated KEMTLS, servers obtain explicit authentication for the stage-5 key, which includes confirmation in the `ClientFinished` message that the client and server have the same transcript. As this transcript includes the negotiation messages, this implies full downgrade resilience once the stage-5 key is accepted.

7.1.5 Strength of the ephemeral KEM

The proof requires that the ephemeral KEM is slightly stronger than passive IND-CPA security: it needs to be secure against a single decapsulation query (IND-1CCA). This is subtle and counterintuitive: one might expect that IND-CPA would be enough for ephemeral key exchange (indeed, we missed this in an early draft of [321]). However, in an AKE security model that replaces the public key of the client and the ciphertext of the server, but allows the

[1] While KEMTLS's implicit authentication in stage 3/4 does not preclude downgrades, TLS 1.3's signature-based explicit authentication at stage 3 does provide transcript authentication. Hence, when KEMTLS and TLS 1.3 are simultaneously supported by a client, an attacker cannot downgrade 1-RTT application data from KEMTLS to TLS 1.3.

[2] Signature-based authentication in TLS 1.3 means that TLS 1.3's downgrade-resilience relies only on the signature and hash being unbroken by the time of acceptance.

adversary to send a different ciphertext back to the client without invalidating the target session at the server, this is unavoidable [194, 220]. For example, in IND-1CCA experiment in the proof of KEMTLS we replace the public key that is sent in the ClientHello message and the ciphertext that is returned by the server in the ServerHello message by the challenge public key and ciphertext; but the adversary is allowed to construct and send a different ciphertext to the client session. This different ciphertext needs to result in a different shared secret at the client and server sessions for the simulation to remain sound, so in this case the reduction uses its single decapsulation query to obtain the expected shared secret in the client session; there is no other way to continue the simulation otherwise.

As IND-1CCA is strictly weaker than IND-CCA, any IND-CCA KEM certainly suffices for the ephemeral KEM, but for most known post-quantum schemes this incurs the cost of re-encryption using the Fujisaki–Okamoto (FO) transform [150]. There are concrete attacks against several non-FO-protected lattice- and isogeny-based KEMs that use several thousand decapsulation queries [144, 154], but none with just a single query. Huguenin-Dumittan and Vaudenay have recently shown that IND-1CCA secure KEMs can be constructed more efficiently than how KEMs are made IND-CCA secure using the FO transform [182]. No IND-1CCA KEMs are part of the NIST PQC standardization project, however, so IND-CCA-secure KEMs remain the safe choice.

7.1.6 Security model

In the following, we describe informally the security model. The model is based on the multi-stage AKE model used by Dowling, Fischlin, Günther, and Stebila [127, 128] to analyze signed-DH in TLS 1.3. We describe the properties associated with each session, as well as the main security properties: Match and Multi-Stage security. The precise formulation of the model appears in section 7.2.

Model syntax

Each participant has a long-term public key and corresponding private key; we assume a public-key infrastructure for certifying these public keys, and that the root certificates are pre-distributed, but certificates are not pre-distributed. Each participant (client or server) can run multiple instances of the protocol,

each of which is called a *session*. Note that a session is a participant's local instance of a protocol execution; the two parties communicating with each other each have their own sessions. Each session consists of multiple *stages* (for KEMTLS, there are 6 stages as marked in figures 5.2 and 5.3 for unilaterally and mutually authenticated KEMTLS respectively).

For each session, each participant maintains a collection of session-specific information, including: the identity of the intended communication partner; the *role* of the session owner (either initiator or responder); the *state of execution* (whether it has accepted a key at a certain stage, or is still running, or has rejected); as well as protocol-specific state. For each stage within a session, each participant maintains stage-specific information, including: the *key* established at the stage (if any); a *session identifier* for that stage; and a *contributive identifier* for that stage. Two stages at different parties are considered partnered if they have the same session identifier. The session identifiers for KEMTLS are the label of the key and the transcript up to that point (see section 7.3). For the first stage, the contributive identifier is the `ClientHello` initially, then updated to the `ServerHello` message; for all other stages, the contributive identifier is the session identifier.

The model also records security properties for each stage key:

1. The level of *forward secrecy* obtained for each stage key. The three levels of forward secrecy we meet are detailed in section 7.1.7 below. The model allows for *retroactive* revision of forward secrecy: the stage-i key may have weak forward secrecy at the time it is established in stage i, but may have full forward secrecy once a later stage $j > i$ has completed (i.e., after receiving an additional confirmation message). The level of forward secrecy also implies whether the key should be considered *implicitly authenticated*.

2. Whether the stage is *explicitly authenticated*: if a party accepts a stage, is it assured that its partner was live and established an analogous stage? Again our model allows for *retroactive* explicit authentication: while a stage-i key may not have explicit authentication when established in stage i, completion of a later stage $j > i$ may imply that a partner to stage i is now assured to exist.

3. Whether the key is intended for *internal* or *external* use. TLS 1.3 and KEMTLS internally use some of the keys established during the

handshake to encrypt later parts of the handshake to improve privacy, whereas other keys are "external outputs" of the handshake to be used for authenticated encryption of application data. Internally used keys must be treated more carefully in the security experiment.

4. Whether the key is *replayable*. This will be relevant in chapter 8 which has replayable stages; in KEMTLS all state keys are not replayable.

Our inclusion of forward secrecy and explicit authentication is an extension to the multi-stage AKE model used for TLS 1.3 [127, 128].

Adversary interaction

The adversary is a probabilistic algorithm that triggers parties to execute sessions and controls the communications between all parties, so it can intercept, inject, or drop any message. As a result, the adversary facilitates all interactions, even between honest parties.

The adversary interacts with honest parties via several queries (which we define in more detail in section 7.2.2). The first two queries model the typical protocol functionality, which is now under the control of the adversary:

- NewSession: Creates a new session at a party with a specified intended partner and role. We also set if mutual authentication is used.
- Send: Delivers a message to a session at a party, which executes the protocol based on its current state, updates its state, and returns any outgoing protocol message.

The next two queries model the adversary's ability to compromise parties' secret information:

- Reveal: Gives the adversary the stage key established in a particular stage in a particular session. This key and the key at the partner session (if it exists) are marked as revealed.
- Corrupt: Gives the adversary a party's long-term secret key. This party is marked as corrupted.

The Reveal and Corrupt queries may make a stage *unfresh*, meaning the adversary has learned sufficiently much information that no security can be expected of this key.

The final query models the challenge to the adversary of breaking a key established in a stage:

- Test: For a session and stage chosen by the adversary, returns either the real key for that stage or a uniformly random key, depending on a hidden bit b fixed throughout the experiment.

Some additional conditions apply to the handling of queries. For keys marked as intended for internal use, the execution of the Send query pauses at the moment the key is accepted, giving the adversary the option to either Test that key or continue without testing. This is required since internal keys may be used immediately for, e.g., handshake encryption, and allowing the adversary to Test the key after it has been used to encrypt data would allow it to trivially win. For keys that are not considered authenticated at the time of the Test query, the query is only permitted if the session has an honest contributive partner, otherwise, the adversary could trivially win.

7.1.7 Security of authenticated key-exchange protocols

There are many extensions to the Bellare–Rogaway model to capture different functionality and security properties of AKE protocols; we refer to [77, Ch. 2] for a summary. A sequence of works [82, 83, 143] split AKE security into two distinct properties: the traditional session-key indistinguishability property dating back to Bellare and Rogaway [32], and a property called Match-security, which models the soundness of the session identifier, ensuring that the session identifier $\pi.\mathsf{sid}$ properly matches the partnered $\pi'.\mathsf{sid}$ and the correctness property that partnered sessions compute the same session key. For well-chosen session identifiers, proving the technical properties of Match-security typically does not depend on any cryptographic assumptions, and instead follows syntactically. This is indeed the case for both TLS 1.3 and KEMTLS, although for KEMTLS we account for delta-correctness of the KEMs: some KEM have a small chance that the recipient of the ciphertext does not recover the same shared secret (definition 2.14). Details are in section 7.2.3.

The Multi-Stage model captures both key indistinguishability and authentication properties, which we describe below. In the proof for Multi-Stage security, we incrementally reduce the capabilities of the adversary. We split the security experiment into many game hops and in each hop, we replace keys or rule out certain attacks. We specify the amount by which the change

reduces the advantage of the now-restricted adversary. This continues until the adversary has no more chance of succeeding as all attacks have been ruled out. We also cover explicit authentication by making sure the adversary cannot maliciously accept in any explicitly authenticated stage. The security bound is obtained as the sum of the reductions in advantage in each of the game hops. The definition of the Multi-Stage experiment and the associated definitions of freshness and malicious acceptance are given in section 7.2.4.

Key indistinguishability

Secrecy of the key established in each stage is through indistinguishability from random, following Bellare–Rogaway [32]. This property is defined via an experiment with the syntax and adversary interaction as specified below. The model is set up with a hidden, uniformly random bit b. This b will be used in Test queries to decide if we will give the adversary the key that was computed in the execution of the protocol, or a truly random, unrelated key. The adversary's task is to decide if it was given the original key or the experiment's replacement; i.e., determine the value of b in the experiment. Any adversary that can guess b better than just flipping a coin can be used to construct reductions to the individual security properties of the components, such as collision resistance of hashes and indistinguishability properties of KEMs, that make up the protocol.

As noted above, the experiment imposes constraints on Reveal queries to prevent the adversary from revealing and testing the same key of some stage in a session or its partner. Depending on the intended forward secrecy goals of the stage key, some Corrupt queries may also be prohibited as described below to prevent the adversary from actively impersonating a party in an unauthenticated session and then testing that key. These restrictions rule out some trivial attacks that would exist if an adversary could call Reveal and Corrupt arbitrarily.

Forward secrecy and implicit authentication

Our multi-stage security definition incorporates three notions of forward secrecy [215] for stage keys:

- *Weak forward secrecy level 1* (wfs1): The stage key is indistinguishable against adversaries who were passive in the test stage (even if the adver-

sary obtains the peer's long-term secret key at any point in time—before or after the stage key was accepted). These keys have no authentication.

- *Weak forward secrecy level 2* (wfs2): The stage key is indistinguishable against adversaries who were passive in the test stage (wfs1) or if the adversary never corrupted the peer's long-term key. These keys are implicitly authenticated if the adversary did not corrupt the peer's long-term key before the stage key was accepted.

- *Forward secrecy* (fs): The stage key is indistinguishable against adversaries who were passive in the test stage (wfs1) or if the adversary did not corrupt the peer's long-term key before the stage accepted. These keys are implicitly authenticated.

These correspond to forward-secrecy levels 1, 3, and 5 in the Noise protocol framework [291].

Explicit authentication

We add an explicit authentication notion to the multi-stage model. When we reach explicit authentication of a stage key, that session has received explicit evidence (e.g., a MAC tag) that the intended peer is live. We also include retroactive authentication, in which an earlier stage is regarded as explicitly authenticated once a later stage accepts.

7.1.8 Security properties of KEMTLS

For KEMTLS, the properties of each stage key in a client instance are the same for unilaterally and mutually authenticated sessions as in both cases the client authenticates the server. They are as follows:

- Stages 1 and 2: these stages have wfs1 security from when they are accepted: they are indistinguishable against passive adversaries. These keys retroactively obtain fs security once stage 6 has accepted. No authentication at the time of acceptance, retroactive explicit authentication once stage 6 has accepted. The keys accepted by stages 1 and 2 are for internal use.

- Stages 3, 4, and 5: these stages have wfs2 security from when they are accepted: they are both indistinguishable against passive adversaries, and secure against adversaries that never corrupt the peer's long-term key. These keys obtain retroactive fs security once stage 6 has accepted. They get implicit authentication at the time of acceptance. Once stage 6 has accepted, they are retroactively explicitly authenticated. The keys accepted in stages 3 and 4 are for internal use; the stage 5 key is for external use.

- Stage 6: This stage immediately has fs security and explicit authentication from the time of acceptance. The key accepted by stage 6 is for external use.

In server instances of KEMTLS, the properties of each stage key are different for unilaterally authenticated and mutually authenticated sessions:

- Stages 1–4: These stages have wfs1 security when they are accepted: they are indistinguishable against passive adversaries. If the server is mutually authenticating the client, these keys (retroactively) obtain fs security once stage 5 accepts. The keys accepted by stages 1–4 are for internal use.

- Stage 5: This stage has fs security and is explicitly authenticated if the server is using mutual authentication. Otherwise, the stage key has wfs1 security. The key accepted in stage 5 is for external use.

- Stage 6: This stage has fs security if mutual authentication is used. It however is never explicitly authenticated as the server never receives a confirmation from the client during the handshake. If the server did not authenticate the client, the key has wfs1 security. The key accepted by stage 6 is for external use.

The following theorem says that KEMTLS is Multi-Stage-secure with respect to the forward secrecy, explicit authentication, and internal/external key-use properties as specified above. We bound the security on the assumptions that the hash function H is collision-resistant, HKDF is a pseudorandom function in either its "salt" or "input keying material" arguments, HMAC is a secure message-authentication code, KEM_s and, for mutual authentication, KEM_c are IND-CCA-secure KEMs, and KEM_e is an IND-1CCA-secure KEM (i.e., KEM_e is secure if a single decapsulation query is allowed).

Theorem 7.1. *Let $\mathcal{A}$ be an algorithm, and let n_s be the number of sessions and n_u be the number of parties. Then the advantage of $\mathcal{A}$ in breaking the multi-stage security of KEMTLS is upper-bounded by*

$$\frac{n_s^2}{2^{|\text{nonce}|}} + \epsilon_{\text{H}}^{\text{COLL}} + 6n_s \cdot \left(\begin{array}{l} n_s \left(\begin{array}{l} \epsilon_{\text{KEM}_e}^{\text{IND-1CCA}} + \epsilon_{\text{HKDF.Ext}}^{\text{PRF-sec}} \\ + 2\,\epsilon_{\text{HKDF.Ext}}^{\text{dual-PRF-sec}} + 3\,\epsilon_{\text{HKDF.Exp}}^{\text{PRF-sec}} \\ + 2\,\epsilon_{\text{HMAC}}^{\text{EUF-CMA}} \end{array} \right) \\ + n_u \left(\begin{array}{l} \epsilon_{\text{KEM}_s}^{\text{IND-CCA}} + \epsilon_{\text{KEM}_e}^{\text{IND-CCA}} \\ + \epsilon_{\text{HKDF.Ext}}^{\text{PRF-sec}} + \epsilon_{\text{HKDF.Ext}}^{\text{dual-PRF-sec}} \\ + 2\,\epsilon_{\text{HKDF.Exp}}^{\text{PRF-sec}} + 2\,\epsilon_{\text{HMAC}}^{\text{EUF-CMA}} \end{array} \right) \end{array} \right).$$

Above we use the shorthand notation $\epsilon_Y^X = \mathsf{Adv}_{Y,\mathcal{B}_i}^X$ for reductions $\mathcal{B}_i$ that are described in the proof; we provide the fully detailed bound in theorem 7.7. The proof of theorem 7.7 appears in section 7.2.4; we provide a sketch below.

7.1.9 Sketch of the proof

We start the proof with a sequence of game hops that assume that there are no reused nonces among the honest sessions and that there are no collisions in any hash function calls, which will be useful in later parts of the proof. The Multi-Stage security experiment is formulated to allow the adversary to make multiple Test queries. In the next game hop, we restrict the adversary to make a single Test query by guessing a to-be-tested session and the to-be-tested stage key using a hybrid argument [167]; this incurs a tightness loss $6n_s$ related to the number of sessions and stages.

The proof then splits into two cases: case A where the (now single) tested session has an honest contributive partner in the first stage; and case B where the tested session does not have an honest contributive partner in the first stage and the adversary does not corrupt the peer's long-term key before the tested stage has been accepted. These two cases effectively correspond to the forward-secrecy levels wfs1, wfs2. We finally show, by assumption on EUF-CMA security of HMAC, that both cases do not maliciously accept, giving us the final forward secrecy level fs.

Case A

In this case we assume that there *does* exist an honest contributive partner to at least the first stage of the tested session. Because we ruled out nonce

collisions, the honest client session and server session are unique, and thus we can speak of *the* client and server sessions. When the tested session is a client session, this means that the adversary did not interfere with the ephemeral key exchange in the ClientHello and ServerHello messages, so the ephemeral shared secret is unknown to the adversary assuming a secure KEM_e.

However, when the tested session is a server session, we only know that the adversary faithfully delivered the ClientHello message to the server; the adversary could have sent its own ServerHello message to the client. This is valid adversary behavior, and such an adversary would be able to compute the handshake encryption keys. In this case we need to correctly respond to the adversary, but in our simulation we do not have the KEM_e secret key, thus we need to make a single query to a decapsulation oracle. This is why we rely on IND-1CCA security, the single-decapsulation-query version of IND-CCA, rather than IND-CPA as might be expected for passive security, as we discussed in section 7.1.5. The use of IND-1CCA security in our proof of KEMTLS is analogous to the proofs of signed-Diffie–Hellman in TLS 1.2 [194, 220] and TLS 1.3 [127, 128] that use a single query to a PRF-ODH oracle. Huguenin-Dumittan and Vaudenay have additionally shown that for TLS 1.3, this PRF-ODH assumption can be replaced by a IND-1CCA-secure KEM, further highlighting their relation [182]. They construct such a KEM from DH, which allows them to rely on the computational Diffie–Hellman assumption in place of the ODH assumption.

All keys derived from the ephemeral key exchange are thus indistinguishable from random, and the remainder of case A is a sequence of game hops which, one-by-one, replace derived secrets and stage keys with random values, under the PRF-security or dual-PRF-security [29] assumption on HKDF. Dual-PRF-security arises since the TLS 1.3 and KEMTLS key schedules sometime use secrets in the "salt" argument of HKDF.Extract, rather than the "input keying material argument". At the end of case A, we have shown the required wfs1 security property for all stage keys, in both unilaterally and mutually authenticated client and server sessions.

Case B

Lacking an honest contributive partner in the first stage means the adversary was actively impersonating the peer to the tested session, and there is no partner at any stage of that session. As a result, any stages aiming for wfs1 security (which only covers passive adversaries) are out of scope. The tested session is

either a client session or a server session attempting mutual authentication. In case B we assume the peer's long-term key is not compromised before the tested session accepts the tested stage. This allows us to rely on the security of encapsulations against the peer's long-term public key.

Case B's sequence of game hops is as follows. First, we guess the identity of the peer that the adversary will attempt to impersonate in the tested session. Then we replace with a random value the shared secret ss_S that a client session encapsulates against the intended server's long-term static key pk_S. If KEM_s is IND-CCA-secure, only the intended server should be able to decapsulate and recover ss_S, and thus ss_S, and any key derived from it (following a sequence of game hops involving the security of HKDF), is an implicitly authenticated key unknown to the adversary. We similarly replace with a random value the shared secret ss_C that a mutually-authenticating server session encapsulates against its intended client's long-term static key pk_C, as well as any key derived from it. At the end of case B, we have shown the indistinguishability of client sessions' stage 3–6 keys, and server sessions' stage 5 and 6 keys under the conditions of case B, and hence their required wfs2 security properties.

Malicious acceptance

Case B allows the adversary to corrupt the intended peer's long-term key after the tested session accepts in stage 5 (if the session is a mutually-authenticating server) or stage 6 (if it is a client). This presents a problem for our reduction from IND-CCA security of KEM_s: how can it correctly answer the adversary's Corrupt query? Up until this bad query occurs, however, our IND-CCA reduction (and indeed, every reduction in case B) is fine, and all keys in the tested session can be shown as indistinguishable from random. This includes key fk_s that the server uses for the MAC authenticating the transcript in the `ServerFinished` message. If the client accepts `ServerFinished` in case B—without a partner to stage 6—then the adversary has forged an HMAC tag. We rely on the EUF-CMA security of HMAC and show that the reduction will never have to answer that Corrupt query. In the sessions that are using mutual authentication, we show the same for fk_c and `ClientFinished` in stage 5.

Contrapositively, assuming all the cryptographic primitives are secure, no stage accepts under the conditions of case B. This yields explicit server-to-client authentication of stage 6, and explicit client-to-server authentication of stage 5 (if mutual authentication is used). We also get retroactive authentication of all previous stages once the explicitly authenticated stage accepts since

their session identifiers are substrings of the stage's sid. Explicit authentication yields forward secrecy (fs) of the stage-6 key at the client, and the stage-5 and 6 keys of a mutually-authenticating server, at the time of acceptance. Retroactively, all stages of client sessions and stages 1–5 of mutually-authenticating server sessions also obtain fs. esh

7.2 Reductionist security model

Our approach adapts the security model and reductionist security analysis of the TLS 1.3 handshake by Dowling, Fischlin, Günther, and Stebila [127, 128] for KEMTLS. In contrast to the model and proof of KEMTLS as presented in [321] the proof below has been updated to the syntax used for the proof of KEMTLS-PDK [320] and been extended to cover mutual authentication.

7.2.1 Model syntax

Set $\mathcal{U}$ denotes the identities of honest participants in the system. Identities $S \in \mathcal{U}$ can be associated with a certified long-term KEM public key pk_S and corresponding private key sk_S. In the server-only authentication version of KEMTLS, participants that only act as clients do not have a long-term key.

Each participant can run multiple instances of the protocol, each of which is called a session. Sessions of a protocol are, for the purposes of modelling, uniquely identified by some administrative label, $\pi \in \mathcal{U} \times \mathbb{N}$, which is a pair (U, n), such that it identifies the n^{th} local session of identity U. In a multi-stage protocol, each session consists of multiple stages, run sequentially with shared state; each stage aims to establish a key. We denote the total number of stages in the protocol by $\mathsf{M} \in \mathbb{N}$.

For each session, each participant maintains the following collection of session-specific information. Many of the values are vectors of length M, with values for each stage.

- $\mathsf{id} \in \mathcal{U}$: the identity of the participant that *owns* this session.
- $\mathsf{pid} \in \mathcal{U} \cup \{*\}$: the identity of the *intended* communication partner. This partner may be unknown, which we indicate by the wildcard symbol '$*$'.
- $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$.

- $\mathsf{status} \in \{\perp, \mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}^{\mathsf{M}}$: the status of each stage key. We set $\mathsf{status}_i \leftarrow \mathsf{accepted}$ when stage i accepts the i^{th} stage key. When rejecting a key, $\mathsf{status}_i \leftarrow \mathsf{rejected}$ and the protocol does not continue. status is initially set to $(\mathsf{running}, \perp^{\times(\mathsf{M}-1)})$.

- $\mathsf{stage} \in \{0, 1, \ldots, \mathsf{M}\}$: the last accepted stage. Initially set to 0, it is incremented to i when status_i reaches accepted.

- $\mathsf{sid} \in (\{0,1\}^* \cup \{\perp\})^{\mathsf{M}}$: the session identifier bitstring in stage i, used for session pairing. Initially sid is set to $\perp$. It is updated to the specified value for each stage when reaching acceptance in that stage.

- $\mathsf{cid} \in (\{0,1\}^* \cup \{\perp\})^{\mathsf{M}}$: the contributive session identifier bitstring in stage i, used for early session pairing. Initially set to $\perp$ and updated as specified for each stage, until reaching acceptance in that stage.

- $\mathsf{key} \in (\mathcal{K} \cup \{\perp\})^{\mathsf{M}}$: the key established in stage i, which is set on acceptance of stage i. Initially set to $\perp$.

- $\mathsf{revealed} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$: records if the i^{th} stage key has been revealed by the adversary. Initially all set to false.

- $\mathsf{tested} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$: records if the i^{th} stage key has been tested by the adversary. Initially all set to false.

- $\mathsf{mutualauth} \in \{\mathsf{true}, \mathsf{false}\}$: Indicates if mutual authentication will be required in this session.

- $\mathsf{auth} \in \{1, \ldots, M, \infty\}^{M}$: Indicates at which stage a stage key is authenticated. Some keys may be considered authenticated right away (e.g., $\mathsf{auth}_i = i$), whereas other keys may only be considered authenticated retroactively (e.g. $\mathsf{auth}_i > i$), after some additional confirmation message has been received. Some may never be authenticated (e.g. $\mathsf{auth}_i = \infty$).

- $\mathsf{FS} \in \{\mathsf{wfs1}, \mathsf{wfs2}, \mathsf{fs}\}^{\mathsf{M}\times\mathsf{M}}$: for $j \geq i$, $\mathsf{FS}_{i,j}$ indicates the type of forward secrecy expected of stage key i, assuming stage j has accepted. The types of forward secrecy are further defined in definition 7.3.

- $\mathsf{use} \in \{\mathsf{internal}, \mathsf{external}\}^{\mathsf{M}}$: use_i indicates if a stage-i key is used internally in the key-exchange protocol. (Internally used keys require a bit of extra care when testing them, while externally used keys may only be used outside the handshake protocol.)

- $\mathsf{replay} \in \{\mathsf{replayable}, \mathsf{nonreplayable}\}^{M}$: Indicates whether a stage is expected to be unique against replay attacks or not. The adversary should not be able to distinguish a replayed accepted key and a random one.

For a session identified by π, we may write $\pi.X$ as shorthand to refer to that session's element X.

We define the *partner* of π at stage i to be any π' such that $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i \neq \perp$ and $\pi.\mathsf{role} \neq \pi'.\mathsf{role}$; a *contributive partner* is defined analogously using contributive identifiers cid. Correctness requires that, in an honest joint execution of the protocol, this predicate holds for all stages on acceptance.

7.2.2 Adversary interaction

Following DFGS [127, 128] our two claimed security properties, Match security and Multi-Stage security, take place within the same adversary interaction model. The adversary $\mathcal{A}$ is a probabilistic algorithm that controls the communication between all parties and thus can intercept, inject or drop any message. In this type of model, even two honest parties require $\mathcal{A}$ to facilitate communication to establish a session.

Some combinations of queries will be restricted; for example, allowing the adversary to both reveal and test a particular session key would allow the adversary to trivially win the test challenge in Multi-Stage security and thus does not model security appropriately. Such a session is marked unfresh.

The first two queries model honest protocol functionality:

- $\mathsf{NewSession}(U, V, \mathit{role}, \mathit{mutualauth})$: Creates a new session π in which we set owner $\pi.\mathsf{id} \leftarrow U$, intended peer $\pi.\mathsf{pid} \leftarrow V$, $\pi.\mathsf{role} \leftarrow \mathit{role}$ and $\pi.\mathsf{mutualauth} \leftarrow \mathit{mutualauth}$. V may be left unspecified ($V = *$).

- $\mathsf{Send}(\pi, m)$: Sends message m to session π. If π has not been created using NewSession, return $\perp$. Otherwise, Send runs the protocol on behalf of $\pi.\mathsf{id}$. It will record the updated state, and return both the response message and $\pi.\mathsf{status}_{\pi.\mathsf{stage}}$. To initiate a session (i.e., have π

compute ClientHello), the adversary may submit the special symbol $m = \mathsf{init}$ if $\pi.\mathsf{role} = \mathsf{initiator}$.

Special handling of acceptance. The adversary may not test any keys that have already been used. Because internal keys may be used immediately, Send will pause execution whenever any key is accepted, and immediately return accepted to the adversary. The adversary may choose to test the session (or do other operations in other sessions). Whenever the adversary decides to continue this session, they may submit $\mathsf{Send}(\pi, \mathsf{continue})$. This will continue the protocol as specified. On this call, we set $\pi.\mathsf{status}_{\pi.\mathsf{stage}+1} \leftarrow \mathsf{running}$, except if the current stage is the last one, and return the next protocol message and $\pi.\mathsf{status}_{\pi.\mathsf{stage}}$.

Whenever stage i accepts, if there exists a partner π' of π at stage i with $\pi'.\mathsf{tested}_i = \mathsf{true}$, we set $\pi.\mathsf{tested}_i \leftarrow \mathsf{true}$. If the stage is an internal-use stage ($\pi.\mathsf{use}_i = \mathsf{internal}$), we also set $\pi.\mathsf{key}_i \leftarrow \pi'.\mathsf{key}_i$ to ensure session keys are used consistently. This ensures the adversary cannot test keys twice.

The next two queries model the adversary's ability to compromise participants and learn some secret information:

- $\mathsf{Reveal}(\pi, i)$: Reveals the stage key $\pi.\mathsf{key}_i$ to the adversary. We record that the key has been revealed by setting $\pi.\mathsf{revealed}_i \leftarrow \mathsf{true}$. If the session does not exist or the stage has not accepted, returns $\perp$.

- $\mathsf{Corrupt}(U)$: Provides the adversary with the long-term secret key sk_U of identity U. We record the time of U's corruption.

The final query models the challenge to the adversary of breaking a key that was established by honest parties:

- $\mathsf{Test}(\pi, i)$: Challenges the adversary on the indistinguishability of stage key $\pi.\mathsf{key}_i$ as follows. If the stage has not been accepted ($\pi.\mathsf{status}_i \neq \mathsf{accepted}$), or the key has already been tested ($\pi.\mathsf{tested}_i = \mathsf{true}$), or there exists a partner π' to π at stage i such that $\pi'.\mathsf{tested}_i = \mathsf{true}$, return $\perp$.

  If the stage-i key use is internal ($\pi.\mathsf{use}_i = \mathsf{internal}$), we require a partner π' to π to exist. To ensure that any partnered session has also not yet used the key, we require $\pi'.\mathsf{status}_{i+1} = \perp$. If there is no partner or this does not hold, return $\perp$.

We now set $\pi.\mathsf{tested}_i \leftarrow \mathsf{true}$.

The Test oracle has a uniformly random bit b, which is fixed throughout the game. If test bit $b = 0$, we sample a uniformly random key $K \leftarrow\!\!\$\ \mathcal{K}$. If test bit $b = 1$, we set $K \leftarrow \pi.\mathsf{key}_i$. To make sure that the selected K is consistent with any later internally used keys, we set $\pi.\mathsf{key}_i \leftarrow K$ if $\pi.\mathsf{use}_i = \mathsf{internal}$.

We then ensure consistency with partnered sessions: for sessions π' that are partner to π at stage i for which $\pi'.\mathsf{status}_i = \mathsf{accepted}$, set $\pi'.\mathsf{tested}_i \leftarrow \mathsf{true}$, and, if $\pi.\mathsf{use}_i = \mathsf{internal}$, also set $\pi'.\mathsf{key}_i \leftarrow K$.

We finally return K to the adversary.

7.2.3 Match security

Match security models sound behavior of session matching: it ensures that, for honest sessions, session identifiers effectively match partnered sessions. (Separately treating the session matching property of AKE protocols is the approach of [82, 83, 127, 128, 143].)

Definition 7.2 (Match security)**.** Let KE be an M-stage key-exchange protocol. Probabilistic adversary $\mathcal{A}$ interacts with KE via the queries defined in section 7.2.2. $\mathcal{A}$ tries to win the following game $G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}}$:

Setup The challenger generates long-term keypairs $(\mathsf{pk}_U, \mathsf{sk}_U)$ for each participant $U \in \mathcal{U}$. All keys are provided to $\mathcal{A}$.

Query The adversary has access to queries NewSession, Send, Reveal, Corrupt and Test. These queries are defined in section 7.2.2.

Stop At some point, the adversary stops with no output.

Let π, π' be distinct, partnered sessions for which $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i \neq \perp$ at some stage $i \in \{1, \ldots, \mathsf{M}\}$ The adversary $\mathcal{A}$ wins the Match security game, denoted

$$G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} = 1,$$

if it can falsify any of the following conditions:

1. At every stage $j \leq i$, $\pi.\mathsf{key}_j = \pi'.\mathsf{key}_j$: both sessions agree on the same key at every stage up to and including stage i.

2. $\pi.\mathsf{role} \neq \pi'.\mathsf{role}$, except if $\pi.\mathsf{role} = \mathsf{responder}$ and $\pi.\mathsf{replay}_i = \mathsf{replayable}$: both sessions must have opposite roles, except if they are both responders in a replayable stage.

3. $\pi.\mathsf{cid}_i = \pi'.\mathsf{cid}_i$: both sessions have the same contributive identifier.

4. At every stage j, if $\pi.\mathsf{status}_j = \mathsf{accepted}$ and $\pi.\mathsf{stage} \geq \pi.\mathsf{auth}_j$, then we require $\pi.\mathsf{mutualauth} = \pi'.\mathsf{mutualauth}$: both sessions agree on the authentication used.

5. At every stage j, if $\pi.\mathsf{status}_j = \mathsf{accepted}$ and $\pi.\mathsf{stage} \geq \pi.\mathsf{auth}_j$, then $\pi.\mathsf{pid} = \pi'.\mathsf{id}$: sessions are partnered with the intended (explicitly authenticated) participant.

6. If $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_j$, then $i = j$: session labels are different in different stages.

7. If $\pi.\mathsf{replay}_i = \mathsf{nonreplayable}$, for any three sessions π, π', π'', if $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i = \pi''.\mathsf{sid}_i \neq \perp$, then $\pi = \pi'$, or $\pi = \pi''$, or $\pi' = \pi''$: at most two sessions share the same session label.

We say that protocol KE is Match-secure if for all $\mathcal{A}$ that run in polynomial time,

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} = \left| \Pr\left[G^{\mathsf{Match}}_{\mathsf{KE},\mathcal{A}} \Rightarrow 1\right] - \frac{1}{2} \right|$$

is negligible in the security parameter.

7.2.4 Multi-Stage security

The Multi-Stage experiment was introduced by Fischlin and Günther [143] and was also used by Dowling, Fischlin, Günther, and Stebila for TLS 1.3 [127, 128]. In this original formulation, secrecy of each stage key is defined as being indistinguishable from a random key, Bellare–Rogaway-style [32]. Our formulation of Multi-Stage is extended to also model explicit authentication.

We first define the terms *fresh* and *maliciously accept*.

Definition 7.3 (Freshness)**.** Stage i of a session π is said to be *fresh* if conditions 1, 2, and 3 and at least one of 4, 5, or 6 hold:

1. the stage key was not revealed ($\pi.\mathsf{revealed}_i = \mathsf{false}$);

2. the stage key of the partner session at stage i, if the partner exists, has not been revealed (for all i, π' such that $\pi.\mathsf{sid}_i = \pi'.\mathsf{sid}_i$, we have that $\pi'.\mathsf{revealed}_i = \mathsf{false}$);

3. If the stage i is replayable, the partner has never been corrupted (if $\pi.\mathsf{replay}_i = \mathsf{replayable}$, then $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was never called);

4. (weak forward secrecy 1) there exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{wfs1}$ and $\pi.\mathsf{status}_j = \mathsf{accepted}$, and there exists a contributive partner at stage i;

5. (weak forward secrecy 2) there exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{wfs2}$ and $\pi.\mathsf{status}_j = \mathsf{accepted}$, and either (a) there exists a contributive partner at stage i or (b) $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was never called;

6. (forward secrecy) there exists $j \geq i$ such that $\pi.\mathsf{FS}_{i,j} = \mathsf{fs}$ and $\pi.\mathsf{status}_j = \mathsf{accepted}$, and either (a) there exists a contributive partner at stage i or (b) $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called before stage j of session π accepted.

Definition 7.4 (Malicious acceptance)**.** Stage i of session π is said to have *maliciously accepted* if all the following conditions hold:

1. $\pi.\mathsf{status}_{\pi.\mathsf{auth}_i} = \mathsf{accepted}$;

2. if stage i is not replayable, there does not exist a unique partner of π at stage i; and

3. $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called before stage $\pi.\mathsf{auth}_i$ of session π accepted.

Now we can define our version of the Multi-Stage security experiment.

Definition 7.5 (Multi-Stage security)**.** Let KE be an M-stage key-exchange protocol, and let $\mathcal{A}$ be a probabilistic adversary interacting with KE via the queries defined in section 7.2.2. The adversary tries to win the game $G^{\text{Multi-Stage}}_{\mathsf{KE},\mathcal{A}}$:

Setup The challenger generates all long-term keys $(\mathsf{pk}_U, \mathsf{sk}_U)$ for all identities $U \in \mathcal{U}$ and picks the uniformly random test bit b (used in the Test queries). The public keys pk_U are provided to $\mathcal{A}$.

Query The adversary has access to queries NewSession, Send, Reveal, Corrupt, and Test. These are defined in section 7.2.2.

Stop At some point, $\mathcal{A}$ stops and outputs their guess b' of b.

Finalize The adversary wins the game if either of the following conditions holds:

1. all tested stages are fresh (definition 7.3), and $b' = b$; or
2. there exists a stage that has maliciously accepted;

in which case the experiment $G^{\text{Multi-Stage}}_{\text{KE},\mathcal{A}}$ outputs 1. Otherwise, the adversary has lost the game, in which case the experiment $G^{\text{Multi-Stage}}_{\text{KE},\mathcal{A}}$ outputs a uniform bit.

The Multi-Stage-advantage of $\mathcal{A}$ is defined as:

$$\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KE},\mathcal{A}} = \left| \Pr\left[G^{\text{Multi-Stage}}_{\text{KE},\mathcal{A}} \Rightarrow 1 \right] - \frac{1}{2} \right|.$$

7.3 Specifics of KEMTLS in the model

For the proofs in the subsequent subsections, KEMTLS is as specified in figures 5.2 and 5.3, with $\mathsf{M} = 6$ stages. The session identifiers sid_i and contributive identifiers cid_i for each stage are defined as follows. Whenever a stage is accepted, its session identifier is set to consist of a label and all (unencrypted) handshake messages up to that point. Note that for mutually authenticated handshakes, the `CertificateRequest`, `ClientCertificate`, and `ServerKemCiphertext` messages are included in these message ranges:

$$\begin{aligned}
\mathsf{sid}_1 &= (\text{“CHTS”}, \quad \texttt{ClientHello} \ldots \texttt{ServerHello}),\\
\mathsf{sid}_2 &= (\text{“SHTS”}, \quad \texttt{ClientHello} \ldots \texttt{ServerHello}),\\
\mathsf{sid}_3 &= (\text{“CAHTS”}, \texttt{ClientHello} \ldots \texttt{ClientKemCiphertext}),\\
\mathsf{sid}_4 &= (\text{“SAHTS”}, \texttt{ClientHello} \ldots \texttt{ClientKemCiphertext}),\\
\mathsf{sid}_5 &= (\text{“CATS”}, \quad \texttt{ClientHello} \ldots \texttt{ClientFinished}),\\
\mathsf{sid}_6 &= (\text{“SATS”}, \quad \texttt{ClientHello} \ldots \texttt{ServerFinished}).
\end{aligned}$$

We use the contributive identifiers cid_i to match up participant sessions in the proof before both sessions have accepted the first session identifier. This means we need special care for the contributive identifiers before the first stage. In stage $i = 1$, the client and server set, upon sending (client) or receiving (server) the `ClientHello` message, $\mathsf{cid}_1 = (\text{“CHTS”}, \texttt{ClientHello})$.

When they next send (server) or receive (client) the ServerHello response, they update this to $\mathsf{cid}_1 = \mathsf{sid}_1$. All other contributive identifiers are set to $\mathsf{cid}_i = \mathsf{sid}_i$ whenever sid_i is set. Finally, if any client or server receives an unexpected message, they terminate.

Every client and server session of unilaterally or mutually authenticated KEMTLS uses the following properties for the replayability and the usage of the six stages:

$$\begin{aligned} \mathsf{replay} &= \left(\mathsf{nonreplayable}^{\times 6}\right), \\ \mathsf{use} &= \left(\mathsf{internal}^{\times 4}, \mathsf{external}^{\times 2}\right). \end{aligned}$$

All KEMTLS client sessions authenticate the server. This means both unilaterally authenticated and mutually authenticated client sessions of KEMTLS have the same properties for when sessions reach explicit authentication and the forward secrecy levels per stage.

The security properties for unilaterally and mutually authenticated KEMTLS client sessions are:

$$\mathsf{auth} = \left(6^{\times 6}\right),$$
$$\mathsf{FS} = \begin{pmatrix} \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} \\ & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} \\ & & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{fs} \\ & & & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{fs} \\ & & & & \mathsf{wfs2} & \mathsf{fs} \\ & & & & & \mathsf{fs} \end{pmatrix}.$$

Every server session of unilaterally authenticated KEMTLS uses:

$$\mathsf{auth} = \left(\infty^{\times 6}\right),$$
$$\mathsf{FS} = \begin{pmatrix} \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & & & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & & & & \mathsf{wfs1} & \mathsf{wfs1} \\ & & & & & \mathsf{wfs1} \end{pmatrix}.$$

As the client is not at all authenticated, we can reach at most wfs1 security:

we are only secure against passive adversaries. We also never reach explicit authentication in any stage.

For mutually authenticated server sessions of KEMTLS, the client *is* authenticated. This gives us the following properties for explicit authentication and forward secrecy in server sessions:

$$\mathsf{auth} = (5, 5, 5, 5, 5, \infty)\,,$$
$$\mathsf{FS}_{i,j} = \begin{pmatrix} \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} & \mathsf{fs} \\ & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} & \mathsf{fs} \\ & & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} & \mathsf{fs} \\ & & & \mathsf{wfs1} & \mathsf{fs} & \mathsf{fs} \\ & & & & \mathsf{fs} & \mathsf{fs} \\ & & & & & \mathsf{fs} \end{pmatrix}.$$

In mutually authenticated KEMTLS, the server's value for auth_6 is still ∞. Server sessions never receive another message during the handshake protocol after accepting the stage-6 key and thus cannot be sure `ServerFinished` arrived at the client.

7.4 Proving the security of KEMTLS

We will now show that both unilaterally and mutually authenticated KEMTLS have the properties that we claimed. We will first show that KEMTLS is Match-secure, after which we will show that it is Multi-Stage secure.

Theorem 7.6. *KEMTLS is* Match*-secure. In particular, any efficient adversary $\mathcal{A}$ has advantage*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathit{KEMTLS},\mathcal{A}} \leq n_s(\delta_e + \delta_s + \delta_c) + n_s^2/2^{|\mathsf{nonce}|},$$

where n_s is the number of sessions, |nonce| *is the length of the nonces r_c and r_s in bits, and the ephemeral and long-term KEM algorithms are assumed to be δ_e-, δ_s- and δ_c-correct, respectively. If π.*mutualauth = false*, $\delta_c = 0$.*

Proof. We need to show each property of Match security (definition 7.2) holds:

1. The session identifiers are defined to contain all handshake messages. KEM messages and hashes of those messages are only inputs into the

key schedule. In stages 1 and 2, the input to the agreed keys is the ephemeral KEM_e shared secret and the messages up to `ServerHello`. For stages 3 and 4, the input to the agreed keys are the previous keys, messages up to `ClientKemCiphertext`, and the static KEM_s shared secret. For the final stages 5 and 6, the input to the keys is the previous keys and the messages up to `ClientFinished` and `ServerFinished` respectively. It is easy to confirm that this is all included in the session identifiers. This means that the parties use the same inputs for their computations. The only way they can arrive at different keys is if any of their computations are not perfectly correct. The ephemeral and long-term KEMs have some small probability of failure (definition 2.14), δ_e, and δ_s and δ_c, respectively, in each of the n_s sessions. This gives us a failure probability of $n_s\,(\delta_e + \delta_s + \delta_c)$. If $\pi.\mathsf{mutualauth} = \mathsf{false}$, there is no chance of decapsulation failure in client authentication, so $\delta_c = 0$.

2. No KEMTLS session that is either initiator or responder will ever accept a wrong-role incoming message, so any pair of two sessions must have both an initiator and a responder. We will later show that at most two sessions have the same sid, implying that this pairing will be unique and thus opposite. There are no replayable stages in KEMTLS.

3. By definition, cid_i is final and equal to sid_i whenever stage i is accepted.

4. The presence of `ClientCertificate` in the transcript decides if the value of either session's $\mathsf{mutualauth} = \mathsf{true}$ before either session accepts the stage-5 key, which is when explicit authentication is first reached in mutually authenticated KEMTLS.

5. The partnered sessions only have to agree once they reach a retroactively authenticated stage. The identity of the server is learned through the `ServerCertificate` sent by the responder. Because Match security only concerns honest sessions, the `ServerCertificate` received by the session that has $\pi.\mathsf{role} = \mathsf{initiator}$ will set the correct pid. The identity of the client is learned through the `ClientCertificate` message sent by the initiator session. Similarly, the `ClientCertificate` received by the responder will set the correct pid.

6. Every stage's session identifier is defined to have a unique label, thus there can be no confusion across distinct stages.

7. The session identifiers include the random nonce and KEM public keys and ciphertexts. For three sessions to have the same identifier, we would need to have a collision of these values picked by honest servers and clients. Without making assumptions on the KEM schemes, we can rely on the distinctness of nonces under the birthday bound on n_s the number of sessions: the probability of failing in n_s sessions is less than $n_s^2/2^{|\text{nonce}|}$, which is negligible in the bit-length of the nonce.

□

Now we can prove the Multi-Stage security of KEMTLS.

Theorem 7.7. *Let $\mathcal{A}$ be an algorithm, and let n_s be the number of sessions and n_u be the number of parties. There exist algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_{16}$, described in the proof, such that*

$$\begin{aligned}\mathsf{Adv}^{\text{Multi-Stage}}_{\mathit{KEMTLS},\mathcal{A}} \leq \frac{n_s^2}{2^{|\text{nonce}|}} + \mathsf{Adv}^{\text{COLL}}_{\text{H},\mathcal{B}_1} \\ + 6n_s \left(\begin{array}{ll} n_s \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\mathcal{B}_2} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3} \\ + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_4} & + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_5} \\ + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_6} & + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_7} \\ + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_8} & + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_9} \end{array} \right) \\ + n_u \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_{10}} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{11}} \\ + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{12}} & + \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_c,\mathcal{B}_{13}} \\ + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{14}} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{15}} \\ + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{16}} \end{array} \right) \end{array} \right).\end{aligned}$$

Proof. The proof follows the basic structure of the proof of DFGS [127, 128] for the TLS 1.3 signed-Diffie–Hellman full handshake. It proceeds by a sequence of games, in each of which we will reduce the advantage of the adversary $\mathcal{A}$ until it has no chance of winning anymore. The security bound is obtained by adding up all the reductions in advantage.

We assume that all tested sessions remain fresh throughout the experiment, as otherwise the adversary loses the indistinguishability game.

Game 0: Multi-Stage game
We define G_0 to be the original Multi-Stage game:

$$\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS},\mathcal{A}} = \mathsf{Adv}^{G_0}_{\mathcal{A}}.$$

Game 1: nonce collisions
We abort if any honest session uses the same nonce r_c or r_s as any other session. Given that there are n_s sessions each using uniformly random nonces of size $|\mathsf{nonce}| = 256$, the chance of a repeat is given by a birthday bound:

$$\mathsf{Adv}^{G_0}_{\mathcal{A}} \leq \mathsf{Adv}^{G_1}_{\mathcal{A}} + \frac{n_s^2}{2^{|\mathsf{nonce}|}}.$$

Game 2: collision resistance
In this game, the challenger will abort if any two honest sessions compute the same hash for different inputs of hash function H. If this happens, it induces a reduction $\mathcal{B}_1$ that can break the collision resistance of H. If a collision occurs, $\mathcal{B}_1$ outputs the two distinct input values. Thus:

$$\mathsf{Adv}^{G_1}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_2}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}.$$

Game 3: single Test query
We now restrict $\mathcal{A}$ to only make a single Test query. This reduces its advantage by at most $1/6n_s$ for the six stages, based on a hybrid argument by Günther [167]. Any single-query adversary $\mathcal{A}_1$ can emulate a multi-query adversary $\mathcal{A}$ by guessing a to-be-tested session in advance. For any other Test queries $\mathcal{A}$ may submit, $\mathcal{A}_1$ can substitute by Reveal queries. $\mathcal{A}_1$ will need to know how sessions are partnered. Early partnering is decided by public information $(\mathsf{sid}_1, \mathsf{sid}_2)$, but later sids are encrypted. However, $\mathcal{A}_1$ can just reveal the handshake traffic keys to decrypt the subsequent information.

We get the following advantage by letting transformed adversary $\mathcal{A}_1$ guess the right session and stage:

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq 6n_s \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}_1}.$$

With this restriction from $\mathcal{A}$ to $\mathcal{A}_1$, we can now refer to *the* session π tested at stage i, and assume that we know the tested session π at the outset.

Case distinction

We now consider two separate cases of game 3. These cases, respectively, roughly correspond to the specified properties of weak forward secrecy: wfs1 and wfs2. By rejecting malicious acceptance, we finally show fs.

The two cases are:

A. (denoted G_A) The tested session π has a contributive partner in stage 1. Formally, there exists $\pi' \neq \pi$ where $\pi'.\mathsf{cid}_1 = \pi.\mathsf{cid}_1$.

B. (denoted G_B) The tested session π does not have a contributive partner in stage 1, and there was no $\mathsf{Corrupt}(\pi.\mathsf{pid})$ query *before* stage i accepted.

As by rejecting malicious acceptance, no cases exist that call $\mathsf{Corrupt}(\pi.\mathsf{pid})$ *after* stage i accepted, these cases are exhaustive.

We will consider the advantage of the adversary separately for these two cases:

$$\begin{aligned}\mathsf{Adv}^{G_3}_{\mathcal{A}_1} &\leq \max\left\{\mathsf{Adv}^{G_A}_{\mathcal{A}_1}, \mathsf{Adv}^{G_B}_{\mathcal{A}_1}\right\} \\ &\leq \mathsf{Adv}^{G_A}_{\mathcal{A}_1} + \mathsf{Adv}^{G_B}_{\mathcal{A}_1}.\end{aligned}$$

Case A: stage 1 contributive partner exists

If the tested session π is a client (initiator) session, then $\pi.\mathsf{cid}_1 = \pi.\mathsf{sid}_1$, and a partner session at stage 1 also exists. Since sid_1 includes both the client and server nonces r_c and r_s via the ClientHello and ServerHello messages, and by game 1 no honest sessions repeat nonces, the contributive partner is unique.

If the tested session π is a server (responder) session, then it is possible that, while the *contributive* partner session at stage 1 exists, the partner session at stage 1 may not exist. However, since cid_1 includes the client nonce (which by game 1 are unique) and contributive partnering includes roles, there is no other honest client session that is a contributive partner at stage 1.

So we can talk about *the* tested session and its unique contributive partner at stage 1. Let π' be the unique honest contributive partner of π at stage 1. In the following, we let π_c denote the one of $\{\pi, \pi'\}$ which is a client (initiator) session, and we let π_s denote the one which is a server (responder) session.

Game A1: guess contributive partner session

In this game, the challenger tries to guess the $\pi' \neq \pi$ that is the honest contributive partner to π at stage 1.

This reduces the advantage of $\mathcal{A}_1$ by a factor of the number of sessions n_s:

$$\mathsf{Adv}^{G_A}_{\mathcal{A}_1} \leq n_s \cdot \mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1}.$$

From this point on, we will make a series of replacements in the tested session π, and its (unique) contributive partner session π' which we know by game A1.

Game A2: ephemeral KEM

In this game, in session π_s, we replace the ephemeral secret ss_e with a uniformly random $\widetilde{\mathsf{ss}_e}$. If π_c received the same ct_e that π_s sent, then we also replace its ss_e with the same $\widetilde{\mathsf{ss}_e}$. All values derived from ss_e in π_s (and π_c, if ss_e was replaced in it) use the randomized value $\widetilde{\mathsf{ss}_e}$.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_2$ against the IND-1CCA security of $\mathsf{KEM_e}$ as follows. $\mathcal{B}_2$ obtains the IND-1CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and challenge shared secret $\mathsf{ss}^\star$. In π_c, it uses $\mathsf{pk}^\star$ in the ClientHello. In π_s, it uses $\mathsf{ct}^\star$ in the ServerHello reply, and $\mathsf{ss}^\star$ as π_s's shared secret ss_e. If $\mathcal{A}_1$ delivers $\mathsf{ct}^\star$ to π_c, then $\mathcal{B}_2$ uses $\mathsf{ss}^\star$ as π_c's shared secret ss_e as well. If $\mathcal{A}_1$ delivers some other $\mathsf{ct}' \neq \mathsf{ct}^\star$ to π_c, then $\mathcal{B}_2$ makes a single query to its IND-1CCA decapsulation oracle with ct' to obtain the required shared secret. All other sessions and parties are simulated.

$\mathcal{A}_1$ eventually terminates its guess of $b = 0$ or $b = 1$. If $\mathsf{ss}^\star$ was the real shared secret, then $\mathcal{B}_2$ has exactly simulated G_{A1} to $\mathcal{A}_1$; if $\mathsf{ss}^\star$ was a random value, then $\mathcal{B}_2$ has exactly simulated G_{A2} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{KEM_e},\mathcal{B}_2}.$$

Game A3: replacing HS

In this game, in session π_s, we replace the handshake secret HS with a uniformly random $\widetilde{\mathrm{HS}}$. If π_c received the same ct_e that π_s sent, then we also replace its HS with the same $\widetilde{\mathrm{HS}}$. All values derived from HS in π_s (and π_c, if HS was replaced in it) use the randomized value $\widetilde{\mathrm{HS}}$.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_3$ against the PRF security of HKDF.Extract in its second argument as follows. When $\mathcal{B}_3$ needs to compute HS in π_s (or π_c, if π_c received the

$G^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3}$

1: $b' \leftarrow\$ \mathcal{B}_3^{\mathcal{O}}()$
2: **return** $[\![b' = b]\!]$

PRF-Oracle $\mathcal{O}(l)$

1: **if** $b = 0$ **then**
2: **return** HKDF.Extract(l, ss_e)
3: **else**
4: **return** R(l)
5: **end if**

$\mathcal{B}_3$'s HKDF.Extract$'(k, l)$

1: **if** $k = \text{ss}_e$ **then**
2: **return** $\mathcal{O}(l)$
3: **else**
4: **return** HKDF.Extract(l, k)
5: **end if**

Distinguisher $\mathcal{B}_3^{\mathcal{O}}()$

1: $\mathcal{B}_3$ simulates KEMTLS using HKDF.Extract$'$ to compute HS.
2: $b' \leftarrow \mathcal{A}_1^{\text{KEMTLS}}()$
3: **return** b'

Figure 7.1: $\mathcal{B}_3$ in game $A3$ uses $\mathcal{A}_1$, which interacts with KEMTLS as described in section 7.2.2, to win the PRF security experiment for HKDF.Extract. R is a function which returns uniformly random values.

same ct_e that π_s sent), it queries its HKDF.Extract challenge oracle (keyed with ss_e) on dES and uses the response as HS. If the response was the real output, then $\mathcal{B}_3$ has exactly simulated G_{A2} to $\mathcal{A}_1$; if the response was a random value, then $\mathcal{B}_3$ has exactly simulated G_{A3} to $\mathcal{A}_1$. Thus:

$$\text{Adv}^{G_{A2}}_{\mathcal{A}_1} \leq \text{Adv}^{G_{A3}}_{\mathcal{A}_1} + \text{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3}.$$

To further illustrate how we construct this reduction to the PRF advantage, we give a pseudocode representation of the PRF experiment and how $\mathcal{B}_3$ uses $\mathcal{A}_1$ to its advantage in figure 7.1.

Game A4: replacing CHTS, SHTS, and dHS

In this game, in session π_s, we replace the values CHTS, SHTS, and dHS with uniformly random values. If π_c received the same ct_e that π_s sent (i.e., has the same value for HS), then we replace its dHS with the same replacement as in π_s. Furthermore, if π_c received the same ct_e that π_s sent and the same ServerHello (i.e., if π_c is a partner to π_s at stage 1 and 2), then we also replace its CHTS and SHTS with the same replacements as in π_s. If π_c received the

same ct_e that π_s sent but not the same ServerHello, then we replace its CHTS and SHTS with independent uniformly random values. All values derived from dHS in π_s (and π_c, if dHS was replaced in it) use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_4$ against the PRF security of HKDF.Expand as follows. When $\mathcal{B}_4$ needs to compute CHTS, SHTS, or dHS in π_s (or π_c, if π_c received the same ct_e that π_s sent), it queries its HKDF.Expand challenge oracle (keyed with HS) on the corresponding labels and transcripts, and uses the responses. If the responses were the real output, then $\mathcal{B}_4$ has exactly simulated G_{A3} to $\mathcal{A}_1$; if the responses were random values, then $\mathcal{B}_4$ has exactly simulated G_{A4} to $\mathcal{A}_1$. Note that if π_c did receive the same ct_e as was sent by π_s, but other parts of the ServerHello message were altered such that π_c and π_s are not partners at stage 1, the adversary may be permitted to query $\mathsf{Reveal}(\pi_c, 1)$; but since the transcript in π_c and π_s is now different, the label input to the HKDF.Expand oracle for CHTS and SHTS is different, so the simulation in $\mathcal{B}_4$ remains good. Thus:

$$\mathsf{Adv}^{G_{A3}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_4}.$$

The stage-1 and stage-2 keys CHTS and SHTS are now uniformly random strings independent of everything else in the game. Thus, the stage-1 and stage-2 keys have been shown to have wfs1 security.

Game A5: replacing AHS

In this game, in session π_s, we replace the secret AHS with a uniformly random $\widetilde{\mathsf{AHS}}$. If π_c shares the values for dHS (i.e., received the same ct_e as was sent by π_s) and ct_S with π_s, we replace its AHS with the same replacement as in π_s. Otherwise, if π_c only shares the same value for ct_S with π_s, we replace its AHS with an independent uniformly random value. All values derived from AHS in π_s (and π_c, if AHS was replaced in it) use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_5$ against the PRF security of HKDF.Extract in its first argument (which we view as the "dual-PRF security" of HKDF.Extract) as follows. We rely on the dual-PRF security of HKDF.Extract in its salt argument instead of using ss_S as the key in the PRF experiment, as the adversary is allowed to Corrupt the sessions involved in the games in case A. When $\mathcal{B}_5$ needs to compute AHS in π_s (or π_c, if π_c has the same value for dHS), it queries its HKDF.Extract challenge oracle (keyed with dHS) on the value of ss_S in the session and uses

the response as AHS. If the response was the real output, then $\mathcal{B}_5$ has exactly simulated G_{A4} to $\mathcal{A}_1$; if the response was a random value, then $\mathcal{B}_5$ has exactly simulated G_{A5} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_5}.$$

Game A6: replacing CAHTS, SAHTS, and dAHS

In this game, in session π_s, we replace the values CAHTS, SAHTS, and dAHS with uniformly random values. If π_c shares the value of AHS with π_s (i.e., has the same ct_e and ct_S) and is a partner to π_s at stage 3 and 4, then we replace its CAHTS, SAHTS, and dAHS with the same replacements as in π_s. If π_c shares the value of AHS with π_s but is not a partner to π_s at stage 3 and 4, then we replace its CAHTS, SAHTS with independent uniformly random values, but we replace dAHS with the same replacement as in π_s. All values derived from dAHS in π_s (and π_c, if dAHS was replaced in it) use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_6$ against the PRF security of HKDF.Expand as follows. When $\mathcal{B}_6$ needs to compute CAHTS, SAHTS, or dAHS in π_s (or π_c, if π_c has the same AHS), it queries its HKDF.Expand challenge oracle (keyed with AHS) on the corresponding labels and transcripts, and uses the responses. If the responses were the real output, then $\mathcal{B}_6$ has exactly simulated G_{A5} to $\mathcal{A}_1$; if the responses were random values, then $\mathcal{B}_6$ has exactly simulated G_{A6} to $\mathcal{A}_1$. Note that if π_c did have the same AHS as π_s, but parts of the transcript were altered such that π_c and π_s are not partners at stage 3 and 4, the adversary may be permitted to query $\mathsf{Reveal}(\pi_c, 3)$ or $\mathsf{Reveal}(\pi_c, 4)$; but since the transcript in π_c and π_s is now different, the label input to the HKDF.Expand oracle for CAHTS and SAHTS is different, so the simulation in $\mathcal{B}_6$ remains good. Thus:

$$\mathsf{Adv}^{G_{A5}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A6}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_6}.$$

The stage-3 and stage-4 keys CAHTS and SAHTS are now uniformly random strings independent of everything else in the game. Thus, the stage-3 and stage-4 keys have been shown to have wfs1 security.

Game A7: replacing MS

In this game, in session π_s, we replace the main secret MS with a uniformly random $\widetilde{\mathrm{MS}}$. If π_c has the same value for dAHS (i.e., has the same values for

ct_e and ct_S), and, if π_c.mutualauth = true, ct_C as π_s, then we replace its MS with the same replacement as in π_s. Otherwise, if π_c.mutualauth = true and π_c only shares the same dAHS with π_s but ct_C is different, we replace MS in π_c with an independent uniformly random value. All values derived from MS in π_s use these newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_7$ against the PRF security of HKDF.Extract in its first argument as follows (which we view as "dual-PRF security" of HKDF.Extract). When $\mathcal{B}_7$ needs to compute MS in π_s (or π_c, if π_c shares the value for dAHS with π_s), it queries its HKDF.Extract challenge oracle (keyed with dAHS) on ss_C or 0 if π.mutualauth = false. It uses the response as MS. If the response was the real output, then $\mathcal{B}_7$ has exactly simulated G_{A6} to $\mathcal{A}_1$; if the response was a random value, then $\mathcal{B}_7$ has exactly simulated G_{A7} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{A6}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A7}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_7}.$$

Game A8: replacing CATS, fk_c, fk_s, and SATS

In this game, in session π_s, we replace the values CATS, fk_c, fk_s, and SATS with uniformly random values. If π_c is a partner of π_s at stage 5, then we replace its CATS, fk_c, and fk_s with the same replacements as in π_s. If π_c is not a partner of π_s at stage 5 but shares the same value for MS with π_s, we replace π_c's CATS with an independent uniformly random value, but still replace fk_c and fk_s with the same replacements as in π_s. If π_c is a partner of π_s at stage 6, then we replace its SATS with the same replacement as in π_s. If π_c is not a partner of π_s at stage 6 but shares the same value for MS, we replace π_c's SATS with a uniformly random value.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_8$ against the PRF security of HKDF.Expand as follows. When $\mathcal{B}_8$ needs to compute CATS, fk_c, fk_s, or SATS in π_s (or π_c, if π_c shares the same value for MS with π_s), it queries its HKDF.Expand challenge oracle (keyed with MS) on the corresponding labels and transcripts, and uses the responses. Note that if π_c does share MS with π_s, but parts of the transcript were altered such that π_c and π_s are not partners at stage 5 and 6, the adversary may be permitted to query $\mathsf{Reveal}(\pi_c, 5)$ or $\mathsf{Reveal}(\pi_c, 6)$; but since the transcript in π_c and π_s is now different, the labels input to the HKDF.Expand oracle for CATS and SATS are different, so the simulation in $\mathcal{B}_8$ remains good. If the response was the real output, then $\mathcal{B}_8$ has exactly simulated G_{A7} to $\mathcal{A}_1$; if the response

was a random value, then $\mathcal{B}_8$ has exactly simulated G_{A8} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{A7}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A8}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_8}.$$

The stage-5 and stage-6 keys CATS and SATS are now uniformly random strings independent of everything else in the game. Thus, the stage-5 and stage-6 keys have been shown to have wfs1 security.

Malicious acceptance
Let bad denote the event that G_{A8} maliciously accepts in the (fresh) tested session in any of the stages i, such that $\pi.\mathsf{auth}_i = j$ and π does not have a session partner in stage j. If π is a client session, we can reach bad in stage $j = 6$ for both unilaterally and mutually authenticated KEMTLS. If π is a server session, we can reach bad in stage $j = 5$, if $\pi.\mathsf{mutualauth} = \mathsf{true}$; otherwise server sessions are never mutually authenticated.

Game A9: identical-until-bad
This game is identical to G_{A8}, except that we abort the game if the event bad occurs. Games G_{A8} and G_{A9} are *identical-until-bad* [33]. Thus,

$$|\Pr[G_{A8} \Rightarrow 1] - \Pr[G_{A9} \Rightarrow 1]| \leq \Pr[G_{A9} \text{ reaches bad}].$$

In game A8, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary cannot distinguish stage keys anymore. By this game, it can no longer reach bad. Thus

$$\mathsf{Adv}^{G_{A9}}_{\mathcal{A}_1} = 0.$$

It remains to bound $\Pr[G_{A9} \text{ reaches bad}]$.

Game A10: HMAC forgery
In this game, if π is a client, if it does not have a session partner in stage 6, π rejects upon receiving the `ServerFinished` message. If π is a server session and $\pi.\mathsf{mutualauth} = \mathsf{true}$, π rejects upon receiving the `ClientFinished` message if it does not have a session partner in stage 5.

Any adversary that behaves differently in G_{A10} compared to G_{A9} can be used to construct an HMAC forger $\mathcal{B}_9$. The only way that G_{A9} and G_{A10} behave differently is if G_{A10} rejects a MAC that should have been accepted as valid. When rejecting `ServerFinished`, if no partner to π_c at stage 6 exists, no

honest π_s exists with the same session identifier and thus transcript. Similarly, if ClientFinished is rejected, no partner to π_s at stage 5 exists, then no honest π_c exists with the same session identifier and transcript. This means no honest π_s ever created a MAC for the transcript that the client verified, or no honest π_c created the MAC for the transcript that the server verified, and thus it must be a forgery. Concluding:

$$\Pr[G_{A9} \text{ reaches bad}] \leq \Pr[G_{A10} \text{ reaches bad}] + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_9}.$$

By the above, the event bad is never reached.

Analysis of game A10

In game A8, all stage keys in the tested session are uniformly random and independent of all messages in the game, so the hidden bit b used in the tested session is now independent all information sent to the adversary. By A10, all events bad are rejected. Thus:

$$\Pr[G_{A10} \text{ reaches bad}] = 0.$$

This concludes case A, yielding:

$$\mathsf{Adv}^{G_A}_{\mathcal{A}_1} \leq n_s \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\mathcal{B}_2} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_4} & +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_5} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_6} & +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_7} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_8} & +2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_9} \end{array} \right).$$

Case B: no contributive partner, peer not corrupted before stage *i* accepted

Since in this case the tested session π does not have a contributive partner in stage 1 (and hence in any stage), stages aiming for wfs1 are outside the scope of this case. This means that we can assume that the tested session π is a client session if π.mutualauth = false. Otherwise, π may also be a server session that mutually authenticates a client session.

We also allow the intended peer V of the tested session π to be corrupted, but not before the tested session accepts the tested stage fixed in game 3. This

models forward secrecy: even if the adversary obtains the peer's long-term key, the tested keys should still be indistinguishable.

Allowing Corrupt in this case means that any reductions that replace ss_S or ss_C are problematic. However, we show by assumption on the EUF-CMA security of HMAC that π cannot be made to maliciously accept at stage $j = 5$ for server sessions with mutual authentication, or stage $j = 6$ for client sessions. If π does accept, it has a partner at stage j, and at all prior stages.

This leads to the following conclusions. Once stage j accepts, all stages are retroactively authenticated. By case A, all stage keys are indistinguishable, even to an adversary that corrupts any long-term key. This yields retroactive fs for all stage keys.

Game B1: guess peer

In this game, we guess the identity V of the intended peer of the test session, and abort if the guess is incorrect (i.e., if $V \neq \pi.\mathsf{pid}$). This reduces the advantage of $\mathcal{A}_1$ by a factor of the number of users n_u:

$$\mathsf{Adv}^{G_B}_{\mathcal{A}_1} \leq n_u \cdot \mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1}.$$

Game B2: long-term KEM

In this game, in session π, a client session, we replace the static shared secret ss_S with a uniformly random $\widetilde{\mathsf{ss}_S}$. Additionally, in any (server) sessions π' of V which received the same ct_S that was sent in π, we replace the static shared secret ss_S with the same $\widetilde{\mathsf{ss}_S}$. All values derived from ss_S in π use the randomized value $\widetilde{\mathsf{ss}_S}$.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_{10}$ against the IND-CCA security of KEM_s as follows. $\mathcal{B}_{10}$ obtains the IND-CCA challenge $\mathsf{pk}^\star, \mathsf{ct}^\star$, and challenge shared secret $\mathsf{ss}^\star$. It uses $\mathsf{pk}^\star$ as the long-term public key of V. In the tested session π, $\mathcal{B}_{10}$ uses $\mathsf{ct}^\star$ as π's encapsulation ct_S in message CKC, and uses $\mathsf{ss}^\star$ as ss_S. In any session of V, if the ciphertext ct_S received in the CKC message is not $\mathsf{ct}^\star$, then $\mathcal{B}_{10}$ queries its IND-CCA decapsulation oracle, and uses the response as ss_S; if the received ciphertext $\mathsf{ct}_S = \mathsf{ct}^\star$, $\mathcal{B}_{10}$ uses $\mathsf{ss}^\star$ as ss_S. By the assumptions of case B, there is never a Corrupt(V) query that needs to be answered. $\mathcal{A}_1$ terminates and outputs its guess of $b = 0$ or $b = 1$. If $\mathsf{ss}^\star$ was the real shared secret, then $\mathcal{B}_{10}$ has exactly simulated G_{B1} to $\mathcal{A}_1$; if $\mathsf{ss}^\star$ was a random value, then $\mathcal{B}_{10}$ has

exactly simulated G_{B2} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM_s},\mathcal{B}_{10}}.$$

Game B3: replacing AHS

In this game, in session π, a client session, we replace the secret AHS with a uniformly random $\widetilde{\mathsf{AHS}}$. Additionally, in any (server) sessions π' of V which received the same ct_S that was sent in π, we replace AHS with random values, maintaining consistency among any sessions of V that use the same ct_S and the same dHS. All values derived from AHS in these sessions use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_{11}$ against the PRF security of HKDF.Extract in its second argument as follows. When $\mathcal{B}_{11}$ needs to compute AHS in π or in any of the sessions of V that received the same ct_S, it queries its HKDF.Extract challenge oracle (keyed with ss_S) on that session's dHS and uses the response AHS. If the HKDF.Expand challenge oracle was returning real outputs, then $\mathcal{B}_{11}$ has exactly simulated G_{B2} to $\mathcal{A}_1$; if it was returning random values, then $\mathcal{B}_{11}$ has exactly simulated G_{B3} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B3}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{11}}.$$

Game B4: replacing CATS, SAHTS and dAHS

In this game, in session π, a client session, we replace the values CAHTS, SAHTS, and dAHS with uniformly random values. Additionally, in any sessions π' of V which use the same AHS as in π, we replace CAHTS, SAHTS with independent uniformly random values and replace dAHS with the same replacement as in π All values derived from dAHS in these sessions use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_{12}$ against the PRF security of HKDF.Expand as follows. When $\mathcal{B}_{12}$ needs to compute CAHTS, SAHTS, or dAHS in π or in any of the sessions of V that received the same ct_S that was sent in π, it queries its HKDF.Expand challenge oracle (keyed with AHS) on the corresponding labels and transcripts, and uses the responses. If the responses were the real output, then $\mathcal{B}_{12}$ has exactly simulated G_{B3} to $\mathcal{A}_1$; if the responses were random values, then $\mathcal{B}_{12}$

has exactly simulated G_{B4} to $\mathcal{A}_1$. Note in particular that while there may be Reveal queries to CAHTS or SAHTS values in sessions at V that used the same ct_S as in π, previous game 1 ensures that other sessions at V use different nonces r_s, and thus have different transcripts (and game 2 ensures distinct transcripts give distinct transcript hashes), so our simulation remains valid even in the face of Reveal queries to sessions of V. Thus:

$$\mathsf{Adv}^{G_{B3}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{12}}.$$

The stage 3 and 4 keys CAHTS and SAHTS in π are now uniformly random strings independent of everything else in the game. Thus, the stage 3 and 4 keys have been shown to have wfs2 security in client sessions.

Game B5: client authentication long-term KEM

We only play this game if π.mutualauth = true. Otherwise, this game is equal to the previous one as there is no reduction in advantage.

We replace the client authentication shared secret in π, a server session, by a uniformly random value $\widetilde{\mathsf{ss}_C}$. If any of V's client sessions π' received the same ct_C as was sent in π_s in ServerKemCiphertext, we make the same replacement in those π'. Any value derived from ss_C in a session where it was replaced will now use $\widetilde{\mathsf{ss}_C}$. Sending any ServerKemCiphertext to π' with π'.mutualauth = false will simply terminate those sessions at no advantage to the adversary.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_{13}$ against the IND-CCA security of KEM_c as follows. $\mathcal{B}_{13}$ obtains the IND-CCA challenge $\mathsf{pk}^\star, \mathsf{ct}^\star$, and challenge shared secret $\mathsf{ss}^\star$. It uses $\mathsf{pk}^\star$ as the long-term public key of V. In the tested session π, $\mathcal{B}_{13}$ uses $\mathsf{ct}^\star$ as π's encapsulation ct_C in message ServerKemCiphertext, and uses $\mathsf{ss}^\star$ as ss_C. In any session of V, if the ciphertext ct_C received in the ServerKemCiphertext message is not $\mathsf{ct}^\star$, then $\mathcal{B}_{13}$ queries its IND-CCA decapsulation oracle, and uses the response as ss_C; if the received ciphertext $\mathsf{ct}_C = \mathsf{ct}^\star$, $\mathcal{B}_{13}$ uses $\mathsf{ss}^\star$ as ss_C. By the assumptions of case B, there is never a $\mathsf{Corrupt}(V)$ query that needs to be answered. If $\mathsf{ss}^\star$ was the real shared secret, then $\mathcal{B}_{13}$ has exactly simulated G_{B4} to $\mathcal{A}_1$; if $\mathsf{ss}^\star$ was a random value, then $\mathcal{B}_{13}$ has exactly simulated G_{B5} to $\mathcal{A}_1$ Thus:

$$\mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{KEM}_c,\mathcal{B}_{13}}.$$

Game B6: replacing MS

In this game, in session π, we replace the main secret MS with a uniformly random $\widetilde{\mathsf{MS}}$. Additionally, in any sessions π' of V which have the same value for ct_C (if any) that was sent or received in π, we replace MS with random values, maintaining consistency among sessions of V that use the same dAHS and, if π.mutualauth = true, ct_C. All values derived from MS in these sessions use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_{14}$ against the PRF security of HKDF.Extract in its first argument as follows (which we view as "dual-PRF security" of HKDF.Extract keyed with dAHS). When $\mathcal{B}_{14}$ needs to compute MS in π or in any of the sessions of V which have the same dAHS as π, it queries its HKDF.Extract challenge oracle on ct_C (or 0, if π.mutualauth = false) and uses the response as MS. If the HKDF.Extract challenge oracle was returning real values, then $\mathcal{B}_{14}$ has exactly simulated G_{B5} to $\mathcal{A}_1$; if it was returning random values, then $\mathcal{B}_{14}$ has exactly simulated G_{B6} to $\mathcal{A}_1$. Thus:

$$\mathsf{Adv}^{G_{B5}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B6}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{14}}.$$

Game B7: replacing CATS, fk_c, fk_s and SATS

In this game, in session π, we replace the value CATS, fk_c, fk_s, and SATS with uniformly random values. Additionally, in any sessions π' of V which share the value of MS with π, we replace CATS and SATS with independent uniformly random values, and replace fk_c and fk_s with the same replacements as in π.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct a distinguisher $\mathcal{B}_{15}$ against the PRF security of HKDF.Expand as follows. When $\mathcal{B}_{15}$ needs to compute CATS, fk_c, fk_s or SATS in π or in any of the sessions of V that have the same value for MS as π, it queries its HKDF.Expand challenge oracle on the corresponding labels and transcripts, and uses the response. If the responses were the real output, then $\mathcal{B}_{15}$ has exactly simulated G_{B6} to $\mathcal{A}_1$; if the response were random values, then $\mathcal{B}_{15}$ has exactly simulated G_{B7} to $\mathcal{A}_1$. Note in particular that while there may be Reveal queries to CATS values in sessions at V that used value of MS as in π, previous game 1 ensures that other sessions at V use different nonces r_s, and thus have different transcripts (and game 2 ensures distinct transcripts give distinct transcript hashes), so our simulation remains valid even in the face of Reveal queries to sessions

of V. Thus:

$$\mathsf{Adv}^{G_{B6}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B7}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_{15}}.$$

The stage 5 key CATS and stage 6 key SATS in π are now uniformly random strings independent of everything else in the game. Thus, the stage 5 and stage 6 keys have been shown to have wfs2 security in client sessions and server sessions that authenticate the client.

Malicious acceptance

Let bad denote the event that G_{B7} maliciously accepts in any of the stages specified for any i as $\pi.\mathsf{auth}_i = j$ in the (fresh) tested session π without a session partner in stage j. If π is a client session, we can reach bad in stage $j = 6$ for both unilaterally and mutually authenticated KEMTLS. If π is a server session, we can reach bad in stage $j = 5$, if $\pi.\mathsf{mutualauth} = \mathsf{true}$; otherwise server sessions are never mutually authenticated.

Game B8: identical-until-bad

This game is identical to G_{B7}, except that we abort if the event bad occurs. Games G_{B7} and G_{B8} are identical-until-bad [33]. Thus,

$$|\Pr[G_{B7} \Rightarrow 1] - \Pr[G_{B8} \Rightarrow 1]| \leq \Pr[G_{B8} \text{ reaches bad}].$$

By game B7, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary cannot distinguish stage keys anymore. By this game, it can no longer reach bad. Thus

$$\mathsf{Adv}^{G_{B8}}_{\mathcal{A}_1} = 0.$$

It remains to bound

$$\Pr[G_{B8} \text{ reaches bad}].$$

Game B9: HMAC forgery

In this game, if it is a client session, π rejects upon receiving the `ServerFinished` message. If π is a server session, it rejects upon receiving the `ClientFinished` message.

Any adversary that behaves differently in G_{B9} compared to G_{B8} can be used to construct an HMAC forger $\mathcal{B}_{16}$. The only way that G_{B8} and G_{B9} behave differently is if G_{B9} rejects a MAC that should have been accepted as valid. When rejecting `ServerFinished`, no honest π_s exists with the same session

identifier and thus transcript. When rejecting ClientFinished, no honest π_c exists with the same session identifier and thus transcript. This means no honest server π' ever created a MAC for the transcript that π verified, and thus it must be a forgery. Concluding:

$$\Pr[G_{B8} \text{ reaches bad}] \leq \Pr[G_{B9} \text{ reaches bad}] + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{16}}.$$

Since this game rejects all ServerFinished messages, the event bad is never reached in client sessions. If π.mutualauth = true, this game also rejects all ClientFinished messages. If π.mutualauth = false, rejecting ClientFinished is out of scope as we only aim for wfs1.

Analysis of game B9

In game B7, all stage keys in the tested session are uniformly random and independent of all messages in the game, so the hidden bit b used in the tested session is now independent of all information sent to the adversary. By B9, all events bad are rejected. Thus:

$$\Pr[G_{B9} \text{ reaches bad}] = 0.$$

This concludes case B, yielding:

$$\mathsf{Adv}^{G_B}_{\mathcal{A}_1} \leq n_u \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_{10}} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{11}} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{12}} & +\mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_c,\mathcal{B}_{13}} \\ +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{14}} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{15}} \\ +\ 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{16}} & \end{array} \right).$$

Combining the bounds in cases A and B yields the theorem. □

8 Security of KEMTLS-PDK

In the previous chapter, we have defined our security model for the reductionist proofs of security. We defined the Match and Multi-Stage security experiments and gave the security properties for KEMTLS. We finally proved the security of KEMTLS. In this chapter, we will prove the security of KEMTLS-PDK in the same model. This will complete the reductionist security analysis of both protocols and precisely describe their security characteristics.

8.1 Overview of the security analysis

The proof of KEMTLS-PDK proceeds in much the same way as the proof of KEMTLS in the previous chapter. Following the approach of Dowling, Fischlin, Günther, and Stebila [127, 128], we model KEMTLS-PDK as a multi-stage key-agreement protocol [143], where each session has several *stages* in each of which a shared secret key is established. This model is adapted from the Bellare–Rogaway security model for authenticated key exchange [32].

Each party (client or server) has a long-term public-key/secret-key pair, and we assume there exists a public-key infrastructure for certifying these public keys. The client is assumed to have access to the public keys for any servers it connects to. Like in the specifics of KEMTLS, the session identifier for KEMTLS-PDK sessions consists of all the messages transmitted up until that point; as KEMTLS-PDK permits pre-distributed public keys, those pre-distributed values will be included in the session identifier.

8.1.1 Security characteristics of KEMTLS-PDK

We briefly summarize some of the security characteristics of KEMTLS-PDK. Some of these are shown in the proof; properties like deniability we only discuss informally.

Key indistinguishability

Like in the previous proof, the Test query captures that keys established should be indistinguishable from random: the goal of the adversary is to guess the hidden bit in the Test query, thereby distinguishing real keys from random. We restrict the adversary from issuing the Test query to stages where the stage key has been exposed via a Reveal query, including partnered sessions.

Explicit authentication

Client sessions in KEMTLS-PDK receive explicit authentication (and fs forward secrecy) right before the stage-4 key is accepted, one round trip earlier than in KEMTLS. Server sessions, in mutually authenticated KEMTLS-PDK, receive explicit authentication right before the stage-5 key is accepted, again one round tip earlier than KEMTLS. In particular, this means that KEMTLS-PDK gives explicit authentication for all client application data (although only implicit authentication for the client certificate).

Negotiation and downgrade resilience

In section 7.1.4 we observed that implicit authentication characteristics of early stages of keys in KEMTLS meant that some application data would be transmitted prior to the client having received explicit authentication from the server of the algorithms negotiated during the handshake. This meant that it would be possible for an adversary to cause a downgrade to a suboptimal (from the server's perspective) algorithm—although still only to an algorithm that the client offered to use. In KEMTLS-PDK, explicit server authentication happens one round trip earlier, in particular prior to client transmission of application data, so KEMTLS-PDK offers full downgrade resilience.

Replayability

In contrast to KEMTLS, KEMTLS-PDK has replayable stages. In particular, stage-1 keys are not guaranteed to be unique at server instances since the same ClientHello message can be replayed multiple times to induce the same stage-1 key. This is the only replayable stage: all subsequent stages are replay-protected.

Anonymity

Like TLS 1.3 and KEMTLS, KEMTLS-PDK does not offer full anonymity, in particular, due to the presence of the ServerNameIndicator extension in the ClientHello message. Our implementation also identifies the server certificate that was encapsulated to. (This identifier could be omitted by using trial

decryption at the server, though if the server has many public keys, this could be prohibitive.) The TLS working group is considering an "Encrypted ClientHello" (ECH) mechanism that relies on the client obtaining a server public key out-of-band to enable identity protection for the server. KEMTLS-PDK's use of a pre-distributed key for encryption of part of the initial client message may be compatible with a variant of ECH, which we leave as future work since ECH has not yet been finalized even for TLS 1.3.

The discussion in the previous paragraph does assume that the ciphertext encapsulated to the long-term KEM public key of the server cannot be used to identify the server's public key. If the ciphertext somehow identifies the public key to which it is encapsulated, then KEMTLS-PDK's unencrypted authentication key exchange in the ClientHello message identifies the server, even if trail decryption is used. A KEM for which the ciphertext and the public key cannot be correlated is said to provide *anonymity*. Fortunately, recent work by Maram and Xagawa has shown that Kyber does have this property [243].

Deniability

As KEMTLS-PDK avoids the use of signatures for authentication, like KEMTLS[1] and unlike TLS 1.3, KEMTLS-PDK offers *offline deniability* [117] in the *universal deniability* setting [189]. This means that a judge, when given a transcript of a protocol execution and all the keys involved, cannot tell whether the transcript is genuine or forged. KEMTLS-PDK does not have the harder-to-achieve online deniability property [124] when one party tries to frame the other or collaborates with the judge.

8.1.2 Different KEMs

Interestingly, if KEMTLS uses KEMs with different cryptographic assumptions for the ephemeral and the long-term KEM, we obtain hybrid confidentiality in the following sense: even if the assumption used for the ephemeral KEM is cryptographically broken at some point, handshakes retain confidentiality as long as the long-term KEM keys are not compromised. With KEMTLS-PDK, some interesting combinations of ephemeral and long-term KEMs—e.g., Classic McEliece and Kyber—may become feasible. We leave it to future work to investigate this further and to formalize the notion of hybrid confidentiality sketched here.

[1] For a discussion of KEMTLS' deniability properties see section 5.4.5.

8.1.3 Security properties of KEMTLS-PDK

We will state the properties in the same model as we specified for KEMTLS in section 7.2. This includes the definitions for multiple forms of forward secrecy (definition 7.3) and explicit authentication. KEMTLS-PDK has 5 stage keys. The properties of each stage key in a client session are as follows:

- Stage 1: this stage is replayable. That means we do not have any forward secrecy guarantees. The key is implicitly authenticated; it is indistinguishable against any adversary that has never corrupted the server. Once stage 4 accepts, this stage is retroactively explicitly authenticated. This key is for internal use.
- Stages 2 and 3: these stages are not replayable. These stages have wfs2 security from when they are accepted: they are both indistinguishable against passive adversaries and secure against adversaries that never corrupt the server. These keys obtain retroactive fs security once stage 4 has accepted. They are implicitly authenticated at the time of acceptance; once stage 4 has accepted they are retroactively explicitly authenticated. The keys accepted in stages 2 and 3 are for internal use.
- Stage 4: this stage is not replayable. The key has fs security and explicit authentication from acceptance. This stage key is for external use.
- Stage 5: this stage is not replayable. The key has fs security from the time that the key is accepted. It however is never explicitly authenticated as the client never receives a confirmation from the server.

In server instances of KEMTLS, the properties of each stage key are:

- Stage 1: this stage is replayable; we do not have any forward secrecy guarantees. The key is implicitly authenticated; it is indistinguishable against any adversary that has never corrupted the server. If the server is mutually authenticating the client, this key obtains retroactive explicit authentication once stage 5 accepts. This key is for internal use.
- Stages 2 and 3: these stages are not replayable. These stages have wfs1 security from when they are accepted: they are indistinguishable against passive adversaries. If the server is mutually authenticating the client, these keys (retroactively) obtain fs security and explicit authentication once stage 5 accepts. The stage 2 and 3 keys are for internal use.

- Stage 4: this stage is not replayable. If the server is mutually authenticating the client, this stage has wfs2 security: the key is indistinguishable against passive adversaries and adversaries that never corrupt the client's long-term key. This stage then also obtains fs security and retroactive explicit authentication once stage 5 accepts. If the server is not authenticating the client, this key only obtains wfs1 security and is never explicitly authenticated. This stage key is for external use.

- Stage 5: this stage is not replayable. If the server is mutually authenticating the client, this stage has fs security and explicit authentication from the time of acceptance. Otherwise, this key only obtains wfs1 security and is never explicitly authenticated. The key accepted by this stage is for external use.

The following theorem says that KEMTLS-PDK is Multi-Stage-secure with respect to the forward secrecy, explicit authentication, and internal/external key-use properties as specified above. We bound the security of KEMTLS-PDK on the assumption that the hash function H is collision-resistant, HKDF is a pseudorandom function in either its "salt" or "input keying material" arguments, HMAC is a secure message-authentication code, KEM_s and, for mutual authentication, KEM_c are IND-CCA secure KEMs, and KEM_e is an IND-1CCA-secure KEM (i.e., KEM_e is secure if a single decapsulation query is allowed).

Theorem 8.1. *Let $\mathcal{A}$ be an algorithm, and let n_s be the number of sessions and n_u be the number of parties. Then the advantage of $\mathcal{A}$ in breaking the multi-stage security of KEMTLS-PDK is upper-bounded by*

$$\frac{n_s^2}{2^{|\mathsf{nonce}|}} + \epsilon_{\mathsf{H}}^{\mathsf{COLL}} + 5n_s \cdot \left(\begin{array}{l} n_s \cdot \left(\begin{array}{c} \epsilon_{\mathsf{KEM}_s}^{\text{IND-CCA}} + 2\,\epsilon_{\mathsf{HKDF.Extract}}^{\text{PRF-sec}} + 3\,\epsilon_{\mathsf{HKDF.Expand}}^{\text{PRF-sec}} \\ + \epsilon_{\mathsf{KEM}_e}^{\text{IND-1CCA}} + \epsilon_{\mathsf{HKDF.Extract}}^{\text{dual-PRF-sec}} + \epsilon_{\mathsf{HMAC}}^{\text{EUF-CMA}} \end{array} \right) \\ n_u \cdot \left(\begin{array}{c} \epsilon_{\mathsf{KEM}_s}^{\text{IND-CCA}} + \epsilon_{\mathsf{HKDF.Extract}}^{\text{PRF-sec}} + 2\,\epsilon_{\mathsf{HKDF.Expand}}^{\text{PRF-sec}} \\ + 2\,\epsilon_{\mathsf{HKDF.Extract}}^{\text{dual-PRF-sec}} + \epsilon_{\mathsf{KEM}_c}^{\text{IND-CCA}} + \epsilon_{\mathsf{HMAC}}^{\text{EUF-CMA}} \end{array} \right) \end{array} \right).$$

Above we use the shorthand notation $\epsilon_Y^X = \mathsf{Adv}_{Y,\mathcal{B}_i}^X$ for reductions $\mathcal{B}_i$ that are described in the proof; we provide the fully detailed bound in theorem 8.3. The proof of theorem 8.3 appears in section 8.2.3; we first provide a sketch.

8.1.4 Sketch of the proof

As in the proof of KEMTLS, we begin the proof with a series of game hops that allow us to not worry about certain attacks later on. We disallow the reuse of nonces in the ClientHello and ServerHello messages and also rule out hash collisions. This ensures that transcripts with different messages are unique in later games. The Multi-Stage security experiment allows the adversary to make multiple Test queries. We again restrict the adversary to make a single Test query by guessing a to-be-tested session and the to-be-tested stage using a hybrid argument [167]. This gives us a tightness loss of $5n_s$ related to the number of sessions and stages.

The proof then splits into two cases: case A where the (now single) tested session has an honest contributive partner in the second stage; and case B where the tested session does not have an honest contributive partner in the second stage and the adversary does not corrupt the peer's long-term key before the tested stage has been accepted. These two cases effectively correspond to the forward-secrecy levels wfs1 and wfs2. We finally show, by assumption on EUF-CMA security of HMAC, that both cases do not maliciously accept, giving us the final forward secrecy level fs.

Case A

In this case, the tested session is assumed to have an honest contributive partner in the second stage. If the tested session is a client session, then we know that this partner is unique; however, if the tested session is a server session then the server may be receiving a replayed message. However, since we rule out nonce repetition, there still exists only one honest client session. In the first game of case A we make the adversary guess the honest contributive partner to the tested session, which results in a tightness loss related to the number of sessions n_s.

Next, we rely on the IND-CCA security of the server's static KEM algorithm KEM_s. After a series of games in which we replace secrets and restrict the adversary's advantage based on PRF-security assumptions, we rely on the IND-1CCA security of the ephemeral KEM KEM_e. Like in KEMTLS, if the tested session is a server session, the adversary may send its own ct_e in a crafted ServerHello message back to the client. To correctly simulate this valid adversary behavior, we need to allow a single decapsulation oracle query.

The game proceeds with another sequence of games in which we replace

keys with random values, relying on the PRF-security and dual-PRF-security assumptions, until all keys are indistinguishable from random. This yields the required wfs1 security property for all stage keys in both unilaterally and mutually authenticated client and server sessions.

Case B

In this case, the tested session is assumed to *not* have an honest contributive partner in stage 2. This means that the adversary is actively impersonating the peer to the tested session. There is no partner at any stage of that session. As a result, any stages aiming for wfs1 security are out of scope. The tested session is either a client session or a server session attempting mutual authentication. In case B we assume the peer's long-term key is not compromised before the tested session accepts the tested stage. This allows us to rely on the security of encapsulations against the peer's long-term public key.

Case B's sequence of game hops again proceeds similarly to those of the multi-stage security proof of KEMTLS. First, we guess the identity of the peer that the adversary will attempt to impersonate to the tested session. Then we replace with a random value the shared secret ss_S that a client session encapsulates against the intended server's long-term static key pk_S. If KEM_s is IND-CCA-secure, only the intended server should be able to decapsulate and recover ss_S, and thus ss_S, and any key derived from it (following a sequence of game hops involving the security of HKDF), is an implicitly authenticated key unknown to the adversary. We similarly replace with a random value the shared secret ss_C that a server session encapsulates against its intended client's long-term static key pk_C, as well as any key derived from it. This yields the indistinguishability of stage keys 2–5 of client sessions, and stages 4 and 5 in server sessions attempting mutual authentication under the conditions of case B, and hence their required wfs2 security properties.

Malicious acceptance

Case B allows the adversary to corrupt the intended peer's long-term key after the tested session accepts in stage 4 (if the session is a client) or stage 5 (if it is a server attempting mutual authentication). Again, reduction from IND-CCA security of KEM_s runs into a problem: how to correctly answer the adversary's Corrupt query. Up until this bad query occurs, however, our IND-CCA reduction (and indeed, every reduction in case B) is fine, and all keys in the tested session can be shown to be indistinguishable from random. This

includes key fk_s that the server uses for the MAC authenticating the transcript in the ServerFinished message. If the client accepts ServerFinished in case B—without a partner to stage 4—then the adversary has forged an HMAC tag. We rely on the EUF-CMA security of HMAC and show that the reduction will never have to answer that Corrupt query. In the sessions that are using mutual authentication, we show the same for fk_c and ClientFinished in stage 5.

Assuming all the cryptographic primitives are secure, no stage accepts under the conditions of case B. This yields explicit server-to-client authentication of stage 4, and explicit client-to-server authentication of stage 5 (if mutual authentication is used). We get retroactive authentication of all previous stages once the explicitly authenticated stage accepts, since their session identifiers are substrings of the stage's sid. Explicit authentication yields forward secrecy (fs) of the stage-5 key at the client, and the stage-5 and 6 keys of a mutually authenticating server, at the time of acceptance. Retroactively, stages 1–4 of the client, and all stages of a mutually-authenticating server, also obtain fs.

8.2 Security proof

The model we use in this analysis is the same as the model used for KEMTLS in chapter 7. The definition of the properties of sessions is given in section 7.2.1 on page 105. We also use the same definitions of Match security (definition 7.2 on page 109), Multi-Stage security (definition 7.5 on page 111), *freshness* (definition 7.3 on page 110), and *malicious acceptance* (definition 7.4 on page 111).

8.2.1 Specifics of KEMTLS-PDK

In our protocol, the number of states is $M = 5$. KEMTLS-PDK without client authentication is shown in figure 6.3 and with client authentication in figure 6.4 The session identifiers are set up as follows in unilaterally authenticated sessions:

$$\begin{aligned}
\mathsf{sid}_1 &= (\text{“ETS”}, \quad \texttt{ServerCertificate}, \texttt{ClientHello}),\\
\mathsf{sid}_2 &= (\text{“CHTS”}, \texttt{SCRT}, \texttt{ClientHello} \ldots \texttt{ServerHello}),\\
\mathsf{sid}_3 &= (\text{“SHTS”}, \texttt{SCRT}, \texttt{ClientHello} \ldots \texttt{ServerHello}),\\
\mathsf{sid}_4 &= (\text{“SATS”}, \texttt{SCRT}, \texttt{ClientHello} \ldots \texttt{ServerFinished}),\\
\mathsf{sid}_5 &= (\text{“CATS”}, \texttt{SCRT}, \texttt{ClientHello} \ldots \texttt{ClientFinished}).
\end{aligned}$$

For mutually authenticated sessions, sid_2 and all subsequent session identifiers implicitly contain the ClientCertificate and ServerKemCiphertext messages in these message ranges. Each identifier is made up of a label, the server's certificate (even though is not transmitted), and all the unencrypted handshake messages up to that point. Finally, if any client or server receives an unexpected message, they terminate.

For the contributive identifiers cid_i we take some special care. For stage 1, we want to ensure client sessions can be tested, even if the adversary drops the client's message to the server. We set $\mathsf{cid}_1 = (\text{“ETS”}, \mathtt{SCRT}, \varnothing)$ initially. When the client sends or the server receives the ClientHello message, we update it to $\mathsf{cid}_1 = (\text{“ETS”}, \mathtt{SCRT}, \mathtt{CH})$.

In case A of our Multi-Stage proof we need to identify the unique pair of honest contributive server and client sessions, even if the adversary drops the server's response to the client. This requires us to set the second contributive identifier $\mathsf{cid}_2 = (\text{“CHTS”}, \mathtt{SCRT}, \mathtt{CH})$ when sending or receiving the ClientHello message. At that time, we also set $\mathsf{cid}_3 = (\text{“SHTS”}, \mathtt{SCRT}, \mathtt{CH})$. If $\pi.\mathsf{mutualauth} = \mathsf{true}$, we update cid_2 and cid_3 by updating the transcripts to $\mathtt{CH} \ldots \mathtt{CCRT}$ when the ClientCertificate message is sent or received. When the ServerHello message is received or sent, they update this to $\mathsf{cid}_i = \mathsf{sid}_i$ for $i = 2, 3$. All other contributive identifiers ($i = 4, 5$) are set when the corresponding sid_i is set.

Every client and server session of unilaterally or mutually authenticated KEMTLS-PDK uses the following properties for the replayability and usage of the five stages:

$$\begin{aligned} \mathsf{replay} &= \left(\mathsf{replayable}, \mathsf{nonreplayable}^{\times 4}\right), \\ \mathsf{use} &= \left(\mathsf{internal}^{\times 3}, \mathsf{external}^{\times 2}\right). \end{aligned}$$

All KEMTLS-PDK client sessions authenticate the server. This means that there is no difference between unilaterally authenticated and mutually authenticated client sessions.

All client sessions use:

$$\mathsf{auth} = (4, 4, 4, 4, \infty),$$
$$\mathsf{FS} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ & \mathsf{wfs2} & \mathsf{wfs2} & \mathsf{fs} & \mathsf{fs} \\ & & \mathsf{wfs2} & \mathsf{fs} & \mathsf{fs} \\ & & & \mathsf{fs} & \mathsf{fs} \\ & & & & \mathsf{fs} \end{pmatrix}.$$

The server is explicitly authenticated when the client accepts stage 4. Note that like in the mutually authenticated server sessions of KEMTLS, the last stage key is never explicitly authenticated; no later *handshake* messages show the client that the server correctly derived this key.

In unilaterally authenticated KEMTLS-PDK, only the server is authenticated. The server never gets explicit authentication of the client, and cannot get better forward secrecy than wfs1: stage keys are only secure against passive adversaries. Unilaterally authenticated server sessions use:

$$\mathsf{auth} = \left(\infty^{\times 5}\right),$$
$$\mathsf{FS} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} \\ & & & \mathsf{wfs1} & \mathsf{wfs1} \\ & & & & \mathsf{wfs1} \end{pmatrix}.$$

When using mutual authentication, i.e, $\pi.\mathsf{mutualauth} = \mathsf{true}$, the client is explicitly authenticated when stage $m = 5$ is accepted. Client authentication allows us to obtain forward secrecy levels wfs2 in stage 4 and (retroactive) fs for all stages in stage 5:

$$\mathsf{auth} = (5, 5, 5, 5, 5),$$
$$\mathsf{FS} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} \\ & & \mathsf{wfs1} & \mathsf{wfs1} & \mathsf{fs} \\ & & & \mathsf{wfs2} & \mathsf{fs} \\ & & & & \mathsf{fs} \end{pmatrix}.$$

8.2.2 Match security

As in prior work [82, 83, 127, 128, 143, 321], we need to show that our model has sound behavior of session matching. Match security ensures that in honest sessions, π.sid and π'.sid correctly match, where π and π' are partnered.

Theorem 8.2. *KEMTLS-PDK is* Match*-secure (definition 7.2). Any efficient adversary $\mathcal{A}$ has advantage*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathit{KEMTLS\text{-}PDK},\mathcal{A}} \leq n_s(\delta_e + \delta_s + \delta_c) + \frac{n_s^2}{2^{|\mathsf{nonce}|}},$$

where n_s is the number of sessions, |nonce| is the length of the nonces r_c, r_s in bits. δ_e is the correctness of the ephemeral KEM, and δ_s and δ_c are the correctness of the long-term KEMs of the server and the client, respectively. If π.mutualauth = false, $\delta_c = 0$.

Proof. We will show the properties of Match-security hold.

1. By definition, the session identifiers contain all handshake messages. The KEM keys and the hashes of the handshake messages are the only inputs into the key schedule. At stage 1, the input to the agreed key is the KEM_s shared secret and the `ClientHello` message. For stages 2 and 3, the input to the agreed keys are the previous key, messages up to and including `ServerHello` and the ephemeral KEM_e shared secret. For final stages 4 and 5, the inputs are the previous keys, messages up to `ServerFinished` and `ClientFinished`, respectively, and, when using mutual authentication, the KEM_c shared secret. These inputs are all included in the session identifiers, which means that both parties use the same inputs to key computations. The only way they could arrive at different keys is when any of the KEMs fail. In each of the n_s sessions, the ephemeral KEM KEM_e may fail with probability δ_e, the static KEM KEM_s may fail with probability δ_s. When using mutual authentication, the static KEM KEM_c may fail with probability δ_c; when not using mutual authentication $\delta_c = 0$.

2. All messages are exclusively sent or received by either role. This means no initiator or responder will accept an incoming message intended for the other role. This implies any pair of two sessions in a non-replay

stage must have an initiator and responder. As at most two sessions in a non-replay phase have the same sid_i (shown later), these pairings are unique and opposite. As the only replayable messages are ClientHello and ClientCertificate messages, not-opposite-role pairings can only be between responders.

3. By definition, cid_i is final and equal to sid_i whenever stage i accepts.

4. The presence of ServerKemCiphertext in the transcript decides if the value of either session's mutualauth = true before either session accepts the stage-4 key, which is when explicit authentication is first reached in mutually authenticated KEMTLS-PDK.

5. Partnered sessions have to agree once they reach a retroactively authenticated stage, so at stage 4 for unilateral authentication and stage 5 for mutual authentication. For Match security, we are only concerned with honest client and server sessions. The client already knows the identity of the server through the pre-distributed key, which is included in the session identifiers at stage 1. The server learns the identity of the client through the ClientCertificate message, which is included in the session identifiers at stage 5. Any honest client will only send its own certificate.

6. As each stage's session identifier has a unique label, this holds trivially.

7. We are only concerned about stages 2 and later. All session identifiers sid contain nonces r_c and r_s embedded in the ClientHello and ServerHello messages. To get any collision between sessions of honest parties, some session would need to pick the same nonce as another session. If this happens, the parties may then be partnered through a regular protocol run to another one. The probability for such a collision is bounded by the birthday bound $n_s^2 \cdot 2^{-|\mathsf{nonce}|}$. Here, n_s is the maximum number of sessions and $|\mathsf{nonce}| = 256$ is the nonces' length in bits.

□

8.2.3 Multi-Stage security

Like in chapter 7 we prove the security of KEMTLS-PDK via Multi-Stage security games as introduced by [143]. The adversary wins, Bellare–Rogaway-style [32],

if they correctly distinguish the keys derived in the protocol from random. They also win if they get the protocol to maliciously accept (definition 7.4); this models the acceptance of an invalid `ServerFinished` or `ClientFinished` message that was forged by the adversary.

Theorem 8.3. *Let adversary $\mathcal{A}$ be a probabilistic polynomial-time algorithm. n_s is the number of sessions and n_u is the number of identities. There exist algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_{19}$, given in the proof, such that*

$$\begin{aligned}
\mathsf{Adv}^{\text{Multi-Stage}}_{\textit{KEMTLS-PDK},\mathcal{A}} \leq\ & \frac{n_s^2}{2^{|\text{nonce}|}} + \mathsf{Adv}^{\text{COLL}}_{\text{H},\mathcal{B}_1} \\
& + 5n_s \cdot \left(\begin{array}{ll}
\mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_2} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3} \\
+ \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_4} & + \mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\mathcal{B}_5} \\
+ \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_6} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_7} \\
+ \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_8} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_9} \\
+ 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{10}} &
\end{array} \right) \\
& + n_u \cdot \left(\begin{array}{ll}
\mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_{11}} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{12}} \\
+ \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{13}} & + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{14}} \\
+ \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_c,\mathcal{B}_{16}} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{15}} \\
+ \mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{17}} & + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{18}} \\
+ 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{19}} &
\end{array} \right).
\end{aligned}$$

Proof. We follow the basic structure of the proof of the KEMTLS handshake as given in chapter 7. This in turn is based on the proofs of the TLS 1.3 handshake by Dowling, Fischlin, Günther, and Stebila [127, 128]. The proof proceeds by a sequence of games in which we keep reducing the advantage of the adversary. As the adversary otherwise loses the game, we assume that all tested sessions remain fresh throughout the experiment.

Game 0: MultiStage game

We define G_0 to be the original Multi-Stage game:

$$\mathsf{Adv}^{\text{Multi-Stage}}_{\text{KEMTLS-PDK},\mathcal{A}} = \mathsf{Adv}^{G_0}_{\mathcal{A}}.$$

Game 1: nonce collisions
If any honest session uses the same nonce r_c or r_s as any other session, the challenger aborts. Given that there are n_s sessions each using uniformly random nonces of size |nonce| = 256, the chance of a repeat is given by a birthday bound:

$$\mathsf{Adv}^{G_0}_{\mathcal{A}} \leq \mathsf{Adv}^{G_1}_{\mathcal{A}} + \frac{n_s^2}{2^{|\mathsf{nonce}|}}.$$

This means we can now rule out nonce collisions in future games.

Game 2: hash collisions
If any two honest sessions compute the same hash for different inputs of hash function H, the challenger aborts. If this event occurs, we obtain a reduction $\mathcal{B}_1$ that can break the collision resistance of H. $\mathcal{B}_1$ outputs the two distinct input values when a collision occurs. This gives us the following:

$$\mathsf{Adv}^{G_1}_{\mathcal{A}} \leq \mathsf{Adv}^{G_2}_{\mathcal{A}} + \mathsf{Adv}^{\mathsf{COLL}}_{\mathsf{H},\mathcal{B}_1}.$$

Game 3: single Test query
By invoking a hybrid argument by Günther [167], we restrict $\mathcal{A}$ to only make a single Test-query. This reduces the advantage at most by $1/5\, n_s$ for the five stages of KEMTLS-PDK. Any single-query adversary $\mathcal{A}_1$ can emulate the original multi-query adversary $\mathcal{A}$ by guessing the to-be-tested session in advance. Any other Tests that $\mathcal{A}$ may submit, $\mathcal{A}_1$ simulates by carefully selected Reveal queries. $\mathcal{A}_1$ needs to know how sessions are partnered from the session identifiers sid. Only the first one is unencrypted, but the later sid can be obtained by $\mathcal{A}_1$ by revealing handshake traffic secrets.

We get the following advantage by letting $\mathcal{A}_1$ guess the right session and stage:

$$\mathsf{Adv}^{G_2}_{\mathcal{A}} \leq 5n_s \cdot \mathsf{Adv}^{G_3}_{\mathcal{A}_1}.$$

This restriction of $\mathcal{A}$ to $\mathcal{A}_1$ means we can now refer to *the* session π at stage i that is tested. We can also assume we know this from the outset.

Case distinction

We now need to consider two separate cases of game 3. These cases, respectively, roughly correspond to the specified properties of weak forward secrecy: wfs1 and wfs2. By rejecting malicious acceptance, we finally show fs.

A. In these games, denoted G_A, the tested session π has a unique contributive partner in stage 2. This means there exists a session $\pi' \neq \pi$ such that $\pi.\mathsf{cid}_2 = \pi'.\mathsf{cid}_2$.

B. In these games, denoted G_B, the tested session π does *not* have a contributive partner in stage 2. In addition, $\mathsf{Corrupt}(\pi.\mathsf{pid})$ was not called *before* stage i of π accepted.

As by rejecting malicious acceptance, no cases exist that call $\mathsf{Corrupt}(\pi.\mathsf{pid})$ *after* stage i accepted, these cases are exhaustive.

The advantage of the adversary can be considered separately for these cases:

$$\begin{aligned}\mathsf{Adv}^{G_3}_{\mathcal{A}_1} &\leq \max\left\{\mathsf{Adv}^{G_A}_{\mathcal{A}_1}, \mathsf{Adv}^{G_B}_{\mathcal{A}_1}\right\} \\ &\leq \mathsf{Adv}^{G_A}_{\mathcal{A}_1} + \mathsf{Adv}^{G_B}_{\mathcal{A}_1}.\end{aligned}$$

Case A: unique stage 2 contributive partner exists

In this case, we assume that π has a π' with whom they share $\pi.\mathsf{cid}_2 = \pi'.\mathsf{cid}_2$. If the tested session π is a client (initiator) session, then $\pi.\mathsf{cid}_2 = \pi.\mathsf{sid}_2$ and a partner session at π' also exists. sid_2 includes the client and server nonces, and by game 1 no honest sessions repeat nonces. This means that the contributive partner at stage 2 is unique.

However, if π has $\mathsf{role} = \mathsf{responder}$, then it may have received a replayed ClientHello message. This would mean a contributive partner session exists at stage 2, but there is no partnered session. However, there exists only one honest client session that is a contributive partner: cid_2 includes the client nonce (unique by game 1) and contributive partnering includes roles.

This means we can speak of a particular tested session, π. Its unique contributive stage-2 partner we call π'. Of these two, we let π_c be the session that is the client ($\mathsf{role} = \mathsf{initiator}$) session. The other session, with $\mathsf{role} = \mathsf{responder}$, is the server session π_s.

Game A1: guess contributive partner session

In this game, the challenger tries to guess the $\pi' \neq \pi$ that is the honest contributive partner to π at stage 2. As the challenger guesses correctly with probability $1/n_s$, this reduces the advantage of $\mathcal{A}_1$ as:

$$\mathsf{Adv}^{G_A}_{\mathcal{A}_1} \leq n_s \cdot \mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1}.$$

In the remainder of case A, we will keep replacing keys in π and π'.

Game A2: static KEM

In this game we replace the shared secret ss_S encapsulated to pk_S by a uniformly random $\widetilde{\mathsf{ss}_S}$. We make this replacement in π_c and π_s, and, as this stage is replayable, in any other sessions π'' of the server S that received ct_S.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_2$ that breaks the IND-CCA security of KEM_s. $\mathcal{B}_2$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and challenge shared secret $\mathsf{ss}^\star$ and uses $\mathsf{pk}^\star$ as the long-term key pk_S of the server S. In π_c, $\mathcal{B}_2$ sends $\mathsf{ct}^\star$ in the `ClientHello` message. $\mathcal{B}_2$ uses $\mathsf{ss}^\star$ for ss_S in both π_c and π_s. If $\mathcal{A}_1$ delivers $\mathsf{ct}^\star$ to some other session π'' of S, $\mathcal{B}_2$ uses $\mathsf{ss}^\star$ as value for ss_S in π''. If $\mathcal{A}_1$ delivers a different $\mathsf{ct}' \neq \mathsf{ct}^\star$ to some other session π'' of S, $\mathcal{B}_2$ queries its IND-CCA decapsulation oracle with ct' to obtain the required shared secret.

Stage 1 cannot maliciously accept since it is replayable. By the definition of freshness (definition 7.3) we also do not need to answer Corrupt queries.

In the end, $\mathcal{A}_1$ terminates and outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_2$ has exactly simulated G_{A1} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_2$ has exactly simulated G_{A2} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{KEM}_s,\mathcal{B}_2}.$$

Game A3: replacing ES

In this game we replace the early handshake secret ES by a uniformly random $\widetilde{\mathrm{ES}}$ in both sessions π_c and π_s.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_3$ that breaks the PRF security of HKDF.Extract in its second argument. When $\mathcal{B}_3$ needs to compute ES in π or π' it queries its HKDF.Extract challenge oracle (keyed with ss_S) on 0 and uses the response as ES. If the response was the real shared secret, $\mathcal{B}_3$ has exactly simulated G_{A2} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_3$ has exactly simulated G_{A3} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A2}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A3}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_3}.$$

Game A4: replacing ETS and dES

In this game, we replace the values ETS and dES by uniformly random values in both sessions π_c and π_s. All values derived from dES in both sessions use the new value $\widetilde{\text{dES}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_4$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_4$ needs to compute ETS or dES in π_c or π_s it queries its HKDF.Expand challenge oracle (keyed with ES) with the corresponding labels and transcripts, and uses the responses. If the response was the real output, $\mathcal{B}_4$ has exactly simulated G_{A3} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_4$ has exactly simulated G_{A4} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A3}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_4}.$$

The stage-1 key ETS is now a uniformly random string independent of anything else in the game. It is, however, not forward-secure.

Game A5: ephemeral KEM

In this game, in session π_s we replace the ephemeral shared secret ss_e with a uniformly random $\widetilde{\mathsf{ss}_e}$. In π_c we replace ss_e with the same $\widetilde{\mathsf{ss}_e}$, but only if it received the same ct_e that π_s sent. If ss_e was replaced in a session by $\widetilde{\mathsf{ss}_e}$, that session will now derive anything originally derived from ss_e from $\widetilde{\mathsf{ss}_e}$ instead.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_5$ that breaks the IND-1CCA security of KEM_e. $\mathcal{B}_5$ obtains the IND-1CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and the challenge ciphertext $\mathsf{ss}^\star$. In π_c, it uses $\mathsf{pk}^\star$ in the ClientHello message. In the server session π_s $\mathcal{B}_5$ sends $\mathsf{ct}^\star$ in the ServerHello reply. It also sets $\mathsf{ss}^\star$ as the shared secret ss_e in π_s. If $\mathcal{A}_1$ sends $\mathsf{ct}^\star$ to π_c, $\mathcal{B}_5$ also sets ss_e to $\mathsf{ss}^\star$ in π_c. But if $\mathcal{A}_1$ sends any other $\mathsf{ct}' \neq \mathsf{ct}^\star$ to π_c, $\mathcal{B}_5$ uses its single query to the IND-1CCA decapsulation oracle to obtain π_c's shared secret.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_5$ has exactly simulated G_{A4} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_5$ has exactly simulated G_{A5} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A4}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{IND-1CCA}}_{\mathsf{KEM}_e,\mathcal{B}_5}.$$

Game A6: replacing HS

In this game we replace the handshake secret HS by a uniformly random $\widetilde{\text{HS}}$ in π_s. If π_c received the same ct_e that π_s sent, we also make a replacement

there. If HS was replaced in a session by $\widetilde{\text{HS}}$, that session will now derive anything originally derived from HS from its newly randomized value instead

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_6$ that breaks the PRF security of HKDF.Extract in its second argument. When $\mathcal{B}_6$ needs to compute HS in π_s (or π_c if it received the correct ct_e) it queries its HKDF.Extract challenge oracle (keyed with ss_e) on that session's dES and uses the response as HS. If the response was the real output, $\mathcal{B}_6$ has exactly simulated G_{A5} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_6$ has exactly simulated G_{A6} to $\mathcal{A}_1$. We obtain:

$$\text{Adv}^{G_{A5}}_{\mathcal{A}_1} \leq \text{Adv}^{G_{A6}}_{\mathcal{A}_1} + \text{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_6}.$$

Game A7: replacing CHTS, SHTS and dHS

In this game, we replace the handshake traffic secrets CHTS and SHTS and the value of dHS by uniformly random values in π_s. If π_c received the same ServerHello and ct_e that π_s sent, and π_s received the same ct_S, we make the same replacements there. If π_c shares the values for ct_e and ct_S (and thus the value of HS) with π_s, but did not receive the same ServerHello as was sent by π_s, we replace π_c's dHS by the same $\widetilde{\text{dHS}}$ as set in π_s, but π_c's CHTS and SHTS are set to independent uniformly random values. If dHS was replaced in a session by a $\widetilde{\text{dHS}}$, that session will now derive anything originally derived from dHS from $\widetilde{\text{dHS}}$ instead.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_7$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_7$ needs to compute CHTS, SHTS or dHS in π_s (or π_c, if it shares the values for ct_e and ct_S with π_s) it queries its HKDF.Expand challenge oracle (keyed with HS) with the corresponding labels and transcripts and uses the responses. If the responses are real values, $\mathcal{B}_7$ has exactly simulated G_{A6} to $\mathcal{A}_1$. If the responses are random values, $\mathcal{B}_7$ has exactly simulated G_{A7} to $\mathcal{A}_1$. Note that if π_c shares the values of ct_e and ct_S with π_s, but other parts of the ServerHello were changed such that π_s and π_c are no longer partnered at stage 2 or 3, the adversary may issue $\text{Reveal}(\pi_c, 2)$ or $\text{Reveal}(\pi_c, 3)$. But since any changes to ServerHello make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for CHTS and SHTS. This means the simulation in $\mathcal{B}_7$ remains good. We obtain:

$$\text{Adv}^{G_{A6}}_{\mathcal{A}_1} \leq \text{Adv}^{G_{A7}}_{\mathcal{A}_1} + \text{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_7}.$$

The stage-2 and stage-3 keys CHTS and SHTS are now uniformly random strings independent of anything else in the game. This means that these keys have been shown to have wfs1 security.

Game A8: replacing MS

In this game we replace main secret MS by a uniformly random value $\widetilde{\mathsf{MS}}$ in π_s. If π_c shares the value for dHS with π_s and, if π.mutualauth = true, received the same ct_C as π_s sent, we replace its MS with the same value. Otherwise, if the value for dHS is the same but ct_C is different in the two sessions, we set MS in π_c to an independent uniformly random value. All values derived from MS use these newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_8$ that breaks the PRF security of HKDF.Extract in its first argument (which we view as the "dual-PRF security"). When $\mathcal{B}_8$ needs to compute MS in π_s (or in π_c, if it shares the value for dHS with π_s) it queries its HKDF.Extract challenge oracle (keyed with dHS) with ss_C or 0 if π.mutualauth = false. It uses the response as MS. If the response is the real value, $\mathcal{B}_8$ has exactly simulated G_{A7} to $\mathcal{A}_1$. If it is a random value, $\mathcal{B}_8$ has exactly simulated G_{A8} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{A7}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A8}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_8}.$$

Game A9: replacing SATS, fk_s, fk_c, and CATS

In this game we replace the values SATS, fk_c, fk_s and CATS with uniformly random values in π_s. If π_c is a partner of π_s at stage 4, we also make identical replacements there. If it is not a partner but does share the value for MS with π_s, we replace π_c's SATS and CATS with independent uniformly random values, but replace fk_c and fk_s with the same values we used in π_s.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_9$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_9$ needs to compute SATS, fk_c, fk_s or CATS in π_s, it queries its HKDF.Expand oracle on the corresponding labels and transcripts. If $\mathcal{B}_9$ needs to compute any of those values in π_c, and π_c has the same value for MS, it does the same in π_c. It uses the responses in those sessions. If the responses are real values, $\mathcal{B}_9$ has exactly simulated G_{A8} to $\mathcal{A}_1$. If the responses are random values, $\mathcal{B}_9$ has exactly simulated G_{A9} to $\mathcal{A}_1$. Note that if π_c had the same value for MS as π_s, but parts of the transcript were changed such that π_s and π_c are no longer

partnered at stage 4, the adversary may issue $\mathsf{Reveal}(\pi_c, 4)$. But since any such changes make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for SATS. Similarly, if π_c had the same value for MS as π_s, but parts of the transcript were changed such that π_s and π_c are no longer partnered at stage 5, the adversary may issue $\mathsf{Reveal}(\pi_c, 5)$. But since any such changes make the transcript in π_s and π_c different, the label input to HKDF.Expand is now different for CATS. This means the simulation in $\mathcal{B}_9$ remains good. We obtain:

$$\mathsf{Adv}^{G_{A8}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{A9}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_9}.$$

The stage-4 key SATS and stage-5 key CATS are now uniformly random strings independent of everything else in the game. This means that the stage-4 and stage-5 keys have been shown to have wfs1 security.

Malicious acceptance

Let bad denote the event that G_{A9} maliciously accepts in stage j in the (fresh) tested session without a session partner in stage j. If the tested session is a client session, $j = 4$. Otherwise, if the session is a server and $\pi.\mathsf{mutualauth} = \mathsf{true}$, then $j = 5$. In server sessions that do not authenticate the client, we do not have explicit authentication.

Game A10: identical-until-bad

This game is identical to game G_{A9}, except that we abort the game if the event bad occurs. Games G_{A9} and G_{A10} are identical-until-bad [33]. Thus,

$$|\Pr[G_{A9} \Rightarrow 1] - \Pr[G_{A10} \Rightarrow 1]| \leq \Pr[G_{A10} \text{ reaches bad}].$$

In game G_{A9}, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary cannot distinguish stage keys anymore. By this game, it can no longer reach bad. Thus:

$$\mathsf{Adv}^{G_{A10}}_{\mathcal{A}_1} = 0.$$

It remains to bound $\Pr[G_{A10} \text{ reaches bad}]$.

Game A11: HMAC forgery

In this game, π_c, if it does not have a session partner in stage 4, rejects upon receiving the ServerFinished message. If π_s.mutualauth = true and π_s does not have a session partner in stage 5, π_s rejects upon receiving the ClientFinished message.

Any adversary that behaves differently in G_{A11} compared to G_{A10} can be used to construct an HMAC forger $\mathcal{B}_{10}$. The only way that G_{A10} and G_{A11} behave differently is if G_{A11} rejects a MAC that should have been accepted as valid. When rejecting ServerFinished, if no partner to π_c at stage 4 exists, no honest π_s exists with the same session identifier and thus transcript. This means no honest π_s ever created a MAC tag for the transcript that the client verified, and thus it must be a forgery. When rejecting ClientFinished, if no partner to π at stage 5 exists, no honest π_c exists with the same session identifier and thus transcript. This means no honest π_c ever created a MAC tag for the transcript that the server verified, and thus it must be a forgery. Concluding:

$$\Pr\left[G_{A10} \text{ reaches bad}\right] \leq \Pr\left[G_{A11} \text{ reaches bad}\right] + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{10}}.$$

By the above, the event bad is never reached.

Analysis of game A11

By game A9, all stage keys are uniformly random and independent of all messages in the game. By game A11, all events bad are rejected. Thus:

$$\Pr\left[G_{A11} \text{ reaches bad}\right] = 0.$$

This concludes case A, yielding:

$$\mathsf{Adv}^{G_A}_{\mathcal{A}_1} \leq n_s \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_2} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_3} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_4} & +\mathsf{Adv}^{\text{IND-1CCA}}_{\text{KEM}_e,\mathcal{B}_5} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_6} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_7} \\ +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_8} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_9} \\ +\ 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{10}} & \end{array} \right).$$

Case B: no contributive partner in stage 2 exists, and peer is not corrupted before stage *i* accepted

In this case, the tested session π does not have a contributive partner in stage 2. This means that stages aiming for wfs1 security are out of scope of this case. If π.mutualauth = false, the tested π can be assumed to a client session. Otherwise, it can both be a server or a client session.

We also allow the intended peer V of the tested session π to be corrupted, but not before the tested session accepted the tested stage selected by game 3. This models forward secrecy: even if the adversary obtains the peer's long-term key, the tested keys should still be indistinguishable.

Allowing Corrupt in this case means that any reduction that replaces ss_C or ss_S is problematic. However, we show by assumption on the EUF-CMA security of HMAC that no client can be made to maliciously accept at stage 4 and no server session at stage 5. This means that if a client accepts in stage 4, then it has a partner at stage 4 and all prior stages. Similarly, if a server accepts in stage 5, then it has a partner at stage 5 and all prior stages.

This allows us to make the following conclusions. Once stage 4 accepts, all client stages are retroactively authenticated. Once stage 5 accepts, all server stages are retroactively authenticated. By case A, all stage keys are indistinguishable, even to an adversary that corrupts any long-term key. This yields retroactive fs security for all stage keys.

Game B1: guessing the intended peer

In this game, we attempt to guess the identity of the peer with which the tested session attempts to connect. If we do not guess this identity V correctly, i.e., this identity $V \neq \pi.\mathsf{pid}$, we abort. This reduces the advantage of $\mathcal{A}_1$ by a factor of the number of users n_u:

$$\mathsf{Adv}^{G_B}_{\mathcal{A}_1} \leq n_u \cdot \mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1}.$$

Game B2: static KEM

In this game we replace the shared secret ss_S in π, a client session, with a uniformly random $\widetilde{\mathsf{ss}_S}$. In any (server) sessions π' of V that received the same ct_S as was sent by π in the ClientHello message, we replace the value of ss_S with the same $\widetilde{\mathsf{ss}_S}$. All values derived from ss_S in π or the sessions of V that received the same ct_S use the new value $\widetilde{\mathsf{ss}_S}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct

an adversary $\mathcal{B}_{11}$ that breaks the IND-CCA security of KEM_s. $\mathcal{B}_{11}$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and challenge shared secret $\mathsf{ss}^\star$ and gives $\mathsf{pk}^\star$ to $\mathcal{A}_1$. In π, $\mathcal{B}_{11}$ sends $\mathsf{ct}^\star$ in the ClientHello message and uses $\mathsf{ss}^\star$ for ss_S. If $\mathcal{A}_1$ delivers $\mathsf{ct}^\star$ to any π' of V, $\mathcal{B}_{11}$ uses $\mathsf{ss}^\star$ as value for ss_S in π'. If $\mathcal{A}_1$ delivers some other $\mathsf{ct}' \neq \mathsf{ct}^\star$, $\mathcal{B}_{11}$ queries its IND-CCA decapsulation oracle with ct' to obtain the required shared secret. By the definition of case B, we will never need to answer any Corrupt(V) queries.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_{11}$ has exactly simulated G_{B1} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{11}$ has exactly simulated G_{B2} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B1}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}_s,\mathcal{B}_{11}}.$$

Game B3: replacing ES

In this game, we replace the early handshake secret ES by a uniformly random value $\widetilde{\mathrm{ES}}$. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacement.

Any $\mathcal{A}_1$ that can detect this change can be used to construct an adversary $\mathcal{B}_{12}$ that breaks the PRF security of HKDF.Extract in its second argument as follows. When $\mathcal{B}_{12}$ needs to compute ES in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, $\mathcal{B}_{12}$ uses its HKDF.Extract challenge oracle (keyed with ct_S) on 0. It uses the response as ES. If the responses the real values, $\mathcal{B}_{12}$ has exactly simulated G_{B2} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{12}$ has exactly simulated G_{B3} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B2}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B3}}_{\mathcal{A}_1} + \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{12}}.$$

Game B4: replacing ETS and dES

In this game we replace the values ETS and dES by uniformly random values in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make the same replacements. All values derived from dES in π and the π' sessions of V that made the replacements use the new value $\widetilde{\mathrm{dES}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{13}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{13}$ needs to compute ETS or dES in π or any of the sessions of V that received

or sent the same ct_S that was sent or received by π, it queries its HKDF.Expand challenge oracle (keyed with ES) with the corresponding labels and transcripts and uses the responses. If the response was the real shared secret, $\mathcal{B}_{13}$ has exactly simulated G_{B3} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{13}$ has exactly simulated G_{B4} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B3}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_{13}}.$$

The stage-1 key ETS is now a uniformly random string independent of anything else in the game. It is, however, not forward-secure.

Game B5: replacing HS

In this game we replace the value of HS by a uniformly random value $\widetilde{\mathsf{HS}}$ in π. Additionally, in any sessions π' of V which either sent or received the same ct_S that was sent or received in π, we make a similar replacement. All values derived from HS in π and the π' of V that made the replacement use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{14}$ that breaks the PRF security of HKDF.Extract in its first argument (which we view as the "dual-PRF security"). When $\mathcal{B}_{14}$ needs to compute HS in π or any of the sessions of V that received or sent the same ct_S that was sent or received by π, it queries its HKDF.Extract challenge oracle (keyed with dES) with that session's ss_e and uses the response. If the response was the real shared secret, $\mathcal{B}_{14}$ has exactly simulated G_{B4} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{14}$ has exactly simulated G_{B5} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B4}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B5}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{14}}.$$

Game B6: replacing CHTS, SHTS and dHS

In this game we replace the values CHTS and SHTS and dHS by uniformly random values in π. Additionally, in any sessions π' of V which share the value of HS (and thus ct_S and ct_e) with π, we make the same replacements. All values derived from dHS in π and the π' sessions of V that made the replacements use the new value $\widetilde{\mathsf{dHS}}$.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{15}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{15}$ needs to compute any of CHTS, SHTS or dHS in π or any of the sessions of

V shares the value of HS with π, it queries its HKDF.Expand challenge oracle (keyed with HS) on the corresponding labels and transcripts and uses the responses. If the response was the real shared secret, $\mathcal{B}_{15}$ has exactly simulated G_{B5} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{15}$ has exactly simulated G_{B6} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B5}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B6}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{15}}.$$

The stage-2 and stage-3 keys CHTS and SHTS in π are now uniformly random independent of anything else in the game. Thus, they have been shown to have wfs2 security in client sessions. Recall that server sessions aim for wfs1 security in this stage, which is out of scope.

Game B7: client authentication static KEM

We only play this game if π.mutualauth = true. Otherwise, this game is equal to the previous one as there is no reduction in advantage.

We replace the client authentication shared secret ss_C in π, a server, by a uniformly random value $\widetilde{\mathsf{ss}_C}$. If any of V's client sessions π' received the same ct_C as π sent in `ServerKemCiphertext`, we make the same replacement in those π' if they have π'.mutualauth = true. Any value derived from ss_C in a session where it was replaced will now use the replacement value $\widetilde{\mathsf{ss}_C}$. Sending any `ServerKemCiphertext` to π' with π'.mutualauth = false will simply terminate those sessions at no advantage to the adversary.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{16}$ that breaks the IND-CCA security of KEM_c. $\mathcal{B}_{16}$ obtains the IND-CCA challenge $\mathsf{pk}^\star$, $\mathsf{ct}^\star$ and the challenge ciphertext $\mathsf{ss}^\star$. $\mathcal{B}_{16}$ uses $\mathsf{pk}^\star$ in the `ClientCertificate` message sent in V's client sessions. In π, $\mathcal{B}_{16}$ uses $\mathsf{ct}^\star$ in the `ServerKemCiphertext` message and sets $\mathsf{ss}^\star$ as the shared secret ss_C. If $\mathcal{A}_1$ sends $\mathsf{ct}^\star$ to any of V's π', $\mathcal{B}_{16}$ also sets ss_C to $\mathsf{ss}^\star$ in those π'. But if $\mathcal{A}_1$ sends any other $\mathsf{ct}' \neq \mathsf{ct}^\star$ to any of V's π', $\mathcal{B}_{16}$ uses the IND-CCA decapsulation oracle to obtain the appropriate shared secret.

In the end, $\mathcal{A}_1$ terminates it outputs its guess of the uniform bit b. If $\mathsf{ss}^\star$ was the real shared secret, $\mathcal{B}_{16}$ has exactly simulated G_{B6} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{16}$ has exactly simulated G_{B7} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B6}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B7}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{IND-CCA}}_{\mathsf{KEM}_c,\mathcal{B}_{16}}.$$

Game B8: replacing MS

In this game, we replace the value of main secret MS by a uniformly random value $\widetilde{\mathsf{MS}}$ in π. Additionally, in any sessions π' of V which share the value for dHS with π, we make the same replacement. All values derived from MS in π and the π' of V that made the replacement use the newly randomized values.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{17}$ that breaks the PRF security of HKDF.Extract in its first argument. When $\mathcal{B}_{17}$ needs to compute MS in π or any of the sessions of V that have the same value for dHS as π, it queries its HKDF.Extract challenge oracle (keyed with dHS) and uses the response as MS. In each of these sessions π'', if π''.mutualauth = false, $\mathcal{B}_{17}$ calls HKDF.Extract with 0. If π''.mutualauth = true, $\mathcal{B}_{17}$ calls HKDF.Extract with ss_C. If the response was the real shared secret, $\mathcal{B}_{17}$ has exactly simulated G_{B7} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{17}$ has exactly simulated G_{B8} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B7}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B8}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{dual-PRF-sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_{17}}.$$

Game B9: replacing SATS, fk_s, fk_c and CATS

In this game we replace the application traffic secrets SATS and CATS, and finished keys fk_s and fk_c by uniformly random values in π. Additionally, in any sessions π' of V which share the value of MS with π we make the same replacements.

Any adversary $\mathcal{A}_1$ that can detect this replacement can be used to construct an adversary $\mathcal{B}_{18}$ that breaks the PRF security of HKDF.Expand. When $\mathcal{B}_{18}$ needs to compute SATS, CATS, fk_s or fk_c in π or any of the sessions of V that share the value for MS with π, it queries its HKDF.Expand challenge oracle (keyed with MS) with the corresponding labels and transcripts and uses the responses. If the response was the real shared secret, $\mathcal{B}_{18}$ has exactly simulated G_{B8} to $\mathcal{A}_1$. If it was a random value, $\mathcal{B}_{18}$ has exactly simulated G_{B9} to $\mathcal{A}_1$. We obtain:

$$\mathsf{Adv}^{G_{B8}}_{\mathcal{A}_1} \leq \mathsf{Adv}^{G_{B9}}_{\mathcal{A}_1} + \mathsf{Adv}^{\text{PRF-sec}}_{\mathsf{HKDF.Expand},\mathcal{B}_{18}}.$$

The stage-4 key SATS and stage-5 key CATS in the tested session π are now uniformly random independent of anything else in the game. Thus, they have been shown to have wfs2 security. If π is a server session, and π.mutualauth = false, wfs2 security of SATS is out of scope.

Malicious acceptance

Let bad denote the event that G_{B9} maliciously accepts in stage j in the (fresh) tested session without a session partner in stage j. If the session is a client, then $j = 4$. If the tested session is a server and π.mutualauth = true we have $j = 5$. In server sessions that do not authenticate the client, we do not have explicit authentication.

Game B10: identical-until-bad

This game is identical to game G_{B9}, except that we abort the game if the event bad occurs. Games G_{B9} and G_{B10} are identical-until-bad [33]. Thus,

$$|\Pr[G_{B9} \Rightarrow 1] - \Pr[G_{B10} \Rightarrow 1]| \leq \Pr[G_{B10} \text{ reaches bad}].$$

In game G_{B9}, all stage keys in the tested session are uniformly random and independent of all messages in the game. The adversary cannot distinguish stage keys anymore. By this game, it can no longer reach bad. Thus:

$$\mathsf{Adv}^{G_{B10}}_{\mathcal{A}_1} = 0.$$

It remains to bound $\Pr[G_{B10} \text{ reaches bad}]$.

Game B11: HMAC forgery

In this game, π, if it is a client, rejects upon receiving the `ServerFinished` message. If π is a server and π.mutualauth = true, π rejects upon receiving the `ClientFinished` message.

Any adversary that behaves differently in G_{B11} compared to G_{B10} can be used to construct an HMAC forger $\mathcal{B}_{19}$. The only way that G_{B10} and G_{B11} behave differently is if G_{B11} rejects a MAC that should have been accepted as valid. When rejecting `ServerFinished`, no partner to π at stage 4 exists, so no honest server session π' exists with the same session identifier and thus transcript. No honest π' ever created a MAC tag for the transcript that the client verified, and thus it must be a forgery. When rejecting `ClientFinished`, no partner to π at stage 5 exists, so no honest client session π' exists with the same session identifier and thus transcript. Concluding:

$$\Pr[G_{B10} \text{ reaches bad}] \leq \Pr[G_{B11} \text{ reaches bad}] + 2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{19}}.$$

Since this game rejects all ServerFinished messages, the event bad is never reached in client sessions. If π.mutualauth = true, this game also rejects all ClientFinished messages. If π.mutualauth = false, rejecting ClientFinished is out of scope as we only aim for wfs1 security.

Analysis of game B11

By game B9, all stage keys are uniformly random and independent of all messages in the game. By game B11, all events bad are rejected. Thus:

$$\Pr\left[G_{B11} \text{ reaches bad}\right] = 0.$$

This concludes case B, yielding:

$$\mathsf{Adv}^{G_B}_{\mathcal{A}_1} \leq n_u \left(\begin{array}{ll} \mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_s,\mathcal{B}_{11}} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{12}} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{13}} & +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{14}} \\ +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{15}} & +\mathsf{Adv}^{\text{IND-CCA}}_{\text{KEM}_c,\mathcal{B}_{16}} \\ +\mathsf{Adv}^{\text{dual-PRF-sec}}_{\text{HKDF.Extract},\mathcal{B}_{17}} & +\mathsf{Adv}^{\text{PRF-sec}}_{\text{HKDF.Expand},\mathcal{B}_{18}} \\ +2\mathsf{Adv}^{\text{EUF-CMA}}_{\text{HMAC},\mathcal{B}_{19}} & \end{array} \right).$$

Combining the bounds in cases A and B yields the theorem. □

8.3 Security of the two protocols

We have now proven the security of KEMTLS and KEMTLS-PDK under the same definitions of Match and Multi-Stage security. In chapter 9, we encode the security properties described in chapter 7 and this chapter in Tamarin, a symbolic protocol analysis tool. The Tamarin model analyses our security claims for both protocols in each other's presence. We will also informally argue that both protocols remain secure in each other's presence.

The arguments of our Match security proofs are compatible: the definitions of (contributive) session identifiers are distinct for KEMTLS and KEMTLS-PDK, as all session identifiers of the latter contain the ServerCertificate message before the transcript. In our Multi-Stage proofs, transcripts of sessions remain unique by the same assumptions that rule out nonce and hash collisions. Both protocols use the same primitives. Any adversary that delivers a (part of a) message from a KEMTLS session to a KEMTLS-PDK session, or vice-versa, can be

handled in the same way as when such (parts of) messages are delivered to a different session of the same protocol. As a result, we can amend our game steps to consider the presence of concurrent KEMTLS(-PDK) sessions.

The fallback mechanism we described in section 6.2.2 has some more subtleties we need to address, and we will only sketch how to approach them. Further analysis of the co-existence of the two protocols remains future work.

We note that the KEMTLS-PDK negotiation mechanisms in ClientHello extensions are part of the transcript. As a result, KEMTLS-PDK sessions cannot be downgraded to KEMTLS, as the transcript computations would fail: the original ClientHello, including its extensions, remains part of the transcript. A rejection of proactive client authentication is more nuanced. Dropping the encrypted handshake message could result in the client and server disagreeing on what was attempted. However, in KEMTLS(-PDK), both parties always know if client authentication was completed through the presence of the ServerKemCiphertext message that contains the ciphertext that the server encapsulated to the client certificate. Additionally, the specification of the proactive client authentication can include an additional ClientHello extension that indicates if a proactive ClientCertificate message will be sent. (In fact, our implementations do this, just for ease of handling the additional message.) Even if the ClientCertificate message is rejected by the server and dropped from the transcript, this extension would then remain part of the transcript to indicate it was present.

9 Formally analyzing KEMTLS in Tamarin

In the previous chapters, we have stated the security properties of KEMTLS and KEMTLS-PDK, and shown how to prove the security of the protocols in the reductionist security model. Pen-and-paper methods are however not the only way to analyze the security of a cryptographic protocol. Computer-aided tools are increasingly used to verify the security of cryptographic protocols and implementations. How computers can help ranges from just checking a human's work, to completely automated analysis, providing attacks or proofs of security. Even within this wide range, a major common characteristic of computer-aided analysis tools is a computer's rigor: it will never read between the lines or gloss over a particular modeling detail.

Contributions

In this chapter, we show how we analyzed the security of KEMTLS and KEMTLS-PDK in Tamarin, a security protocol verification tool that works in the symbolic model. We model KEMTLS(-PDK) twice, in very different styles: once by extending an existing, very detailed model of TLS 1.3; and once by translating our security models and properties from chapters 7 and 8 to Tamarin's modeling language. The two models have very different performance characteristics and allow us to examine the security of the protocols from different perspectives as the models describe different security properties. We can also compare these models, allowing us to comment on the trade-off in symbolic analysis between detail in protocol specification and granularity of security properties.

9.1 Introduction

During the development process of TLS 1.3, there was a strong collaboration between the standardization community with the academic research community. Initial TLS 1.3 protocol designs were based on academic designs [221],

and it was an explicit goal of the TLS 1.3 process to incorporate academic security analysis of new designs before continuing with standardization. Paterson and Van der Merwe described this as a "design-break-fix-release" process rather than the "design-release-break-patch" cycle that was found on prior versions of the standardization and usage of TLS [287]. Many of the security analyses of TLS 1.3 used the reductionist security paradigm [127, 128, 129, 213, 221]. Complementing this manual proof work, computer-aided cryptography [20] was also instrumental in checking TLS 1.3. Analyses were done using the ProVerif [55] and Tamarin [108, 109] symbolic analysis tools, as well as a verified implementation in F* [115].

The proofs of KEMTLS(-PDK) in the initial papers and discussed in the previous chapters adapt the multi-stage key exchange approach used by Dowling, Fischlin, Günther, and Stebila [127, 128] for TLS 1.3. Subsequently, Günther, Rastikian, Towa, and Wiggers proposed and proved an alternative abbreviated handshake,[1] with additional short-lived static keys [168], and found a few minor mistakes in the original security proofs, which were subsequently fixed in online versions of the original papers [320, 321] and the proofs presented in chapters 7 and 8. All of these proofs treat protocol modes independently—one at a time—and do not consider the presence of the other protocol modes.

9.1.1 Contributions

In this chapter, we present two security analyses of all four variants of KEMTLS (the base KEMTLS protocol, with server-only or mutual authentication, and the pre-distributed public keys variant KEMTLS-PDK, also with server-only or mutual authentication) using Tamarin [27, 251].

Our first model, presented in section 9.3, is based on the Tamarin analysis of TLS 1.3 by Cremers, Horvat, Hoyland, Scott, and Van der Merwe [108]. This is a highly detailed model in terms of the protocol specification, closely following the TLS 1.3 wire format. In this model, we show that all KEMTLS variants have equivalent security properties to the main handshake of TLS 1.3 without extensions. We were able to fully automate the proof, unlike the original (though more fully-featured) model which required significant manual effort.

Our second model, presented in section 9.5, is a novel Tamarin model developed from scratch that closely follows the multi-stage key exchange

[1] We briefly discussed this variant of KEMTLS-PDK in section 6.3

security model used in the pen-and-paper proofs. This model focuses on the "cryptographic core", meaning that it is further away from the wire specification and does not model details like message encryption or the record layer. However, it captures more details in the security definitions, using the granular definitions of forward secrecy from [320, 321]. We also analyze the deniability properties. The second model allows us to symbolically verify the forward secrecy and authentication properties specified in the pen-and-paper proofs but goes further by considering all KEMTLS variants simultaneously. This Tamarin model allowed us to identify some minor flaws in the properties stated based on pen-and-paper proofs.

In section 9.7, we compare the features of our two Tamarin models. Having these two models side-by-side illustrates the trade-off between the detail of protocol specification and the granularity of security properties. Ideally, of course, one would achieve both levels of detail simultaneously, but such complexity is challenging both for the humans reading and writing pen-and-paper proofs or authoring Tamarin models, and for computers checking such Tamarin models (where runtime typically scales exponentially with the complexity of the model). Our side-by-side approach with two very different perspectives still yields significant confidence in the soundness of the KEMTLS protocol design and each provides insight into flaws in the earlier models that it was based on.

9.2 Background on symbolic analysis

One approach to proving the security properties of protocols is *symbolic analysis*, which uses formal logic to reason about the properties of an algebraic model of a protocol. Computational tools, such as Tamarin [27, 251] or ProVerif [64], can then be used to check whether certain properties hold in the symbolic model.

In a reductionist analysis, like our pen-and-paper proofs, adversaries are polynomially bounded, and we reduce the security of a protocol to assumptions on the underlying cryptographic primitives, such as hash collisions. These allow us to give a fine-grained analysis through the algebraic properties of the schemes. In symbolic analysis however, generic symbols replace specific values. Operations like encryption are also modeled symbolically: for example, the symbol `senc(a,b)` represents the value `a` being symmetrically

encrypted with the key b. This means that in symbolic analysis cryptographic operations are *perfect*, meaning the adversary can learn nothing about an encrypted message without the correct key. The operations that describe a protocol in a symbolic model take messages and state information and transform them into the next state or emit another protocol message. A tool can then use all operations and symbols to generate every possible protocol run: symbolic adversaries have unbounded numbers of sessions and time.

Many symbolic analyses of protocols use the Dolev–Yao [125] attacker model, in which an attacker can manipulate all messages at will, e.g., by redirecting them, replaying them, dropping them, or manipulating their contents. It can also construct new messages from information previously learned. However, as the cryptographic primitives in symbolic models are assumed to be perfect, the attacker cannot read or modify encrypted or authenticated messages if it does not have the right keys.

Symbolic models can also be extended to give the attacker special extra abilities. For example, one can allow the attacker to reveal private keys or state information of parties through reveal oracles. We record when the attacker uses this oracle, so reveal queries become part of the trace of execution.

Security properties are modeled as predicates over execution traces. In Tamarin, during the execution of the rules of the protocol, we can emit *action facts*. We use these action facts to record, for example, the session's impression of the authentication status or the current keys. We then write lemmas representing security properties as predicates over action facts. For example, a lemma may state that any key recorded in a certain type of action fact must not be known to the adversary unless the adversary cheated by revealing keys. A model checker like Tamarin can then be used to check if the protocol maintains the required security property. Assuming the tool is sound, either the tool will give a proof that the protocol has the required property, find a counter-example, or fail to terminate.

9.3 Model #1: high-resolution protocol specification

In this section, we discuss the natural approach of taking one of the TLS 1.3 models and adapting it to KEMTLS(-PDK). Our work demonstrates that KEMTLS provides security guarantees at least equivalent to those proven by Cremers, Horvat, Hoyland, Scott, and Van der Merwe for the main handshake of TLS 1.3.

9.3.1 The TLS 1.3 model

The Tamarin model of TLS 1.3 [108] is very high-resolution in terms of its modeling of protocol details and adherence to the protocol specification. It covers the cryptographic computations such as the key exchange and the key schedule; for example, calls to HKDF are decomposed into hash function calls. This model also includes the extensions to the basic TLS 1.3 handshake, such as the `HelloRetryRequest` mechanism, pre-shared keys, and resumption via session tickets. Additionally, it models the encryption of handshake messages, the syntax of the protocol messages, and mechanics such as TLS 1.3 extensions.

In terms of security properties, the TLS 1.3 model extends Tamarin's basic Dolev–Yao attacker with the ability to recover secrets from Diffie–Hellman key shares and to reveal the long-term keys of participants. TLS 1.3 is not secure against an attacker who can use these attacks freely but aims to provide confidentiality and integrity against an attacker who is restricted from revealing secrets of the target session. The TLS 1.3 model encodes lemmas capturing most of the security properties claimed by the TLS 1.3 specification [298, Appendix E.1]. They report that proving all lemmas in their model took about a week. Much of this time was spent on manual interaction with Tamarin's prover to guide it to prove some of the more complex lemmas. Verifying the generated proof requires "about a day" and "a vast amount of RAM" [108].

9.3.2 Representing KEMTLS in the model

We now describe how we modified the existing TLS 1.3 model to represent both KEMTLS and its variant with pre-distributed keys, KEMTLS-PDK. The original model is highly modular, which made it relatively easy to modify.

Modeling KEMs

Tamarin does not have a built-in interface to model KEMs. They can be described using Tamarin's asymmetric encryption primitives, as was done in [185]. We choose to model the KEM interface using Tamarin's function API. As the TLS 1.3 model has some support for cryptographic agility in the ephemeral key exchange, we add a public algorithm identifier symbol to each operation. We use fresh values to resemble KEM secret keys, and a function `kempk/2` to represent the public key for the specified algorithm. Tamarin's functions are just symbols and do not describe functionality. Any functionality is handled by writing equations over the symbols. As such, we cannot

easily return two values or generate fresh values in the encapsulate operation. We resolve this by providing an input to the encapsulate operation to stand in for the generated secret. The shared secret is defined through `kemss/2` and also contains the algorithm symbol. We define `kemencaps/3` (KEM encapsulation) over the algorithm, the shared secret, and the public key. The resulting ciphertext can be provided to `kemdecaps/3` (KEM decapsulation) with the algorithm and the secret key. We model the functionality of `kemencaps` and `kemdecaps` by the equation pictured in listing 9.1, where `alg` represents the KEM algorithm, and `seed` and `sk` are the fresh input values.

Listing 9.1: The equation used to model KEMs in model #1

```
equations:
  kemdecaps(alg,
            kemencaps(alg,
                      kemss(alg, seed),
                      kempk(alg, sk)),
            sk)
  = kemss(alg, seed)
```

Modelling KEMTLS

The model of Cremers, Horvat, Hoyland, Scott, and Van der Merwe represents TLS 1.3 through rules that manipulate a specific state object, which keeps track of many protocol variables, such as keys, authentication status, and the currently active handshake mode. Tamarin rules create transitions between these states. Where the protocol branches, such as when the server requests client authentication by sending CertificateRequest, two rules end up in the same next state; in this example, they set the `cert_req` variable differently. The server later uses this variable to decide which of the rules `recv_client_auth` or `recv_client_auth_cert` to use; the latter expects the Certificate, CertificateVerify, and ClientFinished messages, while the former only expects Finished. We handle the public-key infrastructure for KEM public keys in the same way as [108]: we do not model CA certificates and assume an out-of-band binding between public keys and identities.

Ephemeral key exchange in the TLS 1.3 model relies on Tamarin's built-in DH functionality. It also allows the negotiation of two different DH groups.

During the handshake, the client and server generate ephemeral DH secrets for the chosen group. If the server rejects the client's choice of DH group, it falls back to another group through the HelloRetryRequest mechanism. To model the post-quantum ephemeral key exchange in KEMTLS, we replaced the Diffie–Hellman operations by `kemencaps` in place of the server's DH key generation. The client then computes the shared secret via `kemdecaps`.

The authentication rules and states required more careful consideration. In the TLS 1.3 model, the Certificate, CertificateVerify, and Finished messages were sent and received simultaneously. In KEMTLS, we split the handling of these messages, as the peer that is authenticating needs to first receive a ciphertext to decapsulate. Doing this requires more states. Additionally, in KEMTLS, the client sends Finished before the server, which deviates from TLS 1.3. For a diagram of the state machine, we refer to the extended version of the paper is chapter is based on [93, App. B].

To finish our integration of KEMTLS, we made changes to the key schedule to include the computation of the KEMTLS Authenticated Handshake Secret (AHS) and use the correct handshake traffic encryption keys. We also modified the action facts emitted in the various rules to match our KEM operations; lemmas that made use of these action facts were also updated. We disabled the PSK and session ticket features of the original model.

Modeling KEMTLS-PDK

In KEMTLS-PDK, the client has the server's long-term public key beforehand. Access to the public key allows the client to send a ciphertext in the initial ClientHello message. Additionally, the client may attempt client authentication proactively and thus transmit its Certificate before receiving ServerHello from the server. We model this through an additional initial state for the KEMTLS-PDK client. From this state, there are two rules which set the state variable that will decide if the client will send its certificate. KEMTLS-PDK is otherwise implemented as a mostly separate sequence of states and rules, as the key schedule and order of messages are quite different. The client and server still transition through a state shared with KEMTLS, so they can fall back to the "full" handshake.

9.3.3 Security properties

We adapt the lemmas from the Tamarin model for TLS 1.3. Many core lemmas are constructed around the `SessionKey` fact: the client and the server record this fact when the handshake concludes. `SessionKey` contains the actor's final understanding of its and its peer's identities, authentication statuses, and the application traffic keys. We prove all security properties discussed in [108] and briefly explain the most important of these below.

Adversary compromise of secrets

First, we note the extent to which the adversary can compromise ephemeral or long-term secrets. KEMTLS uses ephemeral KEM keys for ephemeral secrecy and long-term KEM keys for authentication. The adversary can reveal actors' long-term secret keys; this records the `RevLtk($actor)` fact. We also allow revealing the ephemeral secret key in individual sessions, recording the `RevEKemSk(tid, $actor, esk)` fact. Variables `tid` ("thread identifier") and `esk` ("ephemeral secret key") track the specific session and secret key.

KEMs are not "symmetric" in the same way that DH key exchange is. Only one party in each KEM key exchange has a secret key that can be targeted by a reveal query. We do not model revealing the shared secret from the ciphertext.

Intermediate session keys, like the Main Secret (MS), cannot be revealed directly. This follows from the design of the original model: in TLS 1.3, these secrets only depend on the ephemeral key exchange, so revealing the ephemeral key exchange in sessions not targeted by a lemma still allows the adversary to obtain those sessions' intermediate session keys. In KEMTLS, this is no longer the case: we mix the shared secrets encapsulated against long-term keys into the key schedule; as a result, our attacker is slightly weaker. The model discussed in section 9.5 does directly allow session key reveal.

(Forward) secrecy of session keys

The outputs of the handshake, as recorded in the `SessionKey` fact, are the application traffic read and write keys `kr` and `kw`. We require these keys to remain secret against various forms of attacks. Forward secrecy requires that if the long-term keys (but not the ephemeral keys) were compromised after the session completes, the session keys remain secure.

We model this in the `secret_session_keys` lemma as shown in listing 9.2. This lemma considers a client or server that believes it has authenticated its peer, where the attacker has not revealed the ephemeral KEM secret

Listing 9.2: The `secret_session_keys` lemma proves application traffic keys are secret.

```
lemma secret_session_keys:
  "All tid actor peer kw kr pas #i.
    SessionKey(tid, actor, peer,
               <pas, 'auth'>, <kw, kr>)@#i &
    not (Ex #r. RevLtk(peer)@#r & #r < #i) &
    not (Ex tid3 esk #r.
          RevEKemSk(tid3, peer, esk)@#r & #r < #i) &
    not (Ex tid4 esk #r.
          RevEKemSk(tid4, actor, esk)@#r & #r < #i)
    ==> not Ex #j. K(kr)@#j"
```

keys. We allow the attacker to reveal the peer's long-term secret key, but only after the `SessionKey` fact was emitted; this is the "forward" secrecy aspect. The attacker should not be able to learn (`not Ex #j. K(kr)@#j`) the target's read key `kr` under these constraints. We similarly prove forward secrecy for each of the intermediate keys in the key schedule: the Handshake Secret (HS), AHS, and MS.

Note that in KEMTLS, the session keys are derived from not just the ephemeral key exchange as in TLS 1.3, but also include the secret encapsulated during the authentication phase of the handshake. This implies that both the ephemeral key and the server's long-term key need to be compromised in client sessions, and the ephemeral key and the server's long-term key in server sessions with mutual authentication. We prove this in our model through a variant of the `secret_session_keys` lemma that allows ephemeral key compromise, as long as the peer's long-term key is never revealed. This lemma is shown in listing 9.3.

Authentication

We model the authentication properties of KEMTLS in the same way as they were modeled for TLS 1.3. The client and server are partnered via the nonces exchanged in the initial messages. The `entity_authentication` lemma captures that if the client, at the end of the handshake protocol, has authenticated their peer, and the peer's long-term keys have not been revealed, then there must be a peer session that started with the same nonces. This lemma

Listing 9.3: The secret_session_keys_ephem_reveal lemma proves application traffic keys are secret even if ephemeral keys are revealed.

```
lemma secret_session_keys_ephem_reveal:
  "All tid actor peer kw kr pas #i.
    SessionKey(tid, actor, peer,
               <pas, 'auth'>, <kw, kr>)@#i &
    not (Ex #r. RevLtk(peer)@#r) &
    ==> not Ex #j. K(kr)@#j"
```

Listing 9.4: The entity_authentication lemma proves that if a client commits to a set of nonces, there is a server that's running with the same nonces.

```
lemma entity_authentication [use_induction]:
  "All tid actor peer nonces cas #i.
    commit(Nonces, actor, 'client', nonces)@#i &
    commit(Identity, actor, 'client',
           peer, <cas, 'auth'>)@#i &
    not (Ex #r. RevLtk(peer)@#r & #r < #i)
    ==> Ex tid2 #j_ea. #j_ea < #i &
          running2(Nonces, peer, 'server', nonces)@#j_ea"
```

is shown in listing 9.4. The lemma mutual_entity_authentication states the same, but with the roles of client and server reversed. As these lemmas allow revealing the targeted actor's long-term keys, these properties also cover key-compromise impersonation attacks. Similarly, in the lemma transcript_agreement, we prove that when the client, after receiving the server's Finished message, commits to a transcript, there exists a server that is running with the same transcript (or their long-term keys have been revealed). The mutual authentication case is stated by the mutual_transcript_agreement lemma, which has the roles reversed.

In TLS 1.3, the verification of the handshake signature immediately ensures authentication. In KEMTLS authentication is only made explicit when the Finished messages are verified. However, the lemmas in the original model already only captured authentication through the Finished messages.

9.3.4 Results

After adding relevant helper lemmas, Tamarin was able to auto-prove all the correctness and security lemmas for Model #1, with all four KEMTLS variants supported simultaneously. Run-times are shown in section 9.4.

Auto-proving and helper lemmas

Many of the lemmas in the model of TLS 1.3 were not able to be auto-proved by Tamarin; instead, the authors had to manually guide Tamarin through parts of the proof. Our goal was to improve the model so that it could be proved automatically, with no manual intervention required.

To help the automated prover, Cremers, Horvat, Hoyland, Scott, and Van der Merwe introduced many intermediate lemmas, many of which state properties of earlier keys or more limited message exchanges. Inheriting these lemmas proved to be both helpful and distracting. Incrementally proving and adjusting the intermediate lemmas to apply to KEMTLS(-PDK) helped us spot bugs and make progress. But starting from their helper lemmas often left us unclear as to why particular intermediate lemmas were necessary to prove the final security properties.

In our experience, Tamarin does not find counterexamples very easily in big models. As a result, we wrote increasingly "smaller" lemmas whenever we ran into a lemma that was hard to prove. This greatly expanded the number of helper lemmas available. While we believe that this helped auto-prove the model, it also resulted in cases where the helper lemmas interacted in bad ways and had to be ignored. (Replacing DH by KEM operations, thus avoiding Tamarin's algebraic analysis of DH group operations, may also have eased analysis.) Additionally, the model of [108] is carefully split over different files to avoid certain helper lemmas from interacting. With much less experience, we joined together most of those files, which in many cases led to Tamarin getting distracted by helper lemmas. Many hours in the manual prover helped us determine which lemmas needed to be marked by `ignore_lemma` annotations. While doing this, it was often helpful to rename lemma variables to be distinguishable, as Tamarin does not indicate what lemmas it tries to apply. For example, to identify the lemma `entity_authentication`, we renamed a time variable `#j` to `#j_ea`.

A bug in the Cremers et al. TLS 1.3 lemmas

While working on the proof, we found that one of the core lemmas in [108]'s TLS 1.3 model seems to have changed after creating the proof. The lemma `session_key_agreement` tried to prove that the client's and servers values of `keys` in the `SessionKey` fact matched. However, variable `keys` is a tuple `<kr, kw>` of the reading and writing keys of each peer. As the server's writing key should match the client's reading key and not the client's writing key, this lemma did not hold. The rendered proofs included in the repository alongside the model and lemmas revealed that in the executed proof, `keys` was split into its elements and equated correctly. We disclosed the bug to one of the authors, and it has been fixed in the upstream repository.

It is not hard to imagine how such a mistake slips into a model if re-proving the smallest changes requires days of manual proving effort. We view this as evidence of the value of auto-proving models: being able to let the computer "do its thing" allows us to make changes more confidently.

9.3.5 Limitations

Although the model is very granular in its description of KEMTLS(-PDK), we do have some limitations. As discussed in section 9.3.3, we do not model intermediate session key reveal. We also have not modeled session resumption or pre-shared key modes with KEMTLS. Finally, we have not attempted to model deniability, which we will model in section 9.5.

9.4 Runtime characteristics of the model

We ran the model on a server with two 20-core Intel Xeon Gold 6230 CPUs, which after hyperthreading gives us 80 threads. The server has 192 GB RAM.

Tamarin runs through all the lemmas in our proof in 28 hours. We note that communication bottlenecks between cores prevent fully utilizing all resources. The model requires 121 GB of RAM to prove all the lemmas, though most individual lemmas need much less memory. In table 9.1, we show some of the lemmas that consumed the most time. Note that it is likely possible to prove them in less time, by hiding more "distracting" helper lemmas or writing smarter oracles, but we did not optimize for this.

Table 9.1: Wall-clock runtimes (hh:mm:ss) and memory usage of a selection of lemmas from the model described in section 9.3

Lemma	Steps	Runtime	Memory
`session_key_auth_agreement`	29 116	6:42:01	16 GB
`session_key_agreement`	57 680	13:56:04	32 GB
`handshake_secret`	29 390	4:40:52	12 GB
`master_secret_pfs`	29 535	2:53:11	76 GB
All lemmas	—	28 hours	121 GB

9.5 Model #2: multi-stage key exchange model

The security properties shown in the original KEMTLS paper [321] and the KEMTLS-PDK paper [320] are stated using the reductionist security paradigm, via the *multi-stage key exchange model* [143], which was adapted for proofs of the TLS 1.3 handshake [127, 128]. Our goal in this section is to translate the reductionist security properties in this model—match security, session key indistinguishability, and authentication—from a pen-and-paper model to being encoded in Tamarin, then have the Tamarin prover confirm these properties hold. Notably, this model discriminates between the several keys established within a single KEMTLS handshake, associating distinct security properties with individual stage keys.

9.5.1 Pen-and-paper proofs

In chapters 7 and 8 we provide theorems and give proofs that KEMTLS and KEMTLS-PDK, respectively, satisfy the match-security and multi-stage security properties; they do not include any proofs for offline deniability. The match-security properties (definition 7.2) are shown information-theoretically, with terms depending on the number of sessions, the correctness probability of the KEMs, and the size of the TLS nonce space. The multi-stage security properties (definition 7.5) are shown under the following computational assumptions: hash function collision resistance, IND-1CCA security of KEM_e, PRF and dual-PRF security of HKDF.Extract, PRF security of HKDF.Expand, EUF-CMA security of HMAC, and IND-CCA security of KEM_c and KEM_s. There is a tightness loss proportional to the number of sessions squared.

9.5.2 Formalizing the reductionist security model in Tamarin

We formalized all four KEMTLS variants (regular and PDK, server-only and mutually authenticated) in Tamarin, along with lemmas capturing correctness, match security, multi-stage security, and deniability, analogous to the definitions from chapters 5 to 8. We now describe the formalization in more detail. In the online version of the paper on which this chapter is based we give a state diagram of the Tamarin model [93, App. D].

Protocol description

This Tamarin formulation of the four KEMTLS variants focuses on the "cryptographic core" of the protocol. Roughly speaking, this is the protocol as formulated in figures 5.2, 5.3, 6.3 and 6.4, which includes cryptographic operations involved in the key exchange, but does not include extra fields and operations arising from the integration of the cryptographic operations into a network protocol. We only address the handshake protocol and exclude TLS message formatting, algorithm negotiation, and data structures such as certificates. We exclude extensions such as TLS 1.3 session resumption or pre-shared key handshakes. Long-term public keys are assumed to be reliably distributed out-of-band. We omit modeling handshake encryption: while the various handshake traffic secrets are established and recorded as accepted in each stage of the protocol, subsequent handshake messages are sent in plaintext. The various primitives based on hash functions (HMAC, HKDF.Extract, HKDF.Expand) are modeled as independent opaque functions, rather than relying on each other and ultimately on a common hash function. As in the pen-and-paper proofs, there are three KEMs, KEM_e, KEM_c, and KEM_s, for ephemeral key exchange, client authentication, and server authentication, respectively. The KEMs are modeled as distinct primitives, meaning that a party cannot use its long-term credential to act as both a client and a server.

Adversary interaction

Among the queries that are given in section 7.2.2, the NewSession and Send queries are not explicitly needed, since the Tamarin model includes rules for each protocol step. The Tamarin model does include Corrupt and Reveal oracles. As Tamarin lemmas using the for-all quantifier already cover all possible sessions, we do not need to specify a session or key under test; thus there is no need for the Test query in the Tamarin model.

Definition of cryptography

For each KEM KEM_x, we define four functions in Tamarin: `KEM_x_PK/1` (to generate a public key from a secret key), `KEM_x_Encaps_ct/2` (to generate a ciphertext from a public key and random coins), `KEM_x_Encaps_ss/2` (to generate, during encapsulation, a shared secret from a public key and random coins), and `KEM_x_Decaps/2` (to decapsulate a ciphertext using a secret key to recover a shared secret). (Two `Encaps` functions are provided because functions in Tamarin only output a single value, so we use two functions to represent the two outputs from encapsulation.) Rewriting rules are provided to model that decapsulation with the appropriate values arrives at the same shared secret as encapsulation.

There are distinct functions for HKDF (modeled by `HKDFExtract/2` and `HKDFExpand/3`), HMAC (`HMAC/2`), and the hash function (`H/1`). We do not attempt to model the fact that HKDF is built from HMAC and that HMAC uses the same hash function `H`; they are all assumed to be independent. The HKDF API is simplified to not include a length parameter as input.

Correctness lemmas

We include a collection of "reachability" lemmas that check that, for every stage in all 4 protocol variants, it is possible to arrive at that stage, with honest client and server sessions having correct owner and peer information, matching contributive and session identifiers, and correct expectations on authentication, forward secrecy, and replayability; the reachability lemmas include checking retroactive upgrading of properties. These lemmas are implemented using Tamarin's **`exists-trace`** feature. There are 47 reachability lemmas in total, generated from a template using the M4 macro language.

We also include lemmas that check that the attacker works, in the sense that the attacker can successfully compute session keys of all stages by using the corruption oracles.

Match security lemmas

The match security lemmas from Definition B.1 of [321], plus the adjustments for replayability in [320], are directly translated into Tamarin. The lemmas are, put simply, predicates over the session-specific variables defined in the model syntax and can be stated analogously since the Tamarin model includes action facts for each session-specific variable.

Match security property 1 (π and π' agree on the same key at every stage

$j \leq i$) is slightly non-trivial to encode in Tamarin, since Tamarin action facts do not let us record stage numbers as integers that can be compared using $\leq$, forcing us to use strings to record stage numbers; thus we use M4 macros to generate a different version of this lemma for each $1 \leq j \leq i \leq 6$.

Since our Tamarin model considers the execution of all 4 KEMTLS variants simultaneously, we add one more match security property: if distinct sessions π and π' are partnered in some stage i, π and π' agree on the protocol variant in use in stage i, and early stages $j \leq i$ that have been retroactively refined. Note that the client and server do not distinguish between KEMTLS-sauth and KEMTLS-mutual in stages 1–4 until stage 6 has accepted (and similarly for KEMTLS-PDK-sauth and KEMTLS-PDK-mutual in stage 1 until stage 2 has accepted).

Session key security and authentication lemmas

Session key security in Tamarin is modeled based on the infeasibility of session key recovery, rather than the indistinguishability of a session key from random. In Tamarin, we write a lemma for each type of forward secrecy a stage key can have. These lemmas directly translate the freshness conditions as given by definition 7.3.

In the pen-and-paper model, some of the forward secrecy predicates include conditions like "there exists a stage $j \geq i$ such that..." to model retroactive upgrading of forward secrecy properties once a later stage accepts. As noted above, we cannot record stage numbers as comparable integers; instead, in the protocol specification we record an action fact of the form `FS(~tid, i, j, 'wfs2')` and check if stage j has accepted before requiring the corresponding forward secrecy property (e.g., `'wfs2'`) to hold.

We also have a lemma for explicit authentication which corresponds to definition 7.4, including the exclusion for uniqueness of replayable sessions for KEMTLS-PDK stage 1.

Deniability lemmas

Whereas the lemmas for the above properties all share the same Tamarin protocol description as explained above, the deniability lemmas use a restatement of the protocol description. To formulate a deniability lemma, we need two versions of the protocol description: honest execution of the protocol using long-term secrets, and simulation using only public keys. The judge in the offline deniability game is passive and receives only transcripts,

so we can collapse the multiple rules for each client and server action into a single rule that generates a full transcript including both client and server operations. The deniability lemmas use Tamarin's observational equivalence feature [28] to check that the real and simulated transcripts are indistinguishable. Deniability for each of the 4 KEMTLS protocol variants is dealt with separately. For each variant (e.g., KEMTLS with server-only authentication), we have one rule that generates real transcripts using long-term secret keys (e.g., `KEMTLS_SAUTH_real`) and one rule that generates simulated transcripts without long-term secret keys (e.g., `KEMTLS_SAUTH_simulated`). Finally, the adversary has access to a rule `real_vs_simulated` which takes as input one real transcript and one simulated transcript. It returns one of these to the adversary using Tamarin's `diff` operator for observational equivalence. By the running Tamarin prover with the `--diff` option to activate observational equivalence mode, Tamarin will check that it is not possible to distinguish which was given to the adversary.

Our definition of deniability is offline deniability in the universal deniability setting against an unbounded judge with full corruption powers. Consequently, the transcripts output includes the parties' long-term secret keys, the session keys computed in the real or simulated transcript, and the random coins allegedly used in the real or simulated transcript.

Using Taramin's observation equivalence feature causes a substantial increase in state space, so for efficiency reasons, we provide an option (using M4 macros) to omit portions of the transcript that are deterministically generated from earlier parts of the transcript and thus (from a mathematical perspective) could not help a distinguisher.

9.5.3 Comparison of pen-and-paper and Tamarin models

In principle, if the same security properties have been encoded in both a pen-and-paper reductionist security model and in a Tamarin model, a full and correct proof in the reductionist security model yields everything that a Tamarin proof could, and potentially more. In particular, reductionist security proofs do not idealize cryptographic primitives as much as Tamarin does. Moreover, a reductionist security proof can be done in the "concrete setting" [30], yielding a precise (non-asymptotic) relationship between the runtime and success probability of an adversary against the protocol versus the runtime and success probability of breaking the underlying cryptographic

assumption. While it would be possible to encode the pen-and-paper proofs of KEMTLS from the original papers into a computer verification tool such as EasyCrypt [26], that would also require the cryptographer to manually write all game hops and reductions, a massive undertaking. To date, there are no proofs of KEMTLS or KEMTLS-PDK using a computer-aided verification tool for reductionist proofs.

Tamarin does not lend itself to writing security properties in precisely the same way as would be used in reductionist models. Although there is no way to objectively justify how close the pen-and-paper and Tamarin models of this section are to each other, subjectively we think they are quite close:

- The protocol specification in Tamarin maps nearly line-for-line onto the protocol figures in the original papers, using the same function interfaces, same key schedule, and same session identifiers.
- The session-specific variables in the pen-and-paper model correspond nearly one-for-one to action facts in the Tamarin model.
- There are Tamarin lemmas for each security property in the pen-and-paper model, and there is a clear mapping between the clauses in the predicates in the pen-and-paper model and the Tamarin model.

The main gap in modeling, as mentioned earlier, is that session key security is modeled via indistinguishability in the pen-and-paper models but via infeasibility of key recovery in the Tamarin model. Though it is possible to verify indistinguishability through Tamarin's observational equivalence features, the effect on the state space as discussed in section 9.5.2 makes this impractical.

A nice feature of our Tamarin models is that there is a fairly clean separation between protocol definition and security properties: files containing lemmas for Match security, Multi-Stage security, and authentication are phrased solely in terms of the action facts of the generic security model (similar to how a good pen-and-paper security model refers abstractly to the protocol API and model variables, rather than mixing in details of protocol instantiation), so these lemmas could be applied to any protocol in the same security model.

9.5.4 Results

Tamarin was able to auto-prove all the lemmas for correctness, reachability, match security, multi-stage session key security, authentication, and deniabil-

ity in Model #2, with all four KEMTLS variants supported simultaneously. We did not need to create any helper lemmas for Tamarin. Run-times are shown in section 9.6.

Bugs in the original papers' security properties

When translating the models into Tamarin, we identified minor mistakes in some of the forward secrecy and authentication properties listed in the original KEMTLS [321] and KEMTLS-PDK [320] papers, highlighting the value of formal verification. In this thesis, we used the corrected properties. The online versions of the source papers, as well as the chapters in this thesis, have also been updated with our corrections.

Summarizing, the errors found were:

- In KEMTLS-mutual: $\mathsf{auth}_3^S = 3$ and $\mathsf{auth}_4^S = 4$ both should have been set to 5; $\mathsf{FS}_{3,3}^S = \mathsf{FS}_{3,4}^S = \mathsf{FS}_{4,4}^S = \mathsf{wfs2}$ should all have been wfs1; and $\mathsf{auth}_6^S = 6$ should have been $\mathsf{auth}_6^S = \infty$.
- In KEMTLS-PDK-sauth: $\mathsf{FS}_{1,j}^C$ and $\mathsf{FS}_{1,j}^S$ should have been 0 for all j; $\mathsf{auth}_5^C = 5$ should have been $\mathsf{auth}_5^C = \infty$; and $\mathsf{FS}_{i,4}^S$ should have been wfs1 for $i = 2, 3, 4$.
- In KEMTLS-PDK-mutual: the message $\mathtt{SKC}$ should have been included in the $\mathtt{SF}$ MAC computation and $\mathtt{SF}$ should have been included in the $\mathtt{CF}$ MAC computation; $\mathsf{FS}_{1,j}^C$ and $\mathsf{FS}_{1,j}^S$ should have been 0 for all j; $\mathsf{auth}_5^C = 5$ should have been $\mathsf{auth}_5^C = \infty$; and $\mathsf{FS}_{4,4}^S = \mathsf{wfs1}$ should have been wfs2.

9.5.5 Limitations

As noted above, the design of the model in this section imposes some limitations. Unlike in section 9.3, we generally did not model non-cryptographic details of the handshake, such as TLS handshake messages, extensions, or the record layer. We also did not model handshake encryption or algorithm negotiation.

We also had, unlike in section 9.3, three distinct KEMs for ephemeral key exchange, server authentication, and client authentication. This implicitly assumes the same certificate is not used for both purposes. We note that a similar assumption on pre-shared keys was the basis of the Selfie attack [131].

Without this limitation, however, we observe a state-space explosion with a major impact on performance. For example, if $\mathsf{KEM}_c = \mathsf{KEM}_s$, the first 10 out of 11 `reachable_*` lemmas take over 8 hours, and the last `reachable_*` did not terminate after 45 hours, compared to all 11 `reachable_*` lemmas taking just over 1 minute with distinct KEM_c and KEM_s.

Our deniability lemmas are for abbreviated transcripts (without messages generated deterministically from earlier parts of the transcript) and omit ephemeral coins. Again, without this limitation, there is a major impact on performance. For example, including full transcripts for KEMTLS-sauth increases runtime from 1 minute to 16 hours, whereas including ephemeral coins increases runtime from 1 minute to 110 minutes.

9.6 Runtime characteristics of the model

Table 9.2 shows the runtime for the various lemmas, for each KEMTLS variant on its own, and when all four KEMTLS variants are run simultaneously. Tamarin was restricted to using 16 cores, and the times shown are wall-clock times. Total CPU time will be greater than the wall-clock time, but typically less than 16× wall-clock time since Tamarin hits communication bottlenecks preventing it from loading all cores to 100 %. Results are measured on the same system as in section 9.4, with `tamarin-prover` version 1.16.1.

Table 9.2 clearly shows that the mutual versions of the protocols require a bit more work for Tamarin than the server-authenticated protocols. Interestingly, this difference is more pronounced for KEMTLS than it is for KEMTLS-PDK. This might be because, in KEMTLS, mutual authentication requires the exchange of significantly more messages. We also see how deniability for mutual authentication is much harder to prove than deniability of unilaterally authenticated KEMTLS(-PDK). Finally, proving the security properties for more than a single protocol has a large effect on the runtime, as we expect given the opportunities for cross-protocol interaction.

9.7 Comparison of models

We discussed two very different models of KEMTLS(-PDK) in the previous sections. These models are examples of how we can view modeling as the art of replacing specifics with generalities. Model #1 stays very close to the wire

Table 9.2: Wall-clock runtime ((hh:)mm:ss) for Tamarin proofs of lemmas from Section 9.5.

Lemma	KEMTLS			KEMTLS-PDK			All 4
	sauth	mutual	both	sauth	mutual	both	**variants**
`reachable_*`	01:17	01:20	04:32	01:46	01:36	04:40	0:13:25
`attacker_works_*`	00:17	00:46	01:16	00:17	00:23	00:53	0:12:04
`match_*`	01:02	01:22	02:55	00:55	01:14	02:46	0:09:53
`sk_sec_nofs_client`	00:05	00:07	00:16	00:05	00:05	00:14	0:00:41
`sk_sec_nofs_server`	00:05	00:06	00:12	00:05	00:06	00:14	0:00:40
`sk_sec_wfs1`	00:21	00:10	01:05	00:17	00:18	00:41	0:03:00
`sk_sec_wfs2`	00:36	00:28	01:30	00:28	00:22	01:23	0:24:28
`sk_sec_fs`	01:20	03:05	06:38	01:21	01:33	05:07	1:39:58
`malicious_accept.`	00:13	01:40	04:13	00:17	00:22	01:39	27:29:37
`deniability` (abbr.)	01:02	12:15	—	00:24	29:10	—	—
Total (excl. `den.`)	05:16	09:05	22:38	05:30	06:00	17:38	30:13:46

format of TLS 1.3 and phrases the security properties in terms of attacks on the ephemeral and long-term keys. It contains more implementation details such as algorithm negotiation, message framing, encryption of handshake messages, and even application data. Model #2 is more abstract in its representation of protocol messages. However, it models the cryptographic properties in a more granular fashion. This more abstract description closely follows the multi-stage pen-and-paper proofs of KEMTLS and KEMTLS-PDK and allowed verifying the properties claimed in the pen-and-paper proofs. Table 9.3 summarizes differences between the two models, a few aspects of which we discuss further below.

9.7.1 Modelling KEMs

The two models differ in the way that they model the KEMs in the protocol. Model #1 uses the same functions for all KEM modes in the protocol (ephemeral key exchange, server authentication, and client authentication). Model #2 has three separate sets of functions for the three different KEM modes; this means the attacker cannot copy ciphertexts or public keys from

Table 9.3: Comparison of features in our two Tamarin models of KEMTLS.

Feature	**Model #1**	**Model #2**
Protocol modelling		
Encrypted handshake messages	✓	✘
HKDF and HMAC decomposed into hash calls	✓	✘
Key exchange and auth. KEMs are the same algorithm	✓	✘
TLS message structure	✓	✘
Algorithm negotiation	✓	✘
Security properties		
Adversary can reveal long-term keys	✓	✓
Adversary can reveal ephemeral keys	✓	✘
Adversary can reveal intermediate session keys	✘	✓
Secrecy of handshake and application traffic keys	✓	✓
Forward secrecy	✓	✓
Multiple flavours of forward secrecy	✘	✓
Explicit authentication	✓	✓
Deniability	✘	✓

one of the modes to another, which should make proving the protocol easier. Interestingly, we saw significantly different performance between these two approaches. The second model proves in a very short time with the three separate KEMs, but runtime blows up if we define all three KEM modes with the same functions; we did not attempt to generate the full proof because it took so long, as we discussed in section 9.5.5. This suggests that splitting the three KEM modes in the first model could result in a speed-up. However, splitting the KEMs in Model #1 did not improve the time to auto-prove lemmas; in fact, a few lemmas even stopped being auto-provable. Ideally, this puzzle would be resolved with a justification that there is a way of safely separating uses of KEMs, allowing us to use whichever form happens to be easier for Tamarin to prove.

9.7.2 Threat model

Both models use Dolev–Yao attackers, but give the attackers slightly different extra abilities as noted in the bottom half of table 9.3. Consequently, the results hold in slightly different circumstances. The attacker in Model #1 can compromise ephemeral keys and long-term keys, but not session keys, whereas the attacker in Model #2 can compromise intermediate session keys and long-term keys, but not ephemeral keys. Revealing the HS intermediate session key allows the second attacker to simulate the abilities of the first, but the reverse does not hold; the attacker in Model #2 is thus slightly stronger.

9.7.3 Ease of use

Work on each of our two models was done by separately by myself and a co-author, while neither of us had modeled a protocol using Tamarin before and who had only had a basic introduction to Tamarin prior to this work. We were surprised that the creation of Model #2 from scratch was simpler and proceeded faster than the work in Model #1 adapting the Cremers, Horvat, Hoyland, Scott, and Van der Merwe TLS 1.3 model to KEMTLS. We attribute this to the higher fidelity of the TLS 1.3 model, requiring more code to model our changes, and to the higher difficulty in proving.

9.8 Conclusion

We presented two Tamarin models checking the security properties of KEMTLS and its variant protocol KEMTLS-PDK. Model #1 is highly detailed in implementation characteristics, close to the wire format of the protocol. Model #2 presents the protocol at a higher level but provides a more precise characterization of security properties. We prove that KEMTLS(-PDK) is secure in both models; importantly these analyses include all four KEMTLS variants supported simultaneously. Additionally, we proved offline deniability properties of KEMTLS(-PDK) in Model #2.

Overall, comparing these two analyses is something of an apples-to-oranges comparison. The two very different approaches allow us to model and test different properties of the protocol. Model #1 is closer to what an implementation would be like, and verifies the security properties in such a scenario. Adopting the Cremers, Horvat, Hoyland, Scott, and Van der Merwe TLS 1.3

model [108] also allowed us to quickly adapt the security claims of TLS 1.3 to our protocols. Model #2, on the other hand, is an adaptation of the multi-stage authenticated key exchange model from the pen-and-paper proofs in [320, 321]. As such, Model #2 in a sense checks the claims in the pen-and-paper proofs, and in fact, uncovered some minor mistakes in those proofs.

Our two models illustrate a common trade-off in formal analysis between the detail of the protocol specification and the granularity of the security properties we can prove. A similar observation was also made in [108], who commented computational analyses could only look at parts of TLS 1.3, rather than considering all the modes at once.

While we proved certain privacy properties, such as deniability, our models can be further expanded to include other privacy properties, such as the proposed Encrypted `ClientHello` extension (previously called ESNI) [301]. These properties have only been proven by using the symbolic protocol analyzer ProVerif [57].

Part III

Implementing and measuring post-quantum TLS

10 Implementing and measuring post-quantum TLS in Rust

To support our work, we implemented post-quantum TLS, OPTLS, KEMTLS, and KEMTLS-PDK on top of Rustls [62], a modern TLS library written in Rust. In chapters 13 and 14, we report measurements from KEMTLS and KEMTLS-PDK obtained from a virtualized network environment. This chapter describes how we implemented the protocols and set up the measurement framework.

10.1 Rustls

Rustls provides a clean implementation of TLS 1.3 that was easier to modify than OpenSSL and provides comparable performance [63]. It uses the Ring [327] library for cryptography and the WebPKI [328] library for certificate validation. Both of these are also written in Rust, although Ring links to C implementations of cryptographic primitives from BoringSSL [163]. Specifically, our implementations are based on Rustls version 0.18.1, and we will describe how we modified that specific version.

In Rustls, we find the source code neatly organized by its role. The key schedule computations, for example, can be found in the file `key_schedule.rs`, while the ciphersuites are found in `cipher.rs`. The more specific logic for TLS clients and servers can be found in `client` and `server` subfolders (i.e., Rust modules), and in turn the TLS 1.2 and TLS 1.3 state machines can be found in the `tls12.rs` and `tls13.rs` files,[1] while shared logic, such as the initiation of a handshake where the protocol version is not yet decided, can be found in `hs.rs`.

The handshake state machines are organized through Rust *structs*, which contain state and have associated functions. One such struct is for example `client::tls13::ExpectCertificate`. Part of the implementation of

[1] Rustls does not support TLS versions prior to 1.2.

`hs::State` is the `handle()` method. This function takes an incoming message and processes it. In the example of `ExpectCertificate`, it will expect a `ServerCertificate` message. If a correct message is received, the function updates the message transcript, validates the certificate, and converts it into a WebPKI object. If anything is awry, it will return an error. The method will then return the next state, in this case the `ExpectCertificateVerify` struct, in which the certificate received in the previous state is stored so the handshake signature can be verified. Often, a state will emit a protocol message as part of this transition.

The record layer, which takes care of transporting the messages and, where relevant, encrypting or decrypting them, is defined on a different logic layer, which makes it much easier to reason about the TLS state machine.

10.2 Implementating post-quantum TLS

As stated above, Rustls already supports TLS 1.3. So in our implementation of post-quantum TLS 1.3, as described in chapter 3, we only need to add support for post-quantum key-exchange and signature algorithms.

10.2.1 Implementing post-quantum key exchange

Rustls only supports ephemeral ECDH key exchange in TLS. ECDH does not exactly have the same API as the post-quantum KEMs we want to replace it with, but fortunately, this is not a problem for how we add KEM-based ephemeral key exchange in TLS 1.3. In Rustls, the key-exchange algorithms are defined along with the ciphersuites in `suites.rs`. The struct `KeyExchange` defines the associated operations, which use a ECDH-style API and directly wraps Ring's `ring::agreement` API. The key exchange is initiated through the `start_ecdhe()` factory method. This method takes a `NamedGroup`, which defines the key exchange group, as input, and returns a new `KeyExchange` struct if successful. The TLS client calls this function when it is constructing its `ClientHello` message, and takes the public key from the `KeyExchange` result. The server, when it is processing the `ClientHello` message, also calls `start_ecdhe()`, but it passes the parameters from the `ClientHello` message to the `server_complete()` method. This function returns, if successful, the `KeyExchangeResult` struct, containing the server's public key and shared secret. The public key is then used in the

ServerHello message, which the client processes in the same way, using the `client_ecdhe()` method, again obtaining the shared secret through a `KeyExchangeResult` struct.

We first change the API used into one that is based on the KEM API. We demote `start_ecdhe()` to a private method and write a new method called `start_kex()`. This method is to be exclusively used by the client, which still takes the public key from the returned `KeyExchange` struct. The server no longer calls `start_ecdhe()`, but instead calls a new `encapsulate()` method. This method transparently wraps the old methods for Ring-based key-exchange algorithms but we have renamed the `public_key` field to `ciphertext` in the returned `KeyExchangeResult` object. The client uses the `decapsulate()` method, renamed from `client_ecdhe`.

The public keys and private keys referenced in the `KeyExchange` struct directly own instances of Ring's `PublicKey` and `EphemeralPrivateKey` structs. Additionally, the identifiers for algorithms are direct references to `ring::agreement::Algorithm` instances. As we now want to add support for new post-quantum algorithms to the newly-created KEM-style API, we have to be able to support non-Ring algorithms. We add a layer of indirection to the API, which will allow us to use the original Ring-backed primitives, but also allow us to add algorithms backed by OQS's `liboqs` [345], for which we wrote and contributed a Rust wrapper. We do this by introducing enums for algorithms and keys. These enums contain a `RingAlg` variant, which we use for the original Ring-API-based things, as well as a `KEM` variant that is used for the `liboqs`-backed algorithms. Whenever we encounter an algorithm or key, we use Rust's enum **`match`** syntax to detect which variant it is and switch the backing implementation logic accordingly. I.e., we call out to Ring when we encounter the `RingAlg` enum variant, and use `liboqs`'s KEM operations when we encounter the `KEM` enum variant. For example, in the new `start_kex()` method shown in listing 10.1, we call `liboqs::kem::keygen` when we encounter the `KEM` enum variant, while we call out to the old `start_ecdhe` method for the `RingAlg` enum variant.

10.2.2 Caching ephemeral key shares

In the experiments with OPTLS, we use algorithms that have heavy computational requirements. It is possible to amortize the cost of ephemeral key generation by either reusing ephemeral keys or by generating them out-of-

Listing 10.1: Example of how Rust enums are used to switch between Ring- and liboqs-provided implementations of key-exchange algorithms.

```
// Generates the public key keyshare
pub fn start_kex(group: NamedGroup)
  -> Option<KeyExchange>
{
  let alg = KeyExchange::named_group_to_ecdh_alg(group)?;
  match alg {
    KexAlgorithm::RingAlg(alg) =>
        Self::start_ecdhe(group, alg),
    KexAlgorithm::KEM(kem) => {
      let (pk, sk) = kem.keypair().unwrap();
      Some(KeyExchange {
        group,
        alg: KexAlgorithm::KEM(kem),
        privkey: KexPrivateKey::KEM(sk),
        pubkey: KexPublicKey::KEM(pk),
      })
    },
  }
}
```

band, in chapter 12 we show results both for handshakes both when ephemeral keys are generated in every handshake, and when they are cached. In our implementation, if key caching is enabled, we store the ephemeral key after generation in a global cache. This code is included in the implementation by a Rust feature flag and is not present in builds where it is disabled.

10.2.3 Post-quantum signature-based authentication

To provide the post-quantum authentication in our TLS experiment, we make use of fully post-quantum certificate chains using post-quantum signature algorithms. This means that we need to use post-quantum CA certificates, intermediate CA certificates, and client and server certificates. The Rustls TLS implementation delegates the verification of signatures on certificates, as well as the verification of handshake signatures, to the WebPKI library. We modified WebPKI version 0.21.0 to add support for `liboqs`-provided implementations of post-quantum signature algorithms. We proceed similarly to our modifications for the key exchange in Rustls: we add a new Rust enum type that wraps the algorithm identifiers to indicate if they are `Ring` or `Oqs` identifiers. The signature verification code again makes use of these enum variants to identify what logic to use. Then, we add supported `SignatureAlgorithms` to WebPKI for each of the post-quantum signature algorithms we need to support, where the `verification_alg` scheme identifier now points to the `liboqs` algorithm identifier.

10.2.4 Caching certificates

For the comparison with KEMTLS-PDK in chapter 14, we implemented new extensions in the `ClientHello` and `ServerHello` messages. The client transmits hashes of the certificates that it knows of in this extension, and the server uses the extension to indicate if it will be making use of this functionality. Note that RFC 7924 [312], which was only specified for TLS 1.2, defines that the extension should transmit the hash of the entire `Certificate` message, and if the server wants to omit the certificates, to replace the entire message by the hash value. In our implementation, we cache individual certificates and the server replaces each certificate in its certificate chain individually by a hash value. This was much easier to implement, and more compatible with the extensions to the `Certificate` message defined by TLS 1.3.

10.3 Implementing post-quantum OPTLS

For experimentation with OPTLS, we first added support for post-quantum NIKEs to the key exchange methods. We added a CSIDH variant to the relevant enums and updated the relevant methods of the `KeyExchange` struct. The code that adds support for CSIDH much resembles the original ECDH code, as both are NIKEs algorithms.

To add support for NIKE algorithms in the client and server certificates, we extended the WebPKI API for end-entity certificates with the method `is_nike_cert()`, that identifies the certificate as containing a NIKE public key. Next, we add a method that allows extracting the NIKE algorithm identifier and certificate public key. In Rustls, we update the states of the state machine up to the processing of the `CertificateVerify` message to keep track of the client's ephemeral key share. In the server's method that emits the `CertificateVerify` message, we compute the OPTLS shared secret by taking the client's ephemeral key share and combining it with the server's certificate private key. In the client implementation of the verification of `CertificateVerify`, we decide based on whether the supplied certificate is a NIKE certificate to either verify a signature or compute the OPTLS shared secret using the client's ephemeral private key and the certificate's public key. The MAC in the `CertificateVerify` message is computed from the OPTLS shared secret in the same way.

10.4 Implementing KEMTLS

To implement KEMTLS, we need to make large changes to the state machine of TLS 1.3. We introduce many new states to handle e.g. the ciphertext and the changed order for the `Finished` messages. The key schedule is greatly extended as well, to incorporate the new handshake secrets and accommodate the change in how the `Finished` keys are computed in KEMTLS.

KEMTLS is negotiated by adding new types of `SignatureAlgorithm` to the TLS handshake. We add a new algorithm called `KEMTLS` which the client uses to indicate support. The server receives the list of client-supported signature algorithms[2] and sees the `KEMTLS` algorithm advertised, it knows it can select

[2] If this way of negotiating KEMTLS is ever standardized, this field should perhaps be renamed to authentication algorithms.

a KEM certificate. (In a production-level implementation of KEMTLS, the `SignatureAlgorithm` identifier should perhaps indicate *which* KEMs are supported.) Our Rustls state machine also makes use of the type of certificate received by the client: if the client receives a certificate with a KEM public key, it knows that it needs to proceed with the KEMTLS handshake. Otherwise, it can transparently fall back to negotiating TLS 1.3.

To add support for KEM public keys in certificates, we made similar modifications to WebPKI as we have described for the NIKE support in the previous section. We add an `is_kem_cert()` method that identifies the endpoint certificate as one containing a KEM public key. We also add `encapsulate()`, which simply encapsulates to the contained public key. Finally, we define a `decapsulate()` method that takes the certificate's private key and the encapsulated ciphertext and returns a shared secret.

10.5 Implementing KEMTLS-PDK

As we extended our TLS 1.3 implementation with not only KEMTLS, but especially KEMTLS-PDK, it admittedly got a lot more convoluted. The client initiates a KEMTLS-PDK handshake by encapsulating to the server's long-term public key. We extend the TLS client configuration structs to allow specifying this certificate as a connection parameter. The encapsulation is sent to the server as a new ClientHello extension, which we called `ProactiveCiphertext`.[3] The extension contains the hash of the certificate that we encapsulate to and the ciphertext. We use a similar extension to indicate in the ServerHello to the client if the server accepted the ciphertext, and that the client should proceed with the KEMTLS-PDK handshake. Otherwise, we allow the client to fall back to the plain KEMTLS handshake.

If we are using client authentication, the client additionally includes an extension that indicates that it will send a client certificate. This allows the server to handle the incoming certificate more easily. We have not implemented any negotiation of algorithms for KEMTLS-PDK client authentication, instead always using a "default" ciphersuite. Note that supporting 128-bit AES-GCM is mandatory for any TLS 1.3-compliant application [298, Sec. 9.1].[4]

[3] The extension was named before we settled on pre-distributed.

[4] Naturally, to protect against very large quantum computers using Grover's algorithm, we would require at least 256-bit keys. We refer to the discussion in section 2.6.

Another option would be to store information about, e.g., the ciphersuite to use alongside the certificate—one could even consider integrating ciphersuite information inside certificates. This would constitute a bigger change to the TLS and public-key infrastructure (PKI) infrastructure but is not unprecedented (c.f. HPKE [21]). Finally, the server currently must process the client certificate and proceed with KEMTLS-PDK if client authentication is used: we do not implement any fallback mechanisms.

10.6 Fast post-quantum cryptography

As mentioned in the previous sections, we relied on Open Quantum Safe's liboqs for most of the implementations of the post-quantum primitives in our experiments. However, to provide a level playing field and examine the best-case scenario for all algorithms, we must use (similarly) optimized implementations for all the primitives involved.[5] All of our experiments have been done with optimized implementations of the post-quantum primitives. Specifically, we used implementations that target the 256-bit AVX2 vector extensions, available on many recent x86-64 CPUs.

Unfortunately, liboqs did not (initially) contain optimized implementations for all schemes that we include in our experiments. For some schemes, we integrated optimized implementations into PQClean (chapter 17), which allowed them to be integrated into liboqs through the project's automated integration scripts. The forks with the optimized implementations are part of the KEMTLS experimental software package.

For reference, we give an overview of all the schemes we used in the experiments in chapters 11, 13 and 14, including the performance of the cryptographic operations on our measurement platform. For KEMs, this overview is given in table 10.1, while for signature schemes it is given in table 10.2. Note that for signature schemes, keygen times are only given for completeness: in our experiments, keys for signature schemes are generated offline.

For all the schemes listed in tables 10.1 and 10.2, we use the as-of-writing most recent versions of the schemes. The only exception is HQC, for which we use a round-3 implementation. This implementation does not differ materially from the current version. For the SPHINCS$^+$ instantiations, we use the

[5] Indeed, we will comment more on the hazards of making comparisons between reference implementations in section 17.3.

"simple" variants, the variant that NIST has indicated they will standardize. We instantiate the hash function with Haraka, accelerated with the Intel AESNI instructions. This is the best-performing variant of SPHINCS$^+$ and should be indicative of the performance of the other hash functions if those are accelerated in hardware. It should be noted that the security of SPHINCS$^+$ instantiations with Haraka is limited to NIST level II and that NIST has indicated that they will likely not standardize it.

Table 10.1: Public key and ciphertext sizes, and computation time for KEMs used in chapters 11, 13 and 14.

Scheme	NIST level	Sizes in bytes		Computation in ms.		
		pk	ct	Keygen	Encaps.	Decaps.
Pre-quantum						
X25519		32	32	0.027	0.103	0.075
Selected for standardization						
Kyber-512	I	800	768	0.020	0.026	0.018
Kyber-768	III	1184	1088	0.022	0.027	0.019
Kyber-1024	V	1568	1568	0.040	0.044	0.036
NIST round-4 candidates						
HQC-128	I	2249	4481	0.066	0.139	0.267
HQC-192	III	4522	9026	0.231	0.460	0.797
HQC-256	V	7245	14 469	0.329	0.628	1.102
McEliece348864	I	261 120	96	55.239	0.026	0.049
McEliece460896	III	524 160	156	148.865	0.037	0.095
McEliece6688128	V	1 044 992	208	184.107	0.062	0.116

10.7 Post-quantum certificates

Rustls and WebPKI do not support creating certificates. To set up our post-quantum root CA certificates, intermediate CA certificates, and leaf certificates, we implemented a certificate generator. We have two small Rust programs to generate KEM and signature keys, as well as a program to sign binary blobs.

Table 10.2: Public key and ciphertext sizes, and computation time for signature schemes used in chapters 11 and 13.

Scheme	NIST level	Sizes in bytes		Computation in ms.		
		pk	sig	Keygen	Sign	Verify
Pre-quantum						
RSA-2048		272	256		0.526	0.016
Selected for standardization						
Dilithium2	II	1312	2420	0.067	0.178	0.064
Dilithium3	III	1952	3293	0.083	0.803	0.081
Dilithium5	V	2592	4595	0.168	0.589	0.161
Falcon-512	I	897	666	16.799	0.642	0.141
Falcon-1024	V	1793	1280	30.422	0.786	0.172
SPHINCS$^+$-128f	I	32	17 088	0.233	5.518	0.443
SPHINCS$^+$-192f	III	48	35 664	0.225	6.234	0.400
SPHINCS$^+$-256f	V	64	49 856	0.491	11.362	0.397
SPHINCS$^+$-128s	I	32	7856	7.590	60.516	0.090
SPHINCS$^+$-192s	III	48	16 224	11.757	118.858	0.135
SPHINCS$^+$-256s	V	64	29 792	6.710	104.567	0.184
Other schemes						
XMSS$_s^{MT}$-I	I	32	979	66 559.800	19.380	8.232
XMSS$_s^{MT}$-III	III	48	1851	96 265.982	28.460	11.899
XMSS$_s^{MT}$-V	V	64	2979	96 319.433	29.482	11.035

Note: SPHINCS$^+$ is short for SPHINCS$^+$-Haraka

These two programs are called by a Python script that generates the relevant ASN.1 encoding, signs it, and outputs X.509 certificates based on environment variables in the build environment. We generate new certificates every time we set up the experiment.

For the pre-quantum RSA and elliptic-curve certificates, we have separately prepared client and root certificates using the OpenVPN Easy-RSA utility [136]. Elliptic-curve based leaf certificates are generated using a shell script that interacts with the OpenSSL command line tools. A shell script is included that sets up the classically-secure PKI fully automatically.

10.8 Code generation

As we need to support many different algorithms, we make use of code generation throughout our integration of the cryptographic primitives into the Rustls and WebPKI libraries. This saved us much effort, especially whenever the list of supported algorithms in `liboqs` changed. We have a python script that identifies the KEMs and signature algorithms in use. For example, for every KEM, it generates an if statement for the lookup function that maps TLS `NamedGroup` algorithm identifiers to the `suites::KexAlgorithm` struct. We output the generated code to function-specific files, which we load directly into the appropriate parts of the codebase using the `include!()` macro. An example is given in listing 10.2.

10.9 Other patches

A minor but not insignificant bump in the road towards experimenting with post-quantum signature algorithms was a very minor mismatch in assumptions between Ring and the TLS standards. Certificates are transmitted in TLS in a particular binary encoding, called DER. In Ring's DER-decoder, they assumed that at most, three-byte encodings (DER length identifier `0x82`) would be used for the length of a certificate object. However, as part of our experiments, we encounter some very large certificates that required four-byte length encodings (DER length identifier `0x84`). This meant that we had to patch the DER decoder to allow these length encodings.

Another example was the decoder for `Certificate` messages in Rustls. The TLS standard allows the certificate message to be up to 16 MiB in size [298,

Listing 10.2: An example of how we use indirection to preserve the existing key exchange functionality, and how generated code is included to support our newly supported algorithms.

```
pub fn named_group_to_ecdh_alg(group: NamedGroup)
  -> Option<KexAlgorithm>
{
  match group {
    NamedGroup::X25519 =>
      Some(KexAlgorithm::RingAlg(
             &ring::agreement::X25519)),
    NamedGroup::secp256r1 =>
      Some(KexAlgorithm::RingAlg(
             &ring::agreement::ECDH_P256)),
    NamedGroup::secp384r1 =>
      Some(KexAlgorithm::RingAlg(
             &ring::agreement::ECDH_P384)),
    group => include!(
      "generated/named_group_to_kex.rs"),
  }
}
```

App. B.3.3]. However, the parser for the certificate message used in Rustls contained the snippet of code shown in listing 10.3. The implementers of Rustls, perhaps in an attempt to avoid memory exhaustion or denial of service attacks, had restricted the size of a certificate significantly compared to the size allowed in the standard, allowing at most 64 KiB of certificates.

Listing 10.3: An example of how implementations can be stricter than what a standard allows.

```
fn read(r: &mut Reader) -> Option<CertificatePayload> {
  // 64KB of certificates is plenty,
  // 16MB is obviously silly
  codec::read_vec_u24_limited(r, 0x10000)
}
```

These two examples in particular are informative of the types of problems that we may run into as we transition to post-quantum cryptography on the internet. In particular, we may see a resurgence of the types of issues seen with so-called "middleboxes" on the internet, such as intelligent firewalls or TLS interception appliances, that intercept and try to validate TLS connections. These middleboxes were a huge hindrance in the development of TLS 1.3, as they did not handle the transition to encrypted handshakes well, often preventing connection establishment [36, 38, 296, 297]. Even the new version number led to many broken connections, while the specification required such TLS parsers to ignore unsupported versions. This is the reason that TLS 1.3 still contains workarounds that make it look more like TLS 1.2 resumption [298, Sec. D.4]. TLS 1.3 clients and servers use the TLS 1.2 version number in their protocol version fields and instead use a TLS extension to indicate support for TLS 1.3. Similarly, before encrypting the handshake messages, most TLS 1.3 implementations still emit the no-longer-necessary `ChangeCipherSuite` message before switching to encrypted traffic.

10.10 Measuring post-quantum TLS on an emulated network

In our experiments we use the example TLS client and server implementations provided by Rustls. We modified the client to connect specified number of times instead of just once. We have also modified the client and server implementations to, e.g., allow specifying cached certificates. We instrumented the handshake implementations to print the number of nanoseconds that have elapsed, starting from either sending or receiving the initial message until operations of interest for both the client and the server.

Part of the measurement setup is a script[6] that prepares all the experiments we are interested in. As the version of Rustls that our implementations are based on hard-codes the client's default ephemeral key-exchange algorithm, we replace the default based on our settings.[7] We then simply compile the example TLS server and client applications for every different key exchange method. We also generate the certificates necessary for the experiment. To make sure all of this is reproducible, we execute all these steps in an isolated Docker container.[8] This fixes the Rust compiler version and isolates the compilation from the host operating system. As Rust statically links binaries, we can use the binaries generated in the container without having to be too careful about keeping in sync with a dynamically linked TLS library.

We follow the same methodology as [281] for setting up emulated networks.[9] The measurements are done using the Linux kernel's network namespacing [59] and network emulation (`NetEm`) features [174]. We create network namespaces for the clients and the servers and create virtual network interfaces in those namespaces. We vary the latency and bandwidth of the emulated network. `NetEm` adds a latency to the *outgoing* packets, so to add a latency of x ms, we add $x/2$ ms of latency to the client and server interfaces; following [281], we consider RTTs of 30.9 ms (representing an transcontinental connection) and 195.5 ms (representing a transpacific connection). We also throttle the bandwidth of the virtual interfaces, considering both 1000 Mbps and 10 Mbps connections. We do not vary the packet loss rate, fixing it at 0 %.

[6] Please refer to `measuring/scripts/experiment.py` in our experiment codebase.
[7] Refer to `measuring/scripts/create-experimental-setup.sh`.
[8] Refer to `Dockerfile` in the repository root.
[9] Refer to `measuring/scripts/setup_ns.sh`.

We ran measurements on a server with two Intel Xeon Gold 6230 (Cascade Lake) CPUs, each featuring 20 physical cores, which gives us 80 hyperthreaded cores in total. For the measurements, we run 40 clients and servers in parallel, such that each process has its own (hyperthreaded) core. We measured 20 000 handshakes for each scheme and set of network parameters.

We also report the sizes of the experimental handshakes. To obtain these numbers, we run the generated TLS client and server with the certificates relevant to the handshake over localhost.[10] We record and process the transmitted TCP packets using `tshark` [356], and use a small Python script to extract the handshake metrics.

10.10.1 Measured network scenarios

In our experiments on the emulated network, we simulate two network environments, following the choices made in [281]. The first environment, which represents a high-bandwidth transcontinental connection, uses a network round-trip latency of 30.9 ms and a network bandwidth of 1000 Mbps. The second environment resembles a transpacific, low-bandwidth connection and uses a network round-trip latency of 195.5 ms and has a bandwidth of 10 Mbps. We do not vary the packet loss rates, as this would mostly affect the results at higher percentiles which we do not report.

[10] Refer to `measuring/scripts/measure-handshake-size.sh`.

11 Performance of post-quantum TLS

In this chapter, we investigate the performance of TLS 1.3 when instantiated with post-quantum primitives. In the chapters that follow, we will compare it with the performance of OPTLS, KEMTLS, and KEMTLS-PDK. We will also compare the instantiations at different security levels, for both unilaterally as well as mutually authenticated handshakes.

Note that we are only comparing the sizes and performance characteristics of the schemes in this and other chapters. It can be argued if this is fair: for example, hash-based schemes, which are generally slow and large, are based on assumptions that are considered to be much more conservative than those assumptions on which much-faster lattice-based schemes are based. However, although one might say a particular scheme performs "best", these comparisons are still useful to estimate the cost of more conservative approaches.

For a description of how we implemented post-quantum TLS, please refer back to section 10.2. The design of the emulated network environment and our choice of parameters are motivated in section 10.10.

11.1 Selecting algorithms for experiments

There are many post-quantum KEM and signature schemes that we could use for our experiments. We select some instantiations that we think are interesting, which we will introduce in this section. As a baseline, we use an instantiation based on ECDH and RSA. We mainly use the algorithms selected for standardization in the NIST PQC standardization project, as well as the remaining round-4 finalists for KEMs [6]. Separately from the NIST PQC standardization project, NIST and the IETF have already standardized stateful hash-based signature schemes XMSS [106, 181] and LMS [106, 249]. These stateful hash-based signature schemes are as conservative as SPHINCS$^+$ but much smaller, so we will present some instantiations that make use of XMSS^{MT}. However, as their stateful nature makes them very sensitive to

user error, we restrict their use to CA certificates. As XMSS only has standardized parameter sets at above NIST PQC security level V, we selected customized parameters for use in CA certificates at the different security levels in section 11.6.

11.1.1 Instantiations of post-quantum TLS 1.3

In each instantiation, we select:

1. an algorithm for ephemeral key exchange, negotiated by the TLS 1.3 client and server;
2. an algorithm for handshake authentication, used in the server's certificate;
3. an algorithm for authentication of the server's certificate by the (intermediate) CA certificate, which we may assume the client to already have;
4. an algorithm to authenticate the intermediate CA certificate by a root CA certificate, which is always assumed to be preinstalled.

For TLS handshakes that use mutual authentication, we additionally select:

5. an algorithm for client authentication, used in the client certificate;
6. an algorithm for authentication of the client certificate by a CA certificate, which is assumed to be preinstalled.

In our experiments, we will try to showcase how algorithms in the NIST PQC standardization project perform, as well as highlight how some careful choices for certificate algorithms can make large differences. We will use the following scenarios:

Pre-quantum The pre-quantum instantiation uses X25519 [41] for key exchange and RSA-2048 [303] for all signatures.

Primary For ephemeral key exchange, this instantiation uses Kyber [319], the only KEM which was selected for standardization for post-quantum key exchange. Dilithium [241], the algorithm which, when it was selected for standardization, was named the *primary* algorithm for post-quantum signatures, is used for all signatures.

Falcon This instantiation uses Kyber for ephemeral key exchange, and Falcon [293] for all signatures. Falcon was also selected for standardization by NIST but its use is not recommended unless its implementation concerns can be properly addressed: Falcon is very sensitive to side channels and requires constant-time 64-bit floating-point operations for signing.

Falcon offline This instantiation uses Kyber for ephemeral key exchange and Dilithium for the (online) handshake signatures of the server and, if mutually authenticating, the client. The CA signatures in certificates are instantiated using Falcon. Because these can be produced offline by the CA, we can assume they can mitigate all implementation concerns: signature verification does not have Falcon's implementation considerations.

SPHINCS$^+$-f This instantiation uses Kyber for ephemeral key exchange. For all signatures SPHINCS$^+$ [184] is used, which is the only NIST selection for standardization that is not based on lattice assumptions. Specifically, this instantiation uses the *fast* variant of SPHINCS$^+$, which has faster runtime but larger signatures. For the hash function, we use Haraka, as explained in section 10.6.

SPHINCS$^+$-s This instantiation is like the SPHINCS$^+$-f-variant, but it uses the *small* variant of SPHINCS$^+$. This variant requires more computation time for signing but has significantly smaller signatures.

Hash-based signatures This conservative instantiation uses Kyber for ephemeral key exchange and SPHINCS$^+$ for the handshake authentication signatures. To minimize the size, we use a custom instantiation of XMSS^{MT} at the appropriate NIST security level for the CA signatures. These instances, which we label XMSS$^{MT}_s$, are described in section 11.6.

Hash-based CA This instantiation uses Kyber for key exchange and Dilithium for the handshake authentication signatures. To minimize the size, we use a custom instantiation of XMSS^{MT} at the appropriate NIST security level for the CA signatures.

HQC This instantiation uses HQC, a round-4 KEM candidate in the NIST PQC standardization project, for ephemeral key exchange. HQC relies

on assumptions based on decoding of quasi-cyclic codes, instead of on assumptions on lattices. For handshake authentication and CA signatures, Dilithium is used.

Note that for presentation purposes, we will refer to these scenarios in our tables and figures by handles, which are composed of the first letters of each of the selected algorithms (though X25519 is represented by the letter 'e' for ECDH). As an example, we denote by KDDD the instantiation that uses Kyber for ephemeral key exchange and uses Dilithium for server authentication and the intermediate and root CA certificates. For an overview of these handles, refer to the tables that show the communication sizes, e.g. table 11.1.

The remaining candidates for post-quantum key exchange in round 4 of the NIST PQC standardization project, are unfortunately not suitable for our experiments. BIKE [17] does not have IND-CCA-secure parameters available, and the public keys of Classic McEliece [7] are too large to use in TLS 1.3.

Note that the use of intermediate CA certificates is not required. Alternatively, there exist proposals in which the intermediate certificate can be cached: the client can then request intermediate CA certificates to not be transferred [202]. To represent these scenarios, we will also show results for experiments that use the intermediate certificate as the root certificate, and thus do not transmit or verify the root certificate.

11.2 Instantiation and results at NIST level I

We measured and compare the performance of TLS 1.3 at NIST security level I. This security level offers security comparable to that given by AES-128. As it is the lowest security level, the parameter sets are the most aggressively chosen. They generally offer the smallest public key, ciphertext, and signature sizes and the shortest computation times. First, we will cover unilaterally authenticated handshakes, in which only the server is authenticated by a certificate and a signature. This scenario is very important to the web, as this is the handshake mode that is almost exclusively used by web browsers [61]. Afterward, we will discuss mutually authenticated handshakes, in which the client also presents a certificate of their identity and signs the handshake. Although this setting is not very relevant to web browsing, it is for example used to secure service-to-service communication or in VPNs.

11.2.1 Unilaterally authenticated TLS 1.3

Table 11.1: Instantiations at NIST level I of unilaterally authenticated post-quantum TLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**		**Int. CA public key**	**Root CA signature**		**Root CA public key**
handle	pk+ct	pk+sig	sig	**Sum**	pk	sig	**Sum**	pk
Pre-quantum	X25519	RSA-2048	RSA-2048	848	RSA-2048	RSA-2048	1 376	RSA-2048
errr	64	528	256		272	256		272
Primary	Kyber-512	Dilithium2	Dilithium2	7 720	Dilithium2	Dilithium2	11 452	Dilithium2
KDDD	1568	3732	2420		1312	2420		1312
Falcon	Kyber-512	Falcon-512	Falcon-512	3 797	Falcon-512	Falcon-512	5 360	Falcon-512
KFFF	1568	1563	666		897	666		897
Falcon offline	Kyber-512	Dilithium2	Falcon-512	5 966	Falcon-512	Falcon-512	7 529	Falcon-512
KDFF	1568	3732	666		897	666		897
SPHINCS$^+$-f	Kyber-512	SPHINCS$^+$-128f	SPHINCS$^+$-128f	35 776	SPHINCS$^+$-128f	SPHINCS$^+$-128f	52 896	SPHINCS$^+$-128f
KSfSfSf	1568	17 120	17 088		32	17 088		32
SPHINCS$^+$-s	Kyber-512	SPHINCS$^+$-128s	SPHINCS$^+$-128s	17 312	SPHINCS$^+$-128s	SPHINCS$^+$-128s	25 200	SPHINCS$^+$-128s
KSsSsSs	1568	7888	7856		32	7856		32
Hash-based signatures	Kyber-512	SPHINCS$^+$-128s	XMSS$_s^{MT}$-I	10 435	XMSS$_s^{MT}$-I	XMSS$_s^{MT}$-I	11 446	XMSS$_s^{MT}$-I
KSsXX	1568	7888	979		32	979		32
HBS-CA	Kyber-512	Dilithium2	XMSS$_s^{MT}$-I	6 279	XMSS$_s^{MT}$-I	XMSS$_s^{MT}$-I	7 290	XMSS$_s^{MT}$-I
KDXX	1568	3732	979		32	979		32
HQC	HQC-128	Dilithium2	Dilithium2	12 882	Dilithium2	Dilithium2	16 614	Dilithium2
HDDD	6730	3732	2420		1312	2420		1312

Communication requirements

In table 11.1, we show the communication sizes of our choices of instantiations. We also give the abbreviated handles by which we will refer to the instantiations in other tables. We give the sum of the data necessary for the ephemeral key exchange, the handshake authentication signature, and the leaf certificate, as this is the minimum amount of data, excluding protocol overhead, that needs to be transferred if no intermediate CA certificates are required. In these experiments, the intermediate CA certificate is assumed to be used as the trusted root certificate. We also give the total amount of public key data

that is transmitted when an intermediate CA certificate is included.

Post-quantum ephemeral key exchange requires much more data than ECDH, and post-quantum signatures have much larger public key and signature sizes than RSA-2048. Falcon is the smallest general-purpose post-quantum signature scheme, while Dilithium is much larger. Finally, we see that the schemes that do not rely on lattice assumptions are much larger than their lattice equivalents. Using HQC-128 instead of Kyber-512 for key exchange requires 5162 additional bytes. The variants based on hash-based signatures also require much more data. The only exception is our custom XMSS parameter set, which appears as an attractive option to reduce the size of the certificate chain when used for CA certificates: using $\text{XMSS}_{\text{s}}^{\text{MT}}$-I in place of SPHINCS$^+$-128s saves 6877 bytes of data (87.5 %). Using $\text{XMSS}_{\text{s}}^{\text{MT}}$-I in place of Dilithium2 saves 1441 bytes of data (59.5 %).

Computational requirements

In table 11.2, we compare the amount of computation that each combination of algorithms requires. The amount given for client operations is the sum of the key generation and decapsulation operations for the ephemeral key exchange, the verification time of the handshake signature, the verification time for the leaf certificate, and if an intermediate certificate is transmitted, the verification time of the intermediate certificate. For reference, the time of individual key-generation, signing, verification, encapsulation, and decapsulation operations are given in tables 10.1 and 10.2.

Comparing the lattice-based experiments KDDD and KFFF with the pre-quantum errr instantiation, it is evident that while post-quantum cryptography may be bigger, it is not necessarily also *slower*: KFFF performs comparable with errr, while KDDD uses much less computation time. However, this does not hold for all schemes. While HQC requires only slightly more computation time than the Kyber-based experiment, the experiments that use hash-based signatures require much more time. But also between the hash-based schemes there exist large differences. The smaller variant of SPHINCS$^+$ requires vastly more computation: to produce the handshake signature with SPHINCS$^+$-128s instead of SPHINCS$^+$-128f requires 54.998 ms more computation (90.9 %). Our custom $\text{XMSS}_{\text{s}}^{\text{MT}}$-I parameters have been tuned for as small a signature as possible, and this is also clearly visible in the computation time: the cost of verifying XMSS signatures adds significantly to the client's computation time.

Table 11.2: Computation time in ms for asymmetric cryptography at NIST level I for each of the unilaterally authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr	0.134	0.629	0.763	0.150	0.629	0.779
KDDD	0.166	0.204	0.370	0.230	0.204	0.434
KFFF	0.320	0.668	0.988	0.461	0.668	1.129
KDFF	0.243	0.204	0.447	0.384	0.204	0.588
KSfSfSf	0.924	5.544	6.468	1.367	5.544	6.911
KSsSsSs	0.218	60.542	60.760	0.308	60.542	60.850
KSsXX	8.360	60.542	68.902	16.592	60.542	77.134
KDXX	8.334	0.204	8.538	16.566	0.204	16.770
HDDD	0.461	0.317	0.778	0.525	0.317	0.842

Handshake performance

In table 11.3, we can see the average times taken in the handshakes instantiated with our selected algorithms for a high-bandwidth, low-latency connection, using a latency of 30.9 ms and a 1000 Mbps link speed. We again give times for when the intermediate CA certificate algorithm is not transmitted and thus used as a root CA certificate, and for the scenario in which the intermediate CA does need to be transmitted and verified. In our experiments, we assume an HTTP-like scenario in which the client requests some data from the server, so the server needs to receive the client's request before it can start transmitting application traffic. In each of these scenarios, we give the amount of time until the client is ready to send a request (i.e., in TLS 1.3, the client has received `ServerFinished` and sent `ClientFinished`), the time until the client receives the response to its request from the server, and the time until the server has completed the handshake (i.e., in TLS 1.3, received `ClientFinished`). Note that the timers of the client and the server are independent, and start counting as soon as the `ClientHello` message is constructed (for the client) or received (for the server).

For the instantiations that use pre-quantum cryptography, or any combination of the fast algorithms Kyber-512, HQC-128, Dilithium2, and Falcon-512,

Table 11.3: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level I with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	65.9	97.0	35.0	66.1	97.2	35.2
KDDD	63.6	94.8	32.7	63.9	95.0	33.0
KFFF	64.6	95.8	33.7	65.0	96.1	34.0
KDFF	63.7	94.8	32.8	64.0	95.2	33.1
KSfSfSf	106.4	137.6	75.5	136.9	168.1	106.0
KSsSsSs	166.7	197.7	135.8	166.9	198.0	136.0
KSsXX	155.5	186.6	124.6	171.1	202.1	140.1
KDXX	97.2	128.3	66.2	113.7	144.8	82.8
HDDD	63.6	94.7	32.7	63.9	95.1	33.0

Table 11.4: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level I with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	397.1	593.1	201.3	397.7	593.7	201.8
KDDD	405.5	602.8	208.8	410.3	610.0	212.8
KFFF	397.3	593.3	200.8	399.5	595.5	203.0
KDFF	398.7	594.7	202.2	405.6	602.8	208.8
KSfSfSf	1177.3	1544.4	963.5	1751.4	2038.1	1524.6
KSsSsSs	914.0	1116.4	715.3	979.2	1217.9	772.5
KSsXX	530.6	735.6	331.2	564.9	776.6	364.9
KDXX	446.9	648.1	240.6	475.4	679.7	266.1
HDDD	405.5	602.7	208.7	410.5	610.1	212.9

we see that the handshake times are roughly a multiple of the connection latency. The client can transmit its request after two times the handshake latency, which matches the two round-trips necessary: one for the TCP connection establishment, and the single round-trip in which the TLS handshake is completed. The response is received in the return round-trip after sending off the request. The server receives `ClientFinished` and thus completes its part of the handshake a single round-trip after transmitting the `ServerFinished` packet. The computational requirements of these algorithms on the chosen platforms are so small that they do not meaningfully contribute to the connection establishment times. With the connection parameters in this experiment, the additional amounts of traffic required for Dilithium2 compared to Falcon-512 or HQC-128 compared to Kyber-512 do not meaningfully contribute to the handshake time.

In the KDXX parameter set which uses $\text{XMSS}_\text{s}^\text{MT}$-I for the CA certificates to reduce the amount of handshake traffic, the additional computation time required to verify the XMSS signatures adds significantly to the handshake time. The additional latency for the experiment in which the root CA is omitted, compared to the KDDD experiment, is more than the 8.2 ms that are required on average to verify the $\text{XMSS}_\text{s}^\text{MT}$-I signature on the server's leaf certificate. We suspect this happens because the additional verification time interacts with the TCP congestion control algorithms. We see a similar delay in the KSsSsSs parameter set, but this instantiation additionally suffers from the larger amount of bytes that need to be transmitted. The amount of data for this selection of algorithms exceeds the initial congestion window (`initcwnd`) set in the TCP slow start algorithm [66], which is the initial limit on the amount of data (measured in MSS) that can be sent on a TCP connection before receiving an acknowledgment packet. The default `initcwnd` on Linux is 10 MSS, which means that after transmitting about 14.5 kB, the server needs to wait for the client to acknowledge the packets that it has received before it will send more. This induces extra round-trip delays. As the instance using SPHINCS^+-128f has very large certificates due to the large signature size of SPHINCS^+-128f, these extra round-trips slow down the connection establishment despite the much faster signing time compared to SPHINCS^+-128s. Still, for this high-bandwidth, low-latency connection, the SPHINCS^+-128f instance is much faster than any instance using SPHINCS^+-128s: the computational requirements are just too large compared to the communication overhead. When including the intermediate CA certificate, KSfSfSf requires 29.9 ms

(15.1 %) less time than KSsSsSs before the client receives the server's response.

In table 11.4, we compare the same metrics for experiments on a high-latency, low-bandwidth connection, using a latency of 195.5 ms latency and 10 Mbps connection bandwidth. With these connection characteristics, we see that the sizes of the public keys, ciphertexts, and signatures start to matter more. KFFF, which has the smallest sum of public-key cryptography objects, has the best performance, coming very close to the performance of the pre-quantum errr instantiation. Comparatively, KDDD, which suffers from the much larger Dilithium2 public keys and signatures, has higher connection establishment times. When including the intermediate CA certificate, KDDD takes an additional 16.3 ms (2.7 %) before the client receives the response from the server, compared to the pre-quantum instance. The performance of the Kyber-512/SPHINCS$^+$-128f instantiation KSfSfSf again clearly suffers under the weight of SPHINCS$^+$-128f signatures, now showing the worst performance of all parameter sets.

The two instantiations that use Kyber-512 and Dilithium2 for the online components of the TLS 1.3 handshake, but rely on different algorithms for CA certificates, KDFF and KDXX, appear promising: by reducing the amount of handshake traffic, they perform fairly reasonably well (though the performance of XMSS$_s^{MT}$-I still hurts KDXX). Especially KDFF, a combination of two to-be-standardized algorithms, seems to combine the best of both worlds: using Dilithium2 for the online handshake authentication avoids the implementation concerns associated with Falcon-512 signature generation while offering performance on par with the all-Falcon KFFF instantiation but even KDXX is only slightly slower.

11.2.2 Mutually authenticated TLS 1.3

Communication requirements

Table 11.5 shows the precise selection of algorithms for our instantiations of mutually authenticated post-quantum TLS 1.3. Like for the unilaterally authenticated instantiations, we also show the sizes of the public-key cryptographic elements necessary for ephemeral key exchange, server authentication, and client authentication, and the total. (For a better presentation, we only show the sum of the size of the public key and the signature that are part of each certificate, even if they use different algorithms). Again, we examine two scenarios for each instantiation, one that omits an intermediate CA certificate

Table 11.5: Instantiations at NIST level I of mutually authenticated post-quantum TLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	**Key exchange**	**Server authentication**	**Client authentication**	**Sum**	**Int. CA certificate**	**Sum**
Pre-quantum	X25519	hs:RSA-2048	hs:RSA-2048		pk:RSA-2048	
		sig:RSA-2048	sig:RSA-2048	1 632	sig:RSA-2048	2 160
errr-rr	64	784	784		528	
Primary	Kyber-512	hs:Dilithium2	hs:Dilithium2		pk:Dilithium2	
		sig:Dilithium2	sig:Dilithium2	13 872	sig:Dilithium2	17 604
KDDD-DD	1568	6152	6152		3732	
Falcon	Kyber-512	hs:Falcon-512	hs:Falcon-512		pk:Falcon-512	
		sig:Falcon-512	sig:Falcon-512	6 026	sig:Falcon-512	7 589
KFFF-FF	1568	2229	2229		1563	
Falcon offline	Kyber-512	hs:Dilithium2	hs:Dilithium2		pk:Falcon-512	
		sig:Falcon-512	sig:Falcon-512	10 364	sig:Falcon-512	11 927
KDFF-DF	1568	4398	4398		1563	
SPHINCS$^+$-f	Kyber-512	hs:SPHINCS$^+$-128f	hs:SPHINCS$^+$-128f		pk:SPHINCS$^+$-128f	
		sig:SPHINCS$^+$-128f	sig:SPHINCS$^+$-128f	69 984	sig:SPHINCS$^+$-128f	87 104
KSfSfSf-SfSf	1568	34 208	34 208		17 120	
SPHINCS$^+$-s	Kyber-512	hs:SPHINCS$^+$-128s	hs:SPHINCS$^+$-128s		pk:SPHINCS$^+$-128s	
		sig:SPHINCS$^+$-128s	sig:SPHINCS$^+$-128s	33 056	sig:SPHINCS$^+$-128s	40 944
KSsSsSs-SsSs	1568	15 744	15 744		7888	
Hash-based signatures	Kyber-512	hs:SPHINCS$^+$-128s	hs:SPHINCS$^+$-128s		pk:XMSS$_s^{MT}$-I	
		sig:XMSS$_s^{MT}$-I	sig:XMSS$_s^{MT}$-I	19 302	sig:XMSS$_s^{MT}$-I	20 313
KSsXX-SsX	1568	8867	8867		1011	
HBS-CA	Kyber-512	hs:Dilithium2	hs:Dilithium2		pk:XMSS$_s^{MT}$-I	
		sig:XMSS$_s^{MT}$-I	sig:XMSS$_s^{MT}$-I	10 990	sig:XMSS$_s^{MT}$-I	12 001
KDXX-DX	1568	4711	4711		1011	
HQC	HQC-128	hs:Dilithium2	hs:Dilithium2		pk:Dilithium2	
		sig:Dilithium2	sig:Dilithium2	19 034	sig:Dilithium2	22 766
HDDD-DD	6730	6152	6152		3732	

hs: certificate public key and handshake signature
pk: certificate public key sig: certificate signature

(and thus uses it as the trusted root certificate) and one that makes use of intermediate CA certificates, transmitting them during the handshake.

The difference in size between the unilaterally authenticated handshakes in table 11.1 and the mutually authenticated handshakes is exactly the number of bytes listed in the client authentication column. As the size of the Kyber-512 key exchange is much smaller than the sum of a public key and signature used in most of our instantiations, we roughly double the sizes of the handshakes when omitting intermediate certificates.

Table 11.6: Computation time in ms for asymmetric cryptography at NIST level I for each of the mutually authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr-rr	0.660	0.661	1.321	0.676	0.661	1.337
KDDD-DD	0.344	0.332	0.676	0.408	0.332	0.740
KFFF-FF	0.962	0.950	1.912	1.103	0.950	2.053
KDFF-DF	0.421	0.409	0.830	0.562	0.409	0.971
KSfSfSf-SfSf	6.442	6.430	12.872	6.885	6.430	13.315
KSsSsSs-SsSs	60.734	60.722	121.456	60.824	60.722	121.546
KSsXX-SsX	68.876	68.864	137.740	77.108	68.864	145.972
KDXX-DX	8.512	8.500	17.012	16.744	8.500	25.244
HDDD-DD	0.639	0.445	1.084	0.703	0.445	1.148

Computational requirements

Table 11.6 shows the amount of computation required for the cryptographic operations using the algorithms in our instantiations. As the client and the server now both need to produce a signature during the handshake and verify a certificate chain, they need to perform the same amount of work when an intermediate CA certificate is not used. Otherwise, the client needs to additionally verify this certificate. As signing is much more expensive than verifying for most algorithms, we see that the total computational load roughly doubles compared to the unilaterally authenticated experiments.

Table 11.7: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level I with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	68.9	100.3	38.3	68.9	100.4	38.3
KDDD-DD	64.2	95.8	33.7	64.4	96.0	34.0
KFFF-FF	66.1	97.9	35.8	66.4	98.2	36.1
KDFF-DF	64.2	96.0	33.9	64.6	96.3	34.2
KSfSfSf-SfSf	117.8	182.3	120.2	148.2	212.8	150.7
KSsSsSs-SsSs	247.9	310.1	248.2	248.2	310.5	248.5
KSsXX-SsX	213.1	254.8	192.9	221.4	263.1	201.2
KDXX-DX	98.7	160.2	98.2	113.4	170.5	108.5
HDDD-DD	64.2	95.8	33.8	64.5	96.1	34.1

Table 11.8: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level I with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	399.9	597.2	205.2	401.0	598.4	206.1
KDDD-DD	404.5	606.9	212.9	412.0	617.7	217.9
KFFF-FF	399.1	597.8	205.3	402.8	601.6	208.8
KDFF-DF	399.8	599.9	207.3	405.1	606.2	211.7
KSfSfSf-SfSf	1224.6	1997.1	1397.7	1598.9	2339.3	1660.3
KSsSsSs-SsSs	849.3	1292.2	877.1	864.8	1296.9	873.4
KSsXX-SsX	583.5	801.8	398.5	594.9	812.3	406.9
KDXX-DX	440.4	671.2	272.1	463.5	692.3	292.1
HDDD-DD	403.2	605.5	211.8	411.3	616.7	217.2

Handshake performance

In tables 11.7 and 11.8, we show the performance of our instantiations of the mutually authenticated TLS 1.3 running on a 30.9 ms latency, 1000 Mbps network and on a 195.5 ms latency, 10 Mbps network. Note that even though the transmission size of some of the instances, when including client authentication, exceeds 15 kB, we point out that the initial congestion window is not shared between the client and the server: thus the client is free to send as much data as it can fit in its congestion window regardless of the size of the server's certificate. Furthermore, the client needs to fully receive the server's certificate and verify the server's handshake signature before it may send the client certificate. This means that TCP congestion control has a chance to catch up and the client's certificate size does not contribute as much to the connection congestion; unlike the server's certificate which is all sent out immediately as the connection is established. Additionally, as the client sends out its request immediately after it transmits its certificate, the server first needs to process the client certificate before it can process the client request. This is visible in our measurements as an increased gap between the client sending its request and receiving the response. We see that large certificates, such as in the instance based on SPHINCS$^+$-128f, especially contribute to longer waiting times before the client receives its response from the server. The KSfSfSf-SfSf experiment is 44.8 ms (32.6 %) slower than the unilaterally authenticated KSfSfSf experiment when omitting intermediate CA certificates on the low-latency network. On the slow network, comparing the same two SPHINCS$^+$-128f experiments shows a 452.7 ms (29.3 %) difference. Note that the relative difference is very similar (while naively we would expect the larger size to take longer), which is due to the TCP congestion control algorithm already having had a chance to scale the client's bandwidth before the client certificate is transmitted: the large server certificate results in acknowledgments being sent out that influence congestion control.

11.3 Instantiation and results at NIST level III

In this section, we measure and compare the performance of TLS 1.3 at NIST PQC security level III. This security level corresponds to, roughly, AES-192 in terms of security. This category is significant, as, among others, the authors of Kyber and Dilithium have recommended using the level-III instantiations

of their schemes [121, 228]. Again, we first examine unilaterally authenticated handshakes, before we look at mutually authenticated handshakes.

11.3.1 Unilaterally authenticated TLS 1.3

Table 11.9: Instantiations at NIST level III of unilaterally authenticated post-quantum TLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**		**Int. CA public key**	**Root CA signature**		**Root CA public key**
handle	pk+ct	pk+sig	sig	**Sum**	pk	sig	**Sum**	pk
Pre-quantum	X25519	RSA-2048	RSA-2048	848	RSA-2048	RSA-2048	1 376	RSA-2048
errr	64	528	256		272	256		272
Primary	Kyber-768	Dilithium3	Dilithium3	10 810	Dilithium3	Dilithium3	16 055	Dilithium3
KDDD	2272	5245	3293		1952	3293		1952
Falcon	Kyber-768	Falcon-1024	Falcon-1024	6 625	Falcon-1024	Falcon-1024	9 698	Falcon-1024
KFFF	2272	3073	1280		1793	1280		1793
Falcon offline	Kyber-768	Dilithium3	Falcon-1024	8 797	Falcon-1024	Falcon-1024	11 870	Falcon-1024
KDFF	2272	5245	1280		1793	1280		1793
SPHINCS$^+$-f	Kyber-768	SPHINCS$^+$-192f	SPHINCS$^+$-192f	73 648	SPHINCS$^+$-192f	SPHINCS$^+$-192f	109 360	SPHINCS$^+$-192f
KSfSfSf	2272	35 712	35 664		48	35 664		48
SPHINCS$^+$-s	Kyber-768	SPHINCS$^+$-192s	SPHINCS$^+$-192s	34 768	SPHINCS$^+$-192s	SPHINCS$^+$-192s	51 040	SPHINCS$^+$-192s
KSsSsSs	2272	16 272	16 224		48	16 224		48
Hash-based signatures	Kyber-768	SPHINCS$^+$-192s	XMSS$^{MT}_s$-III	20 395	XMSS$^{MT}_s$-III	XMSS$^{MT}_s$-III	22 294	XMSS$^{MT}_s$-III
KSsXX	2272	16 272	1851		48	1851		48
HBS-CA	Kyber-768	Dilithium3	XMSS$^{MT}_s$-III	9 368	XMSS$^{MT}_s$-III	XMSS$^{MT}_s$-III	11 267	XMSS$^{MT}_s$-III
KDXX	2272	5245	1851		48	1851		48
HQC	HQC-192	Dilithium3	Dilithium3	22 086	Dilithium3	Dilithium3	27 331	Dilithium3
HDDD	13 548	5245	3293		1952	3293		1952

Communication requirements

In table 11.9, we show the communication sizes of our choices of instantiations. Comparing the instantiations with those at NIST level I, we see that the sizes become significantly larger across all schemes used in the instantiations. For example, Kyber-768 requires 704 bytes (44.9 %) more transmission. As

Dilithium2, which we used in our level-I experiments, already has security level II, we see a modest increase of 1513 bytes (40.5 %) for its public key and signature combined. For all of the other schemes, we get around 50 % increase in sizes; though it should be noted that Falcon-1024 offers NIST security level V.

Table 11.10: Computation time in ms for asymmetric cryptography at NIST level III for each of the unilaterally authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr	0.134	0.629	0.763	0.150	0.629	0.779
KDDD	0.203	0.830	1.033	0.284	0.830	1.114
KFFF	0.385	0.813	1.198	0.557	0.813	1.370
KDFF	0.294	0.830	1.124	0.466	0.830	1.296
KSfSfSf	0.841	6.261	7.102	1.241	6.261	7.502
KSsSsSs	0.311	118.885	119.196	0.446	118.885	119.331
KSsXX	12.075	118.885	130.960	23.974	118.885	142.859
KDXX	12.021	0.830	12.851	23.920	0.830	24.750
HDDD	1.190	1.263	2.453	1.271	1.263	2.534

Computational requirements

When comparing the computation requirements of these instantiations as listed in table 11.10, we see that although instances based on the lattice-based schemes Kyber, Dilithium, and Falcon see significant increases in computation time (Falcon takes about two times as much time), they still do not require much more time than the original pre-quantum instantiation using X25519 and RSA-2048. The two SPHINCS$^+$-192 variants also take around twice as much time, but as the amount of time taken was already quite large, we now see the server requires over 100 ms just for cryptographic computations in the instances using SPHINCS$^+$-192s for handshake authentication.

Handshake performance

This comes together in the average handshake timings shown in tables 11.11 and 11.12. Comparing the computation times, we see that the KFFF instance

performs slightly worse than the KDDD instance. Otherwise, we see that the instances largely follow the pattern established in the level-I experiments.

For the high-bandwidth connection, we see that again the sizes largely do not matter for all instantiations that stay under the limits of the TCP slow start algorithm. However, at NIST security level III, more instantiations now do exceed this limit. In the KDDD and HDDD instances in which an intermediate CA certificate is transferred, the penalty of an additional round-trip can be seen in the connection establishment times due to the large size of the certificates exceeding the approximately 14.5 kB congestion window transmission limit. When the intermediate CA certificate is used as root CA certificate, the `ServerCertificate` stays under the limit and the penalty is avoided.

In the experiments using the variants of SPHINCS$^+$-192, we see that the large sizes of the public keys and certificates affect the handshake performance. The SPHINCS$^+$-192s handshake has large increases in the handshake times as both the computation time and amount of data increase dramatically. The SPHINCS$^+$-192f handshake experiment including an intermediate CA certificate on the 30.9 ms latency, 1000 Mbps network takes 41.7 ms (24.8 %) longer before the client received the response at level III than at level I, much more than the increase in computation time. Note that the handshake time did not increase linearly with the 55 760 bytes (108.6 %) increase of the SPHINCS$^+$-192f public keys and signatures. The TCP congestion control algorithm increases the transmission window, and thus available bandwidth, as more packets get acknowledged. In the same experiment running over the 195.5 ms latency, 10 Mbps network, we see that the increase in handshake time in the same experiment is 87.1 %. This is much more in line with the increase in data that is transmitted, due to the lower connection bandwidth.

As the sizes of the handshakes go up, the benefits increase of choosing the algorithms for CA certificates such that the sizes are minimized. The KDFF instance is the best-performing post-quantum instance on the fast network when an intermediate certificate is transferred, balancing the computational speed of Dilithium3 with the smaller key sizes of Falcon-1024. On the low-bandwidth network, the size of Dilithium3 is too large to keep up with the KFFF instance, and the larger size slows down the handshake time more than the increase in Falcon's computational requirements: KDFF takes 17.8 ms (2.8 %) longer before the client response is received. Even the KDXX instance, which uses the slow-to-verify XMSS$_s^{MT}$-III scheme in the CA certificates,

Table 11.11: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level III with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	65.9	97.0	35.0	66.1	97.2	35.2
KDDD	64.0	95.1	33.1	94.6	125.8	63.7
KFFF	66.5	97.6	35.5	67.0	98.1	36.0
KDFF	64.3	95.4	33.3	65.0	96.1	34.0
KSfSfSf	145.6	176.7	114.6	178.5	209.7	147.6
KSsSsSs	215.3	246.3	184.4	248.1	279.2	217.2
KSsXX	223.3	254.3	192.3	231.9	262.9	201.0
KDXX	103.5	134.6	72.6	117.1	148.1	86.1
HDDD	64.0	95.2	33.1	94.6	125.8	63.7

Table 11.12: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level III with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	397.1	593.1	201.3	397.7	593.7	201.8
KDDD	409.1	607.6	211.6	881.0	1077.0	674.2
KFFF	401.3	597.3	204.8	428.2	631.1	229.5
KDFF	403.2	599.9	206.5	413.6	613.3	215.4
KSfSfSf	2028.4	2510.0	1713.2	3427.2	3814.2	3147.7
KSsSsSs	1156.5	1490.4	938.0	1776.4	2044.8	1553.1
KSsXX	869.0	1083.5	666.6	876.9	1098.1	672.4
KDXX	456.1	659.4	249.7	495.2	703.7	282.2
HDDD	408.4	606.8	211.1	881.0	1077.0	674.2

performs better than KDDD when intermediate CA certificates are used, showing the impact of the transmission overhead on low-bandwidth networks.

11.3.2 Mutually authenticated TLS 1.3

Communication requirements

In table 11.13, we show the communication requirements of our instantiations when using mutual authentication. As with the server authentication requirements in the unilaterally authenticated handshakes, the sizes of the public keys and signatures required for client authentication grow significantly. Already, no post-quantum instantiation has a total handshake size under 10 000 B when excluding intermediate CA certificates, and when including them, only the Falcon-1024-based KFFF-FF instance is just below 15 kB

Computational requirements

Table 11.14 shows the computational requirements for the server and the client. We see that using SPHINCS$^+$-192s for client authentication comes with a very significant computational cost, which now exceeds 100 ms for both the client and the server. We note that although TLS servers are often powerful computers that have stable power supplies, such computational overhead may have significant effects on the battery life of devices such as smartphones.

Handshake performance

Lastly, in tables 11.7 and 11.8, we show the handshake performance for mutually authenticated post-quantum TLS 1.3 handshakes at NIST security level III. For most instantiations, for the same reasons as in the unilaterally authenticated experiments, we do not see large differences between the results at the level I and level III security levels. Comparing the SPHINCS$^+$-192f-based instantiation to its level I instantiation when running on the 195.5 ms latency, 10 Mbps network shows that it takes 1809.6 ms (90.6 %) longer before the client receives its response than the level I instance using SPHINCS$^+$-128f (omitting intermediate CA certificates). The increase in size greatly affects the handshake performance on the low-bandwidth network.

On the high-bandwidth, low-latency network, the KSsXX-SsX instance got 170.2 ms (66.8 %) slower going from security level I to level III, while the unilaterally authenticated KSsXX instance got 67.7 ms (36.3 %) slower between the two security levels. On the high-latency, low-bandwidth network, this gap grows: the mutually authenticated instance was 552.4 ms (68.9 %)

Table 11.13: Instantiations at NIST level III of mutually authenticated post-quantum TLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	**Key exchange**	**Server authentication**	**Client authentication**	**Sum**	**Int. CA certificate**	**Sum**
Pre-quantum	X25519	hs:RSA-2048	hs:RSA-2048		pk:RSA-2048	
		sig:RSA-2048	sig:RSA-2048	1 632	sig:RSA-2048	2 160
errr-rr	64	784	784		528	
Primary	Kyber-768	hs:Dilithium3	hs:Dilithium3		pk:Dilithium3	
		sig:Dilithium3	sig:Dilithium3	19 348	sig:Dilithium3	24 593
KDDD-DD	2272	8538	8538		5245	
Falcon	Kyber-768	hs:Falcon-1024	hs:Falcon-1024		pk:Falcon-1024	
		sig:Falcon-1024	sig:Falcon-1024	10 978	sig:Falcon-1024	14 051
KFFF-FF	2272	4353	4353		3073	
Falcon offline	Kyber-768	hs:Dilithium3	hs:Dilithium3		pk:Falcon-1024	
		sig:Falcon-1024	sig:Falcon-1024	15 322	sig:Falcon-1024	18 395
KDFF-DF	2272	6525	6525		3073	
SPHINCS$^+$-f	Kyber-768	hs:SPHINCS$^+$-192f	hs:SPHINCS$^+$-192f		pk:SPHINCS$^+$-192f	
		sig:SPHINCS$^+$-192f	sig:SPHINCS$^+$-192f	145 024	sig:SPHINCS$^+$-192f	180 736
KSfSfSf-SfSf	2272	71 376	71 376		35 712	
SPHINCS$^+$-s	Kyber-768	hs:SPHINCS$^+$-192s	hs:SPHINCS$^+$-192s		pk:SPHINCS$^+$-192s	
		sig:SPHINCS$^+$-192s	sig:SPHINCS$^+$-192s	67 264	sig:SPHINCS$^+$-192s	83 536
KSsSsSs-SsSs	2272	32 496	32 496		16 272	
Hash-based	Kyber-768	hs:SPHINCS$^+$-192s	hs:SPHINCS$^+$-192s		pk:XMSS$_s^{MT}$-III	
signatures		sig:XMSS$_s^{MT}$-III	sig:XMSS$_s^{MT}$-III	38 518	sig:XMSS$_s^{MT}$-III	40 417
KSsXX-SsX	2272	18 123	18 123		1899	
HBS-CA	Kyber-768	hs:Dilithium3	hs:Dilithium3		pk:XMSS$_s^{MT}$-III	
		sig:XMSS$_s^{MT}$-III	sig:XMSS$_s^{MT}$-III	16 464	sig:XMSS$_s^{MT}$-III	18 363
KDXX-DX	2272	7096	7096		1899	
HQC	HQC-192	hs:Dilithium3	hs:Dilithium3		pk:Dilithium3	
		sig:Dilithium3	sig:Dilithium3	30 624	sig:Dilithium3	35 869
HDDD-DD	13 548	8538	8538		5245	

hs: certificate public key and handshake signature
pk: certificate public key sig: certificate signature

Table 11.14: Computation time in ms for asymmetric cryptography at NIST level III for each of the mutually authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr-rr	0.660	0.661	1.321	0.676	0.661	1.337
KDDD-DD	1.006	0.992	1.998	1.087	0.992	2.079
KFFF-FF	1.171	1.157	2.328	1.343	1.157	2.500
KDFF-DF	1.097	1.083	2.180	1.269	1.083	2.352
KSfSfSf-SfSf	7.075	7.061	14.136	7.475	7.061	14.536
KSsSsSs-SsSs	119.169	119.155	238.324	119.304	119.155	238.459
KSsXX-SsX	130.933	130.919	261.852	142.832	130.919	273.751
KDXX-DX	12.824	12.810	25.634	24.723	12.810	37.533
HDDD-DD	1.993	1.425	3.418	2.074	1.425	3.499

slower while the unilaterally authenticated instance was only 347.9 ms (47.3 %) slower. Somehow the combination of the increase in computational overhead and handshake size adds multiple round-trips worth of latency to the post-quantum TLS 1.3 handshake.

Table 11.15: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level III with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	68.9	100.3	38.3	68.9	100.4	38.3
KDDD-DD	64.9	96.6	34.6	95.4	127.2	65.1
KFFF-FF	69.1	101.4	39.4	69.8	102.2	40.1
KDFF-DF	65.1	97.2	35.2	65.7	97.8	35.8
KSfSfSf-SfSf	166.1	261.4	199.3	198.1	293.5	231.3
KSsSsSs-SsSs	361.6	424.3	362.4	396.4	459.2	397.2
KSsXX-SsX	362.7	425.0	363.1	364.2	426.6	364.6
KDXX-DX	105.6	176.0	114.0	117.7	181.5	119.5
HDDD-DD	64.9	96.7	34.6	95.4	127.2	65.1

Table 11.16: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level III with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	399.9	597.2	205.2	401.0	598.4	206.1
KDDD-DD	409.2	615.6	219.0	795.3	1067.3	559.5
KFFF-FF	404.1	605.1	212.6	410.1	612.3	217.0
KDFF-DF	404.6	607.5	213.3	411.2	616.5	217.6
KSfSfSf-SfSf	2198.3	3806.7	2982.4	3571.3	5450.1	4575.4
KSsSsSs-SsSs	1130.8	1682.2	1178.4	1466.6	2050.8	1489.1
KSsXX-SsX	918.9	1354.2	934.7	926.3	1359.7	939.4
KDXX-DX	459.4	707.9	301.5	496.0	744.6	325.5
HDDD-DD	408.9	615.4	218.8	788.7	1055.5	547.4

11.4 Instantiation and results at NIST level V

In this section, we examine the characteristics of post-quantum TLS instances at NIST PQC security level V. This is the most conservative security level and should offer comparable security to AES-256. This security level is required by the United States National Security Agency (NSA)'s Commercial National Security Algorithm Suite 2.0 [271]. The French national cybersecurity agency agence nationale de la sécurité des systèmes d'information (ANSSI) also recommends security level V [14]. As the most conservative parameter sets, these are generally the largest and slowest-running algorithms to instantiate post-quantum TLS 1.3 with. As such, it will be challenging to instantiate TLS with these algorithms without affecting performance significantly.

11.4.1 Unilaterally authenticated TLS 1.3

Communication requirements

In table 11.17, we show the communication sizes of our choices of instantiations. This level has the most conservative security guarantees, and as such the largest key sizes and computational requirements. If we compare the sums of the public-key cryptography elements that need to be transmitted, we see that even when leaving out the intermediate CA certificates, all but the KFFF instance exceed 10 kB. When we compare the sums including the intermediate CA certificate, all but KFFF and KDFF exceed 15 kB. SPHINCS$^+$-256f even exceeds 150 kB.

Computational requirements

When we compare the computation time that is necessary for the public-key cryptography operations at level V in table 11.18 with the level III requirements, it seems that the increase is less than going from level I to level III. Although HQC-256 requires 0.571 ms (38.4 %) more time than HQC-192, Kyber-1024 only requires 0.052 ms (76.5 %) more than Kyber-768. Dilithium5's signing time is almost identical to Dilithium3's. Falcon-1024, which we used in our level III experiments, already has security level V. The hash-based schemes SPHINCS$^+$ and XMSS$_s^{MT}$ use the same hash primitives for level III and level V, but they truncate the output of the hash function to 192 bytes at level III, while obtaining 256 bytes at level V. SPHINCS$^+$-256s additionally has a slightly differently shaped tree, which optimizes differently for signing and verification

Table 11.17: Instantiations at NIST level V of unilaterally authenticated post-quantum TLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**	**Sum**	**Int. CA public key**	**Root CA signature**	**Sum**	**Root CA public key**
handle	pk+ct	pk+sig	sig		pk	sig		pk
Pre-quantum	X25519	RSA-2048	RSA-2048	848	RSA-2048	RSA-2048	1 376	RSA-2048
errr	64	528	256		272	256		272
Primary	Kyber-1024	Dilithium5	Dilithium5	14 918	Dilithium5	Dilithium5	22 105	Dilithium5
KDDD	3136	7187	4595		2592	4595		2592
Falcon	Kyber-1024	Falcon-1024	Falcon-1024	7 489	Falcon-1024	Falcon-1024	10 562	Falcon-1024
KFFF	3136	3073	1280		1793	1280		1793
Falcon offline	Kyber-1024	Dilithium5	Falcon-1024	11 603	Falcon-1024	Falcon-1024	14 676	Falcon-1024
KDFF	3136	7187	1280		1793	1280		1793
SPHINCS$^+$-f	Kyber-1024	SPHINCS$^+$-256f	SPHINCS$^+$-256f	102 912	SPHINCS$^+$-256f	SPHINCS$^+$-256f	152 832	SPHINCS$^+$-256f
KSfSfSf	3136	49 920	49 856		64	49 856		64
SPHINCS$^+$-s	Kyber-1024	SPHINCS$^+$-256s	SPHINCS$^+$-256s	62 784	SPHINCS$^+$-256s	SPHINCS$^+$-256s	92 640	SPHINCS$^+$-256s
KSsSsSs	3136	29 856	29 792		64	29 792		64
Hash-based signatures	Kyber-1024	SPHINCS$^+$-256s	XMSS$_s^{MT}$-V	35 971	XMSS$_s^{MT}$-V	XMSS$_s^{MT}$-V	39 014	XMSS$_s^{MT}$-V
KSsXX	3136	29 856	2979		64	2979		64
HBS-CA	Kyber-1024	Dilithium5	XMSS$_s^{MT}$-V	13 302	XMSS$_s^{MT}$-V	XMSS$_s^{MT}$-V	16 345	XMSS$_s^{MT}$-V
KDXX	3136	7187	2979		64	2979		64
HQC	HQC-256	Dilithium5	Dilithium5	33 496	Dilithium5	Dilithium5	40 683	Dilithium5
HDDD	21 714	7187	4595		2592	4595		2592

time. The performance of these schemes is thus very similar at level III and V, with SPHINCS$^+$-256s signing being slightly faster than SPHINCS$^+$-192s.

Table 11.18: Computation time in ms for asymmetric cryptography at NIST level V for each of the unilaterally authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr	0.134	0.629	0.763	0.150	0.629	0.779
KDDD	0.398	0.633	1.031	0.559	0.633	1.192
KFFF	0.420	0.830	1.250	0.592	0.830	1.422
KDFF	0.409	0.633	1.042	0.581	0.633	1.214
KSfSfSf	0.870	11.406	12.276	1.267	11.406	12.673
KSsSsSs	0.444	104.611	105.055	0.628	104.611	105.239
KSsXX	11.295	104.611	115.906	22.330	104.611	126.941
KDXX	11.272	0.633	11.905	22.307	0.633	22.940
HDDD	1.753	1.217	2.970	1.914	1.217	3.131

Handshake performance

In tables 11.19 and 11.20, we show the handshake times for the instantiations at NIST security level V. In the results for the instances run on the 30.9 ms latency, 1000 Mbps network, we see that the experiments that are under the `initcwnd` size limit still complete in roughly the same amount of time, at multiples of the round trip latency. However, when including the intermediate certificate we see that all instances but KFFF and KDFF suffer at least an extra round-trip worth of handshake time due to exceeding the limit of the initial congestion window. Compared to the pre-quantum experiment, KDDD now requires 29.8 ms (30.6 %) more time if an intermediate certificate needs to be transmitted. In the low-bandwidth, 195.5 ms latency experiment KDDD even needs 488.1 ms (82.2 %) more time. Even the KFFF instance sees a 34.6 ms (5.8 %) increase in time when an intermediate certificate is included, again illustrating the impact of the larger certificates on low-bandwidth connections. If Falcon-1024 is not an option for the online handshake signatures, using it for CA certificates in the KDFF instantiation can mitigate the performance penalty

Table 11.19: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level V with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	65.9	97.0	35.0	66.1	97.2	35.2
KDDD	64.5	95.6	33.5	95.9	127.0	65.0
KFFF	66.4	97.5	35.4	67.1	98.2	36.2
KDFF	64.6	95.7	33.6	65.2	96.4	34.3
KSfSfSf	169.8	200.9	138.8	198.2	229.4	167.2
KSsSsSs	238.9	270.0	208.0	247.0	278.1	216.1
KSsXX	215.1	246.2	184.2	230.6	261.7	199.7
KDXX	110.9	142.0	80.0	121.1	152.2	90.2
HDDD	64.5	95.6	33.5	95.9	127.1	65.0

Table 11.20: Average handshake times in ms for unilaterally authenticated post-quantum TLS experiments at NIST level V with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr	397.1	593.1	201.3	397.7	593.7	201.8
KDDD	416.4	616.5	216.6	885.8	1081.8	689.3
KFFF	401.3	597.3	204.8	426.8	628.3	227.5
KDFF	407.2	604.8	209.9	418.7	619.5	218.5
KSfSfSf	2748.2	3589.5	2482.5	4321.3	5100.0	3707.2
KSsSsSs	2047.6	2397.9	1819.3	2707.4	3445.2	2421.1
KSsXX	1172.1	1515.5	958.9	1205.3	1584.0	980.5
KDXX	474.2	680.6	265.8	879.9	1076.6	671.7
HDDD	418.4	618.9	218.0	885.7	1081.7	689.2

compared to KDDD a bit, by improving the performance to a slowdown of 25.8 ms (4.3 %) compared to the pre-quantum instance errr.

The performance of the instances using SPHINCS$^+$-256 variants is not too much slower than SPHINCS$^+$-192 on the fast network, even when the large intermediate CA certificates are transmitted. The instance based on SPHINCS$^+$-256f (KSfSfSf) is 19.7 ms (9.4 %) slower at level V than at level III, much less than the 42 608 bytes (39.8 %) increase in signature size. However, it is still 102.3 ms (80.6 %) slower than the KDDD instantiation, and 132.1 ms (135.9 %) slower than using pre-quantum cryptography. On the low-bandwidth network, the additional bandwidth requirements very significantly slow down the TLS handshakes. Here, KSfSfSf is 1285.7 ms (33.7 %) slower at level V compared to level III. The size-optimized instance that uses SPHINCS$^+$-256s, KSsSsSs, is 1654.8 ms (32.4 %) faster than KSfSfSf, easily making up for the additional computational requirements, but still 4018.2 ms (371.4 %) slower than KDDD. Again, we see that using XMSS$_s^{MT}$-V instead of SPHINCS$^+$-256s leads to a significant increase in handshake performance: the KSsXX instance takes 1861.1 ms (54.0 %) less time before the client receives a response.

11.4.2 Mutually authenticated TLS 1.3

Communication requirements

Table 11.21 shows how we instantiate mutually authenticated post-quantum TLS 1.3 at level V and the sizes of the public keys and signatures. Now, all instantiations exceed 10 kB even when omitting intermediate CA certificates. The Kyber-1024 and Falcon-1024-based KFFF-FF instance is only marginally larger at level V than at level III, however, as we already used level-V secure Falcon-1024 in the level III instantiation.

Computational requirements

Table 11.22 shows the computational requirements of the cryptography in these instantiations at level V. We observe the same things as for the unilaterally authenticated handshakes, except the client now also needs to produce a signature while the server has to verify this signature and the signature in the client certificate.

Table 11.21: Instantiations at NIST level V of mutually authenticated post-quantum TLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	**Key exchange**	**Server authentication**	**Client authentication**	**Sum**	**Int. CA certificate**	**Sum**
Pre-quantum	X25519	hs:RSA-2048	hs:RSA-2048		pk:RSA-2048	
		sig:RSA-2048	sig:RSA-2048	1 632	sig:RSA-2048	2 160
errr-rr	64	784	784		528	
Primary	Kyber-1024	hs:Dilithium5	hs:Dilithium5		pk:Dilithium5	
		sig:Dilithium5	sig:Dilithium5	26 700	sig:Dilithium5	33 887
KDDD-DD	3136	11 782	11 782		7187	
Falcon	Kyber-1024	hs:Falcon-1024	hs:Falcon-1024		pk:Falcon-1024	
		sig:Falcon-1024	sig:Falcon-1024	11 842	sig:Falcon-1024	14 915
KFFF-FF	3136	4353	4353		3073	
Falcon offline	Kyber-1024	hs:Dilithium5	hs:Dilithium5		pk:Falcon-1024	
		sig:Falcon-1024	sig:Falcon-1024	20 070	sig:Falcon-1024	23 143
KDFF-DF	3136	8467	8467		3073	
SPHINCS$^+$-f	Kyber-1024	hs:SPHINCS$^+$-256f	hs:SPHINCS$^+$-256f		pk:SPHINCS$^+$-256f	
		sig:SPHINCS$^+$-256f	sig:SPHINCS$^+$-256f	202 688	sig:SPHINCS$^+$-256f	252 608
KSfSfSf-SfSf	3136	99 776	99 776		49 920	
SPHINCS$^+$-s	Kyber-1024	hs:SPHINCS$^+$-256s	hs:SPHINCS$^+$-256s		pk:SPHINCS$^+$-256s	
		sig:SPHINCS$^+$-256s	sig:SPHINCS$^+$-256s	122 432	sig:SPHINCS$^+$-256s	152 288
KSsSsSs-SsSs	3136	59 648	59 648		29 856	
Hash-based signatures	Kyber-1024	hs:SPHINCS$^+$-256s	hs:SPHINCS$^+$-256s		pk:XMSS$_s^{MT}$-V	
		sig:XMSS$_s^{MT}$-V	sig:XMSS$_s^{MT}$-V	68 806	sig:XMSS$_s^{MT}$-V	71 849
KSsXX-SsX	3136	32 835	32 835		3043	
HBS-CA	Kyber-1024	hs:Dilithium5	hs:Dilithium5		pk:XMSS$_s^{MT}$-V	
		sig:XMSS$_s^{MT}$-V	sig:XMSS$_s^{MT}$-V	23 468	sig:XMSS$_s^{MT}$-V	26 511
KDXX-DX	3136	10 166	10 166		3043	
HQC	HQC-256	hs:Dilithium5	hs:Dilithium5		pk:Dilithium5	
		sig:Dilithium5	sig:Dilithium5	45 278	sig:Dilithium5	52 465
HDDD-DD	21 714	11 782	11 782		7187	

hs: certificate public key and handshake signature
pk: certificate public key sig: certificate signature

Handshake performance

In tables 11.23 and 11.24, we show the handshake performance. In both tables, the results are in line with what we have seen for the unilaterally authenticated handshakes and the level III experiments.

Table 11.22: Computation time in ms for asymmetric cryptography at NIST level V for each of the mutually authenticated post-quantum TLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
errr-rr	0.660	0.661	1.321	0.676	0.661	1.337
KDDD-DD	0.987	0.955	1.942	1.148	0.955	2.103
KFFF-FF	1.206	1.174	2.380	1.378	1.174	2.552
KDFF-DF	0.998	0.966	1.964	1.170	0.966	2.136
KSfSfSf-SfSf	12.232	12.200	24.432	12.629	12.200	24.829
KSsSsSs-SsSs	105.011	104.979	209.990	105.195	104.979	210.174
KSsXX-SsX	115.862	115.830	231.692	126.897	115.830	242.727
KDXX-DX	11.861	11.829	23.690	22.896	11.829	34.725
HDDD-DD	2.342	1.539	3.881	2.503	1.539	4.042

Table 11.23: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level V with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	68.9	100.3	38.3	68.9	100.4	38.3
KDDD-DD	65.4	97.5	35.5	96.9	129.0	66.9
KFFF-FF	69.0	101.4	39.3	69.8	102.1	40.1
KDFF-DF	65.5	97.7	35.7	66.2	98.5	36.4
KSfSfSf-SfSf	230.9	333.7	271.5	237.3	340.6	278.4
KSsSsSs-SsSs	387.5	481.2	419.2	387.9	481.6	419.6
KSsXX-SsX	346.2	408.8	346.9	346.5	409.1	347.1
KDXX-DX	107.0	184.3	122.3	122.3	192.1	130.1
HDDD-DD	65.5	97.6	35.5	96.8	128.9	66.8

Table 11.24: Average handshake times in ms for mutually authenticated post-quantum TLS experiments at NIST level V with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
errr-rr	399.9	597.2	205.2	401.0	598.4	206.1
KDDD-DD	426.0	641.7	233.3	801.1	1042.0	601.9
KFFF-FF	404.0	605.1	212.6	409.9	612.1	216.9
KDFF-DF	410.2	617.9	220.5	417.9	626.8	223.7
KSfSfSf-SfSf	3117.7	4918.7	3790.7	4577.6	7095.0	5936.6
KSsSsSs-SsSs	1504.9	2422.4	1834.0	2255.4	3630.6	2887.9
KSsXX-SsX	1198.6	1809.2	1282.4	1312.4	1973.6	1442.1
KDXX-DX	483.9	753.1	328.9	628.4	895.1	478.5
HDDD-DD	427.2	642.9	234.2	785.5	1032.3	590.9

11.5 Discussion

Summarizing the results for our experiments across NIST security level I, III, and V, we see that Kyber and Dilithium offer reasonable performance. The size of Dilithium does affect the handshake times significantly at levels III and V, due to these instances exceeding the initial congestion window size. Falcon offers comparable or better performance, as the AVX2-accelerated implementation is highly performant and its public keys and signatures are the smallest. If using Falcon for online signatures is not an option due to its requirement of constant-time 64-bit floating-point arithmetic for signing, using it with Dilithium for online signatures as in our KDFF instances is very attractive, although this does increase the size of the codebase. Using SPHINCS$^+$ does not seem very attractive for TLS 1.3: both in computation time and size of signatures, the scheme very significantly affects the handshake performance. If any application greatly prefers using a hash-based signature scheme, it seems using alternatives to SPHINCS$^+$ for CA certificates can offer very large performance improvements, especially at higher security levels.

Our conclusions are also reflected by figure 11.1: all instantiations are mostly gathered together around the 3-RTT line. Only the instances that use large certificates or that use slower algorithms such as XMSS$_s^{MT}$ move away from this line. In the bottom plot, we compare all instantiations using SPHINCS$^+$: they are so large and comparatively slow that the top graph fits in the lower left corner of the bottom graph.

Our experiment is not the first to look at the performance of post-quantum TLS. Comparing our results with [281, 325, 326, 333, 351], we see that we arrive at similar results. For high-bandwidth connections, the increase in computation time is very moderate for any scheme that stays under the `initcwnd` limit on the number of MSS that can be sent before receiving a TCP acknowledgment from the recipient. As Sikeridis, Kampanakis, and Devetsikiotis first observed in [326], we see that combining two different algorithms for *online* handshake authentication and *offline* CA certificates can greatly influence connection establishment times.

More signatures on the web

As Westerbaan [351] highlighted, there are many more signatures in a typical TLS handshake on the web than we included in our experiments. The Google Chrome and Safari web browsers require at least two CT [235] proofs to be

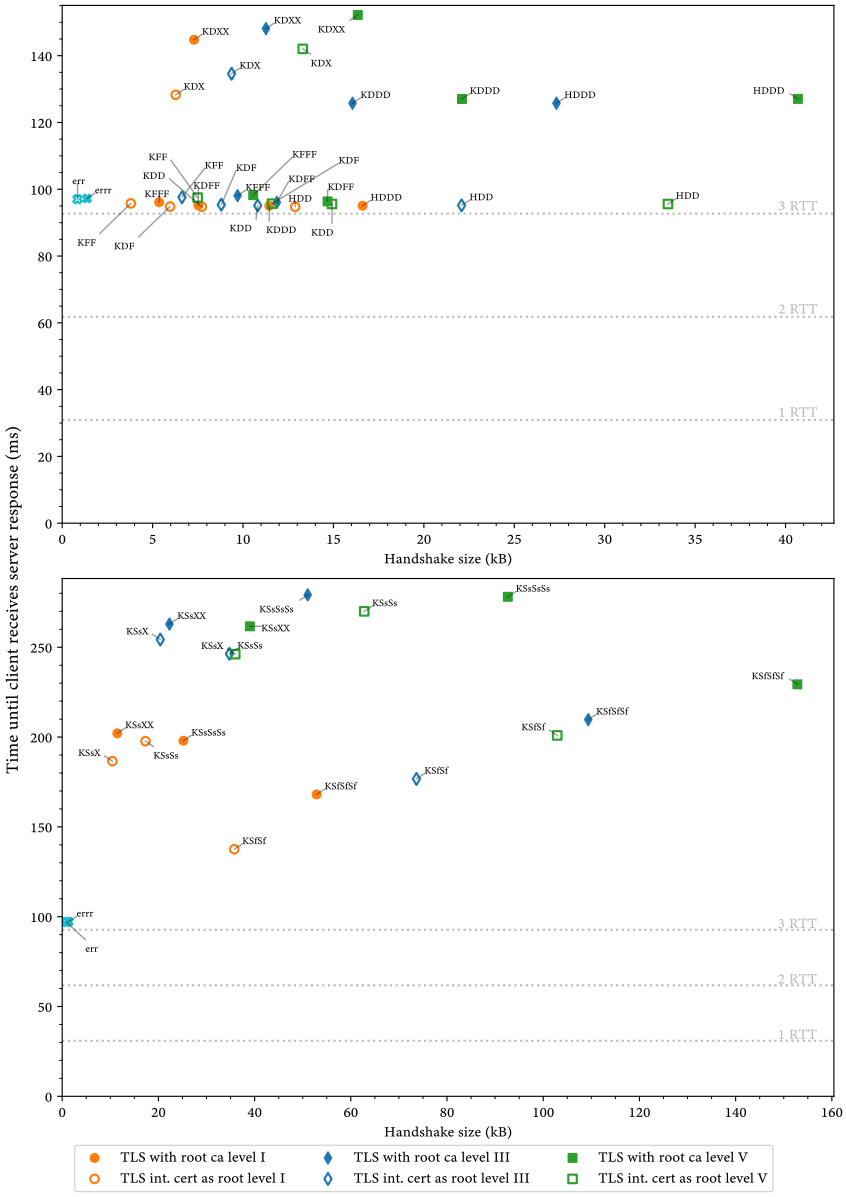


Figure 11.1: Handshake timings of post-quantum TLS experiments

included for any certificate [15, 103]. This means that in every certificate, there are two additional signatures from CT logs. The online certificate status protocol (OCSP) [311], which allows a server to show that its certificate is not revoked, also consists of a signed statement from the certificate authority. CT and OCSP thus add another three offline signatures to the typical TLS handshake traffic. If all the signature algorithms in the TLS handshake are instantiated with Dilithium2, this adds up to 17 144 bytes of public keys and signatures that are sent in the `ServerCertificate` message. This easily exceeds a 10 MSS initial congestion window size. This means using different algorithms for online and offline signatures and/or standardizing mechanisms that allow removing or suppressing some signatures [202] may be vital to the performance of TLS on the web.

Increasing the congestion window

The initial congestion window size default of 10 MSS is a fairly recent development, first suggested by Google [132]. This suggestion was implemented by Linux in 2011, before being standardized by the IETF in RFC 6928 in 2013 [104]. Before, the recommendation was a congestion window "between 2 and 4 segments". It has been argued that the congestion window can be increased to allow for the larger sizes of post-quantum cryptographic algorithms, e.g. in [40]. Indeed, many content delivery networks already use much larger congestion windows [306]. Increasing the initial congestion window across the internet may be a valid strategy for handling the increase in TLS handshake traffic. However, too large values could result in congestion and packet loss. RFC 6928 [104, Appendix A] additionally highlights that raising the congestion window may have adverse effects on internet connection speeds in the developing world. Evaluating what would be the best value for `initcwnd` and the impact of changing the value on the congestion control behavior of TCP across the internet is beyond the scope of this thesis, however, and we leave this for other work.

Other considerations

Finally, we have not explored the computational load of our instantiations. The algorithms have very different computational characteristics. If an algorithm has very high computational requirements, this may result in a limitation on the number of connections that a server may be able to support. Sikeridis, Kampanakis, and Devetsikiotis examined this for unilaterally authenticated

TLS 1.3 using round-2 schemes in [326] and showed that a server running Dilithium may be able to support more connections than a server running Falcon. In our experiments, the server was also exactly as computationally powerful as the client: after all, they ran on the same machine. Often, however, this is not the case: clients may, for example, have battery life concerns, such as smartphones or laptops. Further examining these issues in post-quantum TLS remains for future work. Another example of asymmetry between clients and servers is the case of the performance of post-quantum TLS on microcontrollers, which are much less powerful and may have low-bandwidth links. We will discuss this issue in chapter 16.

11.6 Appendix: XMSS at different NIST security levels

The security of XMSS parameter sets specified in RFC 8391 [181] reach NIST security level V (equivalent to AES-256) and above. This high level of security has only a very minor impact on computational performance, but it does have a significant impact on signature size. The NIST standard [106] also considers parameter sets targeting security level III (equivalent to AES-192); the simple modification is to truncate all hashes to 192 bits. This is possible because the security of XMSS and its multi-tree variant XMSS^{MT} is guaranteed by a tight reduction from second-preimage resistance [46, 188]; collisions in the underlying hash function do not affect the security of XMSS. We straightforwardly extend XMSS to parameter sets targeting NIST security levels I and III. For level I, hashes are simply truncated to 128 bits; we obtain this by using SHAKE-128 [268] with 128 bits of output. For level III, we use SHAKE-256 truncated to 192 bits of output. SPHINCS$^+$ similarly constructs its level-I and level-III parameter sets. Aside from minor details, XMSS can be seen as a "sub-step" of SPHINCS$^+$; see [183]. To complete our set of parameters, we also specify a variant at NIST security level V, which obtains 256 bits from SHAKE-256.

We define XMSS$^{MT}_s$ as an instantiation of XMSS^{MT} using two trees of height 12 each, i.e., a total tree height of 24, which limits the maximum number of signatures per public key to $2^{24} \approx 16.7$ million. Increasing this maximum number of signatures to, for example, $2^{30} \approx 1$ billion increases signature size by only 96 bytes and has negligible impact on verification speed. It does have an impact on key-generation speed and signing latency, but as mentioned

in section 13.6.3, the latency of signing is not very relevant when used by certificate authorities as in our benchmarks.

Multi-tree XMSS is particularly well-suited for efficient batch signing. The idea is to compute one whole tree (of height h/d) on the lowest level and use it on-the-fly to sign $2^{h/d}$ messages. The computational effort per signature is then essentially reduced to one WOTS$^+$ key-pair generation.

We set the Winternitz parameter in XMSS$^{\text{MT}}_{\text{s}}$ to $w = 256$ to optimize for signature size. Changing to the more common $w = 16$ would increase the signature size by about a factor of 2 and speed up verification by about a factor of 8.

12 Performance of post-quantum OPTLS

In this chapter, we investigate the performance of OPTLS when instantiated with CSIDH [91]. CSIDH is the only post-quantum NIKE that currently offers IND-CCA security and has readily available implementations; the Swoosh [153] scheme announced in early 2023 only proposes parameters for an actively-secure version. We will show the bandwidth characteristics and performance of OPTLS when we use the aggressive 512- and 1024-bit versions of CSIDH. The security of CSIDH is hotly debated; we will also discuss the expected performance using CSIDH at higher security levels.

12.1 Instantiating OPTLS

When instantiating OPTLS, we need to select a NIKE that is used for ephemeral key exchange as well as for server authentication. Because the authentication key exchange in OPTLS combines the client's ephemeral key share with the server's long-term NIKE public key, these need to be the same algorithm. We additionally require a signature scheme that a CA uses to authenticate the server's certificate; in our experiments we use Falcon-512. To simplify our experiments, we leave out intermediate certificates in this chapter.

No NIKE are currently under consideration in the NIST PQC standardization project. As discussed, the only NIKE that currently has passively-secure parameters is CSIDH. To provide the best possible scenario for OPTLS, we use the CTIDH instantiations of CSIDH by Banegas, Bernstein, Campos, Chou, Lange, Meyer, Smith, and Sotáková [19]. This is the most efficient publicly available implementation of CSIDH as of writing. The security analysis of the original CSIDH parameter sets (512 and 1024 bits) shows they are very aggressively chosen; we will return to this subject and the effect of more conservative parameters on the performance of OPTLS in the discussion in section 12.3.

Table 12.1: Communication sizes and computation requirements for the client and the server for OPTLS instantiated with specific NIKEs.

	Transmission size (bytes)			Computation time (ms)	
NIKE	**KEX**	**HS. auth**	**Certificate signature** (Falcon-512)	**Ephem. keygen**	**Key agreements**
CTIDH512	128	64	666	47.7	95.3
CTIDH1024	256	128	666	182.3	366.5

In table 12.1, we show the communication and computational requirements for OPTLS when instantiated with CTIDH512 and CTIDH1024. In the OPTLS key exchange, both parties transmit an ephemeral public key. For server authentication, only the server transmits its long-term public key: this key is combined with the already-transmitted client key exchange public key. Finally, for completeness, we show the size of the Falcon-512 signature that we use for certificate authentication. The size of these OPTLS handshakes is very small compared to the smallest post-quantum TLS handshakes, but CSIDH has significant computational requirements. The times shown in table 12.1 are the time required for key generation in ephemeral key exchanges, if this is done during the handshake, and the time required to compute the two shared secrets for key exchange and authentication. Both the client and the server need to perform these computations during the handshake, but the cost of ephemeral key generation may be amortized if ephemeral key shares are reused (see also the discussion on forward secrecy in section 2.5.1). In our handshake measurements, we include results in which the ephemeral keys have been generated before the measured handshakes. However, the time required by each peer for key agreement for CTIDH512 already greatly exceeds the 30.9 ms latency in our low-latency network experiments, and for CTIDH1024 it is even much more than the 195.5 ms latency in the high-latency network experiments. Note that, unlike ephemeral key generation, both the client and the server need to perform these computations every handshake, and they cannot be pre-computed, as their inputs are unique to every handshake.

12.2 OPTLS handshake performance

In table 12.2 we show the handshake performance of OPTLS when instantiated with CTIDH512 and CTIDH1024 when running over two different network environments. As discussed in section 12.1, show the handshake performance when ephemeral keys are generated during each handshake, and the performance when ephemeral keys are cached or generated out-of-band. In the latter case, no key generations take place during the measured handshakes. In both scenarios, a large effect of the CSIDH computational requirements is seen. Even when the cost of ephemeral key generation is excluded from the handshake performance, the client's response is received more than 200 ms later than in the post-quantum TLS experiments at NIST PQC security level I, discussed in section 11.2. When using CTIDH1024, the overhead of the OPTLS handshake takes nearly a full second on the low-latency network, even if ephemeral keys are cached. Generating fresh ephemeral keys in each handshake increases the delay by another 429.0 ms for CTIDH1024. On the high-latency network, the results are very similar. The significant computational requirements of CSIDH result in handshakes that take about two times as long as comparable post-quantum TLS 1.3 handshakes for CSIDH512; CTIDH1024 is almost twice as slow.

Table 12.2: Handshake latencies in ms for OPTLS instantiated with specific NIKE when run over two different network environments. Latencies are given with and without ephemeral key share reuse.

		Handshake latencies (RTT, bandwidth)					
		30.9 ms, 1000 Mbps			**195.5 ms, 10 Mbps**		
NIKE	**Cached ephem. keys**	**Client sent req.**	**Client recv. resp.**	**Server HS done**	**Client sent req.**	**Client recv. resp.**	**Server HS done**
CTIDH512	✘	372.7	403.8	315.1	644.8	840.8	448.9
	✓	284.5	315.6	253.6	604.6	800.6	408.6
CTIDH1024	✘	1346.4	1377.5	1127.1	1402.2	1598.1	1188.4
	✓	917.4	948.4	886.5	1157.7	1353.6	961.7

12.3 Discussion

Based on our results, post-quantum OPTLS does not offer competitive performance compared to post-quantum TLS 1.3, at least when instantiated with CTIDH512 or CTIDH1024. The large amount of computation severely affects handshake times; additionally, the time spent on cryptographic computations cannot be used for any applications running on top of TLS. Only in extremely restricted-bandwidth scenarios might the small size of the handshake offer a benefit over using signed TLS, or as we will see in chapters 13 and 14, KEMTLS(-PDK).

We have examined the performance of two small instances of CSIDH, but the security of CSIDH is the subject of ongoing, significant debate [42, 49, 58, 71, 289]. Specifically, CSIDH is vulnerable to a quantum algorithm by Kuperberg [225], but different authors come to different conclusions as to the cost of implementing the attack. Depending on analysis, the quantum security of CSIDH512 ranges from 29 to 139 bits [91]; in the evaluation of Chávez-Saab, Chi-Domínguez, Jaques, and Rodríguez-Henríquez, CSIDH512 offers just 63 bits of quantum security and CSIDH1024 just 72 bits [97]. In our results, we see that even CSIDH512 is likely too slow for most deployments of TLS. This leaves the question: what if we require higher-security instantiations of CSIDH?

Higher-security CSIDH

Chávez-Saab, Chi-Domínguez, Jaques, and Rodríguez-Henríquez presented the first CSIDH implementation at higher security levels going all the way from primes of size 2000 bits up to 9000 bits [97]. In [88], Campos, Chavez-Saab, Chi-Domínguez, Meyer, Reijnders, Schwabe, and I discuss new parameters for high-security CSIDH and CTIDH, specifically chosen to allow for highly-optimized implementations that outperform prior results. Following the conclusions from the quantum circuit estimations in [97], we use primes of 2048 and 4096 bits targeting NIST PQC security level I, 5120 and 6144 bits targeting level II, and 8192 and 9216 bits targeting level III. Each pair of instantiations represents a choice between more "aggressive" assumptions (with attacker circuit depth bounded by 2^{60}) or more "conservative" assumptions (attacker circuit depth bounded by 2^{80}). As stressed in [97], this choice of parameters does not take into account the cost of calls to the CSIDH evaluation oracle and is likely to underestimate security.

Table 12.3: Public key cryptography transmission sizes in bytes and time in seconds until client receives and sends Finished messages for OPTLS, TLS 1.3, and KEMTLS.

				Handshake latencies (RTT, bandwidth)			
		Transmission		**30.9 ms, 1000 Mbps**		**195.5 ms, 10 Mbps**	
		KEX	**Auth**	**SFIN recv**	**CFIN sent**	**SFIN recv**	**CFIN sent**
OPTLS (pregen)	CSIDH p2048	544	938	24.468	24.468	24.288	24.288
	CTIDH p2048	512	922	7.346	7.346	7.203	7.203
	CTIDH p4096	1024	1178	36.321	36.321	36.299	36.299
	CTIDH p5120	1280	1306	28.701	28.701	28.580	28.580
OPTLS (ephemeral)	CSIDH p2048	544	938	43.642	43.642	43.486	43.486
	CTIDH p2048	512	922	10.042	10.042	9.882	9.882
	CTIDH p4096	1024	1178	50.039	50.039	49.951	49.951
	CTIDH p5120	1280	1306	42.383	42.383	42.163	42.163
TLS	Kyber-512–Falcon-512	1568	2229	0.064	0.064	0.428	0.428
	Kyber-512–Dilithium2	1568	4398	0.063	0.063	0.519	0.519
	Kyber-768–Falcon-1024	2272	3739	0.065	0.065	0.497	0.497
KEMTLS	Kyber-512	1568	2234	0.094	0.063	0.593	0.396
	Kyber-768	2272	2938	0.094	0.063	0.597	0.400

All instantiations use Falcon-512 for the certificate authority; the CA public key is not transmitted. Bytes necessary for authentication includes bytes for the Falcon-512 CA signature on the server's certificate. These benchmark results have been obtained independently from the other results in chapters 11 to 14.

Extrapolating the slowdown when using CTIDH1024 instead of CTIDH512 seen in table 12.2 suggests that especially the proposed level II and level III parameter sets will be extremely impractical for use in OPTLS. In table 12.3 we report advance results from [88]. When using CTIDH above 5000 bits, handshake times need to be measured in the tens of seconds. This means that unless the performance of CSIDH can be improved by several orders of magnitude, CSIDH, especially at these higher security levels, does not seem practical for use in interactive protocols such as TLS.

Swoosh

Just around the time of writing, a new proposal for a post-quantum NIKE was put forward, called Swoosh [152]. This NIKE is based on the module-learning with errors (MLWE) problem, and the authors claim a post-quantum security level of more than 120 bits. However, the Swoosh public key for the passively secure version is approximately 220 kB; additional zero-knowledge proofs

need to be included for an actively-secure version. This public key size not only exceeds the 64 kB limit on key share entries, but also exceeds the TCP Slow Start initial congestion window (`initcwnd`), and thus would require multiple round-trips to transfer the public key. As a result, right now, Swoosh is not the missing NIKE scheme that makes OPTLS a viable alternative for a TLS handshake with signatures.

12.4 Conclusions

It appears that OPTLS cannot be practically used in a post-quantum setting, as CSIDH is currently too slow. Until new NIKEs are proposed with better runtime performance than CSIDH but smaller sizes than Swoosh this seems unlikely to change, as our experimental results are based on highly-optimized implementations of CSIDH512 and CSIDH1024. If higher security levels than CSIDH1024 are required, the handshake time of OPTLS when instantiated with, e.g., CSIDH4096 is around 50 seconds, based on the performance of highly-optimized implementations of CSIDH and CTIDH at higher security levels [88]. This suggests that CSIDH is not suitable for interactive network protocols, like TLS.

13 Performance of KEMTLS

In this chapter, we investigate the performance of KEMTLS. We first instantiate the protocol with post-quantum cryptographic primitives at NIST PQC security levels I, III, and V. We then compare how they perform in both unilaterally authenticated handshakes and mutually authenticated handshakes. We will also compare KEMTLS against the results for post-quantum TLS 1.3 from chapter 11.

For a description of how we implemented KEMTLS, please refer back to section 10.4. The design of the emulated network environment and our choice of parameters are motivated in section 10.10.

As in previous chapters, we are only comparing the sizes and performance characteristics of the schemes used to instantiate KEMTLS in this chapter. It can be argued that this compares apples to oranges: different security assumptions both affect the size and performance, but also have different levels of confidence associated with them. However, these comparisons are still useful to estimate the cost of choosing different assumptions based on such considerations.

13.1 Selecting algorithms for KEMTLS experiments

As we did for the experiments in chapter 11, we base our main selection of algorithms on the NIST PQC standardization project, although we also choose some additional algorithms such as the customized XMSS parameters at lower security levels than NIST level V, which we described in section 11.6.

In each instantiation, we select:

1. an algorithm for ephemeral key exchange, negotiated by the KEMTLS client and server;

2. an algorithm for handshake authentication, used in the server certificate;

3. an algorithm for authentication of the server's certificate by the (intermediate) CA certificate, which we may assume the client to already have;

4. an algorithm for authentication of the intermediate CA certificate by a root CA certificate, which is always assumed to be preinstalled.

For KEMTLS handshakes that use mutual authentication, we additionally select:

5. an algorithm for client authentication, used in the client certificate;

6. an algorithm for authentication of the client's certificate by a CA certificate, which is assumed to be preinstalled.

In our experiments, we will try to showcase how the different algorithms in the NIST PQC standardization project perform. We base our choices on those made for the post-quantum TLS 1.3 experiments in section 11.1, so that we may compare the performance of TLS 1.3 and KEMTLS. We will use the following scenarios:

Primary This instantiation uses Kyber [319], the only KEM which was selected for standardization for post-quantum key exchange. We use Kyber for both the ephemeral key exchange and handshake authentication. Dilithium [241], the algorithm which, when it was selected for standardization, was named the *primary* algorithm for post-quantum signatures, is used for the signatures in certificates.

Falcon This instantiation uses Kyber for ephemeral key exchange and handshake authentication, but uses Falcon [293], the other digital signature algorithm that NIST selected for standardization, for the signatures in the certificates. Note that, as there is no signing in the KEMTLS handshake, Falcon's implementation concerns do not affect KEMTLS handshakes.

SPHINCS$^+$-f This instance uses Kyber for ephemeral key exchange and handshake authentication. It uses SPHINCS$^+$ [184] for the CA signatures in certificates. SPHINCS$^+$ is the only algorithm that was selected for standardization by NIST in 2022 that is not based on lattice assumptions. Specifically, this instantiation uses the *fast* variant of SPHINCS$^+$, which has faster runtime but larger signatures. For the hash function, we use Haraka, as explained in section 10.6.

SPHINCS$^+$-s This instantiation is like the SPHINCS$^+$-f-variant, but it uses the *small* variant of SPHINCS$^+$, which has slow signing time but smaller signatures.

Hash-based CA This conservative instantiation uses Kyber for ephemeral key exchange and handshake authentication but uses a customized instantiation of XMSS^{MT} for the CA signatures in certificates. This gives us conservative hash-based signatures for the long-lived certificates.

HQC This instantiation uses HQC, a round-4 KEM candidate in the NIST PQC standardization project, for ephemeral key exchange and handshake authentication. HQC relies on assumptions based on decoding of quasi-cyclic codes, instead of on assumptions on lattices.

Note that for presentation purposes, we will refer to these scenarios in our tables and figures by handles, which are composed of the first letters of each of the selected algorithms. For an overview of these handles, refer to the tables that show the communication sizes, e.g. table 13.1.

As in the TLS 1.3 experiments, the remaining candidates for post-quantum key exchange in round 4 of the NIST PQC standardization project, are unfortunately not suitable for our experiments. BIKE [17] does not have IND-CCA secure parameter sets available, and Classic McEliece [7] has a too-large public key for use in TLS 1.3.

As before, we measure the performance of our instantiations when using the intermediate certificate in place of a root CA certificate, thus excluding it from the `ServerCertificate` message; and the performance when the server does transmit the intermediate certificate. This represents scenarios in which no intermediate certificates are used, as well as accounts for proposals that call for omitting or caching intermediate CA certificates, like [202].

13.2 Instantiation and results at NIST level I

We measured and compare the performance of TLS 1.3 at NIST security level I, which offers security comparable to that given by AES-128. As it is the lowest security level, the parameter sets are the most aggressively chosen and generally offer the smallest public key, ciphertext, and signature sizes and the shortest computation times. We will first discuss unilaterally authenticated

handshakes, in which only the server is authenticated by a certificate and a signature. This scenario is the most important to web browsing, as this is the handshake mode that is almost exclusively used by web browsers [61]. Afterward, we will discuss mutually authenticated handshakes, in which the client also presents a certificate of their identity and signs the handshake. This is for example used to secure service-to-service communication or in VPNs.

13.2.1 Unilaterally authenticated KEMTLS

Table 13.1: Instantiations at NIST level I of unilaterally authenticated KEMTLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**		**Int. CA public key**	**Root CA signature**		**Root CA public key**
handle	pk+ct	pk+ct	sig	**Sum**	pk	sig	**Sum**	pk
Primary	Kyber-512	Kyber-512	Dilithium2	5 556	Dilithium2	Dilithium2	9 288	Dilithium2
KKDD	1568	1568	2420		1312	2420		1312
Falcon	Kyber-512	Kyber-512	Falcon-512	3 802	Falcon-512	Falcon-512	5 365	Falcon-512
KKFF	1568	1568	666		897	666		897
SPHINCS$^+$-f	Kyber-512	Kyber-512	SPHINCS$^+$-128f	20 224	SPHINCS$^+$-128f	SPHINCS$^+$-128f	37 344	SPHINCS$^+$-128f
KKSfSf	1568	1568	17 088		32	17 088		32
SPHINCS$^+$-s	Kyber-512	Kyber-512	SPHINCS$^+$-128s	10 992	SPHINCS$^+$-128s	SPHINCS$^+$-128s	18 880	SPHINCS$^+$-128s
KKSsSs	1568	1568	7856		32	7856		32
Hash-based CA	Kyber-512	Kyber-512	XMSS$_s^{MT}$-I	4 115	XMSS$_s^{MT}$-I	XMSS$_s^{MT}$-I	5 126	XMSS$_s^{MT}$-I
KKXX	1568	1568	979		32	979		32
HQC	HQC-128	HQC-128	Dilithium2	15 880	Dilithium2	Dilithium2	19 612	Dilithium2
HHDD	6730	6730	2420		1312	2420		1312

Communication requirements

In table 13.1, we show how we instantiate the KEMTLS experiments at security level I. Specifically, we show the sizes of the public-key cryptography items: the public keys, ciphertexts, and signatures that are used in the handshake for ephemeral key exchange and authentication. We include the total amount of data required for public-key cryptography items in two scenarios: one where the intermediate CA certificate is used in place of a root certificate (and

thus not transferred), and the scenario in which a full certificate chain is used and the intermediate CA certificate is thus included in the transmission. We also give the abbreviated handles, by which we will refer to our instantiations. These handles are generally composed of the first letters of each scheme, in the order of key exchange, handshake authentication, intermediate CA certificate, and root CA certificate.

Comparing the sizes of our instantiations, we observe similar things as we saw for post-quantum TLS 1.3 in section 11.2. Again, the instance that uses Falcon-512, KKFF, is significantly smaller than the KKDD instance that uses Dilithium2 for certificate signatures. The KKXX instance that uses $\text{XMSS}_{\text{s}}^{\text{MT}}$-I instead of Falcon-512 for handshake signatures is only slightly larger than KFFF when using the intermediate CA certificate in place of the root certificate, and slightly smaller than KFFF when the intermediate CA certificate is included. Using HQC-128, the NIST PQC round-4 KEM candidate which is not based on lattice assumptions, leads to a large increase in handshake size. Only the instance that uses SPHINCS$^+$-128f is larger.

Table 13.2: Computation time in ms for asymmetric cryptography at NIST level I for each of the unilaterally authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD	0.128	0.044	0.172	0.192	0.044	0.236
KKFF	0.205	0.044	0.249	0.346	0.044	0.390
KKSfSf	0.507	0.044	0.551	0.950	0.044	0.994
KKSsSs	0.154	0.044	0.198	0.244	0.044	0.288
KKXX	8.296	0.044	8.340	16.528	0.044	16.572
HHDD	0.536	0.406	0.942	0.600	0.406	1.006

Computational requirements

In table 13.2, we see the computational requirements for the KEMTLS experiments at security level I. As we use Kyber-512 for authentication in most instantiations, the server only needs to perform one encapsulation, in the ephemeral Kyber-512 key exchange, and one decapsulation on a ciphertext

for the handshake authentication. The client additionally needs to perform verification of signatures on the certificates, which differ between the instantiations. Except for the KKXX instantiation, which uses the slow-to-verify $\text{XMSS}_{\text{s}}^{\text{MT}}$-I scheme, all computation times are very small.

Handshake performance

In tables 13.3 and 13.4, we show the handshake times for KEMTLS when ran over a 30.9 ms latency, 1000 Mbps network and a 195.5 ms latency, 10 Mbps network. We show the time it takes from the start of the client's connection setup to the moment the client has sent its request (immediately after it sent the `ClientFinished` message), the time after which the client receives a response to this request, and the time it takes for the server, measured from the moment it receives the `ClientFinished` message until the server has completed the handshake and is ready to process the client's request.

Similarly to what we saw for the post-quantum TLS 1.3 experiments at level I in section 11.2, most handshakes, except KKSfSf, and KKSsSs if an intermediate certificate is transmitted, stay well under the 10 MSS initial congestion window limit. As the computation times for all algorithms but $\text{XMSS}_{\text{s}}^{\text{MT}}$-I are very small, we see that, on the high-bandwidth network, the handshake times are dominated by the number of round-trips necessary, Only in KKXX do we see that the handshake suffers a delay for signature verification of the certificates, especially if the client needs to verify both the server's certificate and the intermediate CA certificate.

On the 195.5 ms latency, 10 Mbps network, the large size of some experiments adds notable delay. Especially the largest SPHINCS^{+}-128f instance has very long handshake times: it takes 609.5 ms (95.4 %) longer before receiving the server's response than the KKDD instance if an intermediate certificate is included. Using $\text{XMSS}_{\text{s}}^{\text{MT}}$-I instead of either SPHINCS^{+} variant brings the handshake performance more in line with the performance of the KKDD instance when intermediate certificates are used. This may be preferable if hash-based signatures are preferred over Falcon-512 or Dilithium2.

Comparison with post-quantum TLS 1.3

In table 13.5, we compare similar post-quantum TLS 1.3 and KEMTLS instances. We show the size of the handshake when either excluding or including an intermediate CA certificate from the transmission, and we show the time until the client receives the first response from the server in both scenarios. We

Table 13.3: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level I with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	63.0	94.4	32.3	63.4	94.8	32.6
KKFF	63.1	94.5	32.4	63.5	94.9	32.8
KKSfSf	95.2	126.6	64.5	96.9	128.3	66.2
KKSsSs	63.5	94.9	32.8	95.0	126.4	64.3
KKXX	90.1	121.5	59.3	111.3	142.7	80.6
HHDD	63.5	95.6	33.5	63.8	95.9	33.8

Table 13.4: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level I with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	398.0	595.0	202.3	435.5	638.8	236.2
KKFF	395.6	592.5	199.9	397.3	594.3	201.7
KKSfSf	884.7	1081.7	689.1	937.6	1248.3	652.3
KKSsSs	455.8	667.1	252.8	883.6	1080.5	687.9
KKXX	429.5	628.5	229.2	475.1	677.3	262.8
HHDD	398.6	599.3	206.7	410.2	613.8	215.7

Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	**Δ%**	**With int.**	**Δ%**	**No int.**	**Δ%**	**With int.**	**Δ%**
TLS	KDDD	7720	−28.0 %	11 452	−18.9 %	94.8	−0.4 %	95.0	−0.3 %
KEMTLS	KKDD	5556		9288		94.4		94.8	
TLS	KFFF	3797	+0.1 %	5360	+0.1 %	95.8	−1.3 %	96.1	−1.2 %
KEMTLS	KKFF	3802		5365		94.5		94.9	
TLS	KDFF	5966	−36.3 %	7529	−28.7 %	94.8	−0.3 %	95.2	−0.3 %
KEMTLS	KKFF	3802		5365		94.5		94.9	
TLS	KSfSfSf	35 776	−43.5 %	52 896	−29.4 %	137.6	−8.0 %	168.1	−23.6 %
KEMTLS	KKSfSf	20 224		37 344		126.6		128.3	
TLS	KSsSsSs	17 312	−36.5 %	25 200	−25.1 %	197.7	−52.0 %	198.0	−36.2 %
KEMTLS	KKSsSs	10 992		18 880		94.9		126.4	
TLS	KSsXX	10 435	−60.6 %	11 446	−55.2 %	186.6	−34.9 %	202.1	−29.4 %
KEMTLS	KKXX	4115		5126		121.5		142.7	
TLS	HDDD	12 882	+23.3 %	16 614	+18.0 %	94.7	+0.9 %	95.1	+0.9 %
KEMTLS	HHDD	15 880		19 612		95.6		95.9	

also show the relative differences between the TLS 1.3 and KEMTLS instances. Replacing the handshake signature by a KEM key exchange for authentication results, for most of the instantiations, in a significant reduction in handshake size. This is because a Kyber-512 public key and ciphertext are smaller than a public key and signature for any of the candidate signature schemes but Falcon-512: indeed, only the KFFF TLS 1.3 instantiation is slightly smaller than its KKFF KEMTLS equivalent. However, the HQC-128 KEM is much larger, and as such the HQC-based KEMTLS instance is much larger than the similar TLS 1.3 instantiation. But otherwise, we see that even for the KDFF TLS 1.3 instantiation, which tries to optimize the handshake size while avoiding using the implementation concerns associated with Falcon-512, the equivalent KKFF KEMTLS handshake (which also only needs Falcon signature verification) is more efficient in both size and handshake latency.

We also see that Kyber-512 is computationally more efficient than all signature schemes: this results in a small reduction in time before the client receives a response from the server for the instances that fit in the initial congestion window. In the comparison between KSsSsSs and KKSsSs, we see that using KEMTLS reduces the size of the experiment (when the intermediate CA certificate is omitted from transmission) such that it fits in the initial congestion window, and thus we observe a large reduction in handshake latency.

13.2.2 Mutually authenticated KEMTLS

Table 13.6: Instantiations at NIST level I of mutually authenticated KEMTLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	**Key exchange**	**Server authentication**	**Client authentication**	**Sum**	**Int. CA certificate**	**Sum**
Primary	Kyber-512	hs:Kyber-512	hs:Kyber-512		pk:Dilithium2	
		sig:Dilithium2	sig:Dilithium2	9 544	sig:Dilithium2	13 276
KKDD-KD	1568	3988	3988		3732	
Falcon	Kyber-512	hs:Kyber-512	hs:Kyber-512		pk:Falcon-512	
		sig:Falcon-512	sig:Falcon-512	6 036	sig:Falcon-512	7 599
KKFF-KF	1568	2234	2234		1563	
SPHINCS$^+$-f	Kyber-512	hs:Kyber-512	hs:Kyber-512		pk:SPHINCS$^+$-128f	
		sig:SPHINCS$^+$-128f	sig:SPHINCS$^+$-128f	38 880	sig:SPHINCS$^+$-128f	56 000
KKSfSf-KSf	1568	18 656	18 656		17 120	
SPHINCS$^+$-s	Kyber-512	hs:Kyber-512	hs:Kyber-512		pk:SPHINCS$^+$-128s	
		sig:SPHINCS$^+$-128s	sig:SPHINCS$^+$-128s	20 416	sig:SPHINCS$^+$-128s	28 304
KKSsSs-KSs	1568	9424	9424		7888	
Hash-based CA	Kyber-512	hs:Kyber-512	hs:Kyber-512		pk:XMSS$_s^{MT}$-I	
		sig:XMSS$_s^{MT}$-I	sig:XMSS$_s^{MT}$-I	6 662	sig:XMSS$_s^{MT}$-I	7 673
KKXX-KX	1568	2547	2547		1011	
HQC	HQC-128	hs:HQC-128	hs:HQC-128		pk:Dilithium2	
		sig:Dilithium2	sig:Dilithium2	25 030	sig:Dilithium2	28 762
HHDD-HD	6730	9150	9150		3732	

hs: certificate public key and authentication ciphertext
pk: certificate public key sig: certificate signature

Communication requirements

Table 13.6 shows the sizes for the public-key cryptography elements required to instantiate mutually authenticated KEMTLS with the selected algorithms. Again, we compare scenarios in which the intermediate CA certificate is used as a root, for example, because the client has it cached or because no intermediates are used, and scenarios in which an intermediate certificate is transmitted as part of the handshake. For a better presentation, we only show the sums of the size of the public key and the ciphertext for KEMs, and sums of the size of the public key and the signature in each certificate, even if they use different algorithms. For the sizes of the individual public keys, ciphertexts, and signatures, refer to section 10.6.

The difference in size between the unilaterally authenticated handshakes and the mutually authenticated handshakes is exactly the size listed in the client authentication column. This pushes up the communication requirements, but as we will discuss later in the comparison with TLS 1.3, the use of Kyber-512 for client authentication keeps the increase modest.

Table 13.7: Computation time in ms for asymmetric cryptography at NIST level I for each of the mutually authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD-KD	0.146	0.134	0.280	0.210	0.134	0.344
KKFF-KF	0.223	0.211	0.434	0.364	0.211	0.575
KKSfSf-KSf	0.525	0.513	1.038	0.968	0.513	1.481
KKSsSs-KSs	0.172	0.160	0.332	0.262	0.160	0.422
KKXX-KX	8.314	8.302	16.616	16.546	8.302	24.848
HHDD-HD	0.803	0.609	1.412	0.867	0.609	1.476

Computational requirements

For mutual authentication, the server needs to additionally assert the validity of the client's certificate by verifying its signature. It also needs to encapsulate a shared secret to the KEM public key in the certificate. The client only needs to perform an additional decapsulation. This makes it so that the compu-

tational requirements for the client and server are now very similar: only the ephemeral key exchange requires slightly additional work from the client (keygen and decapsulation operations versus the server's encapsulation). If an intermediate certificate is used, the client also needs to verify one additional signature. This barely increases the total computation time, except when $\text{XMSS}_{\text{s}}^{\text{MT}}$-I is used in KKXX-KX.

Handshake performance

We show the time from the start of the client's connection until the client can send a request to the server, the time until the client receives the response, and the server's time from receiving `ClientHello` to completing the handshake in tables 13.8 and 13.9 for both the low-latency, high-bandwidth and the high-latency, low-bandwidth network environments. When we compare the time until the client can send or receive a response in unilaterally authenticated KEMTLS, we see that all handshakes take at least one additional round-trip to complete. As we discussed in section 5.5, KEMTLS requires an additional round-trip for authentication. The client cannot transmit its certificate before it has (implicitly) authenticated the server: otherwise, we cannot protect the client's identity from active attackers. We also wait to obtain the client authentication ciphertext from the server before we allow the client to transmit its request, which would otherwise be unauthenticated. Otherwise, we see that most instances perform comparably; only KKSfSf-KSf, KKXX-KX, and KKSsSs-KSs stand out on the 30.9 ms latency, 1000 Mbps connection: their sizes, or in the case of $\text{XMSS}_{\text{s}}^{\text{MT}}$-I, the computational overhead, result in unavoidable increases. On the 195.5 ms latency, 10 Mbps network however, we see that the computation time of $\text{XMSS}_{\text{s}}^{\text{MT}}$-I is made up by the smaller size of the certificates: it performs 108.8 ms (11.6 %) faster than the KKSsSs-KSs handshake when the intermediate certificate is not transmitted.

Comparison with post-quantum TLS 1.3

In table 13.10, we compare mutually authenticated KEMTLS instances with similar post-quantum TLS 1.3 instantiations. Like in the unilaterally authenticated experiments, using Kyber-512 in the place of any signature scheme but Falcon-512 results in a significant size reduction. HQC-128, however, is larger than most signature schemes; using it in KEMTLS results in additional handshake overhead when used in the place of Dilithium2. When comparing the time until the client receives the response to its request from the server,

Table 13.8: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level I with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	94.8	126.0	63.9	95.1	126.4	64.2
KKFF-KF	94.8	126.0	63.9	95.2	126.4	64.3
KKSfSf-KSf	158.3	189.5	127.4	160.1	191.3	129.2
KKSsSs-KSs	95.4	126.6	64.5	126.9	158.2	96.0
KKXX-KX	130.0	161.2	99.1	151.7	182.8	120.8
HHDD-HD	97.0	128.2	66.1	97.3	128.6	66.4

Table 13.9: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level I with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	597.3	793.4	400.8	685.1	881.7	486.9
KKFF-KF	594.9	791.0	398.4	596.4	792.5	399.9
KKSfSf-KSf	1578.9	1775.0	1382.5	1333.9	1562.9	1094.2
KKSsSs-KSs	736.7	935.0	538.4	1001.9	1216.1	811.3
KKXX-KX	629.8	826.2	433.2	658.0	854.4	461.2
HHDD-HD	660.2	870.6	471.1	638.4	838.0	437.8

Table 13.10: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between mutually authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
TLS	KDDD-DD	13 872	−31.2 %	17 604	−24.6 %	95.8	+31.6 %	96.0	+31.6 %
KEMTLS	KKDD-KD	9544		13 276		126.0		126.4	
TLS	KFFF-FF	6026	+0.2 %	7589	+0.1 %	97.9	+28.7 %	98.2	+28.7 %
KEMTLS	KKFF-KF	6036		7599		126.0		126.4	
TLS	KDFF-DF	10 364	−41.8 %	11 927	−36.3 %	96.0	+31.3 %	96.3	+31.2 %
KEMTLS	KKFF-KF	6036		7599		126.0		126.4	
TLS	KSfSfSf-SfSf	69 984	−44.4 %	87 104	−35.7 %	182.3	+3.9 %	212.8	−10.1 %
KEMTLS	KKSfSf-KSf	38 880		56 000		189.5		191.3	
TLS	KSsSsSs-SsSs	33 056	−38.2 %	40 944	−30.9 %	310.1	−59.2 %	310.5	−49.1 %
KEMTLS	KKSsSs-KSs	20 416		28 304		126.6		158.2	
TLS	KSsXX-SsX	19 302	−65.5 %	20 313	−62.2 %	254.8	−36.7 %	263.1	−30.5 %
KEMTLS	KKXX-KX	6662		7673		161.2		182.8	
TLS	HDDD-DD	19 034	+31.5 %	22 766	+26.3 %	95.8	+33.8 %	96.1	+33.7 %
KEMTLS	HHDD-HD	25 030		28 762		128.2		128.6	

we see that the additional round-trip leads to a significant delay in most experiments. However, in the experiments that use (large-sized) hash-based signature schemes, using Kyber-512 in place of a signature scheme leads to comparable or better performance. Especially using Kyber-512 instead of SPHINCS$^+$-128s results in a large performance improvement, as the latter requires a lot of time to produce handshake signatures.

13.3 Instantiation and results at NIST level III

We measure and compare the characteristics and performance of KEMTLS when instantiated with primitives that offer at least NIST PQC security level III. This security level is comparable to AES-192 in terms of security. It is also the security level that is recommended by the authors of Kyber and Dilithium for

general use [121, 228]. We first compare unilaterally authenticated handshakes and then examine mutually authenticated handshakes.

13.3.1 Unilaterally authenticated KEMTLS

Table 13.11: Instantiations at NIST level III of unilaterally authenticated KEMTLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**		**Int. CA public key**	**Root CA signature**		**Root CA public key**
handle	pk+ct	pk+ct	sig	**Sum**	pk	sig	**Sum**	pk
Primary	Kyber-768	Kyber-768	Dilithium3	7 837	Dilithium3	Dilithium3	13 082	Dilithium3
KKDD	2272	2272	3293		1952	3293		1952
Falcon	Kyber-768	Kyber-768	Falcon-1024	5 824	Falcon-1024	Falcon-1024	8 897	Falcon-1024
KKFF	2272	2272	1280		1793	1280		1793
SPHINCS$^+$-f	Kyber-768	Kyber-768	SPHINCS$^+$-192f	40 208	SPHINCS$^+$-192f	SPHINCS$^+$-192f	75 920	SPHINCS$^+$-192f
KKSfSf	2272	2272	35 664		48	35 664		48
SPHINCS$^+$-s	Kyber-768	Kyber-768	SPHINCS$^+$-192s	20 768	SPHINCS$^+$-192s	SPHINCS$^+$-192s	37 040	SPHINCS$^+$-192s
KKSsSs	2272	2272	16 224		48	16 224		48
Hash-based CA	Kyber-768	Kyber-768	XMSS$_s^{MT}$-III	6 395	XMSS$_s^{MT}$-III	XMSS$_s^{MT}$-III	8 294	XMSS$_s^{MT}$-III
KKXX	2272	2272	1851		48	1851		48
HQC	HQC-192	HQC-192	Dilithium3	30 389	Dilithium3	Dilithium3	35 634	Dilithium3
HHDD	13 548	13 548	3293		1952	3293		1952

Communication requirements

In table 13.11, we show the communication requirements for KEMTLS when instantiated with the selected primitives at security level III. Going from level I to level III increases the required amount of communication for the KKDD experiment by 2281 bytes (41.1 %) when omitting intermediate CA certificates. The KKFF instance, which is based on Kyber-768 and Falcon-1024 even increases by 2022 bytes (53.2 %) (omitting intermediate certificates), although Falcon-1024 offers NIST level V security. All instantiations, except those using SPHINCS$^+$-192 or HQC-192, stay within the roughly 15 kB limit that the congestion window imposes on the server's first transmission, however, even when intermediate certificates are transmitted.

Table 13.12: Computation time in ms for asymmetric cryptography at NIST level III for each of the unilaterally authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD	0.149	0.046	0.195	0.230	0.046	0.276
KKFF	0.240	0.046	0.286	0.412	0.046	0.458
KKSfSf	0.468	0.046	0.514	0.868	0.046	0.914
KKSsSs	0.203	0.046	0.249	0.338	0.046	0.384
KKXX	11.967	0.046	12.013	23.866	0.046	23.912
HHDD	1.569	1.257	2.826	1.650	1.257	2.907

Computational requirements

In table 13.12, we show the computational requirements for KEMTLS when instantiated with the selected primitives at security level III. Compared to the level I computational requirements, we see a small increase in the time required for asymmetric cryptography operations.

Handshake performance

Tables 13.13 and 13.14 show how the instantiations of KEMTLS perform over a 30.9 ms latency, 1000 Mbps, and a 195.5 ms latency, 10 Mbps network. As the experiments that do not use SPHINCS$^+$ do not cross the 10 MSS `initcwnd` limit, we see that the times taken on the fast, high-bandwidth network are very similar; the KKSsSs instance does now cross this limit in the scenario where intermediate CA certificates are not used and thus has a 31.4 ms (33.1 %) increase in time before the client receives the application data response from the server. Interestingly, the HQC-192-based HHDD handshake does not require an additional round-trip when the intermediate CA certificate is omitted: even though the server needs to transmit 16 841 bytes, which is over the 10 MSS `initcwnd` limit. This may be due to the large size of the HQC-192 public key that the client sends to the server in the `ClientHello` message: this large message could already be tuning the congestion window size. The low-bandwidth connection has minor delays due to the extra data, e.g., the KKDD instance with intermediate CA certificate is 45.3 ms (7.1 %) slower.

Table 13.13: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level III with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	63.2	94.6	32.5	63.7	95.1	33.0
KKFF	63.4	94.8	32.7	64.2	95.6	33.5
KKSfSf	96.6	128.1	65.9	130.6	162.0	99.9
KKSsSs	94.9	126.3	64.2	96.2	127.6	65.5
KKXX	103.7	135.1	73.0	116.8	148.1	86.0
HHDD	64.3	97.1	35.0	64.7	97.6	35.5

Table 13.14: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level III with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	398.5	595.7	203.1	465.7	684.1	259.8
KKFF	396.7	593.9	201.3	404.1	602.0	207.9
KKSfSf	977.2	1299.9	667.4	2010.1	2498.4	1613.3
KKSsSs	884.0	1081.2	688.6	901.7	1201.2	612.4
KKXX	451.5	652.5	248.3	505.5	712.5	286.0
HHDD	418.8	636.0	231.4	416.4	626.2	224.8

Table 13.15: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level III.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
TLS	KDDD	10 810	−27.5 %	16 055	−18.5 %	95.1	−0.5 %	125.8	−24.4 %
KEMTLS	KKDD	7837		13 082		94.6		95.1	
TLS	KFFF	6625	−12.1 %	9698	−8.3 %	97.6	−2.8 %	98.1	−2.5 %
KEMTLS	KKFF	5824		8897		94.8		95.6	
TLS	KDFF	8797	−33.8 %	11 870	−25.0 %	95.4	−0.6 %	96.1	−0.5 %
KEMTLS	KKFF	5824		8897		94.8		95.6	
TLS	KSfSfSf	73 648	−45.4 %	109 360	−30.6 %	176.7	−27.5 %	209.7	−22.7 %
KEMTLS	KKSfSf	40 208		75 920		128.1		162.0	
TLS	KSsSsSs	34 768	−40.3 %	51 040	−27.4 %	246.3	−48.7 %	279.2	−54.3 %
KEMTLS	KKSsSs	20 768		37 040		126.3		127.6	
TLS	KSsXX	20 395	−68.6 %	22 294	−62.8 %	254.3	−46.9 %	262.9	−43.7 %
KEMTLS	KKXX	6395		8294		135.1		148.1	
TLS	HDDD	22 086	+37.6 %	27 331	+30.4 %	95.2	+2.0 %	125.8	−22.4 %
KEMTLS	HHDD	30 389		35 634		97.1		97.6	

Comparison with post-quantum TLS 1.3

In table 13.15, we compare the size and performance of post-quantum TLS 1.3 with similar instantiations of KEMTLS. Similar to the comparison at security level I, KEMTLS saves significant amounts of handshake traffic compared to TLS 1.3, except when using HQC-256, which has much larger communication requirements than Dilithium3. A notable difference with the previous security level is that the KKFF instance now is smaller than the KFFF instance: Kyber-768 is a bit smaller than Falcon-1024, although it does have security level V. Also, we see that while KDDD, when including an intermediate CA certificate, requires an additional round-trip due to exceeding the TCP slow start limits, the KKDD KEMTLS instance manages to stay within these limits and is thus a round-trip faster.

13.3.2 Mutually authenticated KEMTLS

Table 13.16: Instantiations at NIST level III of mutually authenticated KEMTLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	Key exchange	Server authentication	Client authentication	Sum	Int. CA certificate	Sum
Primary	Kyber-768	$_{hs:}$Kyber-768	$_{hs:}$Kyber-768		$_{pk:}$Dilithium3	
		$_{sig:}$Dilithium3	$_{sig:}$Dilithium3	13 402	$_{sig:}$Dilithium3	18 647
KKDD-KD	2272	5565	5565		5245	
Falcon	Kyber-768	$_{hs:}$Kyber-768	$_{hs:}$Kyber-768		$_{pk:}$Falcon-1024	
		$_{sig:}$Falcon-1024	$_{sig:}$Falcon-1024	9 376	$_{sig:}$Falcon-1024	12 449
KKFF-KF	2272	3552	3552		3073	
SPHINCS$^+$-f	Kyber-768	$_{hs:}$Kyber-768	$_{hs:}$Kyber-768		$_{pk:}$SPHINCS$^+$-192f	
		$_{sig:}$SPHINCS$^+$-192f	$_{sig:}$SPHINCS$^+$-192f	78 144	$_{sig:}$SPHINCS$^+$-192f	113 856
KKSfSf-KSf	2272	37 936	37 936		35 712	
SPHINCS$^+$-s	Kyber-768	$_{hs:}$Kyber-768	$_{hs:}$Kyber-768		$_{pk:}$SPHINCS$^+$-192s	
		$_{sig:}$SPHINCS$^+$-192s	$_{sig:}$SPHINCS$^+$-192s	39 264	$_{sig:}$SPHINCS$^+$-192s	55 536
KKSsSs-KSs	2272	18 496	18 496		16 272	
Hash-based CA	Kyber-768	$_{hs:}$Kyber-768	$_{hs:}$Kyber-768		$_{pk:}$XMSS$_s^{MT}$-III	
		$_{sig:}$XMSS$_s^{MT}$-III	$_{sig:}$XMSS$_s^{MT}$-III	10 518	$_{sig:}$XMSS$_s^{MT}$-III	12 417
KKXX-KX	2272	4123	4123		1899	
HQC	HQC-192	$_{hs:}$HQC-192	$_{hs:}$HQC-192		$_{pk:}$Dilithium3	
		$_{sig:}$Dilithium3	$_{sig:}$Dilithium3	47 230	$_{sig:}$Dilithium3	52 475
HHDD-HD	13 548	16 841	16 841		5245	

hs: certificate public key and authentication ciphertext
pk: certificate public key sig: certificate signature

Communication requirements

We show the sizes of our instantiations for mutual authentication in table 13.16. Adding a client certificate to the handshake traffic pushes up the total handshake size past 10 kB for all instances when an intermediate CA certificate is included in the communication. The SPHINCS$^+$-192f instance KKSfSf-KSf even comes in at well over 113 kB of traffic.

Computational requirements

In table 13.17, we show how much time the client and the server need to perform all of the computations for the public-key cryptography operations. We see a similar increase in computational requirements as in the unilaterally

Table 13.17: Computation time in ms for asymmetric cryptography at NIST level III for each of the mutually authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD-KD	0.168	0.154	0.322	0.249	0.154	0.403
KKFF-KF	0.259	0.245	0.504	0.431	0.245	0.676
KKSfSf-KSf	0.487	0.473	0.960	0.887	0.473	1.360
KKSsSs-KSs	0.222	0.208	0.430	0.357	0.208	0.565
KKXX-KX	11.986	11.972	23.958	23.885	11.972	35.857
HHDD-HD	2.366	1.798	4.164	2.447	1.798	4.245

authenticated case, but the server now needs to perform a bit more work to authenticate the client's certificate.

Handshake performance

Tables 13.18 and 13.19 show how the mutually authenticated instantiations perform on the 30.9 ms latency, 1000 Mbps and the 195.5 ms latency, 10 Mbps networks. On the low-latency network, we see that the KKDD-KD and KKFF-KF instances are the only experiments that manage to keep the handshake duration to around a multiple of the number of round-trips. The HQC-192-based HHDD-DD instance now requires well more than the congestion window limits and thus needs additional round-trips: it is 32.4 ms (25.3 %) slower than the level-I handshake if an intermediate handshake is omitted from the transmission. If an intermediate certificate is included, this further increases to 63.3 ms (49.2 %) more than the level-I HHDD-HD instance.

In the high-latency, low-bandwidth network, the increase in size pushes up the handshake times even more. When an intermediate certificate is included, the increase in the sizes of Kyber-768 and Dilithium3 compared to their lower-security variants leads to a 84.5 ms (9.6 %) increase in time before the client receives a response from the server. The SPHINCS$^+$-192s based KKSsSs-KSs instance requires 839.6 ms (89.8 %) more if an intermediate certificate is not included before the client receives its response.

Table 13.18: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level III with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	95.1	126.4	64.2	95.6	126.8	64.7
KKFF-KF	95.3	126.5	64.4	95.9	127.2	65.1
KKSfSf-KSf	160.8	192.1	130.0	194.9	226.2	164.0
KKSsSs-KSs	158.1	189.3	127.2	159.4	190.7	128.5
KKXX-KX	140.0	171.2	109.1	159.0	190.1	128.1
HHDD-HD	129.4	160.6	98.5	160.6	191.9	129.7

Table 13.19: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level III with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	607.2	803.5	410.7	769.1	966.2	569.2
KKFF-KF	596.9	793.1	400.5	611.7	808.1	414.7
KKSfSf-KSf	1588.8	1885.9	1351.7	2263.3	2623.2	1928.4
KKSsSs-KSs	1578.4	1774.5	1382.0	1661.9	1860.6	1402.3
KKXX-KX	644.7	841.4	447.6	675.7	872.6	478.4
HHDD-HD	944.6	1168.9	747.5	1255.6	1491.2	1054.2

Table 13.20: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between mutually authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level III.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
TLS	KDDD-DD	19 348	−30.7 %	24 593	−24.2 %	96.6	+30.8 %	127.2	−0.3 %
KEMTLS	KKDD-KD	13 402		18 647		126.4		126.8	
TLS	KFFF-FF	10 978	−14.6 %	14 051	−11.4 %	101.4	+24.7 %	102.2	+24.5 %
KEMTLS	KKFF-KF	9376		12 449		126.5		127.2	
TLS	KDFF-DF	15 322	−38.8 %	18 395	−32.3 %	97.2	+30.1 %	97.8	+30.0 %
KEMTLS	KKFF-KF	9376		12 449		126.5		127.2	
TLS	KSfSfSf-SfSf	145 024	−46.1 %	180 736	−37.0 %	261.4	−26.5 %	293.5	−22.9 %
KEMTLS	KKSfSf-KSf	78 144		113 856		192.1		226.2	
TLS	KSsSsSs-SsSs	67 264	−41.6 %	83 536	−33.5 %	424.3	−55.4 %	459.2	−58.5 %
KEMTLS	KKSsSs-KSs	39 264		55 536		189.3		190.7	
TLS	KSsXX-SsX	38 518	−72.7 %	40 417	−69.3 %	425.0	−59.7 %	426.6	−55.4 %
KEMTLS	KKXX-KX	10 518		12 417		171.2		190.1	
TLS	HDDD-DD	30 624	+54.2 %	35 869	+46.3 %	96.7	+66.2 %	127.2	+50.9 %
KEMTLS	HHDD-HD	47 230		52 475		160.6		191.9	

Comparison with post-quantum TLS 1.3

In table 13.20, we compare equivalent instances of post-quantum TLS 1.3 and KEMTLS. Although mutually authenticated KEMTLS requires an additional round-trip compared to TLS 1.3, we see that in the instance based on Kyber-768 and Dilithium3, KDDDD-DD and KKDD-KD, KEMTLS performs slightly faster than TLS 1.3 if an intermediate CA certificate is part of the handshake. This is due to KEMTLS managing to stay inside the `initcwnd` limit, while the TLS 1.3 instance gains this round-trip due to the amount of handshake traffic and thus ends up with the same amount of round-trips as KEMTLS.

As we saw for level I, HHDD-HD, which is much larger than the HDDD-DD instance, is much slower than TLS 1.3. This is due to Dilithium3 being much smaller than HQC-192. Going the other way, we see that when we replace a SPHINCS$^+$-192 signature with a Kyber-768 key exchange, the handshake sizes and handshake times are drastically reduced.

13.4 Instantiation and results at NIST level V

In this section, we examine the performance and characteristics of KEMTLS when instantiated with post-quantum KEMs and signature schemes at NIST PQC security level V. This highest security level is required by the United States National Security Agency (NSA)'s Commercial National Security Algorithm Suite 2.0 [271] and recommended by French national cybersecurity agency agence nationale de la sécurité des systèmes d'information (ANSSI) [14]. These parameter sets are generally the slowest-running and largest, and will thus affect the performance of KEMTLS the most.

13.4.1 Unilaterally authenticated KEMTLS

Table 13.21: Instantiations at NIST level V of unilaterally authenticated KEMTLS handshakes and the sizes of the public-key cryptography elements in bytes.

		Leaf certificate			Int. CA certificate			Offline
Experiment	**Key Exchange**	**Handshake auth.**	**Int. CA signature**	**Sum**	**Int. CA public key**	**Root CA signature**	**Sum**	**Root CA public key**
handle	pk+ct	pk+ct	sig		pk	sig		pk
Primary	Kyber-1024	Kyber-1024	Dilithium5	10 867	Dilithium5	Dilithium5	18 054	Dilithium5
KKDD	3136	3136	4595		2592	4595		2592
Falcon	Kyber-1024	Kyber-1024	Falcon-1024	7 552	Falcon-1024	Falcon-1024	10 625	Falcon-1024
KKFF	3136	3136	1280		1793	1280		1793
SPHINCS$^+$-f	Kyber-1024	Kyber-1024	SPHINCS$^+$-256f	56 128	SPHINCS$^+$-256f	SPHINCS$^+$-256f	106 048	SPHINCS$^+$-256f
KKSfSf	3136	3136	49 856		64	49 856		64
SPHINCS$^+$-s	Kyber-1024	Kyber-1024	SPHINCS$^+$-256s	36 064	SPHINCS$^+$-256s	SPHINCS$^+$-256s	65 920	SPHINCS$^+$-256s
KKSsSs	3136	3136	29 792		64	29 792		64
Hash-based CA	Kyber-1024	Kyber-1024	XMSS$_s^{MT}$-V	9 251	XMSS$_s^{MT}$-V	XMSS$_s^{MT}$-V	12 294	XMSS$_s^{MT}$-V
KKXX	3136	3136	2979		64	2979		64
HQC	HQC-256	HQC-256	Dilithium5	48 023	Dilithium5	Dilithium5	55 210	Dilithium5
HHDD	21 714	21 714	4595		2592	4595		2592

Communication requirements

In table 13.21, we show how we instantiate KEMTLS at NIST level V and how much data is required for ephemeral key exchange and authentication. Mov-

ing up from level III to level V again results in a large increase in the KKDD instantiation. When excluding intermediate CA certificates from the handshake data, the increase in size is 3030 bytes (38.7 %), when they are included, we need 4972 bytes (38.0 %) more than the level III experiment. As we already used Falcon-1024 in the level III experiment, the increase in size is much more modest for the KKFF experiment and limited to the switch to Kyber-1024: it requires only 1728 bytes (19.4 %) either with or without intermediate CA certificates. Only this experiment and the $XMSS_s^{MT}$-V-based KKXX experiment manage to stay well under 15 kB when intermediate CA certificates are included, all other instantiations well exceed the limit after which extra round-trips will be required by the TCP Slow Start algorithm.

Table 13.22: Computation time in ms for asymmetric cryptography at NIST level V for each of the unilaterally authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD	0.281	0.080	0.361	0.442	0.080	0.522
KKFF	0.292	0.080	0.372	0.464	0.080	0.544
KKSfSf	0.517	0.080	0.597	0.914	0.080	0.994
KKSsSs	0.304	0.080	0.384	0.488	0.080	0.568
KKXX	11.155	0.080	11.235	22.190	0.080	22.270
HHDD	2.220	1.730	3.950	2.381	1.730	4.111

Computational requirements

The computation times for asymmetric cryptography operations required by the level V instantiations of our experiments are listed in table 13.22. Compared to the level III Kyber-768, Kyber-1024 requires 0.017 ms (63.0 %) more for encapsulation and 0.017 ms (89.5 %) more for decapsulation. If we go back to the level I Kyber-512 parameter set, Kyber-1024 requires 0.018 ms (69.2 %) more for a single encapsulation and 0.018 ms (100.0 %) more for decapsulation. Although the relative increase is quite large, compared to even our low-latency experiment, which has a round-trip latency of 30.9 ms, these time increases are insignificant and barely contribute to the handshake la-

tency. For Dilithium signature verification, it is a similar story: Dilithium5 is 0.080 ms (98.8 %) slower than Dilithium3 and 0.097 ms (151.6 %) slower than Dilithium2 when verifying a signature. Likewise, Falcon-1024 signature verification is 0.031 ms (22.0 %) slower than Falcon-512. Even though we perform multiple of these operations, the total time required for computation remains well under 1 ms for almost all instantiations. The exception is the $\text{XMSS}_{\text{s}}^{\text{MT}}$-V parameter set, which takes a very large amount of time to verify a signature; although for this scheme the increase in verification time starting from $\text{XMSS}_{\text{s}}^{\text{MT}}$-I to $\text{XMSS}_{\text{s}}^{\text{MT}}$-V is only 2.803 ms (34.1 %).

Handshake performance

When we compare the times shown in table 13.23 for KEMTLS handshakes run over a 30.9 ms latency, 1000 Mbps network to those for NIST security level III, we see the same results for handshakes that do not include intermediate CA certificates. The only exception in this scenario is the KKSfSf handshake, which now requires another additional 31.7 ms (24.8 %) to transfer the large SPHINCS^{+}-256f signature in the server's certificate. When intermediate CA certificates are transferred, the KKDD and HHDD handshakes, which now exceed 18 kB, need an additional round-trip compared to the level III instantiation: it takes 31.2 ms (32.8 %) more time before the client receives the response from the server.

In table 13.24, we show the performance for KEMTLS handshakes run over the high-latency 195.5 ms ms, 10 Mbps network. Aside from the increase in round-trips that we also observe in the results for the low-latency network, we see additional effects of the large sizes of the level V parameters.

Comparison with post-quantum TLS 1.3

In table 13.25, we compare how KEMTLS instantiated at NIST security level V compares to similar instantiations of post-quantum TLS 1.3. For KKDD, KEMTLS still offers a smaller handshake size than the TLS 1.3 KDDD instantiation, but both handshakes are now sufficiently large that they need the same number of round-trips again. The KKFF instantiation with Falcon-1024 is now slightly larger than the KFFF instance, as Falcon-1024 is close in size to the Kyber-1024 instance. In the level III experiment, we had to pair this level V scheme with a level III KEM, but now these are matched again and the difference in size has disappeared. For HQC-256-based HHDD, we see that HQC-256 is so much larger than Dilithium5 that we suffer a very large amount

Table 13.23: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level V with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	63.5	94.9	32.8	94.8	126.3	64.2
KKFF	63.5	95.0	32.9	64.2	95.7	33.6
KKSfSf	128.3	159.8	97.7	163.0	194.6	132.4
KKSsSs	95.9	127.4	65.2	128.9	160.5	98.3
KKXX	110.3	141.7	79.6	120.6	151.9	89.8
HHDD	65.5	130.7	68.5	96.9	162.1	100.0

Table 13.24: Average handshake times in ms for unilaterally authenticated KEMTLS experiments at NIST level V with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD	421.8	622.2	225.4	854.5	1078.5	685.9
KKFF	397.2	594.9	202.3	404.5	603.0	208.7
KKSfSf	1626.8	1960.5	1311.2	2936.1	3489.8	2368.2
KKSsSs	841.2	1121.5	573.4	1719.7	2124.0	1290.1
KKXX	465.9	668.5	261.4	512.2	721.2	296.7
HHDD	467.7	966.0	556.0	947.1	1580.3	1180.6

Table 13.25: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level V.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
TLS	KDDD	14 918	−27.2 %	22 105	−18.3 %	95.6	−0.7 %	127.0	−0.6 %
KEMTLS	KKDD	10 867		18 054		94.9		126.3	
TLS	KFFF	7489	+0.8 %	10 562	+0.6 %	97.5	−2.6 %	98.2	−2.6 %
KEMTLS	KKFF	7552		10 625		95.0		95.7	
TLS	KDFF	11 603	−34.9 %	14 676	−27.6 %	95.7	−0.7 %	96.4	−0.7 %
KEMTLS	KKFF	7552		10 625		95.0		95.7	
TLS	KSfSfSf	102 912	−45.5 %	152 832	−30.6 %	200.9	−20.5 %	229.4	−15.2 %
KEMTLS	KKSfSf	56 128		106 048		159.8		194.6	
TLS	KSsSsSs	62 784	−42.6 %	92 640	−28.8 %	270.0	−52.8 %	278.1	−42.3 %
KEMTLS	KKSsSs	36 064		65 920		127.4		160.5	
TLS	KSsXX	35 971	−74.3 %	39 014	−68.5 %	246.2	−42.4 %	261.7	−42.0 %
KEMTLS	KKXX	9251		12 294		141.7		151.9	
TLS	HDDD	33 496	+43.4 %	40 683	+35.7 %	95.6	+36.7 %	127.1	+27.6 %
KEMTLS	HHDD	48 023		55 210		130.7		162.1	

of handshake time overhead as well as more handshake traffic. Finally, in the hash-based schemes, we see the large effect of replacing a SPHINCS$^+$-256 signature with a Kyber-1024 KEM key exchange, which has only gotten more significant as the size of the hash-based schemes has increased much more than the size of Kyber.

13.4.2 Mutually authenticated KEMTLS

Communication requirements

In table 13.26, we show how we instantiate mutually authenticated KEMTLS with the sizes required for the public-key cryptography elements. At level V, any mutually authenticated handshake transfers more than 10 kB, even when intermediate CA certificates are omitted. When intermediate CA certificates

Table 13.26: Instantiations at NIST level V of mutually authenticated KEMTLS experiments and the sizes of the public-key cryptography elements transmitted in bytes.

Experiment handle	**Key exchange**	**Server authentication**	**Client authentication**	**Sum**	**Int. CA certificate**	**Sum**
Primary	Kyber-1024	hs:Kyber-1024	hs:Kyber-1024		pk:Dilithium5	
		sig:Dilithium5	sig:Dilithium5	18 598	sig:Dilithium5	25 785
KKDD-KD	3136	7731	7731		7187	
Falcon	Kyber-1024	hs:Kyber-1024	hs:Kyber-1024		pk:Falcon-1024	
		sig:Falcon-1024	sig:Falcon-1024	11 968	sig:Falcon-1024	15 041
KKFF-KF	3136	4416	4416		3073	
SPHINCS$^+$-f	Kyber-1024	hs:Kyber-1024	hs:Kyber-1024		pk:SPHINCS$^+$-256f	
		sig:SPHINCS$^+$-256f	sig:SPHINCS$^+$-256f	109 120	sig:SPHINCS$^+$-256f	159 040
KKSfSf-KSf	3136	52 992	52 992		49 920	
SPHINCS$^+$-s	Kyber-1024	hs:Kyber-1024	hs:Kyber-1024		pk:SPHINCS$^+$-256s	
		sig:SPHINCS$^+$-256s	sig:SPHINCS$^+$-256s	68 992	sig:SPHINCS$^+$-256s	98 848
KKSsSs-KSs	3136	32 928	32 928		29 856	
Hash-based CA	Kyber-1024	hs:Kyber-1024	hs:Kyber-1024		pk:XMSS$_s^{MT}$-V	
		sig:XMSS$_s^{MT}$-V	sig:XMSS$_s^{MT}$-V	15 366	sig:XMSS$_s^{MT}$-V	18 409
KKXX-KX	3136	6115	6115		3043	
HQC	HQC-256	hs:HQC-256	hs:HQC-256		pk:Dilithium5	
		sig:Dilithium5	sig:Dilithium5	74 332	sig:Dilithium5	81 519
HHDD-HD	21 714	26 309	26 309		7187	

hs: certificate public key and authentication ciphertext
pk: certificate public key sig: certificate signature

are included, all experiments exceed 15 kB.

Computational requirements

Table 13.27 shows the total time necessary for all the asymmetric cryptographic operations for the client and the server. We can see that the KKDD-KD instantiation requires the smallest amount of computation for both the client and the server, but a millisecond is still a very short amount of time compared to even the fast network setup with a 30.9 ms round-trip latency.

Handshake performance

Table 13.28 show the performance of mutually authenticated KEMTLS instantiated with primitives with NIST security level V on the 30.9 ms round-trip latency, 1000 Mbps network. Like in the unilaterally authenticated results, all level V experiments but the KKFF-KF instantiation require, when intermedi-

Table 13.27: Computation time in ms for asymmetric cryptography at NIST level V for each of the mutually authenticated KEMTLS instantiations at the client and server.

Handle	Intermediate cert. as root			With intermediate CA cert.		
	Client	Server	Sum	Client	Server	Sum
KKDD-KD	0.317	0.285	0.602	0.478	0.285	0.763
KKFF-KF	0.328	0.296	0.624	0.500	0.296	0.796
KKSfSf-KSf	0.553	0.521	1.074	0.950	0.521	1.471
KKSsSs-KSs	0.340	0.308	0.648	0.524	0.308	0.832
KKXX-KX	11.191	11.159	22.350	22.226	11.159	33.385
HHDD-HD	3.322	2.519	5.841	3.483	2.519	6.002

ate CA certificates are included, additional round-trips for data as they exceed the TCP Slow Start algorithm's initial congestion window size. When the CA certificate is omitted, however, the KKDD-KD and KKFF-KF instantiations still have a three-round-trip handshake latency. The difference between these two experiments, when intermediate certificates are omitted, is negligible on the high-bandwidth network, even though KKDD-KD is 6630 bytes (55.4 %) larger than KKFF-KF.

Table 13.29 shows the performance of mutually authenticated KEMTLS on the 195.5 ms latency, low-bandwidth 10 Mbps network. Here we see that KKDD-KD is 79.7 ms (10.0 %) slower than KKFF-KF when intermediate CA certificates are omitted due to the larger size taking more time to transfer over the low-bandwidth connection. When they are included, the difference in handshake time dramatically increases to 465.7 ms (55.6 %). Going from security level III to security level V leads to a 337.2 ms (34.9 %) increase for KKDD-KD on the low-bandwidth network, an arguably large increase in cost compared to the gain in security margin.

Comparison with post-quantum TLS 1.3

In table 13.30, we compare the KEMTLS instantiations with similarly instantiated post-quantum TLS 1.3 experiments. We see mostly the same things that we have seen already in the table for NIST security level I; all KEMTLS instantiations have an extra round-trip compared to what is needed for TLS 1.3. The

Table 13.28: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level V with 30.9 ms latency and 1000 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	95.5	126.7	64.6	126.8	158.0	95.9
KKFF-KF	95.5	126.7	64.6	96.2	127.4	65.3
KKSfSf-KSf	223.9	255.2	193.0	258.7	290.0	227.8
KKSsSs-KSs	159.9	191.2	129.1	193.2	224.4	162.3
KKXX-KX	147.7	178.9	116.8	163.3	194.4	132.4
HHDD-HD	136.2	167.4	105.3	167.7	199.0	136.8

Table 13.29: Average handshake times in ms for mutually authenticated KEMTLS experiments at NIST level V with 195.5 ms latency and 10 Mbps bandwidth. Server and client timers are independent.

Handle	Intermediate cert. as root			With root certificate		
	Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
KKDD-KD	671.6	874.4	479.4	1099.5	1303.4	905.9
KKFF-KF	598.6	794.7	402.1	640.8	837.7	443.0
KKSfSf-KSf	2951.0	3300.3	2789.5	4859.3	5208.0	4719.4
KKSsSs-KSs	1489.0	1771.8	1302.5	2082.3	2362.1	1783.1
KKXX-KX	664.9	862.6	468.1	696.2	895.6	496.3
HHDD-HD	1444.3	1640.7	1245.4	1280.5	1517.1	1027.3

Table 13.30: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between mutually authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level V.

Experiment		Handshake size (bytes)				Time until response (ms)			
		No int.	**Δ%**	**With int.**	Δ%	**No int.**	Δ%	**With int.**	Δ%
TLS	KDDD-DD	26 700	−30.3 %	33 887	−23.9 %	97.5	+29.9 %	129.0	+22.5 %
KEMTLS	KKDD-KD	18 598		25 785		126.7		158.0	
TLS	KFFF-FF	11 842	+1.1 %	14 915	+0.8 %	101.4	+25.0 %	102.1	+24.7 %
KEMTLS	KKFF-KF	11 968		15 041		126.7		127.4	
TLS	KDFF-DF	20 070	−40.4 %	23 143	−35.0 %	97.7	+29.7 %	98.5	+29.4 %
KEMTLS	KKFF-KF	11 968		15 041		126.7		127.4	
TLS	KSfSfSf-SfSf	202 688	−46.2 %	252 608	−37.0 %	333.7	−23.5 %	340.6	−14.9 %
KEMTLS	KKSfSf-KSf	109 120		159 040		255.2		290.0	
TLS	KSsSsSs-SsSs	122 432	−43.6 %	152 288	−35.1 %	481.2	−60.3 %	481.6	−53.4 %
KEMTLS	KKSsSs-KSs	68 992		98 848		191.2		224.4	
TLS	KSsXX-SsX	68 806	−77.7 %	71 849	−74.4 %	408.8	−56.2 %	409.1	−52.5 %
KEMTLS	KKXX-KX	15 366		18 409		178.9		194.4	
TLS	HDDD-DD	45 278	+64.2 %	52 465	+55.4 %	97.6	+71.6 %	128.9	+54.4 %
KEMTLS	HHDD-HD	74 332		81 519		167.4		199.0	

only difference is the KSfSfSf-SfSf TLS 1.3 experiment, which is now 78.5 ms (23.5 %) slower than the KEMTLS experiment when using intermediate CA certificates as root certificates, while at security level I KEMTLS was 7.2 ms (3.9 %) faster in this experiment.

13.5 Summary

Summarizing the results from the experiments across NIST security levels I, III, and V, we see that KEMTLS generally performs well when instantiated with Kyber, Dilithium, and Falcon, the algorithms selected by NIST for standardization. We show the handshake sizes and the time until the client receives a response from the server in figure 13.1. Only at level V does the Kyber-Dilithium KKDD parameter set no longer fit in the initial congestion window,

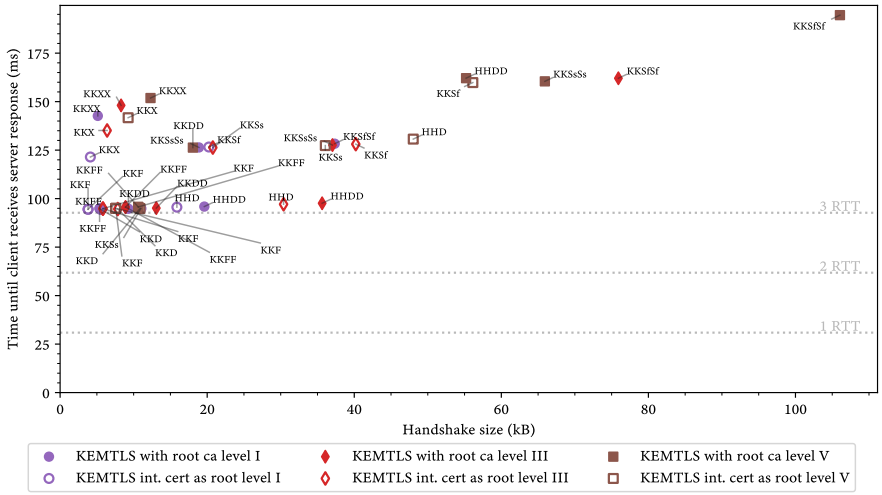


Figure 13.1: Handshake timings of KEMTLS experiments

requiring an additional round-trip. Using Falcon in KEMTLS for the certificate signature algorithms offers the best performance: this signature scheme is very small, and the certificate signatures are only computed when the certificates are generated, avoiding the implementation concerns of Falcon's signing operation. The HQC KEM, a round-4 candidate in the NIST PQC standardization project, is significantly larger than Kyber, and this results in significantly larger handshake sizes. The HHDD instance using HQC-256, the level V parameter set, requires an additional round-trip due to the large sizes of the KEM's public key and ciphertext.

Using hash-based schemes in KEMTLS seems not very attractive. The instantiations using SPHINCS$^+$ are very large and have significant overhead due to the transmission size. Using SPHINCS$^+$-s does seem more viable than using SPHINCS$^+$-f: as only signature verification is required we do not suffer the large computational effort of producing SPHINCS$^+$-s signatures. The XMSS$_s^{MT}$ scheme suffers from the overhead of its signature verification, although this is made up at level V by its smaller size when comparing it to the level V SPHINCS$^+$-based instantiations.

13.6 Comparing KEMTLS and post-quantum TLS 1.3

13.6.1 Handshake size and performance

In general terms, the difference between KEMTLS and TLS 1.3 for unilaterally authenticated handshakes is exactly the effect of replacing one post-quantum signature scheme public key and signature with a KEM public key and ciphertext in the handshake. As a result, we see for example that the size of the handshake is reduced when we replace Dilithium with Kyber, and very slightly increases when we replace Falcon with Kyber. In figure 13.1, where we show how the KEMTLS and post-quantum TLS 1.3 instances based on Kyber, Dilithium, and Falcon compare in terms of size and handshake latency, this effect can be seen. As Kyber is faster than both algorithms, the KEMTLS instances are always at least slightly faster as well. More significant differences are visible when comparing the level III TLS 1.3 instantiation KDDD with the KEMTLS instance KKDD: the additional size of Dilithium pushes the TLS 1.3 instance just over the TCP slow start algorithm's initial congestion window, so TLS 1.3 requires an additional round-trip.

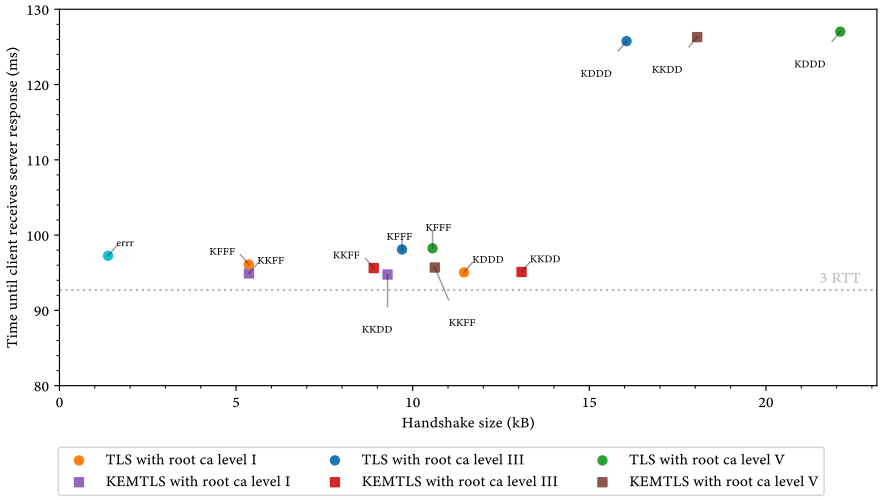


Figure 13.2: Handshake timings of the KDDD and KFFF TLS 1.3 and the KKDD and KKFF KEMTLS experiments.

13.6.2 Fewer CPU cycles

Most KEMTLS instances are more computationally efficient than the equivalent TLS 1.3 instances. For example, the level I KKDD KEMTLS instance requires 0.198 ms (45.6 %) less computation time than the TLS 1.3 KDDD instance (when intermediate CA certificates are included). While this had only a minor effect on our experiments, which ran on a powerful server, this may be more consequential for embedded or battery-operated devices that have less powerful CPUs. It may also have a measurable impact on the maximum number of connections that a server can handle, but our experiment was not set up to measure this. This remains an avenue for future work.

13.6.3 No handshake signatures

As we have already mentioned, KEMTLS only requires signature verification for both unilaterally authenticated and mutually authenticated handshakes. As a result, KEMTLS servers no longer need efficient and secure implementations of signing, a routine that has been the target of various side-channel attacks [35, 80, 156, 196, 359]. When client authentication is used, the same goes for clients; otherwise, the effect is reduced because clients still need code to verify signatures in certificates. However, this code does not deal with any secret data and thus does not need side-channel protection. Reducing the amount of (trusted) code is particularly attractive for embedded devices, which typically have tight constraints on code size and are often exposed to a variety of side-channel attacks. As embedded devices may be more likely to use mutual authentication than web browsing, being able to reuse the KEM implementation for both ephemeral key exchange and client authentication instead of requiring additional signature generation code can be very beneficial. If raw public keys are used, as specified in RFC 7250, signature verification code can even be omitted entirely [358]. We further discuss the performance of KEMTLS and the size of the trusted code base in chapter 16.

Requirements for post-quantum signatures

The signatures on the certificates are only generated in the more confined secure environment of certificate authorities. This means that KEMTLS also enables using signature algorithms that may not be suitable for all deployments. Notably, stateful hash-based signatures are very sensitive to how the state is

managed, but this may be doable when implemented in hardware security modules employed by CAs. As mentioned before, in KEMTLS we can also use Falcon for the certificate signatures while avoiding its signing algorithm in the server and client, without requiring implementations of three algorithms like in the TLS 1.3 KDFF instance (an algorithm for key exchange, for signing and verification in the handshake and for signature verification of certificates).

Going further, many post-quantum signature schemes can tweak parameters to make different trade-offs between signature size, signing speed, public-key size, and verification speed. One common direction to optimize for is signing speed, or more precisely signing *latency* reported as the number of clock cycles for a single signature. The common motivation for this optimization is the use of online signatures in handshake protocols like the one used in TLS 1.3 (and earlier versions) or the SIGMA handshake approach [216] used, for example, in the Internet key-exchange protocol (IKE) [208].

In KEMTLS, signatures are only needed for certificates and thus computed offline. This eliminates the requirement for low-latency signing; what remains important (depending to some extent on certificate-caching strategies) is signature size, public-key size, verification latency, and—at least for certificate authorities—signing *throughput*. However, throughput can easily be achieved for any signature scheme by signing the root of an XMSS or LMS tree and using the leaves of that tree to sign a batch of messages. This for example discussed in [250, Sec. 6]; we also referred to this in section 11.6.

13.7 Conclusion

KEMTLS offers attractive reductions in handshake size while also reducing computational requirements. In some instantiations, KEMTLS can allow a handshake to remain below the `initcwnd` limit, and thus avoid an additional round-trip due to the TCP Slow Start algorithm. Otherwise, performance was usually only slightly faster than equivalent TLS 1.3 instantiations using Kyber, Dilithium, and Falcon in our experiments. KEMTLS also allows reducing the trusted code base and picking signature algorithms that have implementation or deployment considerations on TLS servers and clients. This can for example be used to achieve small-code-size implementations of mutually authenticated KEMTLS that use Kyber and Falcon, which only require protected implementations for Kyber.

14 Performance of KEMTLS-PDK

In this chapter, we examine the performance of post-quantum KEMTLS-PDK. Specifically, we first look at how it can be instantiated and the sizes of these instantiations. Then, we show the performance of the instantiations of KEMTLS-PDK handshakes when run over a high-bandwidth, low-latency network, and a low-bandwidth, high-latency network. We do this for NIST PQC security levels I, III, and V, and for both unilaterally and mutually authenticated handshakes.

KEMTLS-PDK avoids transmission of the server's certificates, so for a fair comparison we compare to a variant of TLS 1.3 in which we allow the client to *cache* the server's certificate. We denote this variant by cTLS. We will also compare the performance against the results obtained for KEMTLS, previously discussed in chapter 13.

For a description of how we implemented KEMTLS-PDK, please refer back to section 10.5. The implementation of the caching mechanism is discussed in section 10.2.4. The design of the emulated network environment and our choice of parameters are motivated in section 10.10.

As in previous chapters, we are only comparing the sizes and performance characteristics of the schemes used to instantiate KEMTLS-PDK in this chapter. It can be argued that this compares apples to oranges: different security assumptions both affect the size and performance, but also have different levels of confidence associated with them. However, these comparisons are still useful to estimate the cost of choosing different assumptions based on such considerations.

14.1 Instantiations

As we did for the experiments in chapters 11 and 13, we base our main selection of algorithms on the NIST PQC standardization project.

In each instantiation, we select:

1. an algorithm for ephemeral key exchange, negotiated by the KEMTLS client and server, and;

2. an algorithm for handshake authentication, used in the server certificate. This algorithm is a KEM for KEMTLS-PDK and a signature scheme in cTLS experiments.

For KEMTLS-PDK handshakes that use mutual authentication, we additionally select:

3. an algorithm for client authentication, used in the client certificate, and;

4. an algorithm for authentication of the client's certificate by a CA certificate, which is assumed to be preinstalled.

Note that we do not select algorithms for signatures by CAs on server certificates: as those are assumed to be already available, we can assume that the certificate has been verified on installation.

In our experiments, we try to showcase how the different algorithms in the NIST PQC standardization project perform, like in previous chapters. We will use the following scenarios:

Primary In this scenario we use Kyber [319], the only KEM which was selected for standardization for post-quantum key exchange. We use Kyber for both the ephemeral key exchange and in KEMTLS-PDK for handshake authentication. In cTLS and for authentication of KEMTLS-PDK client certificates, we use Dilithium [241], the algorithm which was named the *primary* algorithm for post-quantum signatures when selected for standardization, for the signature in the handshake.

HQC This instantiation uses HQC [3], a round-4 KEM candidate in the NIST PQC standardization project, for ephemeral key exchange and handshake authentication. HQC relies on assumptions based on decoding of quasi-cyclic codes, instead of on assumptions on lattices. In cTLS and for authentication of KEMTLS-PDK client certificates, we use Dilithium signatures.

McEliece In this scenario, we minimize the size of the KEMTLS-PDK handshake by using the Classic McEliece algorithm [7] for handshake authentication. Classic McEliece has a very large public key, but the ciphertext,

the only thing we transmit for handshake authentication in KEMTLS-PDK, is very small. McEliece's public key is too large for transmission in the ClientHello message, so for the ephemeral key exchange we again use Kyber. To further minimize the size of this handshake and to compare to minimized cTLS handshakes, we use Falcon [293] as the signature scheme for the client certificate CA signature and in the signatures in cTLS.

BIKE [17], the remaining round-4 candidate in the NIST PQC standardization project, does not have an IND-CCA secure parameter set. Consequently, it cannot be securely used in KEMTLS or cTLS.

Note that for presentation purposes, we will refer to these scenarios in our tables and figures by handles, which are composed of the first letters of each of the selected algorithms. We separate the algorithms used for client authentication from the ephemeral key exchange and server authentication algorithms using a hyphen. For example, the instantiation using Kyber for ephemeral key exchange and server authentication, Kyber for client authentication, and Dilithium for the CA signature is denoted KK-KD. For an overview of these handles, refer to the tables that show the communication sizes, e.g. table 14.1.

Recall that in our design of KEMTLS-PDK, we assume that there are many more clients than servers. Therefore, for mutually authenticated handshakes, we submit the client's certificate to the KEMTLS-PDK server and do not require the server to store all client public keys.

14.2 Instantiation and results at NIST level I

In this section, we measure and compare the performance of KEMTLS-PDK with cTLS instantiated with post-quantum primitives at NIST PQC security level I. The level I parameter sets of KEMs and signature schemes are the smallest and generally the most performant. We discuss both unilaterally and mutually authenticated handshakes.

Communication requirements

In table 14.1 we show which algorithms are used for key exchange and server authentication. We also show which algorithms are used to achieve mutually authenticated handshakes: specifically the client authentication and client authentication CA certificate algorithms. We also show how much the client

and mutually authenticating server need to store for the instantiation.

Table 14.1: Instantiations of unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) at NIST security level I with the sizes of public-key cryptography elements in bytes.

		Unilateral authentication				Mutual authentication			Cached data	
	Experiment handle	**Key exchange**	**Server authentication**	**Sum**		**Client auth.**	**Certificate signature**	**Sum**	**Server public key**	**Client auth. CA pk**
KEMTLS-PDK	**Primary** KK(-KD)	Kyber-512 1568	Kyber-512 ct 768	2 336		Kyber-512 pk+ct 1568	Dilithium2 2420	6 324	Kyber-512 800	Dilithium2 1312
KEMTLS-PDK	**Falcon** KK(-KF)	Kyber-512 1568	Kyber-512 ct 768	2 336		Kyber-512 pk+ct 1568	Falcon-512 666	4 570	Kyber-512 800	Falcon-512 897
KEMTLS-PDK	**HQC** HH(-HD)	HQC-128 6730	HQC-128 ct 4481	11 211		HQC-128 pk+ct 6730	Dilithium2 2420	20 361	HQC-128 2249	Dilithium2 1312
KEMTLS-PDK	**McEliece** KM(-KF)	Kyber-512 1568	McEliece348864 ct 96	1 664		Kyber-512 pk+ct 1568	Falcon-512 666	3 898	McEl.348864 261 120	Falcon-512 897
cTLS	**Primary** KD(-DD)	Kyber-512 1568	Dilithium2 sig 2420	3 988		Dilithium2 pk+sig 3732	Dilithium2 2420	10 140	Dilithium2 1312	Dilithium2 1312
cTLS	**Falcon** KF(-FF)	Kyber-512 1568	Falcon-512 sig 666	2 234		Falcon-512 pk+sig 1563	Falcon-512 666	4 463	Falcon-512 897	Falcon-512 897
cTLS	**HQC** HD(-DD)	HQC-128 6730	Dilithium2 sig 2420	9 150		Dilithium2 pk+sig 3732	Dilithium2 2420	15 302	Dilithium2 1312	Dilithium2 1312

pk: certificate public key ct: authentication ciphertext sig: handshake signature

In general, this table shows that KEMTLS-PDK is much more efficient than cTLS when instantiated with Kyber-512 and Dilithium2. The KK instance, which does not use mutual authentication, is 1652 bytes (70.7 %) smaller than the Kyber-512 and Dilithium2-based KD instantiation. This is exactly the difference in size between a Kyber-512 ciphertext and a Dilithium2 signature. Using Classic McEliece 348864 for authentication of the server further reduces the size of the handshake. As a Falcon-512 signature is smaller than a Kyber-512 ciphertext, that instantiation is smaller than all but the KM(-KF) KEMTLS instantiation; however, as we will see in the next paragraph, Kyber-512 is more computationally efficient. Finally, as we have seen in our experiments with TLS 1.3 and KEMTLS in chapters 11 and 13, HQC-128's performance is hampered by the large size of its public key and ciphertext.

Computational requirements

In table 14.2, we show the computational requirements of the instantiations of KEMTLS-PDK and cTLS. We show the total time needed for asymmetric cryp-

Table 14.2: Computation time in ms for asymmetric cryptography at NIST level I for each of the unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) instantiations at the client and server.

	Handle	Unilateral auth.			Mutual auth.		
		Client	Server	Both	Client	Server	Both
PDK	KK(-KD)	0.064	0.044	0.108	0.082	0.134	0.216
PDK	KK-KF				0.082	0.211	0.293
PDK	HH(-HD)	0.472	0.406	0.878	0.739	0.609	1.348
PDK	KM(-KF)	0.064	0.075	0.139	0.082	0.242	0.324
cTLS	KD(-DD)	0.102	0.204	0.306	0.280	0.332	0.612
cTLS	KF(-FF)	0.179	0.668	0.847	0.821	0.950	1.771
cTLS	HD(-DD)	0.397	0.317	0.714	0.575	0.445	1.020

tographic operations by the client and the server. In KEMTLS-PDK, the client generates a KEM public key and decapsulates a ciphertext for the ephemeral key exchange. It also encapsulates a ciphertext to the server's long-term KEM public key. The server performs the encapsulation in the ephemeral key exchange and decapsulates the ciphertext encapsulated to its long-term public key. If using mutual authentication, the client additionally decapsulates the ciphertext which the server additionally encapsulates to the client's long-term public key. Note that there is only one aspect in which KEMTLS-PDK and cTLS differ from KEMTLS and TLS 1.3: in KEMTLS-PDK and cTLS, the clients do not need to verify the CA signatures on the certificates. This is because we assume that they have been verified when they were cached by the client.

Using KEMTLS-PDK thus does not avoid much computation compared to KEMTLS, but avoiding signatures in the KEMTLS-PDK handshake does save computation time compared to cTLS when using any algorithm but the comparatively slow HQC-128: In the KK instantiation, the server requires 0.160 ms (363.6 %) more for computation in the Kyber-512 plus Dilithium2 KD experiment. Although Classic McEliece 348864 has a very large public key, its raw encapsulation and decapsulation operations are very efficient: KD is 0.167 ms (120.1 %) slower than KM.

Table 14.3: Average handshake times in ms for unilaterally authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level I. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK	62.8	93.9	31.9	394.6	590.6	197.4
	HH	63.5	94.6	32.5	398.4	594.4	198.1
	KM	65.9	97.0	35.0	397.1	593.1	200.4
cTLS	KD	63.6	94.7	32.7	396.5	592.5	200.0
	KF	64.6	95.7	33.7	396.5	592.5	199.9
	HD	63.6	94.7	32.7	396.4	592.4	199.9

Table 14.4: Average handshake times in ms for mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level I. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK-KD	63.2	94.4	32.3	398.4	594.4	200.4
	KK-KF	63.3	94.5	32.4	397.0	593.0	199.0
	HH-HD	65.0	96.1	34.1	417.2	617.4	210.3
	KM-KF	66.4	97.6	35.5	399.6	595.7	201.7
cTLS	KD-DD	64.2	95.8	33.7	399.3	615.5	221.1
	KF-FF	66.0	97.8	35.8	397.8	596.8	204.3
	HD-DD	64.2	95.8	33.7	399.5	616.3	221.8

Handshake performance

In tables 14.3 and 14.4, we show the average time that elapsed from the initiation of the connection until the moment that the client can send a request and when the client received a response from the server. For the server, we additionally show the time from receiving the `ClientHello` message to the completion of the handshake (receiving `ClientFinished`). We again show these handshake times a 30.9 ms low RTT, 1000 Mbps high-bandwidth network and a 195.5 ms high RTT, 10 Mbps low-bandwidth network.

Table 14.3 shows the results for unilaterally authenticated handshakes, while table 14.14 shows the results for mutually authenticated handshakes. The times for the different experiments are very similar. This is not surprising as even though relatively there are significant differences in computation time, these are not very significant compared to the handshake latency. The KEMTLS-PDK instances complete the handshake in a very similar time to the cTLS equivalents. Surprisingly, the KM instantiation that uses Classic McEliece 348864 requires slightly more time than the KK instantiation that uses only Kyber; while based on the benchmarks of the cryptographic operations, as we've discussed for table 14.2, and based on the smaller handshake size Classic McEliece should be faster. This may be due to the large public key hurting performance when it is loaded in and out of caches in our larger benchmark.[1]

When using mutual authentication, shown in table 14.4, the large size of the HH-HD experiment slows it down compared to all other experiments. It requires 23.0 ms (3.9 %) more than the KK-KD KEMTLS-PDK experiment. cTLS equivalent HD-DD requires 1.1 ms (0.2 %) less time. Otherwise, we see that the small size of, for example, the KK-KD experiment results in a minor performance improvement: the client receives the response 1.4 ms (1.5 %) faster on average compared to KD-DD on the fast network.

Comparison to KEMTLS

Compared to KEMTLS, KEMTLS-PDK avoids the transmission of the server's public key and the certificate chain. For example, using the KEMTLS-PDK KK instantiation saves 3220 bytes (58.0 %) compared to the KKDD KEMTLS instantiation that uses Dilithium2 and is 1466 bytes (38.6 %) smaller than the

[1] As an anecdotal example of benchmarks not reporting real-world performance, Google removed the assembly implementation of NTRU-HRSS [186] from BoringSSL in favor of the reference code, reporting that the larger AVX2 code negatively affected cache pressure [234].

Table 14.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between post-quantum KEMTLS and KEMTLS-PDK instances at NIST level I.

	Experiment		Handshake size (bytes)				Time until response (ms)			
			No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
unilateral authentication	KEMTLS	KKDD	5 556	−58.0 %	9 288	−74.8 %	94.4	−0.5 %	94.8	−0.9 %
	KEMTLS-PDK	KK	2 336		2 336		93.9		93.9	
	KEMTLS	KKFF	3 802	−38.6 %	5 365	−56.5 %	94.5	−0.6 %	94.9	−1.1 %
	KEMTLS-PDK	KK	2 336		2 336		93.9		93.9	
	KEMTLS	KKFF	3 802	−56.2 %	5 365	−69.0 %	94.5	+2.7 %	94.9	+2.2 %
	KEMTLS-PDK	KM	1 664		1 664		97.0		97.0	
	KEMTLS	HHDD	15 880	−29.4 %	19 612	−42.8 %	95.6	−1.0 %	95.9	−1.3 %
	KEMTLS-PDK	HH	11 211		11 211		94.6		94.6	
mutual authentication	KEMTLS	KKDD-KD	9 544	−33.7 %	13 276	−52.4 %	126.0	−25.1 %	126.4	−25.3 %
	KEMTLS-PDK	KK-KD	6 324		6 324		94.4		94.4	
	KEMTLS	KKFF-KF	6 036	−24.3 %	7 599	−39.9 %	126.0	−25.0 %	126.4	−25.3 %
	KEMTLS-PDK	KK-KF	4 570		4 570		94.5		94.5	
	KEMTLS	KKFF-KF	6 036	−35.4 %	7 599	−48.7 %	126.0	−22.5 %	126.4	−22.8 %
	KEMTLS-PDK	KM-KF	3 898		3 898		97.6		97.6	
	KEMTLS	HHDD-HD	25 030	−18.7 %	28 762	−29.2 %	128.2	−25.0 %	128.6	−25.2 %
	KEMTLS-PDK	HH-HD	20 361		20 361		96.1		96.1	

smallest instantiation of KEMTLS that uses Falcon-512 when omitting intermediate CA certificates in both experiments. If they are included, we save even more data. The smaller size of the handshake and the small reduction in necessary computations also results in a minor reduction in handshake times for unilaterally authenticated handshakes. We further compare KEMTLS-PDK with similar KEMTLS handshakes, the latter both with and without intermediate CA certificates, in table 14.5.

When compared to mutually authenticated KEMTLS, KEMTLS-PDK has a much faster handshake: KEMTLS requires a full additional round-trip to transmit the client certificate.[2] The KEMTLS KK-KD handshake takes 31.6 ms (33.5 %)

[2] As discussed in section 5.5, KEMTLS cannot send the client certificate earlier, because then it cannot be transmitted securely; in KEMTLS-PDK, we can encrypt the client certificate using the key encapsulated to the server's long-term key in the ClientHello message.

longer than the KEMTLS KKDD-KD handshake before the client receives a response from the server.

14.3 Instantiation and results at NIST level III

Next, we show how KEMTLS-PDK can be instantiated using primitives at NIST PQC security level III and how this affects the size of the handshake and the performance. We compare KEMTLS to cached-TLS 1.3 (cTLS) instantiations using similar primitives. Security level III is comparable to AES-192 in terms of security and the security level that is recommended by the authors of Kyber and Dilithium for general use [121, 228].

Table 14.6: Instantiations of unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) at NIST security level III with the sizes of public-key cryptography elements in bytes.

		Unilateral authentication			Mutual authentication			Cached data	
	Experiment handle	**Key exchange**	**Server authentication**	**Sum**	**Client auth.**	**Certificate signature**	**Sum**	**Server public key**	**Client auth. CA pk**
KEMTLS-PDK	**Primary** KK(-KD)	Kyber-768 2272	Kyber-768 ct 1088	3 360	Kyber-768 pk+ct 2272	Dilithium3 3293	8 925	Kyber-768 1184	Dilithium3 1952
KEMTLS-PDK	**Falcon** KK(-KF)	Kyber-768 2272	Kyber-768 ct 1088	3 360	Kyber-768 pk+ct 2272	Falcon-1024 1280	6 912	Kyber-768 1184	Falcon-1024 1793
KEMTLS-PDK	**HQC** HH(-HD)	HQC-192 13 548	HQC-192 ct 9026	22 574	HQC-192 pk+ct 13 548	Dilithium3 3293	39 415	HQC-192 4522	Dilithium3 1952
KEMTLS-PDK	**McEliece** KM(-KF)	Kyber-768 2272	McEliece460896 ct 156	2 428	Kyber-768 pk+ct 2272	Falcon-1024 1280	5 980	McEl.460896 524 160	Falcon-1024 1793
cTLS	**Primary** KD(-DD)	Kyber-768 2272	Dilithium3 sig 3293	5 565	Dilithium3 pk+sig 5245	Dilithium3 3293	14 103	Dilithium3 1952	Dilithium3 1952
cTLS	**Falcon** KF(-FF)	Kyber-768 2272	Falcon-1024 sig 1280	3 552	Falcon-1024 pk+sig 3073	Falcon-1024 1280	7 905	Falcon-1024 1793	Falcon-1024 1793
cTLS	**HQC** HD(-DD)	HQC-192 13 548	Dilithium3 sig 3293	16 841	Dilithium3 pk+sig 5245	Dilithium3 3293	25 379	Dilithium3 1952	Dilithium3 1952

pk: certificate public key ct: authentication ciphertext sig: handshake signature

Communication requirements

Table 14.6 shows how many bytes we need to transmit the public keys, ciphertexts, and signatures when instantiating KEMTLS-PDK and cTLS handshakes using the listed NIST security level III algorithms. As we have previously seen in the experiments with post-quantum TLS 1.3 and KEMTLS in sections 11.3

and 13.3, the lack of a Falcon parameter set with security level III makes the KF cTLS instantiations larger than the Kyber-768-based KEMTLS-PDK instantiations. KF is a significant 1124 bytes (46.3 %) larger than the instantiation that uses Classic McEliece 460896; though this comes at the significant cost of the KM client having to store 524 160 bytes. In mutually authenticated handshakes, however, KEMTLS-PDK still transmits full client certificates, and the large size of Dilithium3 pushes the KK-KD instance over the size of the KF-FF experiment. If Falcon-1024 can be used in the client authentication CA certificate, the size of the handshake can be reduced further.

Table 14.7: Computation time in ms for asymmetric cryptography at NIST level III for each of the unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) instantiations at the client and server.

	Handle	Unilateral auth.			Mutual auth.		
		Client	Server	Both	Client	Server	Both
PDK	KK(-KD)	0.068	0.046	0.114	0.087	0.154	0.241
	KK-KF				0.087	0.245	0.332
	HH(-HD)	1.488	1.257	2.745	2.285	1.798	4.083
	KM(-KF)	0.078	0.122	0.200	0.097	0.321	0.418
cTLS	KD(-DD)	0.122	0.830	0.952	0.925	0.992	1.917
	KF(-FF)	0.213	0.813	1.026	0.999	1.157	2.156
	HD(-DD)	1.109	1.263	2.372	1.912	1.425	3.337

Computational requirements

The time that is required for the asymmetric cryptography computations in our experiments is listed in table 14.7. Unlike the level I instantiations, the KM(-KF) instantiations now require slightly more time for computation than the KK(-KD) experiments.

Handshake performance

Table 14.8 shows the time until certain events in the handshakes for unilaterally authenticated KEMTLS-PDK and cTLS when ran over low-latency, high-bandwidth connections and when ran over high-latency, low-bandwidth connections. The Kyber-768 KEMTLS-PDK instantiation KK performs the best of

Table 14.8: Average handshake times in ms for unilaterally authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level III. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK	62.8	93.9	31.9	394.9	591.0	197.4
	HH	64.4	95.5	33.4	412.9	614.1	204.0
	KM	69.2	100.3	38.3	400.6	596.7	203.6
cTLS	KD	64.0	95.1	33.0	397.7	593.7	201.2
	KF	66.1	97.2	35.1	398.4	594.4	201.9
	HD	64.0	95.1	33.0	397.7	593.7	201.2

Table 14.9: Average handshake times in ms for mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level III. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK-KD	63.4	94.6	32.5	403.3	600.1	203.0
	KK-KF	63.7	94.9	32.8	398.9	594.9	200.8
	HH-HD	96.6	127.8	65.6	808.7	1042.7	455.5
	KM-KF	70.1	101.2	39.1	409.0	607.1	207.3
cTLS	KD-DD	64.7	96.5	34.5	419.7	670.7	259.4
	KF-FF	68.7	101.1	39.1	400.9	602.5	210.0
	HD-DD	64.8	96.6	34.5	420.2	671.9	260.4

all handshakes, although by a small margin. Surprisingly, the client waits 6.4 ms (6.8 %) longer before it receives the response from the server in KM than in KK, which is not explained by either computation time (based on benchmarks of the primitives) or the transmission size, which is lower in KM than in KD. This suggests that loading the Classic McEliece 460896 public key, which is very large, affects this handshake's real-world performance. On the low-bandwidth network, the large size of the HQC-192 HH handshake results in a slowdown: it takes 3.8 ms (0.6 %) longer than the KK handshake on this network.

In the mutually authenticated handshakes, as shown in table 14.9, the HH-HD instantiation is the slowest. It needs an additional round-trip due to the large size of the HQC-192 public keys and ciphertexts, which exceed the `initcwnd` size of the TCP Slow Start algorithm [66]. On the low-bandwidth connection, this is exacerbated by the slow speed of the connection: HH-HD requires 442.6 ms (73.8 %) longer than KK-KD before the client receives the response from the server.

Comparison to KEMTLS

In table 14.10, we compare the level III instantiations of KEMTLS-PDK against similar KEMTLS instantiations. As the sizes of the post-quantum signature schemes' public keys and signatures are larger in level III, the absolute reduction in handshake size is larger than we saw in the comparison for level I in table 14.5. Relatively, we see that the reductions in size are similar. Finally, that difference in handshake times between KEMTLS and KEMTLS-PDK is slightly larger for unilaterally authenticated handshakes than we saw in the level I experiments.

Table 14.10: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between post-quantum KEMTLS and KEMTLS-PDK instances at NIST level III.

	Experiment		Handshake size (bytes)				Time until response (ms)			
			No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
unilateral authentication	KEMTLS	KKDD	7 837	−57.1 %	13 082	−74.3 %	94.6	−0.7 %	95.1	−1.2 %
	KEMTLS-PDK	KK	3 360		3 360		93.9		93.9	
	KEMTLS	KKFF	5 824	−42.3 %	8 897	−62.2 %	94.8	−1.0 %	95.6	−1.8 %
	KEMTLS-PDK	KK	3 360		3 360		93.9		93.9	
	KEMTLS	KKFF	5 824	−58.3 %	8 897	−72.7 %	94.8	+5.8 %	95.6	+4.9 %
	KEMTLS-PDK	KM	2 428		2 428		100.3		100.3	
	KEMTLS	HHDD	30 389	−25.7 %	35 634	−36.7 %	97.1	−1.6 %	97.6	−2.1 %
	KEMTLS-PDK	HH	22 574		22 574		95.5		95.5	
mutual authentication	KEMTLS	KKDD-KD	13 402	−33.4 %	18 647	−52.1 %	126.4	−25.2 %	126.8	−25.4 %
	KEMTLS-PDK	KK-KD	8 925		8 925		94.6		94.6	
	KEMTLS	KKFF-KF	9 376	−26.3 %	12 449	−44.5 %	126.5	−25.0 %	127.2	−25.4 %
	KEMTLS-PDK	KK-KF	6 912		6 912		94.9		94.9	
	KEMTLS	KKFF-KF	9 376	−36.2 %	12 449	−52.0 %	126.5	−20.0 %	127.2	−20.4 %
	KEMTLS-PDK	KM-KF	5 980		5 980		101.2		101.2	
	KEMTLS	HHDD-HD	47 230	−16.5 %	52 475	−24.9 %	160.6	−20.5 %	191.9	−33.4 %
	KEMTLS-PDK	HH-HD	39 415		39 415		127.8		127.8	

14.4 Instantiation and results at NIST level V

In this section, we examine the performance and characteristics of KEMTLS-PDK when instantiated at NIST PQC security level V. The United States National Security Agency (NSA)'s Commercial National Security Algorithm Suite 2.0 [271] requires level V, and it is recommended by French national cybersecurity agency agence nationale de la sécurité des systèmes d'information (ANSSI) [14]. Level V instantiations of the post-quantum primitives are generally the slowest-running and largest, and will thus affect the performance of KEMTLS-PDK the most.

Communication requirements

Table 14.11 lists the algorithms which are used to instantiate KEMTLS-PDK and cTLS at NIST security level V. It also shows the sizes of the public keys, ciphertexts, and signatures that need to be transmitted. We also show how

Table 14.11: Instantiations of unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) at NIST security level V with the sizes of public-key cryptography elements in bytes.

		Unilateral authentication			Mutual authentication			Cached data	
	Experiment handle	Key exchange	Server authentication	Sum	Client auth.	Certificate signature	Sum	Server public key	Client auth. CA pk
KEMTLS-PDK	**Primary** KK(-KD)	Kyber-1024 3136	Kyber-1024 ct 1568	4 704	Kyber-1024 pk+ct 3136	Dilithium5 4595	12 435	Kyber-1024 1568	Dilithium5 2592
KEMTLS-PDK	**Falcon** KK(-KF)	Kyber-1024 3136	Kyber-1024 ct 1568	4 704	Kyber-1024 pk+ct 3136	Falcon-1024 1280	9 120	Kyber-1024 1568	Falcon-1024 1793
KEMTLS-PDK	**HQC** HH(-HD)	HQC-256 21 714	HQC-256 ct 14 469	36 183	HQC-256 pk+ct 21 714	Dilithium5 4595	62 492	HQC-256 7245	Dilithium5 2592
KEMTLS-PDK	**McEliece** KM(-KF)	Kyber-1024 3136	McEliece6688128 ct 208	3 344	Kyber-1024 pk+ct 3136	Falcon-1024 1280	7 760	McEl.6688128 1 044 992	Falcon-1024 1793
cTLS	**Primary** KD(-DD)	Kyber-1024 3136	Dilithium5 sig 4595	7 731	Dilithium5 pk+sig 7187	Dilithium5 4595	19 513	Dilithium5 2592	Dilithium5 2592
cTLS	**Falcon** KF(-FF)	Kyber-1024 3136	Falcon-1024 sig 1280	4 416	Falcon-1024 pk+sig 3073	Falcon-1024 1280	8 769	Falcon-1024 1793	Falcon-1024 1793
cTLS	**HQC** HD(-DD)	HQC-256 21 714	Dilithium5 sig 4595	26 309	Dilithium5 pk+sig 7187	Dilithium5 4595	38 091	Dilithium5 2592	Dilithium5 2592

pk: certificate public key ct: authentication ciphertext sig: handshake signature

much data needs to be stored by the client and the server for the public keys that we assume to be pre-distributed or cached. For security level V, this means that using Classic McEliece 6688128 in KM(-KF) requires storing 1 044 992 bytes on the client's side. While this is a very large amount of data for, for example, embedded platforms, the KM handshake is also the smallest in terms of bytes that need to be transmitted. Thus, if network environments are very constrained, it may still be an attractive option. Using HQC-256, on the other hand, requires transmission of so much data that using HH seems impractical compared to Kyber-1024.

Computational requirements

How much time the KEMTLS-PDK and cTLS instantiations require for the asymmetric cryptography operations encapsulate, decapsulate, signing, and verification is shown in table 14.12. All KEMTLS-PDK instantiations but the one using HQC-256 require much less computation than the cTLS instantiations: the most efficient KEMTLS-PDK instantiation, KK, the server requires 0.553 ms (691.2 %) less computation time than the cTLS KD instantiation with the lowest computational requirements.

Table 14.12: Computation time in ms for asymmetric cryptography at NIST level V for each of the unilaterally and mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) instantiations at the client and server.

	Handle	Unilateral auth.			Mutual auth.		
		Client	Server	Both	Client	Server	Both
PDK	KK(-KD)	0.120	0.080	0.200	0.156	0.285	0.441
	KK-KF				0.156	0.296	0.452
	HH(-HD)	2.059	1.730	3.789	3.161	2.519	5.680
	KM(-KF)	0.138	0.160	0.298	0.174	0.376	0.550
cTLS	KD(-DD)	0.237	0.633	0.870	0.826	0.955	1.781
	KF(-FF)	0.248	0.830	1.078	1.034	1.174	2.208
	HD(-DD)	1.592	1.217	2.809	2.181	1.539	3.720

Table 14.13: Average handshake times in ms for unilaterally authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level V. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK	62.9	94.0	31.9	395.4	591.4	197.4
	HH	96.7	127.9	34.8	885.5	1081.6	200.4
	KM	75.6	106.7	44.6	407.0	603.1	209.9
cTLS	KD	64.3	95.4	33.3	399.3	595.3	202.7
	KF	66.1	97.3	35.2	398.5	594.5	202.0
	HD	64.3	95.4	33.3	399.2	595.2	202.7

Table 14.14: Average handshake times in ms for mutually authenticated KEMTLS-PDK and cached-TLS 1.3 (cTLS) experiments at NIST level V. Server and client timers are independent.

	Handle	30.9 ms RTT, 1000 Mbps			195.5 ms RTT, 10 Mbps		
		Client sent req.	Client recv. resp.	Server HS done	Client sent req.	Client recv. resp.	Server HS done
PDK	KK-KD	63.7	94.8	32.7	407.0	605.0	206.6
	KK-KF	63.8	95.0	32.9	401.4	597.6	202.1
	HH-HD	133.0	164.2	71.1	1102.0	1371.7	521.6
	KM-KF	76.8	107.9	45.8	419.1	618.5	214.9
cTLS	KD-DD	65.2	97.3	35.2	455.1	732.6	289.5
	KF-FF	68.7	101.1	39.0	401.0	602.7	210.2
	HD-DD	65.1	97.2	35.1	455.2	733.0	289.8

Handshake performance

The performance of unilaterally authenticated KEMTLS-PDK and cTLS handshakes is shown in table 14.13. We show the times until the client can send a request, receives the response, and the time that the server spends in the handshake, both on a low-latency, high-bandwidth network and a high-latency, low-bandwidth network. The KEMTLS-PDK KK handshake is the most performant; the similar KD cTLS instantiation requires 1.4 ms (1.5 %) more time before the client receives the response from the server. As in the level III experiments, the large size of the HQC-256 public key and ciphertext results in poor handshake performance: HH requires an additional round-trip as the server's handshake traffic exceeds the `initcwnd` size. On the 10 Mbps low-bandwidth network, the size leads to further slowdown: HH is 490.2 ms (82.9 %) slower than KK. The Classic McEliece6688128-based KM KEMTLS-PDK handshake is the smallest handshake, but like we have seen in the experiments at the lower security levels its performance appears to be held back by the very large public key having to be loaded in and out of memory; even though this is not reflected by the benchmark results in table 14.12.

The performance of mutually authenticated handshakes is shown in table 14.14. The performance on the high-bandwidth, low-latency network is very close to the unilaterally authenticated handshakes, again except for the

HQC-256 HH-HD experiment. Transmitting another HQC-256 public key and ciphertext compared to the unilaterally authenticated HH KEMTLS-PDK instantiation results in further slowdowns: the HH-HD experiment is 69.3 ms (73.1 %) slower than the Kyber-1024 based KK-KD handshake.

Table 14.15: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between post-quantum KEMTLS and KEMTLS-PDK instances at NIST level V.

	Experiment		Handshake size (bytes)				Time until response (ms)			
			No int.	Δ%	With int.	Δ%	No int.	Δ%	With int.	Δ%
unilateral authentication	KEMTLS	KKDD	10 867	−56.7 %	18 054	−73.9 %	94.9	−1.0 %	126.3	−25.6 %
	KEMTLS-PDK	KK	4 704		4 704		94.0		94.0	
	KEMTLS	KKFF	7 552	−37.7 %	10 625	−55.7 %	95.0	−1.1 %	95.7	−1.8 %
	KEMTLS-PDK	KK	4 704		4 704		94.0		94.0	
	KEMTLS	KKFF	7 552	−55.7 %	10 625	−68.5 %	95.0	+12.3 %	95.7	+11.5 %
	KEMTLS-PDK	KM	3 344		3 344		106.7		106.7	
	KEMTLS	HHDD	48 023	−24.7 %	55 210	−34.5 %	130.7	−2.2 %	162.1	−21.1 %
	KEMTLS-PDK	HH	36 183		36 183		127.9		127.9	
mutual authentication	KEMTLS	KKDD-KD	18 598	−33.1 %	25 785	−51.8 %	126.7	−25.2 %	158.0	−40.0 %
	KEMTLS-PDK	KK-KD	12 435		12 435		94.8		94.8	
	KEMTLS	KKFF-KF	11 968	−23.8 %	15 041	−39.4 %	126.7	−25.1 %	127.4	−25.5 %
	KEMTLS-PDK	KK-KF	9 120		9 120		95.0		95.0	
	KEMTLS	KKFF-KF	11 968	−35.2 %	15 041	−48.4 %	126.7	−14.8 %	127.4	−15.3 %
	KEMTLS-PDK	KM-KF	7 760		7 760		107.9		107.9	
	KEMTLS	HHDD-HD	74 332	−15.9 %	81 519	−23.3 %	167.4	−2.0 %	199.0	−17.5 %
	KEMTLS-PDK	HH-HD	62 492		62 492		164.2		164.2	

Comparison to KEMTLS

Table 14.15 compares similar KEMTLS and KEMTLS-PDK handshakes. In all unilaterally authenticated handshakes but the HH handshake, KEMTLS-PDK saves over 50 % of handshake traffic compared to KEMTLS handshakes that use intermediate CA certificates. This size reduction can also reduce the number of round-trips necessary: the KEMTLS KKDD experiment, when including intermediate CA certificates in the handshake, exceeded the initial congestion window size and needed another round-trip. Due to this extra round-trip, KEMTLS requires 32.3 ms (34.4 %) more time than KEMTLS-PDK until the client

receives the server's response.

14.5 Discussion

If it is possible to store the server's long-term public key, for instance, because it is part of firmware or bundled in a software library, KEMTLS-PDK can greatly reduce the bandwidth cost compared to KEMTLS handshakes. It is more efficient than similar approaches in TLS, such as our cTLS implementation, as these still rely on larger digital signatures. KEMTLS-PDK has similar benefits to KEMTLS compared to TLS 1.3, in that the trusted code base of applications does not need to contain signature generation code. If the client has the server public key preinstalled, for example in a firmware image, signature verification code could also be omitted entirely.

14.5.1 Using Classic McEliece

As we have previously discussed in section 8.1.2, KEMTLS-PDK allows choosing algorithms, specifically Classic McEliece, for the server authentication key exchange that would otherwise not fit in the KEMTLS or TLS 1.3 key exchange. Not only does this result in the smallest KEMTLS-PDK instantiations, but Classic McEliece also has the longest history of analysis of any KEM: the scheme goes back to 1978 [248]. The very large public keys of Classic McEliece do appear to have a noticeable effect on the handshake latency, and may also be prohibitive for embedded applications that do not have large amounts of flash storage. These public keys are also so large, that using Classic McEliece is only beneficial if the public key never or only rarely needs to be updated; distributing Classic McEliece public keys for short-term session resumption likely does not result in net data savings.

14.5.2 TLS 1.3's pre-shared key resumption

We did not compare KEMTLS-PDK to TLS session resumption using a symmetric pre-shared key (for authentication) and DH (or KEMs) for forward secrecy. The reason is that in this scenario clients need to keep a sensitive secret key that is shared with the server; we believe this makes KEMTLS-PDK usable in more scenarios than TLS 1.3 with pre-shared keys. Refer to section 6.2 for more discussion of KEM public keys versus symmetric keys. For scenarios

that currently do use TLS 1.3 with symmetric-key resumption, such as web browsing, quantifying the difference in performance between KEMTLS-PDK and (KEM)TLS with symmetric key resumption remains an open question.

In chapter 15 we examine the performance of post-quantum TLS 1.3, KEMTLS, and KEMTLS-PDK in a more realistic scenario, measuring the handshake performance between two data centers.

14.6 Conclusions

When the long-term public key of the server can be stored by the client, KEMTLS-PDK offers savings both in terms of handshake size and handshake latency. Additionally, KEMTLS-PDK achieves this without having to manage sensitive symmetric shared secret keys. At the same time, KEMTLS-PDK achieves the same benefits as KEMTLS, such as the reduced size of the trusted code base and allowing to pick signature algorithms for the intermediate CA certificates that have implementation concerns, such as Falcon, without increasing the amount of code necessary.

15 Measuring the performance of KEMTLS over the internet

15.1 Introduction

In the previous chapters, we investigated the impact of post-quantum cryptography on TLS 1.3, OPTLS, KEMTLS, and KEMTLS-PDK using an emulated network setup and a dummy application. In this chapter, we analyze the impact of transitioning to post-quantum cryptography in TLS in a more realistic setting. We first show how to experiment in production systems, by delegating trust from existing, CA-issued certificates to not-yet-standardized post-quantum schemes. We measure TLS 1.3 and KEMTLS-PDK handshakes using post-quantum algorithms on a real-world system: a distributed network that is subject to actual internet traffic conditions and spans two continents. We record the latency of these handshakes and compare them against a baseline TLS 1.3 handshake, considering both server-only and mutual authentication. For this experiment, we modified the Go programming language's TLS library, providing another example of how post-quantum TLS and KEMTLS can be implemented.

Please note that due to the nature of the work in this chapter, which took place in 2021, results have not been updated to the latest versions of the referenced schemes. In this chapter, we are using NIST PQC standardization project round-2 and round-3 parameter sets.

15.2 Delegating trust from existing certificates

There are no theoretical obstacles for transitioning TLS 1.3 to a post-quantum world: indeed, we described this in chapter 3. In this chapter, we will refer to the post-quantum instantiation of TLS 1.3 by PQTLS.

In principle, the main hurdle to experimenting with post-quantum algorithms in TLS seems the matter of implementing them. There are, however,

practical considerations that go further for real-world experiments. CAs must adapt their software to include post-quantum signatures, and, historically, the Web PKI and other X.509 PKIs have limited which algorithms can be used. It could take a long time until new algorithms are widely deployed. These changes may occur in the future but, for experimentation and rapid deployment, these issues become limitations.

In this chapter, we use a practical approach to overcome this problem. Specifically, we rely on a delegation mechanism for handshake authentication. A delegated credential (DC) is an authenticated credential valid for a short period (at most 7 days) that can be used to decouple the handshake authentication algorithm from the authentication algorithms used in the certificate chain. The DC contains a public key to be used for authentication in the handshake and, in turn, it is authenticated by the end-entity certificate. This mechanism allows separating sensitive long-term certificate keys from the machines that terminate TLS connections, which instead can be issued short-lived DCs. This is especially useful for globally distributed content-delivery networks, where the location in which the TLS server is physically located may not be fully trusted. The document describing this technique, "Delegated Credentials for TLS and DTLS" [24], was standardized at the IETF in July 2023.

Using delegated credentials comes with other advantages for our experiments. Unlike a regular certificate, a delegated credential is smaller and has no other extensions, such as revocation lists and certificate statuses, which makes it a perfect fit for experiments where the size of parameters is important. Additionally, DCs are generated and validated only in TLS clients and servers, which reduces the number of codebases or systems where we needed to roll out new algorithms. This is especially relevant when using client authentication.

While the draft specification for DCs states that the DC authentication algorithm "is expected to be the same as the sender's `CertificateVerify.algorithm`", the draft allows the use of different authentication algorithms than the algorithm that was used in the server or client certificate. We use this to our benefit, to perform the first PQTLS and KEMTLS handshakes that have a CA-issued authentication root. We simply delegate a post-quantum signature scheme or KEM public key as DC from the "classical" CA-issued certificate. This allows us to circumvent restrictions posted by CA infrastructure.

We note that if full post-quantum security is required, every link in the certificate chain needs to use post-quantum secure algorithms. Authentication

is only as strong as its weakest link, so until every link has post-quantum security we do not have a fully post-quantum authenticated protocol. This includes the root CA certificate, as well as any intermediate CA certificates. We emphasize that the use of DCs in our experiment thus does not grant post-quantum authentication, but only serves to enable the experiments.

To authenticate a TLS server, a client relies on a chain of certificates: a root CA certificate, typically followed by at least one intermediate CA certificate, and then the leaf certificate of the server. Intermediate and server certificates may be cached, preinstalled, or suppressed, which means that less data needs to be transmitted during the handshake; but these mechanisms are not widely deployed. This means that for a full post-quantum TLS 1.3 handshake, peers typically transmit the whole certificate chain (except for the root CA certificate) and verify all signatures (at least three signatures or other proofs of authentication). Note that the use of DCs increases the amount of data transmitted and the number of signature verification operations, as it adds another step in the authentication chain to the root CA certificate.

15.3 Implementing post-quantum TLS in Go

Go is an open-source, high-level programming language, whose standard library includes support for the TLS protocol (including TLS 1.3). The way Go's standard library is organized allows us to make modifications to its internals without requiring third-party libraries. Go also has mechanisms to interact with low-level features of the computer architecture. This is particularly useful for accessing architecture-specific capabilities, which are only available through assembly code.

The submissions to the NIST PQC standardization project include (optimized) implementations in C and platform-specific assembly code. We relied on those implementations for the Rustls-based experiments in chapters 13 and 14 through a wrapper around the `liboqs` library. The OQS project also has a wrapper for Go through the `cgo` programming interface. However, we can observe performance degradation compared to the native version, due to the way `cgo` interacts with native Go code. The CIRCL [141] library provides AVX2-optimized implementations for several post-quantum algorithms natively in Go, including SIDH and SIKE [197].[1] As part of the work in this

[1]Although SIDH and SIKE been broken since this experiment was completed [50],

chapter, we contributed AVX2-optimized implementations of the Dilithium signature scheme (round-2 parameters) and the Kyber key encapsulation mechanism to CIRCL. Table 15.1 shows performance timings of cryptographic operations measured on a 4.2 GHz Core i7-8650U processor.

Table 15.1: Average time in ms for performing KEM and signature scheme operations.

KEM	Encapsulate	Decapsulate
X25519	0.062	0.032
X448	0.249	0.150
Kyber512	0.016	0.016
SIKEp434	4.020	6.720

Signature	Sign	Verify
Ed25519	0.028	0.059
Ed448	0.104	0.192
Dilithium3	0.191	0.077
Dilithium4	0.144	0.218

Go provides a clean implementation of TLS 1.3. However, the implementation is conservative regarding the type of extensions and algorithms that it supports. Changing the TLS 1.3 implementation to include delegated credentials and PQTLS required including some extensions and adding certain algorithm identifiers. It also meant adding a mechanism for generating and validating delegated credentials, as well as adding the support for the delegated credentials X.509 extension in generated certificates. We also added support for the cached information extension [312], which indicates that the client already has the server's certificate, and modified it to work with TLS 1.3 for KEMTLS-PDK.

Integrating KEMTLS and KEMTLS-PDK was more challenging. Doing so required the interruption of the original TLS 1.3 handshake flow, the order in which client and server exchange messages, depending on whether the client is using server-only authentication, or is mutually authenticating. If the client is using prior knowledge of the server's certificate and thus using KEMTLS-PDK, there is yet another sequence of messages. This differs from the standard TLS 1.3 handshake that follows the same flow of messages regardless if server-only or mutual authentication is performed. These differences were an important lesson learned during our implementation as it was often a source of errors. We did not integrate the KEMTLS-PDK mutual authentica-

the results are indicative of algorithms that sacrifice computation time for reduced size.

tion mechanism, as it was not clear when the original work was done how to securely transmit the client certificate.

15.4 Experimenting over the public internet

We analyze the effects on TLS handshakes connecting over the public internet when using post-quantum algorithms. We measure the time it takes for a TLS 1.3 handshake using certificate-based authentication to complete. We then compare all experiments to this baseline measurement.

15.4.1 The network environment

To test and measure TLS connections, we chose a service that operates under common internet conditions and spans different geographical locations. Drand [338], the target application for our experimentation, is a distributed randomness beacon written in Go. Linked servers produce publicly-verifiable random numbers at fixed time intervals. A threshold signature scheme prevents collusion or biasing the generation of numbers. Network nodes communicate with one another using the gRPC protocol [244] with TLS authentication. Additionally, public randomness is exposed through an HTTPS endpoint. Note that we only experiment with the transport encryption; we do not change the distributed random number generation protocol.

Changes to the Drand code base are minimal, being mostly limited to the configuration. We needed to provide and configure a certificate with the DC extension enabled for servers and clients. We also set which protocol will be initiated (KEMTLS or PQTLS) in the TLS configuration. If KEMTLS-PDK was to be used, a "regular" KEMTLS handshake is run first, from which information is cached (the `ServerCertificate` message), and then the obtained certificate is used in a fresh KEMTLS-PDK handshake by passing it at the TLS configuration level. We added configuration options for ease of experimentation: in a more realistic scenario stating which key exchange and authentication algorithms are supported should be enough to trigger the appropriate protocol execution.

At runtime, fresh DCs are generated each time that a request arrives. However, these credentials can be further cached and stored, so they can be reused between connections. A mechanism that routinely checks the validity of these credentials can also be implemented. This shows that DCs can be easily implemented and used without needing to constantly modify certificate storage

or retrieval. It is worth noting that adding DCs increases the number of signature validations: the certificate chain has to be validated, the DC has to be validated, and the handshake has to be validated.

15.4.2 Experiment setup

We build a Drand cluster with one leader node and three worker peers. These ran independently in a data center located in Portland, USA. The connections of each internal node and the external HTTPS interface are configured to support post-quantum handshake protocols.

A Drand client retrieves randomness from the Drand network. We opted for locating the client far from the Drand network itself, so it is located in Lisbon, Portugal. With this setup, our experiment faces the same traffic conditions found in transatlantic connections.

We choose a combination of cryptographic algorithms for setting up the following handshake configurations:

TLS 1.3 handshake using Ed25519 certificates for authentication (baseline).

TLS 1.3+DC handshake with Ed25519 certificate and delegated credentials either using Ed25519 or Ed448 algorithms for authentication.

PQTLS handshake with SIKEp434 or Kyber-512 for key exchange, and hybrid signatures using round-two Dilithium level 3 or level 4, respectively, paired with Ed25519 and Ed448 for authentication (the authentication keys are transmitted via DCs).

KEMTLS handshake with SIKEp434 or Kyber-512 for both key exchange and authentication (the authentication keys are transmitted via DCs).

KEMTLS-PDK handshake using the same configuration as KEMTLS (server authentication only).

15.4.3 Measurements

For each client-to-server connection, we measure the time elapsed until completion of the TLS handshake, that is until the client can send encrypted application data, for each different handshake configuration. We also measure the elapsed time for each flight of the handshake, i.e., the time elapsed that a

peer (server or client) waits for receiving messages from their counterpart. We initiate two timers: one for the client (which starts when the ClientHello message was constructed and sent) and one for the server (which starts when the ClientHello message is received). Therefore, the first and second flights, as seen in the tables, do not include network latency, as the timer is started prior to the message being sent or just when it is received, respectively. Note that the RTTs from the third flight onward are affected by the conditions of the state of the network. We tested the scenarios over an average-latency network.

To reduce the effects caused by the state of the network, the Drand client was instructed to fetch randomness from the Drand server consecutively over one hour. The total number of connections during this period amounts to approximately 5000 connections. We report the average timings in tables 15.2 and 15.3. We also measure the total time until the session is completed (note that these times include the sending and receiving of encrypted application data). The average times to session completion are listed in tables 15.4 and 15.5.

In server-only authentication, the handshake consists of the following flights:

1. ($C \Rightarrow S$) Sending ClientHello for (PQ)TLS, KEMTLS and KEMTLS-PDK.
   KEMTLS-PDK: As part of ClientHello we include ClientKemCiphertext and a hash of the cached server's ServerCertificate message.

2. ($C \Leftarrow S$) Processing of ClientHello.
   TLS 1.3 and PQTLS: reply with the ServerHello, Certificate, CertificateVerify and Finished messages.
   KEMTLS: reply with the ServerHello and Certificate messages.
   KEMTLS-PDK: reply with the ServerHello and ServerFinished messages.

3. ($C \Rightarrow S$) Processing of received messages based on the protocol.
   TLS 1.3 and PQTLS: processing of ServerHello, Certificate, CertificateVerify and Finished messages. Reply with ClientFinished and immediate transmission of encrypted application data.
   KEMTLS: processing of ServerHello and Certificate. Reply with the ClientKemCiphertext and ClientFinished messages and immediate transmission of encrypted application data.
   KEMTLS-PDK: processing of ServerHello and Finished messages. Reply with the ClientFinished message and immediate transmission of

encrypted application data.

4. ($C \Leftarrow S$) Processing of received messages based on the protocol.
   TLS 1.3 and PQTLS: processing of ClientFinished message and of encrypted application data.
   KEMTLS: processing of ClientKemCiphertext and Finished messages. Reply with the ServerFinished message.
   KEMTLS-PDK: processing of ClientFinished message and encrypted application data.

In mutual authentication, the handshake consists of the following flights:

1. ($C \Rightarrow S$) Sending ClientHello for (PQ)TLS and KEMTLS.

2. ($C \Leftarrow S$) Processing of ClientHello.
   TLS 1.3 and PQTLS: reply with the ServerHello, Certificate, CertificateVerify, and CertificateRequest messages, followed by the Finished message.
   KEMTLS: reply with the ServerHello, the Certificate, and the CertificateRequest messages.

3. ($C \Rightarrow S$) Processing of received messages based on the protocol.
   TLS 1.3 and PQTLS: processing of the ServerHello, Certificate, CertificateVerify, and CertificateRequest messages, followed by the Finished message. Reply with the ClientCertificate, the CertificateVerify, and the Finished messages, and immediate transmission of encrypted application data.
   KEMTLS: processing of the ServerHello, the Certificate, and the CertificateRequest messages. Reply with the ClientKemCiphertext and Certificate messages.

4. ($C \Leftarrow S$) Processing of received messages based on the protocol.
   TLS 1.3 and PQTLS: processing of the received ClientCertificate, CertificateVerify, and Finished messages, and received encrypted application data.
   KEMTLS: processing of ClientKemCiphertext and Certificate messages. Reply with ServerKemCiphertext message.

5. ($C \Rightarrow S$) **KEMTLS** only: Includes processing of the ServerKemCiphertext message and sending of the ClientFinished message. Immediate sending of encrypted application data.

6. ($C \Leftarrow S$) **KEMTLS** only: Includes processing of the ClientFinished message and any application data. Sending of the ServerFinished message.

Table 15.2: Average time in ms of messages for server-only authenticated connections. Note that timings are measured per client and server: each has an independent timer. All experiments use the same intermediate and root CA certificates.

Handshake	KEX	Auth	Handshake flight			
			1	2	3	4
TLS 1.3	X25519	Ed25519	0.227	0.436	123.838	180.202
TLS 1.3+DC	X25519	Ed25519	0.243	0.489	156.954	186.868
TLS 1.3+DC	X25519	Ed448	0.242	0.907	165.395	183.124
PQTLS	Kyber-512	Dilithium3	0.350	0.701	173.814	198.256
PQTLS	SIKEp434	Dilithium4	2.533	4.856	441.732	212.924
KEMTLS	Kyber-512	Kyber-512	0.412	0.217	157.123	187.147
KEMTLS	SIKEp434	SIKEp434	3.058	7.215	352.840	291.592
KEMTLS-PDK	Kyber-512	Kyber-512	0.623	0.327	181.132	189.442
KEMTLS-PDK	SIKEp434	SIKEp434	9.573	12.507	396.818	287.550

15.5 Discussion

As noted above, we initiate two timers in each of our measurements: one for the client (which starts when the ClientHello message is constructed and sent) and one for the server (which starts when the ClientHello message is received). This is why the first and second flights in tables 15.2 and 15.3 have small timings: they do not take into account network latency. Starting from the time of the third flight, which occurs after the client received the

Table 15.3: Average time in ms of messages for mutually authenticated connections. Note that timings are measured per client and server: each has an independent timer. For presentation reasons, we have abbreviated Dilithium, Kyber-512, and SIKEp434. All experiments use the same intermediate and root CA certificates.

Handshake	Kex	Auth	Handshake flight					
			1	2	3	4	5	6
TLS 1.3	X25519	Ed25519	0.113	0.420	111.358	121.349		
TLS 1.3+DC	X25519	Ed25519	0.148	0.546	129.638	178.90		
TLS 1.3+DC	X25519	Ed448	0.154	0.221	137.131	192.283		
PQTLS	Kyber	Dilith. 3	0.125	1.326	231.232	191.187		
PQTLS	SIKE	Dilith. 4	3.324	7.294	459.888	216.077		
KEMTLS	Kyber	Kyber	0.244	0.303	231.752	175.490	375.202	346.308
KEMTLS	SIKE	SIKE	2.450	6.206	431.445	228.414	510.591	436.301

server's response, the impact of network latency can be seen. We also note that encrypted application data is sent already on the 3rd flight of all experiments, except for KEMTLS with mutual authentication (as the client has to wait two flights before it can send application data), which increases the measured times with the application overhead.

When adding delegated credentials to the TLS 1.3 handshake, a peer receiving a delegated credential must validate that it was signed by the appropriate end-entity certificate (which is sent as part of the handshake) and must validate the certificate chain, as well. In our measurements, we observed a short increase in the latency of the flights when DCs are added; but the impact is almost negligible (especially, in the second flight when the DCs are received).

This is not the case when adding either post-quantum signatures or post-quantum KEMs for certain algorithms. The first observable difference appears in the `ClientHello` in both server-only authentication and mutual authentication: this message advertises both classic and post-quantum key-exchange algorithms because this could be the realistic scenario for systems when transitioning to post-quantum cryptography. The timings increase especially when using SIKEp434 as a KEM in both KEMTLS and PQTLS because its decap-

sulation takes on average 6.7 ms (when using the implementation from the CIRCL library). The predominant factor that slows down PQTLS compared to TLS 1.3 is the number of signature validations, but this is similar (when using Kyber-512 and Dilithium3) to using Ed448.

The biggest drawback of using KEMTLS is the number of round-trips that it has to perform, especially for mutual authentication. The KEM cryptographic operations do not seem to heavily impact the connection if the underlying algorithm operations are fast. An ideal scenario for post-quantum cryptography is the use of KEMs for both confidentiality and authentication provided that the number of round trips does not increase, which is the case of KEMTLS-PDK for server authentication. This prediction matches with the results: KEMTLS-PDK with Kyber-512 performs best for server-only authentication.

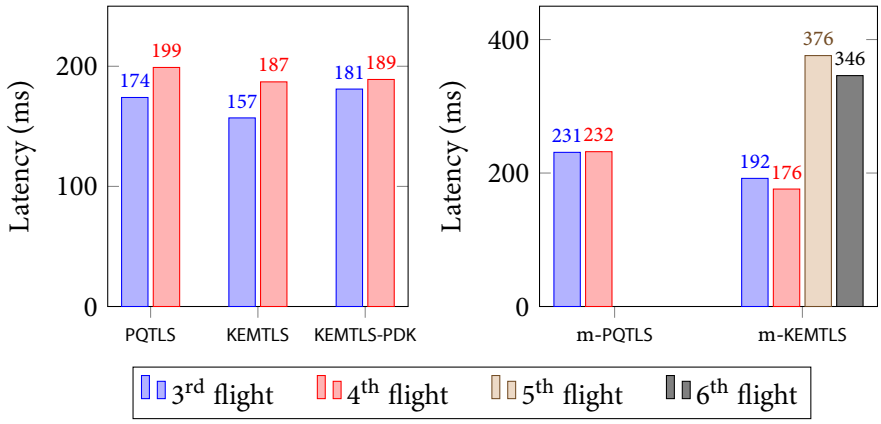


Figure 15.1: Comparison of: on the left, server-authenticated handshakes for the 3rd, and 4th flights; on the right, mutually authenticated handshakes with the additional 5th and 6th flights. Both measurements are of the Kyber-512 instantiations. Note that the timers of client- and server-initiated flights are independent.

We see more differences if we compare completion times for server-only and mutually authenticated handshakes in tables 15.4 and 15.5. In the first case, KEMTLS performs faster than PQTLS and, in both cases, a client can immediately send application data on the third flight (when the client sends its `ClientFinished`). Nevertheless, for KEMTLS, the server still has to wait for the `ClientFinished` to arrive and to send their `ServerFinished` in turn, before it can

Table 15.4: Average total handshake completion time (in ms) for server-only authentication.

Handshake	Key exchange	Authentication	Handshake time	
			Server	Client
TLS 1.3	X25519	Ed25519	187.296	552.518
TLS 1.3+DC	X25519	Ed25519	197.568	578.097
TLS 1.3+DC	X25519	Ed448	220.576	614.366
PQTLS	Kyber-512	Dilithium3	199.025	556.203
PQTLS	SIKEp434	Dilithium4	219.401	634.546
KEMTLS	Kyber-512	Kyber-512	200.237	792.168
KEMTLS	SIKEp434	SIKEp434	277.304	901.292
KEMTLS-PDK	Kyber-512	Kyber-512	209.872	583.582
KEMTLS-PDK	SIKEp434	SIKEp434	200.126	561.068

Table 15.5: Average total handshake completion time (in ms) for mutual authentication.

Handshake	Key exchange	Authentication	Handshake time	
			Server	Client
TLS 1.3	X25519	Ed25519	190.587	592.801
TLS 1.3+DC	X25519	Ed25519	179.653	549.760
TLS 1.3+DC	X25519	Ed448	222.902	541.695
PQTLS	Kyber-512	Dilithium3	191.939	542.599
PQTLS	SIKEp434	Dilithium4	223.470	609.646
KEMTLS	Kyber-512	Kyber-512	352.448	881.928
KEMTLS	SIKEp434	SIKEp434	571.057	1096.708

send application data. As we discussed in section 7.1.4, the `ServerFinished` message completes the handshake and provides full downgrade resilience and forward secrecy for the whole connection. However, this extra half-round trip forces the server to wait before sending application data, which could not be an ideal scenario for all real-world systems. If we look at figure 15.1, we see that the best protocol in terms of latency is KEMTLS. Using KEMTLS-PDK is best, as it allows earlier sending of application data by the server and has stronger notions of the security properties.

For mutual authentication, we see that KEMTLS has the biggest impact on the handshake completion timings. This is the result of the extra RTT that is needed for mutual authentication in KEMTLS. SIKEp434, on average, increases the handshake timings by approximately 10 ms compared with Kyber-512 for the verification of the peer's Certificate in both cases. For this reason, the PQTLS completion time is also slowed down when using SIKEp434 even without the extra round-trip addition. Although we do not provide timings for KEMTLS-PDK with mutual authentication, our timings can provide insight into the cost of the operations and the relevance of the algorithm selection.

15.5.1 Optimizations

The cost of transmitting post-quantum parameters is tangible in our measurements. These costs can be further optimized by using a form of certificate compression [158], although compression will likely mostly reduce the size of certificate metadata rather than the size of (high-entropy) public keys. Suppression of the intermediate certificates [202] is an alternative idea, but it relies on clients being kept up-to-date with the server's intermediates. In either case, the costs of the post-quantum encapsulation, decapsulation, signing, and signature verification operations remain.

15.6 Conclusions

The experiments reported on in this chapter are the first that integrate different post-quantum handshake alternatives to the TLS 1.3 handshake into a real-world system. These results have shown us how post-quantum algorithms can impact the handshake completion time, and, therefore, impact the establishment of real-world connections. In general, on the reliable, high-bandwidth

network that we used, the different post-quantum TLS 1.3 handshake alternatives do not have a handshake completion time that is much different from a regular TLS 1.3 handshake. The only somewhat exception to this is KEMTLS, as the extra half or full round trip that is added does increase the completion time, especially for mutual authentication. For this reason, KEMTLS-PDK should be investigated more, as it could reduce the completion time.

In this chapter, we implemented post-quantum algorithms in native Go. We also adapted the Go TLS library to different handshake configurations and added support for new TLS extensions. We developed a measurement framework that allows performing transatlantic post-quantum TLS 1.3 connections for retrieving random numbers from a Drand network.

We remark that an important piece to achieving cryptographic agility in the transition to post-quantum algorithms is the use of delegated credentials. They allowed us to advertise post-quantum KEMs or post-quantum signatures without generating new certificates or asking certificate authorities to support new algorithms. Even though authentication is only as strong as the weakest link, and our deployment thus does not provide post-quantum secure authentication, this approach enables the investigation of performance characteristics and the discovery of deployment constraints.

In future work, the network characteristics can be further diversified. We measured a reliable, high-bandwidth datacenter-to-datacenter connection; how post-quantum TLS will behave on user-to-server connections remains an open question. We also only investigated a limited number of algorithms and did not implement KEMTLS-PDK with mutual authentication. Finally, the transition of the public-key infrastructure to post-quantum cryptography remains an open question.

16 Measuring the performance of KEMTLS in embedded systems

In this chapter, we compare KEMTLS to TLS 1.3 in an embedded setting. To gain meaningful results, we present implementations of KEMTLS and TLS 1.3 on a Cortex-M4-based platform. These implementations are based on the popular WolfSSL embedded TLS library and hence share a majority of their code. In our experiments, we consider both protocols with the round-3 finalist signature schemes and KEMs in the NIST PQC standardization project, except for Classic McEliece which has too-large public keys. Both protocols are compared in terms of runtime, memory usage, traffic volume, and code size. We show benchmark results from network settings relevant to the Internet of Things, namely low-latency broadband, LTE-M, and Narrowband IoT. These show that in the embedded context, KEMTLS can reduce handshake time by up to 38 %, can lower peak memory consumption, and can save traffic volume compared to TLS 1.3.

16.1 Introduction

In the comparisons between post-quantum TLS 1.3 and KEMTLS in previous chapters, we focused on high-end hardware and high-bandwidth connections. However, TLS is used for more than just protecting web browsing on desktop computers. The Internet of Things (IoT) increasingly interconnects embedded devices over the internet. Especially device-to-cloud communication is an omnipresent IoT use case. New communication protocols like Matter [105] (formerly Connected Home over IP) mark a new trend by using IPv6 and forcing every embedded device to establish individual end-to-end-secure connections. From a security perspective, this makes perfect sense. However, for embedded software developers this poses a challenge. Key establishment, digital signatures, and certificate transmission are already problematic for low-cost, resource-constrained devices. With the advent of post-quantum

cryptography, it will become even more challenging to establish TLS connections from those embedded devices.

There has been some work investigating the performance of post-quantum TLS on embedded devices rather than the large-scale, high-performance computers we discussed in previous chapters. Bürstinghaus-Steinbach, Krauß, Niederhagen, and Schneider have experimented with Kyber and stateless hash-based signature scheme SPHINCS$^+$, integrating it in the TLS 1.2 implementation of mbedTLS [247] and showing the performance on various Arm boards [85]. More recently, Tasopoulos, Li, Fournaris, Zhao, Sakzad, and Steinfeld have evaluated the performance of post-quantum TLS 1.3 on embedded systems [339]. They investigated the performance of the NIST finalist KEMs and the Dilithium and Falcon signature algorithms in WolfSSL's TLS 1.3 implementation.

16.1.1 Contribution

This chapter investigates if KEMTLS' advantages transfer to the embedded realm, by comparing KEMTLS and post-quantum TLS 1.3 (PQTLS) in an embedded setting. For this purpose, KEMTLS and PQTLS were implemented including all NIST finalist signature schemes and KEMs, except for Classic McEliece which has too-large public keys. As the PQTLS and KEMTLS implementations share large parts of their code base, we can directly compare their performance. Our analysis focuses on the relevant trade-offs embedded systems engineers face. We benchmark runtime, memory usage, code size, and bandwidth consumption of our KEMTLS and PQTLS instantiations. The benchmark results were obtained by running our implementations on a Cortex-M4-based platform. Our experiments were conducted with a technology stack that is typically used in real-world deployments, in which the embedded device is a TLS client talking to a TLS server running on a high-end computer. This computer also simulated different network technologies throughout the experiments. We briefly introduce post-quantum cryptography, specifically in the context of embedded devices. To support our results, the implementation and experimental setup will then be explained in detail. Finally, we present and discuss the results of our measurements.

Table 16.1: Comparison of NIST PQC round-3 finalists at NIST PQC security level I. We show the size (in bytes) of data transmitted during a protocol exchange, offline data, and operation timings (from [102, 204]) on ARM Cortex M4.

	bytes transmitted			**stored**	**computation (≈Kcycles)**		
Signatures	pk	sig	sum	secret	keygen	sign	verify
Dilithium ★	1312	2420	3732	2528	1597	4095	1572
Falcon ★	897	690	1587	1281	163 994	39 014	473
Rainbow †	161 600	66	161 666	103 648	94	907	238
KEMs	pk	ct	sum	secret	keygen	encaps	decaps
Kyber ★	800	768	1568	1632	440	539	490
NTRU †	699	699	1398	953	2867	565	538
SABER †	672	736	1408	1568	352	481	453

★: Scheme was selected for standardization.
†: Scheme was eliminated from the NIST PQC standardization project.

16.2 Post-quantum cryptography on embedded devices

Public-key cryptography was already challenging for embedded systems in a pre-quantum setting. The more expensive post-quantum algorithms will make this worse. To gain a better understanding of PQC algorithm performance on embedded systems, the pqm4 [204] project collects implementations for the Cortex-M4 platform and benchmarks them. Table 16.1 shows size and performance trade-offs between the NIST PQC round-3 finalists based on numbers from [204] and [102]. Here it is important to mention that these numbers are accomplished on a clocked-down Cortex-M4. Using such a slowed-down embedded processor is customary for measuring algorithm run times because it avoids flash wait states. In a real deployment, code would be fetched from a fast ROM instead of a flash. For our experiments, we are not exclusively interested in PQC algorithm runtime, but in the performance of the overall system. Therefore, we do not clock down our CPU. The ramifications of this are detailed in section 16.3.2.

16.3 Experimental setup

The following section describes the experimental setup used to acquire our results. Both protocols were benchmarked for handshake times, run-times of algorithms, peak memory usage, code size, and network traffic. Handshake times were measured in three network environments relevant to the IoT domain. This includes regular "broadband" internet, as well as two low-power wide-area network standards, LTE-Machine Type Communication (LTE-M) and Narrowband-IoT (NB-IoT), developed by the 3rd Generation network Partnership Project (3GPP). We give the characteristics employed for these environments in table 16.2. While the performance characteristics of LTE-M and NB-IoT are based on numbers of the 3GPP [1], the broadband scenario is based on realistic RTTs of client-to-cloud communication within Western Europe using a consumer-grade connection [288].

Table 16.2: Connection characteristics

Name	Abbrev.	Bandwidth	RTT time
Broadband [288]	BB	1 Mbit	26 ms
LTE Machine Type Communication [1]	LTE-M	1 Mbit	120 ms
Narrowband-IoT [1]	NB-IoT	46 kbit	3 s

16.3.1 Cryptographic primitives

As KEMTLS is a post-quantum protocol, it is not specifically designed for transitional security. Although KEMTLS does not preclude their use, we do not consider mixed classic/post-quantum certificates or hybrid (post-quantum plus elliptic-curve) key-exchange methods in our experiments. For comparability, our PQTLS implementation is also exclusively using post-quantum algorithms. We evaluated all combinations of NIST PQC round-3 finalists, except for the KEM Classic McEliece. Classic McEliece's public keys are too large to fit into memory and do not fit in the `ClientHello`'s `KeyShareEntry` extension [298, Sec. 4.2.8].

Both KEMTLS and PQTLS make use of a CA that signs certificates. The CA's certificate, containing the CA's public key used for signature verification, is stored on the client device. Only leaf certificates, transmitted by the server

during the handshake, differ in KEMTLS and TLS 1.3. For PQTLS they include the public key of a signature algorithm, in KEMTLS a KEM public key. In our experiments, we omit intermediate certificates.

We only evaluate primitives at the lowest security level, NIST level I. These are the smallest and most efficient parameter sets.

16.3.2 Implementation

All benchmarks were conducted on a Silicon Labs STK3701A board, also known as the "Giant Gecko". This board was chosen because it features a 72 MHz ARM Cortex-M4F embedded processor and offers large enough memory (2 MB flash storage, 512 kB SRAM) to fit Rainbow public keys. As Cortex-M4 is the designated NIST PQC reference platform for embedded devices, there are optimized assembly implementations available for most finalist algorithms. The pqm4 project collects these implementations and provides extensive benchmarks [204]. All PQC implementations used for benchmarking were taken from the pqm4 project. Only minor modifications, such as adding *verify* functions to signature schemes, fixing alignment issues and name-spacing symbol names had to be conducted. The code was compiled using GCC version 11.1, with the -O3 speed optimization flag. In contrast to experiments run within the pqm4 project, we do not clock down the processor to avoid wait states. Instead, the processor runs at full speed. This makes sense since we are not exclusively interested in the run times of the primitives but in the performance of the overall system. Running the processor at full speed makes the PQC algorithms consume more cycles due to flash wait states and higher costs of memory accesses. However, since the PQC algorithms do not consume more wall-clock time, the actual handshake durations are not negatively affected. The Giant Gecko board exclusively takes on the role of an embedded (KEM)TLS client, wanting to connect to a backend server. To validate certificates send in the handshake, we flash the CA's root certificate into the Giant Gecko's persistent memory during setup. For efficiency, the CA directly signs the server's certificate. This avoids the need for transmitting intermediate CA certificates, reducing the size of the certificate chain. As both endpoints in embedded scenarios are usually under some level of manufacturer control, this is a common deployment. Communication to the backend server is done via the Giant Gecko's Ethernet port, which is directly connected to a high-end computer. This host computer

simulates different network environments by using Linux's `netem` network emulation framework [174]. The network emulation framework is set up to throttle bandwidth and delay RTTs according to the aforementioned network environments. The KEMTLS implementation describe in chapter 10 and used for the experiments in chapter 13 is used as server software. When running an iteration of the experiment, the corresponding PQC algorithms and the CA certificate are linked into the binary using Zephyr's *West* build tool. We then flash the binary onto the board via JLink. Benchmark results are received via serial communication.

16.3.3 Platform

To have a realistic setup, we employed a typical embedded systems software stack. In our case that includes an embedded real-time operating system (RTOS) with an open-source TCP/IP stack and added TLS support. For reproducibility, we used the Apache-licensed Zephyr RTOS [361]. Zephyr supports over 200 boards and is backed by the Linux Foundation and multiple large corporations involved with developing embedded systems, such as NXP, NORDIC, and Memfault. It provides its own optimized embedded network stack and allows cycle-accurate runtime measurements (given a board's hardware supports it). Our application code runs as the exclusive Zephyr thread, eliminating scheduling costs. We added PQTLS and KEMTLS support to the operating system via a custom WolfSSL module. KEMTLS certificate generation were generated by the script described in section 10.7. Post-quantum certificates for PQTLS were generated using a fork of OpenSSL's command-line tool maintained by the Open Quantum Safe project [345]. The TLS 1.3 cipher suite `TLS_CHACHA20_POLY1305_SHA256` was used in all experiments.

16.3.4 WolfSSL integration

Previous work [85, 339] also uses WolfSSL for running benchmarks on embedded systems. We decided to use WolfSSL for the same reasons as the mentioned works and to make comparisons with our results easier. WolfSSL is designed to be memory efficient and fast on embedded systems. It already supports TLS 1.3 and has a clean implementation of TLS's state machine. This makes it an ideal basis for implementing PQTLS and KEMTLS. Adding post-quantum algorithms to WolfSSL is straightforward. WolfSSL's crypto

provider, called WolfCrypt, has a clean API that can be extended easily. As the KEM Kyber was already included in WolfSSL by Bürstinghaus-Steinbach, Krauß, Niederhagen, and Schneider [85], we did not need to make changes to the TLS 1.3 state machine. Apart from including the relevant ASN.1 object identifiers for KEMs and post-quantum signatures, only small changes, such as increasing the maximum size of certificates, had to be applied. Our embedded KEMTLS implementation is based on the same WolfSSL version as our PQTLS implementation. The majority of the code is identical in the PQTLS and KEMTLS implementation. However, adding support for KEMTLS to WolfSSL still required significant effort. Apart from altering the certificate/ASN.1 parser to allow KEM keys in certificates (and using those), WolfSSL's internal state machine, key derivations, and state structures had to be modified. In both our PQTLS and KEMTLS experiments, the client only performs signature verification, so no code for signing was linked into the final binary.

16.4 Results

For developers of embedded systems, the trade-offs between ROM (code size), RAM (memory usage), network traffic, and CPU time (runtime of code) are most crucial. In this section, we present our findings regarding the consumption of these resources by KEMTLS and PQTLS using NIST PQC round-3 finalists.

The runtime of algorithms impacts the device's energy consumption. This is especially relevant for battery-powered devices that rely on the possibility to hibernate when inactive. Network traffic also affects energy consumption, as operating an antenna is usually a very energy-consuming operation. Depending on the underlying wireless technology, network traffic can also be expensive in terms of network provider fees. Our results are representative of Cortex-M4-based platforms in general. Hence we focus on benchmarks that are independent of our specific evaluation board. As energy consumption varies heavily based on a board's design, choice of peripherals, and transmission technology we did not include direct energy measurements in our results. Instead, we present code size, consumed memory, handshake traffic, handshake duration, and runtime of PQC primitives. All KEMTLS and PQTLS instantiations were run 1000 times, with each run using a different CA and leaf certificate. The presented benchmarks are averaged over all runs. NIST PQC

signature algorithm finalist Rainbow, which is included as a representative for multivariate-based cryptography, is only present in the KEMTLS results. This is because Rainbow public keys are very large: there was not enough memory to fit Rainbow along another signature scheme, and Rainbow public keys would have a disproportionate impact if used in leaf certificates. It could therefore not be included in the PQTLS benchmarks. We emphasize that all employed PQC algorithms were optimized for speed, and not stack consumption.

16.4.1 Storage and memory consumption

Both protocol implementations are roughly the same size. Excluding the implementations of the post-quantum primitives, the code size is around 111 kB. table 16.3 shows combinations of PQC algorithms with their measured code size. For KEMTLS, only instantiations with one KEM used for both ephemeral key exchange and authentication are shown. Including two KEMs does not give an advantage, but increases code size. However, for completeness, a table with all combinations can be found in section 16.7. Similarly, PQTLS instantiations with the same signature algorithm used for CA and leaf certificates are shown. Additionally, we include the combination of Dilithium and Falcon, where Dilithium is used as the handshake signature algorithm. This combination was suggested by Sikeridis, Kampanakis, and Devetsikiotis to make use of Dilithium's faster signing times for servers without hardware support for Falcon's double-precision floating-point operations [326].

The table also shows the PQC code's share of the overall code size as a percentage. Also included in table 16.3 is memory consumption. Shown is the peak of consumed memory, in both heap and stack, during the handshake. This includes the memory consumed by the protocol implementation and PQC primitives.

In contrast to PQTLS, KEMTLS uses a KEM encapsulation instead of a signature verification to authenticate the connection. KEMTLS, therefore, needs code for KEM encapsulation, whereas PQTLS does not. PQTLS on the other hand needs the code for two distinct verification algorithms if different signature algorithms are used for CA and leaf certificates. Instantiations with NTRU ephemeral key exchange are notable outliers in terms of code size, requiring over 200 kB of code. This is in line with results reported by PQM4 [204]. Interestingly, this big increase in code size cannot be observed when NTRU is used exclusively for authentication. This is because the client requires key

generation and decapsulation code for ephemeral key exchange, whereas authentication via KEM only requires encapsulation functionality. (Note that due to the Fujisaki–Okamoto transform used to obtain IND-CCA security in NTRU, the decapsulation code contains the encapsulation code as well). Whenever Rainbow is used, the CA certificate containing a Rainbow public key takes up between 33 % and 53 % of the overall consumed storage space. This, however, does not disqualify Rainbow from usage on embedded systems, due to its small signature and very fast verification times (see section 16.4.2).

Table 16.3: Code and CA certificate sizes (and as a percentage of total ROM size), and peak memory usage in the experiments. Parameter sets used are NIST level I.

	KEX	Auth.	CA	PQC code	(%)	CA size	(%)	Memory
KEMTLS	Kyber	Kyber	Dilithium	29.0 kB	(20.1%)	3.9 kB	(2.7%)	49.7 kB
	Kyber	Kyber	Falcon	25.7 kB	(18.6%)	1.7 kB	(1.2%)	52.8 kB
	Kyber	Kyber	Rainbow	29.8 kB	(9.8%)	161.8 kB	(53.4%)	167.0 kB
	NTRU	NTRU	Dilithium	203.4 kB	(63.9%)	3.9 kB	(1.2%)	49.7 kB
	NTRU	NTRU	Falcon	200.0 kB	(63.9%)	1.7 kB	(0.6%)	52.8 kB
	NTRU	NTRU	Rainbow	204.0 kB	(42.8%)	161.8 kB	(33.9%)	182.9 kB
	SABER	SABER	Dilithium	31.5 kB	(21.5%)	3.9 kB	(2.7%)	49.7 kB
	SABER	SABER	Falcon	28.2 kB	(20.0%)	1.7 kB	(1.2%)	52.8 kB
	SABER	SABER	Rainbow	32.2 kB	(10.5%)	161.8 kB	(53.0%)	167.9 kB
PQTLS	Kyber	Dilithium	Dilithium	29.0 kB	(20.1%)	4.0 kB	(2.8%)	58.0 kB
	Kyber	Dilithium	Falcon	34.4 kB	(23.3%)	1.8 kB	(1.2%)	60.0 kB
	Kyber	Falcon	Falcon	25.8 kB	(18.6%)	1.8 kB	(1.3%)	56.2 kB
	NTRU	Dilithium	Dilithium	203.4 kB	(63.8%)	4.0 kB	(1.3%)	56.6 kB
	NTRU	Dilithium	Falcon	208.7 kB	(64.9%)	1.8 kB	(0.6%)	58.6 kB
	NTRU	Falcon	Falcon	200.1 kB	(63.9%)	1.8 kB	(0.6%)	54.8 kB
	SABER	Dilithium	Dilithium	31.5 kB	(21.5%)	4.0 kB	(2.7%)	58.0 kB
	SABER	Dilithium	Falcon	36.8 kB	(24.6%)	1.8 kB	(1.2%)	60.0 kB
	SABER	Falcon	Falcon	28.2 kB	(20.0%)	1.8 kB	(1.3%)	56.2 kB

Further, the results show that the lattice-based schemes perform well in terms of memory consumption. The consumed memory is mainly driven by stack usage of the PQC signature algorithms. Only Rainbow is an exception here. With a Rainbow-powered CA certificate, the very large public key

has to be loaded into memory and held during signature verification. This requires a large allocation of heap space. Using a custom certificate loader implementation it would be possible to store the public key in an already usable form in flash. Then the public key could directly be streamed in from flash (similar to [161]), without the need to hold it in memory entirely. However, since we present comparable results of reusable code, we did not include this kind of optimization for an individual algorithm.

16.4.2 Handshake times

Apart from storage and memory consumption, handshake times are key in an embedded environment. Table 16.4 shows handshake times for different transmission technologies measured in millions of cycles. A complete table, with all possible instantiations, can be found in section 16.7. In figure 16.1 we show the handshake times and traffic for the broadband and NB-IoT scenarios. In a real deployment, the device would likely go into a low-power mode or sleep instead of actively polling data during a slow transmission. This behavior however depends highly on the specifics of the embedded system and its transmission technology. Therefore, to achieve reproducible results, the CPU was running at a constant speed of 72 MHz during all experiments. This also makes a direct translation to wall time possible. The table also shows the percentage of cycles spend on the underlying PQC primitives. The remaining cycles are spent in the TLS state machine, memory operations, or waiting for I/O.

Time spent in crypto operations is significant in the broadband and LTE-M setting. Whereas the NB-IoT transmission is so slow, that the share of cycles spent in cryptographic operations is very low (0.8 %–1.7 %). In low-bandwidth/high-RTT settings like NB-IoT, the transmission size of certificates and public keys is the main driving factor of runtime. Loading large public keys from storage into memory is a relevant factor as well, slowing down the otherwise fast Rainbow signature algorithm. Cycles spent to access memory and storage also become increasingly negligible when using slow transportation mediums. This is visible in figure 16.1b, where the instantiations with similarly sized handshake traffic appear in clusters.

Both PQTLS and KEMTLS use a KEM for key exchange. While the performance of the module lattice KEMs Kyber and SABER is similar, they both outperform NTRU for this task. This is mainly due to the rather slow key

Table 16.4: TLS handshake traffic and runtime for various scenarios. Parameter sets used are NIST level I.

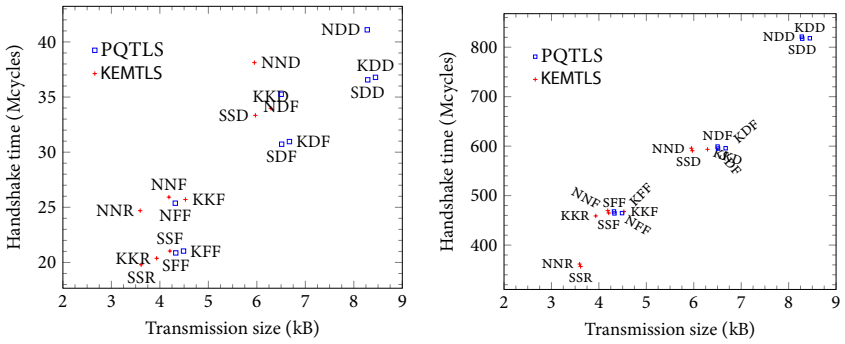
	KEX	Auth.	CA	Handshake traffic	Handshake time in Mcycles (% of crypto) BB (%)	LTE-M (%)	NB-IoT (%)
KEMTLS	Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
	Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
	Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
	NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
	NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
	NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
	SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
	SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
	SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)
PQTLS	Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
	Kyber	Dilithium	Falcon	6.7 kB	14.7 (35.4%)	31.0 (16.8%)	595.8 (0.9%)
	Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
	NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
	NTRU	Dilithium	Falcon	6.5 kB	19.0 (50.3%)	35.3 (27.2%)	599.2 (1.6%)
	NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
	SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
	SABER	Dilithium	Falcon	6.5 kB	14.5 (34.2%)	30.7 (16.2%)	595.2 (0.8%)
	SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)

generation of NTRU increasing handshake time. Slow key generation is also the reason why PQTLS and KEMTLS instantiations using NTRU have the highest percentage of cycles spent in PQC operations.

All KEMs outperform Dilithium when used for authentication. This makes sense as Dilithium's verification routine is slower than the encapsulation routine of all investigated KEMs. Dilithium's performance also suffers from its large public key and signature, which increase the required transmission size. In slow, bandwidth-constrained network environments, such as NB-IoT, this drawback becomes even more apparent. Rainbow performs well in terms of handshake times when used as a CA certificate. Not only because it has a fast, bitsliced Cortex-M4 implementation. Since the large Rainbow public key is stored on the client device, only the small signature has to be transmitted during the handshake. Rainbow's small signature and fast runtime make it a good fit for CA certificates if the storage and memory demands can be afforded. The instantiations with Rainbow offer the fastest KEMTLS handshake times throughout all transmission mediums. Additionally, the shortest NB-IoT handshake times use KEMTLS with Rainbow and SABER. Falcon on the other hand performs very well on the Cortex-M4 platform in our experiments. In terms of runtime, it even outperforms KEMs for server authentication. However, this is only true for the client side. Signing operations using Falcon are considerably more expensive than KEM decapsulation. But these operations are conducted on the server side, increasing server load, which is not part of our measurements. Additionally, Falcon's public key and signature sizes are comparable to the sizes of the KEM's public keys and ciphertexts. So it is not surprising that PQTLS instantiations using Falcon perform well. In the broadband and LTE-M setting, PQTLS with Falcon and SABER performs as well as KEMTLS with Rainbow and SABER.

16.5 Discussion

Our results show that KEMTLS with server-only authentication uses less memory than PQTLS and has similar code sizes. Due to Falcon's verification algorithm being very efficient, in terms of bandwidth and computation time, PQTLS with Falcon performs as well as or better than any KEMTLS instantiation. The only exceptions are the KEMTLS instantiations using SABER or NTRU with Rainbow, where the ability of KEMTLS to use Rainbow due to lower memory



(a) Broadband (b) Narrowband-IoT

Figure 16.1: Handshake times and traffic for instantiations of KEMTLS and PQTLS. Letters represent the algorithms Dilithium, Falcon, Kyber, NTRU, Rainbow, and SABER in the roles of ephemeral key exchange, handshake authentication, and CA, in that order.

usage saves a few bytes. Thus, these instances become the best-performing in the NB-IoT scenario. Falcon also performs better than Dilithium on the client side, in any scenario.

Although we have not measured client authentication or an embedded server, we can extrapolate from our results and other work. As reported by pqm4 [204] and Sikeridis, Kampanakis, and Devetsikiotis [326], Falcon's signing algorithm, especially without hardware support, is significantly more costly than the Dilithium signing algorithm or any of the KEM operations. This suggests that Falcon is perhaps not generically suitable for post-quantum authentication.

Sikeridis, Kampanakis, and Devetsikiotis also suggested using a combination of Dilithium and Falcon for PQTLS, in scenarios where there is no hardware support for Falcon's constant-time double-precision floating-point operations in signing [326]. Dilithium would be put in the leaf certificate, to make use of its efficient signing times for online handshake signatures. Falcon's smaller public key and signature sizes would be beneficial for the CA certificate algorithm, which signs the leaf certificate only once, but the signature is transmitted many times. However, our results show that for embedded clients that only need to do signature validation Falcon is preferable over Dilithium, especially in very low bandwidth scenarios like NB-IoT.

16.6 Conclusion and future work

In this chapter, we compared the performance of KEMTLS and TLS 1.3 using NIST PQC round-3 finalists in an embedded environment. This environment was represented by a Cortex-M4-based client communicating with a desktop-class server. We showed that a KEMTLS client consumes less memory than TLS 1.3, due to the smaller memory footprint of KEMs. The code size did not differ between KEMTLS and TLS 1.3. Since only server authentication was used, both protocols require a signature verify function and KEM for key exchange. Our run times show that in both protocols PQC primitives require a significant amount of computational time during the handshake, sometimes requiring over 50 % of the entire handshake time. Even in the LTE-M setting, the percentage of cycles spent in PQC computations is considerable. However, in the bandwidth-constrained NB-IoT setting, handshake times are mostly driven by handshake size. In these conditions, Rainbow's very small signatures are an advantage. While Dilithium is generally outperformed by KEMs when used for authentication, Falcon performs very well due to its efficient verification algorithm. However, signing in Falcon is a very costly operation. Future work should therefore investigate KEMTLS and TLS 1.3 using client authentication, and embedded KEMTLS and post-quantum TLS 1.3 servers. In both of these applications, the embedded TLS 1.3 client needs to produce handshake signatures. This would increase the cost of using signatures instead of KEMs significantly, leading to new trade-offs. Another avenue of research is the pre-distributed key setting, where the client already knows the server's public key. In this setting, bandwidth can be reduced even further, which may be compelling for the NB-IoT application.

Finally, we would like to repeat that we used implementations that were optimized for runtime, not code size, and only were protected against side-channel attacks based on execution time. As of writing, the discussion on the effects of protecting the post-quantum schemes against different types of side-channel and fault attacks is actively ongoing, with new papers appearing frequently. For applications sensitive to those kinds of attacks, it remains an open question as to how our results will scale to the cost of the relevant protections (although transmission sizes, and the reported effects, should remain the same).

16.7 Appendix: Extended benchmark results

In tables 16.5 and 16.7 we report code sizes, CA certificate sizes, and memory usage for all experiments we ran. Tables 16.6 and 16.8 provides all results for the handshake traffic and handshake timing metrics.

Table 16.5: Code and CA certificate sizes (and as a percentage of total ROM size), and peak memory usage in the experiments: PQTLS experiments.

KEX	Auth.	CA	PQC code (%)	CA size (%)	Memory
Kyber	Dilithium	Dilithium	29.0 kB (20.1%)	4.0 kB (2.8%)	58.0 kB
Kyber	Dilithium	Falcon	34.4 kB (23.3%)	1.8 kB (1.2%)	60.0 kB
Kyber	Falcon	Dilithium	34.4 kB (23.0%)	4.0 kB (2.7%)	60.7 kB
Kyber	Falcon	Falcon	25.8 kB (18.6%)	1.8 kB (1.3%)	56.2 kB
NTRU	Dilithium	Dilithium	203.4 kB (63.8%)	4.0 kB (1.3%)	56.6 kB
NTRU	Dilithium	Falcon	208.7 kB (64.9%)	1.8 kB (0.6%)	58.6 kB
NTRU	Falcon	Dilithium	208.7 kB (64.4%)	4.0 kB (1.2%)	59.3 kB
NTRU	Falcon	Falcon	200.1 kB (63.9%)	1.8 kB (0.6%)	54.8 kB
SABER	Dilithium	Dilithium	31.5 kB (21.5%)	4.0 kB (2.7%)	58.0 kB
SABER	Dilithium	Falcon	36.8 kB (24.6%)	1.8 kB (1.2%)	60.0 kB
SABER	Falcon	Dilithium	36.8 kB (24.2%)	4.0 kB (2.6%)	60.7 kB
SABER	Falcon	Falcon	28.2 kB (20.0%)	1.8 kB (1.3%)	56.2 kB

Table 16.6: TLS handshake traffic and runtime for various PQTLS scenarios

PQTLS			Handshake	Handshake time in Mcycles (% of crypto)		
KEX	Auth.	CA	traffic	BB (%)	LTE-M (%)	NB-IoT (%)
Kyber	Dilithium	Dilithium	8.4 kB	19.9 (35.9%)	36.8 (19.5%)	818.1 (0.9%)
Kyber	Dilithium	Falcon	6.7 kB	14.7 (35.4%)	31.0 (16.8%)	595.8 (0.9%)
Kyber	Falcon	Dilithium	6.3 kB	15.5 (33.0%)	29.0 (17.6%)	586.4 (0.9%)
Kyber	Falcon	Falcon	4.5 kB	10.9 (30.1%)	21.0 (15.6%)	464.6 (0.7%)
NTRU	Dilithium	Dilithium	8.3 kB	24.3 (47.6%)	41.1 (28.1%)	821.3 (1.4%)
NTRU	Dilithium	Falcon	6.5 kB	19.0 (50.3%)	35.3 (27.2%)	599.2 (1.6%)
NTRU	Falcon	Dilithium	6.1 kB	19.9 (47.8%)	33.4 (28.5%)	590.6 (1.6%)
NTRU	Falcon	Falcon	4.3 kB	15.2 (50.3%)	25.4 (30.2%)	468.0 (1.6%)
SABER	Dilithium	Dilithium	8.3 kB	19.7 (35.2%)	36.6 (19.0%)	817.3 (0.8%)
SABER	Dilithium	Falcon	6.5 kB	14.5 (34.2%)	30.7 (16.2%)	595.2 (0.8%)
SABER	Falcon	Dilithium	6.1 kB	15.3 (32.0%)	28.8 (17.0%)	586.2 (0.8%)
SABER	Falcon	Falcon	4.3 kB	10.7 (28.5%)	20.9 (14.6%)	464.0 (0.7%)

Table 16.7: Code and CA certificate sizes (and as a percentage of total ROM size), and peak memory usage in the experiments: KEMTLS experiments.

KEX	Auth.	CA	PQC code	(%)	CA size	(%)	Memory
Kyber	Kyber	Dilithium	29.0 kB	(20.1%)	3.9 kB	(2.7%)	49.7 kB
Kyber	Kyber	Falcon	25.7 kB	(18.6%)	1.7 kB	(1.2%)	52.8 kB
Kyber	Kyber	Rainbow	29.8 kB	(9.8%)	161.8 kB	(53.4%)	167.0 kB
Kyber	NTRU	Dilithium	41.0 kB	(26.3%)	3.9 kB	(2.5%)	49.7 kB
Kyber	NTRU	Falcon	37.7 kB	(25.0%)	1.7 kB	(1.1%)	52.8 kB
Kyber	NTRU	Rainbow	41.7 kB	(13.3%)	161.8 kB	(51.4%)	182.9 kB
Kyber	SABER	Dilithium	44.9 kB	(28.1%)	3.9 kB	(2.4%)	49.7 kB
Kyber	SABER	Falcon	41.7 kB	(26.9%)	1.7 kB	(1.1%)	52.8 kB
Kyber	SABER	Rainbow	45.7 kB	(14.3%)	161.8 kB	(50.8%)	167.9 kB
NTRU	Kyber	Dilithium	216.3 kB	(65.3%)	3.9 kB	(1.2%)	49.7 kB
NTRU	Kyber	Falcon	213.0 kB	(65.4%)	1.7 kB	(0.5%)	52.8 kB
NTRU	Kyber	Rainbow	217.1 kB	(44.3%)	161.8 kB	(33.0%)	182.9 kB
NTRU	NTRU	Dilithium	203.4 kB	(63.9%)	3.9 kB	(1.2%)	49.7 kB
NTRU	NTRU	Falcon	200.0 kB	(63.9%)	1.7 kB	(0.6%)	52.8 kB
NTRU	NTRU	Rainbow	204.0 kB	(42.8%)	161.8 kB	(33.9%)	182.9 kB
NTRU	SABER	Dilithium	219.7 kB	(65.6%)	3.9 kB	(1.2%)	49.7 kB
NTRU	SABER	Falcon	216.4 kB	(65.7%)	1.7 kB	(0.5%)	52.8 kB
NTRU	SABER	Rainbow	220.4 kB	(44.7%)	161.8 kB	(32.8%)	182.9 kB
SABER	Kyber	Dilithium	44.5 kB	(27.9%)	3.9 kB	(2.4%)	49.7 kB
SABER	Kyber	Falcon	41.3 kB	(26.8%)	1.7 kB	(1.1%)	52.8 kB
SABER	Kyber	Rainbow	45.3 kB	(14.2%)	161.8 kB	(50.8%)	167.9 kB
SABER	NTRU	Dilithium	43.9 kB	(27.6%)	3.9 kB	(2.5%)	49.7 kB
SABER	NTRU	Falcon	40.6 kB	(26.4%)	1.7 kB	(1.1%)	52.8 kB
SABER	NTRU	Rainbow	44.6 kB	(14.0%)	161.8 kB	(50.9%)	182.9 kB
SABER	SABER	Dilithium	31.5 kB	(21.5%)	3.9 kB	(2.7%)	49.7 kB
SABER	SABER	Falcon	28.2 kB	(20.0%)	1.7 kB	(1.2%)	52.8 kB
SABER	SABER	Rainbow	32.2 kB	(10.5%)	161.8 kB	(53.0%)	167.9 kB

Table 16.8: TLS handshake traffic and runtime for various KEMTLS scenarios

KEMTLS			Handshake	Handshake time in Mcycles (% of crypto)		
KEX	Auth.	CA	traffic	BB (%)	LTE-M (%)	NB-IoT (%)
Kyber	Kyber	Dilithium	6.3 kB	17.1 (30.2%)	34.0 (15.2%)	593.6 (0.9%)
Kyber	Kyber	Falcon	4.5 kB	12.3 (27.2%)	25.7 (13.0%)	467.8 (0.7%)
Kyber	Kyber	Rainbow	3.9 kB	11.3 (25.1%)	20.4 (13.9%)	459.0 (0.6%)
Kyber	NTRU	Dilithium	6.1 kB	17.1 (31.5%)	34.1 (15.8%)	592.2 (0.9%)
Kyber	NTRU	Falcon	4.4 kB	12.4 (28.8%)	21.7 (16.4%)	466.2 (0.8%)
Kyber	NTRU	Rainbow	3.8 kB	11.4 (27.0%)	20.5 (15.0%)	358.1 (0.9%)
Kyber	SABER	Dilithium	6.1 kB	16.8 (30.0%)	33.6 (15.0%)	591.5 (0.8%)
Kyber	SABER	Falcon	4.4 kB	12.0 (26.6%)	21.5 (14.9%)	465.4 (0.7%)
Kyber	SABER	Rainbow	3.8 kB	11.0 (24.5%)	20.2 (13.4%)	357.4 (0.8%)
NTRU	Kyber	Dilithium	6.1 kB	21.3 (44.8%)	38.2 (25.0%)	596.9 (1.6%)
NTRU	Kyber	Falcon	4.4 kB	16.6 (46.4%)	25.9 (29.7%)	470.8 (1.6%)
NTRU	Kyber	Rainbow	3.8 kB	15.5 (46.3%)	24.7 (29.1%)	462.4 (1.6%)
NTRU	NTRU	Dilithium	6.0 kB	21.3 (46.0%)	38.1 (25.6%)	595.8 (1.6%)
NTRU	NTRU	Falcon	4.2 kB	16.6 (47.8%)	25.9 (30.6%)	469.7 (1.7%)
NTRU	NTRU	Rainbow	3.6 kB	15.7 (47.4%)	24.7 (30.1%)	361.6 (2.1%)
NTRU	SABER	Dilithium	6.0 kB	20.8 (45.1%)	37.7 (24.9%)	594.9 (1.6%)
NTRU	SABER	Falcon	4.2 kB	16.2 (46.6%)	25.6 (29.5%)	468.9 (1.6%)
NTRU	SABER	Rainbow	3.6 kB	15.3 (46.3%)	24.3 (29.1%)	360.9 (2.0%)
SABER	Kyber	Dilithium	6.1 kB	16.8 (29.4%)	33.6 (14.7%)	593.0 (0.8%)
SABER	Kyber	Falcon	4.4 kB	11.9 (26.1%)	22.7 (13.7%)	466.8 (0.7%)
SABER	Kyber	Rainbow	3.8 kB	11.0 (23.7%)	20.2 (12.8%)	458.3 (0.6%)
SABER	NTRU	Dilithium	6.0 kB	16.8 (30.8%)	33.7 (15.3%)	591.5 (0.9%)
SABER	NTRU	Falcon	4.2 kB	12.0 (27.9%)	21.5 (15.5%)	465.6 (0.7%)
SABER	NTRU	Rainbow	3.6 kB	11.0 (25.8%)	20.2 (14.1%)	357.6 (0.8%)
SABER	SABER	Dilithium	6.0 kB	16.3 (29.4%)	33.3 (14.4%)	590.8 (0.8%)
SABER	SABER	Falcon	4.2 kB	11.6 (25.5%)	21.0 (14.1%)	464.8 (0.6%)
SABER	SABER	Rainbow	3.6 kB	10.7 (23.1%)	19.8 (12.5%)	356.8 (0.7%)

17 Improving software quality in standardization projects

To facilitate the experiments we reported on in the preceding chapters, we integrated the reference and accelerated implementations of most of the post-quantum KEMs and signature schemes in the NIST PQC standardization project. Unfortunately, most of the implementations, which were written in the C programming language, had quality problems. Additionally, the calling interface that NIST defined for interacting with the schemes was not suitable for experimentation in TLS. We set up the *PQClean* community effort, in which we collected both cleaned-up reference and platform-specific implementations of the competitors in the NIST project. In this chapter, we summarize some of the lessons we learned in this effort: by applying standard practices from the software engineering community, we think NIST could have saved themselves, as well as the cryptographic community experimenting with post-quantum schemes, a lot of time and effort.

17.1 Introduction

The selection of cryptographic algorithms for use in applications and standards is increasingly accomplished via public competitions, in which researchers are invited to submit algorithms that are then subject to public review. One significant case study of such a process is the PQC standardization project of NIST. In 2016, NIST announced its intention to standardize quantum-resistant digital signatures and public-key encryption / KEMs in a multi-year public process that continues as of March 2022. This PQC standardization project was to be modeled after NIST's earlier public competitions[1] that lead to the Advanced Encryption Standard (1997–2001, 15 submissions

[1] The PQC standardization process was not explicitly called a "competition" as it might result in several winners.

total, 2 rounds) and the Secure Hash Algorithm (SHA-3) (2007–2015, 51 submission total, 2 rounds).

The PQC standardization project is NIST's largest cryptography standardization effort to date. There were 82 initial submissions of which 69 have been accepted as "complete and proper" submissions. By now, those have been winnowed down over three rounds of public evaluation to 7 finalists and 8 alternate candidates. The first algorithms were selected for standardization in July 2022; a fourth round for 4 KEMs meriting further investigation and an "on-ramp" for new signature scheme designs were also announced.

Compared with the AES and SHA-3 competitions, the PQC standardization project is more complex in several ways, beyond the sheer number of submissions and rounds. The PQC standardization project involves two distinct cryptographic primitives (digital signatures and KEMs) compared to one in each of the previous competitions (block ciphers for AES, hash functions for SHA-3). There is also a much greater variety of mathematical constructions used to build the candidates. As a consequence, there are much more pronounced differences between the speed and output size characteristics of the various candidates. Whereas the AES competition prescribed just three sizes—128-, 192-, or 256-bit keys, all with 128-bit block sizes—post-quantum KEM candidates (even just among round 3 finalists and alternates) have public keys ranging in size from 197 bytes to more than 1.3 MB and encapsulations from 128 bytes to 21 kB; and round 3 signature candidates have public keys from 32 bytes to 1.9 MB and signatures from 66 bytes to 209 kB.

Certainly, this scale and variety made NIST's selection task harder, and also meant a greater burden on the community in reviewing and evaluating the candidates, both in terms of security and performance. The wide range of size and speed characteristics meant that standalone microbenchmarks would not suffice to evaluate the suitability of candidates for adoption, and instead would require integration and testing of candidates in a variety of contexts. This is where the importance of software implementations plays a greater role.

As with previous competitions, NIST required submissions to be accompanied by software implementations. Each submission needed to include a reference implementation and an optimized implementation for Intel x64, both in ANSI C (no assembly or intrinsics allowed, limited use of external libraries), along with known-answer test (KAT) values to check correctness. Submissions could also include additional implementations for other platforms or microarchitectures. NIST provided a C application programming

interface (API) for each cryptographic primitive as well as a test harness for known-answer tests. (See section 17.2 for a detailed review of NIST's original submission requirements and how they evolved over later rounds.)

In addition to NIST's internal benchmarking, there have been many community and industry projects building on software implementations submitted to NIST. These include: the SUPERCOP benchmarking project [48]; the PQClean project [207] for standalone C implementations on Intel and ARMv8; the pqm4 project [204] for ARM Cortex M4 platform; and the Open Quantum Safe project [333] with a library of C implementations as well as integrations of those algorithms into popular libraries, applications, and protocols. There have also been many research papers and industry experiments building on the above-mentioned projects or directly on software submissions to NIST.

Due to the lack of consistency, organization, and quality of submitted software, each of the above initiatives has involved a repetition of time and effort in getting submitted software to compile and run cleanly.

Admittedly, not all cryptographers should be expected to have the software engineering expertise to create production-quality software, and indeed expecting so may disincentivize the submission of mathematically innovative proposals. Nonetheless, a public cryptography standardization initiative does need software of sufficient quality that works in a variety of settings and has performance characteristics representative of production implementations, in order for good decisions to be made.

We argue that the NIST PQC standardization effort—and future public cryptography standardization—could be improved by having a more extensive software framework prepared in advance by the organizers for submitters, relying on modern continuous integration and testing tools. Our goal is to lay out the requirements for such a framework, based on our experience in the PQClean project, where we assembled a collection of standalone C implementations of NIST PQC submissions and developed a continuous integration testing framework to improve the software we assembled.

17.1.1 Organization of this chapter

As this chapter is slightly separate from the other chapters that report on cryptographic protocols, we give a brief overview of its organization. In section 17.2, we review the submission requirements issued by NIST over the lifetime of the PQC standardization project to date, specifically as related to software imple-

mentations; understanding the software submission requirements provides context to the types and quality of software submitted.

In section 17.3, we begin to examine what went wrong in the process concerning software implementations. Our main observations in this section are (a) that the reference implementations were not ready to meet all the needs expected of them; (b) that "ANSI C" as the language for both reference and optimized implementations may not be the best choice; in particular as (c) no enforcement of standard software-engineering techniques.

In section 17.4, we propose that future cryptography competitions could be improved by having the organizers provide an extensive testing framework for implementations, and we enumerate desired features of such a framework.

In section 17.5 we present details of our PQClean framework, which is an open-source collection of C implementations of NIST PQC candidates, along with an extensive array of compile- and runtime tests via a range of continuous integration tools. Through the process of adding PQC algorithms to PQClean and running our test framework, we identified flaws in the implementations of almost every of the 17 schemes from the NIST PQC project that have been added to PQClean (until 2022); these are summarized in table 17.1.

Much of sections 17.3 to 17.5 focus on C implementations. In section 17.6, we look beyond PQClean's central focus on "cleaning" C implementations, and discuss alternatives to C for representing specifications as well as extensions beyond testing frameworks for cryptographic standardization processes.

We wrap up in section 17.7 with conclusions and recommendations.

17.1.2 Related work

The risk associated with flaws in cryptographic software has been well-known for decades [11, 171, 317]. There are many potential causes for such flaws, which are important to distinguish to help put this chapter's focus into context. There can be cryptographic weaknesses in the cryptographic algorithm itself (weak parameters or broken cryptographic assumptions), applicable to any particular implementation, which is therefore outside the scope of this chapter's focus. The cryptographic implementation could be used in a context that does not match the implementation's threat model: side-channel attacks against implementations without countermeasures, for example. It is also possible that applications and protocols might improperly use otherwise good cryptography algorithms and implementations. The latter is quite

common; for example, Lazar, Chen, Wang, and Zeldovich [237] evaluated 269 flaws for cryptographic software reported in the Common Vulnerabilities and Exposures (CVE) database from 2011–2014 and found that only 17 % were in cryptographic libraries, whereas 83 % were "misuses of cryptographic libraries by individual applications".

Our focus is on when the implementation of a cryptographic algorithm (for which there are no known cryptanalytic attacks) contains flaws and the software development steps that lead to those flaws. Blessing, Specter, and Weitzner [67] examined vulnerabilities specifically in open-source C/C++ cryptographic libraries, and found that only 27 % of vulnerabilities were cryptographic issues, whereas 37 % were memory safety or resource management issues, 11 % involved improper input validation, and 5 % were numeric issues. There are also high-profile examples that seem to derive from particular C coding styles, such as the lack of braces leading to the so-called `goto fail` bug [231].

In the context of cryptographic standardization projects, Mouha, Raunak, Kuhn, and Kacker [260] studied software implementations submitted to the NIST SHA-3 competition. Using solely black-box testing, they found a total of 68 bugs in 41 of the 86 reference implementations submitted, none of which were discovered by the test suite provided by NIST.

A 2015 survey by Braga and Dahab [78] surveys techniques for the development of secure cryptographic software. They identify a sequence of three levels of cryptographic software development:

1. cryptographic library programming and verification;
2. cryptographic software programming and verification; and
3. cryptographic software testing.

For each level, they identify a range of techniques that can be used to reduce the risk of flaws, including using secure languages, secure code generation, applying static and dynamic analysis tools, and using functional tests and adversarial tests (fault injection, fuzzing).

One trend is to create implementations of cryptographic primitives in domain-specific or specialized languages and then generate lower-level implementations from there, with compilers and code generators yielding certain assurances. Examples include the HACL* library written in F* that generates

C code [363]; and Jasmin [9] which generates EasyCrypt code that can be verified for security and functional correctness and x86_64 assembly code for execution; Jasmin can even include mitigations against microarchitecture attacks such as SPECTRE [25].

For implementations that are originally written in C, there are some tools and techniques available for aiding in secure software development, including a range of general-purpose static and dynamic analysis tools. One specialized technique in the context of cryptography is the use of Valgrind to detect control flow based on secret data, an example of which is the TIMECOP project [273].

17.2 NIST PQC software submission requirements

In this section, we review software requirements laid out by NIST for the PQC standardization project as it evolved. Figure 17.1 shows a timeline of the main events in the process. This includes the start of each of the rounds and the deadlines by which submissions had to be updated, as well as the projected timeline for publishing the new standards. In 2022, NIST announced that they would be looking for new signature schemes to be submitted in the "on-ramp"; they expect to finish that process in "18–24 months" [267].

17.2.1 Call for proposals and round 1 submissions

NIST issued a call for proposals in December 2016 which included a set of submission requirements and evaluation criteria [270]. In addition to the design documents, submissions were required to include several components related to software and testing:

- A reference implementation, written in ANSI C, intended to "promote understanding of how the submitted algorithm may be implemented", in which "clarity… is more important than… efficiency" [270, §2.C.1].
- An optimized implementation, also written in ANSI C, targeting the Intel x64 processor.
- A statement by the implementations' owners granting certain rights to use the implementation "for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if

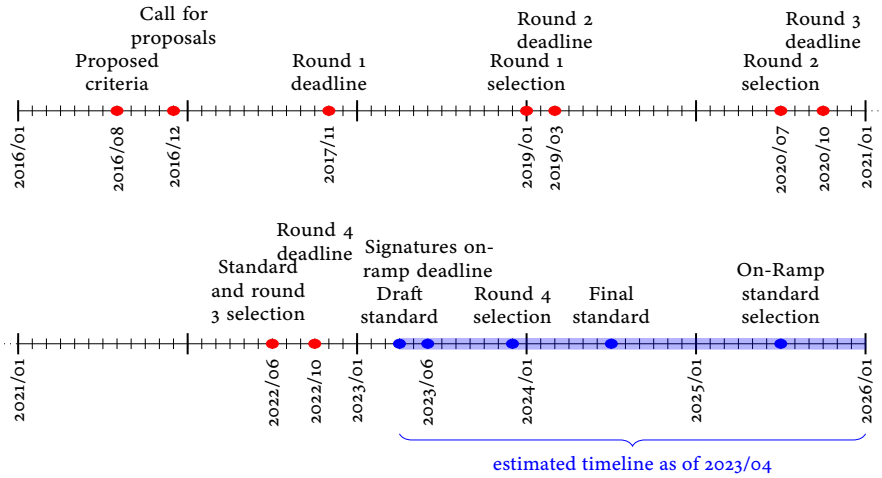


Figure 17.1: Timeline of NIST PQC standardization project. Dates after the end of 2022 are estimates based on recent NIST announcements.

the corresponding cryptosystem is selected for standardization and as a standard" [270, §2.D.3].

- KAT values to check the correctness of reference and optimized implementations [270, §5.B].

Submitters could at their discretion include additional implementations for other platforms, for example using intrinsics or assembly [270, §2.C.1].

The evaluation criteria in [270] referred to software and testing in several aspects:

- Performance: schemes will be evaluated based on their computational cost in software and hardware [270, §4.B.2].
- Side channel aspects: schemes that can be made side-channel resistant efficiently are more desirable than those that cannot, and "optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not" [270, §4.A.6].
- Flexibility: schemes that "can be implemented securely and efficiently

> on a wide variety of platforms" or for which implementations "can be parallelized to achieve higher performance" are desirable [270, §4.C.1].

NIST stated that the goal of the evaluation process during round 1 was to "narrow the candidate pool for more careful study and analysis" and that this "will be done primarily on security, efficiency, and intellectual property considerations" [270, §5.A]. NIST indicated that submitters would be able to provide updated optimized implementations for evaluation in round 2.

The call for proposals indicated that correctness and efficiency testing would be performed by NIST on the "NIST PQC Reference Platform, an Intel x64 running Windows or Linux and supporting the GCC compiler" [270, §5.B]. NIST further stated: "At a minimum, NIST intends to perform an efficiency analysis on the reference platform; however, NIST invites the public to conduct similar tests and compare results on additional platforms (e.g., 8-bit processors, digital signal processors, dedicated CMOS, etc.). NIST may also perform efficiency testing using additional platforms".

In addition to the text of the submission requirements, several technical notes on API and testing and some corresponding source code was included.

NIST's API notes [265] described the C API for signature schemes, public-key encryption schemes, and key encapsulation mechanisms. The API was derived from the eBATS (ECRYPT Benchmarking of Asymmetric Systems) API in the eBACS project [48]. One notable characteristic of the API was that the signature API generated "attached signatures" (where the output of the signing function is a single variable-length "signed message" containing both the message and the signature together) rather than "detached signatures" (where the output of the signing function is a typically fixed-length signature digest without message). The API also included a function providing random bytes, a pseudorandom expander, and an optional deterministic random bit generator to facilitate known-answer tests.

NIST provided a short document [266] describing the process for generating known-answer test values, as well as a source-code archive [269] containing C files for generating KATs for signature schemes, public-key encryption schemes, and KEMs, and a C header file and implementation of a seeded pseudorandom number generator for generating consistent KAT values. The document also included an example Makefile for building and running the KAT programs.

17.2.2 Round 1 selection and round 2 submissions

In January 2019, NIST issued a status report [5] on round 1, including a discussion of its selection of round 2 candidates. The report noted that evaluation criteria used for selecting round 2 candidates were, in order of importance, "security, cost and performance, and algorithm and implementation characteristics" [5, §2.3]. Among the comments on individual schemes selected for round 2 were comments on speed (either positively or negatively) as well as side-channel attacks and constant-time implementations.

The round 1 status report included a statement that, as a next step, NIST was interested in more performance data, including "optimized implementations written in assembly code or using instruction set extensions, and analyses of implementation suitability of candidate algorithms in constrained platforms" [5, §4].

17.2.3 Round 2 selection and round 3 submissions

In July 2020, NIST issued a status report [4] on round 2, including a discussion of its selection of round 3 candidates. At around the same time, NIST issued an additional note with guidelines for submitting tweaks for round 3 [264].

The status report on round 2 noted that the evaluation period saw better data, especially for constant-time implementations on Intel x64 as well as implementations for ARM Cortex-M4 and hardware implementations, and observed that this included information about resources required by implementations, such as RAM or gate counts. The report indicated NIST's desire to see "more and better data for performance in the third round" including for "implementations that protect against side-channel attacks, such as timing attacks, power monitoring attacks, fault attacks, etc." [4, §2.2]. NIST concluded the report with a clear request for performance evaluation of implementations during round 3: "NIST hopes that with only seven finalists and eight alternate candidates, the public review period will include more work on side-channel resistant implementations, performance data in internet protocols, and performance data for hardware implementations in addition to more rigorous cryptanalytical study" [4, §4].

The guidelines for submitting tweaks for round 3 [264] relaxed the requirements on the optimized implementation included in the submission: "the reference implementation should still be in ANSI C; however, the optimized

implementation is not required to be in ANSI C" and recommended "providing an AVX2 (Haswell) optimized implementation and [...] other optimized software implementations (e.g., microcontrollers) and hardware implementations (e.g., FPGAs)".

17.2.4 Round 3 selections, signature on-ramp, and round 4 submissions

In July 2022, NIST issued a status report [6] on round 3, including a discussion of its selection of the first schemes selected for standardization as well as the round 4 candidates. In the guidelines for submitting round 4 tweaks, NIST did not specify any new requirements for software [263]. At the same time, NIST announced the call for new submissions for signature schemes. In the guidelines for submitting new signature schemes [261], announced in September 2022, NIST followed the same requirements as set for the initial NIST standardization project, again requiring "ANSI C".

17.3 Problems with NIST PQC reference implementations

In this section, we argue that the reference implementations submitted to the NIST post-quantum competition did not achieve the declared goal of *promoting the understanding of how the submitted algorithm may be implemented* as well as they could have. We give an overview of what we believe to be the reasons for this; in short, those are a combination of

1. differing expectations of what a reference implementation should accomplish;
2. the choice of ANSI C as the primary programming language; and
3. insufficient use of some standard software-development tools and techniques, paired with a lack of experience with writing cryptographic software in the submission teams.

17.3.1 Reference implementation expectations

Let us first look at what one might reasonably expect from a reference implementation or what such an implementation might be used for. NIST's

call made it clear that the central goal is about *clarity* of the code. Once the programming language—here, ANSI C—is fixed, it is not hard to start heated debates about what exactly "clarity of code" means; however, this is not the central problem. What is much more important is that reference code is typically used for much more than just "promoting understanding", including:

Generation of test vectors This is probably the most obvious use case for a reference implementation aside from portraying what the proposed algorithms look like in code. Being able to reliably generate test vectors is the foundation for any kind of regression testing of more optimized implementations.

Basic performance evaluation A more controversial question is if a reference implementation should also be used as a baseline for performance evaluation. The call made it pretty clear that performance should *not* be a focus for the reference implementation. However, many "reference implementations" submitted to NIST did include pieces of code that were clearly optimized for speed rather than for readability. Also, many papers did report benchmarks of reference implementations [111, 130, 205, 238, 326], sometimes without clear warnings that such benchmarks say absolutely nothing about the performance of the proposed scheme.

Starting point for optimization A very common approach for performance-oriented implementations, typically with platform-specific optimizations, is to start from a reference implementation. The first step is to identify the routines that take most of the CPU cycles and then step-by-step replace those with optimized routines, often written in assembly and targeting a specific microarchitecture. To enable this approach, it is important that the reference implementation builds for and runs on the platform targeted for optimization. In particular, when considering embedded platforms, the use of large external libraries is often a problem.

Use in protocol experiments Through the course of the NIST PQC standardization project, various efforts have investigated how to upgrade *protocols* to have post-quantum security, using the primitives (and implementations) submitted to NIST; see, e.g., [43, 110, 185, 321, 326]. Integration into cryptographic-protocol frameworks often requires namespacing of

the code. A meaningful performance evaluation additionally requires implementations that are optimized for the benchmark platform and adhere to all security guidelines requested by that platform, most notably, not leaking secrets through timing.

Use in (performance-uncritical) production-level experiments Several early adopters experimented with post-quantum primitives in production software; prominent examples are the experiments by Google and Cloudflare with post-quantum TLS [226, 232, 233] and Infineon's implementations of post-quantum cryptography in contactless smartcards [294] and TPMs [151]. While server-side deployments typically need highly optimized software, a portable (reference) implementation may be perfectly reasonable for less performance-critical client-side deployment. This, however, requires that this implementation is secure in the sense of the threat model the respective application uses.

Portability Many of these possible use cases of a reference implementation require that code is portable to different platforms, both hardware (e.g., 32-bit vs. 64-bit platforms or platforms with different endianness) and software (e.g., different operating systems or compilers). In the PQC FAQ, NIST clearly states that *"key requirements are that the submission code should be written in a cross-platform manner […]"*, however without clarifying exactly what this means.

Some of the use cases for a reference implementation have strong synergies. For example, optimized code is also more useful to generate test vectors, and code suitable for academic protocol-level experiments is more likely to also be suitable for use in production software. However, some of these possible use-case scenarios have opposing requirements on a reference implementation and, ironically, almost all of them have some requirements that do not help to promote understanding. Notably, pursuing higher performance often requires unrolling implementations or more advanced (non-"schoolbook") implementations of, e.g., field arithmetic. This may distract from the overall structure of the scheme.

17.3.2 The problems with "ANSI C"

The first problem with requesting software written in "ANSI C" is that the term is not well defined. As pointed out in a posting to NIST's pqc-forum mailing list by Saarinen, "ANSI C" is commonly understood to refer to the ISO/ANSI C90 standard, which does not even define the `long long` data type used by the API. This issue with the definition of "ANSI C" was clarified in an FAQ entry saying that *"implementations written in C99 and C11 are both perfectly fine"* [307]. Furthermore, with regards to the use of the NTL library that is written in C++, NIST clarified that implementations making use of NTL should still be *"as ANSI C-like as possible, only using C++ functionality where absolutely required in order to interact with NTL"*.

However, these clarifications do not address two other issues inherent to the C programming language: the fact that the language is underspecified and that it offers very little support to the programmer to write safe and correct programs.

Underspecification of C

The C programming language is intentionally underspecified to enable compilation to very fast binary code on a broad range of platforms. Krebbers [222] summarizes three different kinds of underspecification:

1. *Implementation-defined behavior* leaves it to the compiler to make decisions about the semantics of certain expressions. These decisions need to be consistent, i.e., when compiling code for one target, all occurrences of the same kind of expression are required to have the same semantics. Also, these semantics should be documented. It is near impossible to write (cryptographic) software without any expression that falls into this category. For instance, even the number of bits in one byte (`char`) is not fixed by the C semantics. Whether `char` is signed or unsigned is also left to implementations. An example of implementation-defined behavior that causes more issues in real-world implementations is the endianness of integers.

2. *Unspecified behavior* leaves it to the compiler to define the semantics of certain expressions. These decisions do *not* have to be consistent throughout the compilation and also do not need to be documented.

One example is the evaluation order. Carefully written code avoids the pitfalls of unspecified behavior, either by entirely avoiding expressions that fall into this category or by ensuring that semantics does not depend on the compiler's decision.

3. *Undefined behavior* allows the compiler to do anything; for example, it would be within the specification of the C programming language to generate a binary that deletes all data in the user's home directory when encountering a case of undefined behavior. The most notable examples of undefined behavior are related to memory safety (i.e., reading or writing out of bounds), but this also includes, for example, signed-integer overflow, division by zero, modifications of string literals, or dereferencing a `NULL`-pointer. Undefined behavior in a program is generally a bug and very often a security-critical one.

Trusting the programmer

The C programming language, by its design philosophy, gives a lot of power to the programmer but also puts a lot of trust in the programmer. For example, C by itself has no mechanism to prevent programmers from accessing memory at invalid locations, has no mechanisms to ensure that all heap allocations are eventually freed (and freed only once), and has no mechanism to check for integer overflows. C does not guarantee that variables are initialized before they are read, it also features a rather weak type system with somewhat unintuitive rules for implicit casts. This all together makes C a great programming language to write highly optimized software, but it also makes it very easy to write programs with bugs that often have severe security implications—in particular in cryptographic software.

17.3.3 Software-engineering issues

Many issues with software implementations submitted to NIST PQC could have been avoided by following standard software-development practices:

Compiler warnings

A first step to avoiding common pitfalls is to enable compiler warnings and ensure that code compiles without any warnings. This may sound like a rather

straightforward thing to do, but it turns out that "compilation without warnings" is much more complex. First, one needs to determine what warnings should be enabled; enabling "all" warnings with `-Wall` in GCC or Clang does by far not enable all warnings, neither do the "extra" (`-Wextra`) or "pedantic" (`-Wpedantic`) warnings levels. Compilation with the Microsoft compiler uses different flags and, unsurprisingly, also issues different warnings. However, even standard warning levels are suitable to identify many common patterns that are typically related to bugs.

Static analysis

More generally, there exist many tools for static analysis of C code, i.e., tools that analyze code without actually running it [65, 86, 329, 342]. These tools are typically able to find larger classes of bugs than those identified through compiler warnings. Aside from the general-purpose static-analysis analysis tools, there exist also tools specifically to check that cryptographic software is free of timing leaks [10]. However, the use of these tools is not a default practice even in the development of widely used cryptographic libraries [195]. One example of a bug that, for example, gcc's static analyzer can catch is in the following piece of code that we found in the reference implementation of a NIST round-1 submission shown in listing 17.1.

Listing 17.1: A buggy piece of code included in one of the NIST round-1 submissions

```
int64_t* extEuclid(int64_t a, int64_t b) {
    int64_t array[3];
    int64_t *dxy = array;
    // ...
    return dxy;
}
```

The problem with this piece of code is that the function returns a pointer to a local stack variable. If the calling code dereferences this returned pointer—what else would it do with a pointer?—the result is undefined behavior. In principle both GCC (via the `-Wreturn-local-addr` flag) and Clang (via the `-Wreturn-stack-address` flag) issue compiler warnings for this kind of behavior. However, in our experiments, the indirection through `dxy` hides

the bug from this compile-time analysis. The reason that this bug did not trigger undefined behavior in this submission is simply that the function was never called from any context—spotting such dead code is another use case for static analysis.

Dynamic analysis

Another approach to finding bugs is running the code with instrumentation or inside special environments. The most commonly used tools for dynamic analysis, at least under Linux, are Valgrind [274, 323], and the AddressSanitizer [322] and UndefinedBehaviorSanitizer [343] included with the Clang compiler. As for static analysis, there also exist specialized tools to identify timing leaks through dynamic analysis [273].

Testing

Extensive testing is still one of the cheapest and most widely used techniques to ensure that software behaves as intended. NIST provided a rather minimalistic framework to generate test vectors for regression and compatibility testing to be used by submitters. Unfortunately, the framework did not include any negative tests (i.e., tests that ensure failure on invalid inputs) or basic tests that the API was used as intended. For example, one of the reference implementations submitted to round 1 of NIST PQC used out-of-bound accesses of the form

```
sk[CRYPTO_SECRETKEYBYTES + j]
```

for positive values of `j` to access bytes of the *public key*. The implementation simply assumed that the public key happens to be stored behind the secret key in memory. This is something that any reasonable testing framework should catch.

We do not mean to suggest that submitters to public cryptographic standardization efforts like NIST PQC should be familiar and up-to-date with all the intricacies of these tools for engineering (cryptographic) software. On the contrary: Our proposal, which we detail in the next section, is that the standardization body soliciting submissions ensures a basic level of code quality by providing a suitable code-analysis and testing framework.

17.4 Proposed features of a software framework for cryptographic competitions

In this section, we propose a testing framework for software submitted to future cryptographic competitions. This section aims to be technology agnostic; we discuss our specific realization, PQClean, which accomplishes many of these goals, in section 17.5. Where necessary, we focus on the C programming language as it remains prevalent for cryptographic software.

Our proposal could be implemented either as an offline solution using virtualization, or online using continuous integration. The benefit of the latter is that most testing can be executed in the cloud without requiring setup by or using resources of each submission team.

We propose that such a testing framework be made available together with the call for proposals, or alternatively, not later than 6 months before the submission deadline for software implementations.

The framework should include at least the following features.

Build system The framework should include a build system that enables a reasonable level of compiler warnings and refuses to compile if any warnings exist. This would help to avoid many obvious mistakes and would save many hours for other researchers to fix the same bugs.

Provide a working example Along with the testing framework, a working example should be provided to serve as a reference of what is expected from submitters. In the case of NIST PQC standardization, this could have been a KEM and signature scheme based on RSA or elliptic curves.

Automated functional tests Straightforward functional tests should be implemented. For example, for digital signature schemes, a generated signed message should verify correctly. Failure test cases should also be included, e.g., a signed message should not verify under a different public key or a modified ciphertext in an IND-CCA KEM should not decrypt (or decrypt to a different value, for implicit failure KEMs).

Verify test vectors In addition to asking submission teams to submit test vectors, the framework should check if the software satisfies the test vectors. We believe that this is best done by including a hash of the test vectors in the submission, saving space in the case of large parameter

sets. (Several round 1 submissions in the NIST PQC standardization project had KAT file archives over 75 MB.)

Provide code building blocks The framework should provide core functionality that is likely to be used by most schemes. For the NIST PQC competition, this primarily consists of hash functions (e.g., SHA-2, SHA-3), extendable output functions (XOFs) (e.g., SHAKE), the AES function, and functions outputting random bytes. This ensures that performance differences between implementations are not due to different implementations of the same building blocks. Requiring implementations to use the same APIs also eases evaluation and modifications by other teams. Submissions should not be allowed to ship their own version of the same functions, though the competition organizers will need to consider how to deal with requests for specialized versions of these functions (e.g., vectorized implementations).

Test using all major toolchains The submitted software should support all major toolchains. In the case of the C programming language, at least GCC, Clang, and Microsoft's compiler (CL) should be supported. The framework should make sure that the code compiles with all of them. As compilers change over time, it is important to fix a version for each of them.

Test on all major platforms To ensure that all submitted code is platform-independent, the tests should be executed on a variety of platforms, including 32-bit and 64-bit systems, and little-endian and big-endian architectures.

Leverage modern static and dynamic analysis The framework should enable the static analysis included in the toolchains. Additionally, Valgrind and AddressSanitizer should be used for dynamic software analysis for detecting memory problems.

Enforce namespacing As implementations from multiple submissions are likely going to be used in the same software (e.g., in a library) their namespaces should be properly separated. The framework should enable and enforce appropriate namespacing and visibility, requiring unique names for public API functions (e.g., `mykem_lvl1_encaps`

rather than `crypto_kem_encaps`) and unique names or limited visibility for internal symbols (e.g., static functions within a compilation unit).

Enforce code style and documentation To improve the readability of code, submissions should be formatted in the same way. The call for submissions should include the coding guidelines and the framework should check if the code is formatted accordingly. In the same vein, a common syntax for documenting code should be defined. The framework should automatically check if a bare minimum of documentation for each function in the source code exists.

Benchmarking code The framework should allow basic benchmarks to ensure that all teams run benchmark their code in the same way. The benchmarking results should be reported in the submission document. Advanced benchmarking (multiple platforms, multiple compilers) may be provided by the competition organizer, in which case it should be a public platform with clear submission procedures and transparent results reporting, or can be taken on by third-party projects like SUPERCOP.

Additionally, the framework could include the more advanced features in the following.

Verify that code is constant time Code intended for use in actual software needs to have runtime independent of any secret data, i.e., avoiding branches depending on secrets, secret-dependent memory accesses, and variable-time instructions depending on secrets. Tools like ctverif [10] could be used to detect such timing leakage. Alternatively, dynamic checking through Valgrind with uninitialized secret data can be used to catch most of the variable-time code. For some code (e.g., rejection sampling) this approach may result in false positives. In that case, submitters need to be able to mark the finding as a false positive and provide a rationale for why it is not a security issue.

Disallow dynamic memory allocation Dynamic memory allocations, e.g. using `malloc`, present a problem on smaller bare-metal platforms. Additionally, they are often a source of bugs that would have been prevented

> by exclusively using stack memory. Thus, we recommend disallowing dynamic memory allocation and enforcing it using a test. In the majority of cases, cryptographic code is only using fixed-sized buffers which makes switching to stack memory straightforward. If variable-sized buffers are needed, the software can usually be rewritten to allocate the worst-case size. Admittedly disallowing dynamic memory allocation can cause other problems. Some PQC algorithms have rather large memory usage, and some platforms do have problems with large stack sizes, especially within threads. Typically, 8 MB of stack is the limit on Linux. If such large buffers are required, dynamic memory allocation may be acceptable.

To lower the burden for initial software submissions, some of the requirements could be optional, i.e., failing tests will merely trigger a warning. In subsequent evaluation rounds, more requirements could become mandatory to gradually increase the quality of implementations. In case any of the requirements are not fulfilled in the submission, we recommend publicly disclosing a list of problems with each submission. The submission team should then be able to remedy the issues within a reasonable time frame. For this to work transparently, source code should be hosted in a code versioning system (e.g., git) that is accessible to everyone. Note, however, that there need to be clear guidelines on the scope of updates allowed. As specifications are usually frozen during each evaluation round, changes that alter test vectors or algorithmic interoperability should not be allowed while algorithm specifications are meant to be frozen.

17.5 The PQClean framework

The goal of the PQClean project is to build a repository of highly-tested, high-confidence implementations, which may be valuable to other projects. As such, PQClean is a collection of schemes and implementations, but it is not a software library; the schemes and implementations all exist independently and PQClean offers no API other than the scheme's interface. We organize the implementations in PQClean like in the SUPERCOP [48] project: the implementations are organized by type (KEM or signature scheme), then by scheme and specific instantiation (e.g., Kyber-512). Each instantiation might have several implementations. PQClean supports C and assembly code; for

some schemes, we also have ARMv8-A or AVX2-specific code. Due to the nature of assembly code, optimized implementations may not be available on all operating systems.

17.5.1 Common files

PQClean makes some common primitives available to each implementation. This includes (incremental) hashing primitives, AES, and random number generators. Anyone extracting implementations from PQClean can re-use these, or provide their own implementation based on our API. For hashing and encryption primitives, we additionally provide initialization and cleanup functions. This allows them to be implemented by heap-based primitives, as for example provided by OpenSSL [279]. PQClean does not attempt to offer the most efficient or any machine-optimized implementations of the primitives.

17.5.2 Meta information

In each instantiation folder, there exists a `META.yml` file, in which some scheme metadata is tracked. This information includes some scheme-specific information, like authors; instantiation-specific information, like key sizes; and information on each of the implementations present, like version and optionally compatibility information. This machine-readable information can be helpful for any automated tool using the framework, including the internal testing framework, as well as for any projects that generate code that wraps implementations from PQClean.

17.5.3 Namespacing

PQClean enforces that all exported symbols, like function names and global values, have a predictable and unique name. They are "namespaced" by prefixing with `PQCLEAN_`, then the name of the scheme and parameter set, and the name of the implementation. This ensures that no symbols conflict between, for example, different schemes or different implementations. Without this separation, it would be difficult to build software that uses several primitives or selects implementations based on CPU feature detection.

17.5.4 Automated testing

Currently, there are 22 automated tests in the PQClean testing harness, against which each scheme is evaluated. The tests range from simply compiling the source code with compiler warnings or checking the existence of license files, to parsing source files to exclude certain types of patterns. These tests include the following; symbols indicate which flaws in table 17.1 the test can potentially identify:

- ⋆ that the scheme compiles correctly, without compiler warnings;
- ♠ `Makefile` correctness: that all scheme source files are correctly specified as dependencies;
- ♣ functional correctness: that the key generation, KEM, and signature operations function as intended, even on unaligned buffers; we also check if corrupted ciphertexts and signatures fail to verify;
- † that scheme keys, ciphertexts, and signatures match test vectors, including NIST's KAT test vectors;
- ⋄ running functional tests with sanitizers (Clang's address sanitizer [322], memory sanitizer [334], and undefined behavior sanitizer [343]) and Valgrind [323];
- ± specification of signedness of `char`;
- = existence of certain timing-suspicious boolean operations;
- ✓ `clang-tidy` [342] linting and static analysis;
- © existence of license files;
- • symbol namespacing;
- • no usage of dynamic memory;
- • consistent style and formatting.

The testing framework is based on Pytest [223], which allows us to generate tests for each scheme and implementation flexibly, and generates convenient output. For tests that compile code, we isolate the source files and compilation targets so that tests can be executed in parallel.

Table 17.1: Flaws found, and which tests might have detected them, in how many of the 10 KEMs and 7 signature schemes that have ever been included in PQClean.

Flaw	KEMs	Sigs	Tests
Endianness assumptions †	7	2	⋆ Compilation test
Platform-specific behavior ♣, ±, †, ✓	4	0	♠ `Makefile` checks
Alignment assumptions ⋆, ♣, ⋄	4	4	♣ Functional tests
Signed integer overflow ⋆, ⋄, ✓	3	1	† Test vectors
Memory safety ⋄	3	4	⋄ Sanitizers
Other undefined behavior ⋆, ♣	1	1	± Signedness of `char`
Integer sizes ⋄, ⋆, †	6	3	= Timing-suspicious ops.
Global state	2	1	✓ `clang-tidy`
Licensing unclear ©	3	1	© License file
Dead code ⋆, ♠	3	4	
Variable-Length Arrays ⋆	4	1	
Compiler extensions ⋆	5	2	
Non-constant time =	4	0	

17.5.5 Testing platform and platform diversity

The automated tests are run on each commit and pull request to PQClean. We also run them periodically on the `master` branch. This ensures that the implementations continue to be validated as compilers and tools get updated.

The testing platform is based on GitHub Actions [159]. This service provides Linux, Windows, and macOS runners, on which we run our automated tests. We run all tests on Linux and macOS with a recent version of GNU GCC as well as with Clang. Windows tests use Microsoft's CL compiler. Results are publicly visible through GitHub's user interface.

Although the native architecture of these systems is the 64-bit, little-endian Intel x64 architecture, we also run tests on 32-bit Intel x86, 32-bit ARMv7 and 64-bit ARMv8, and big-endian PowerPC. We use user-mode QEMU [346] emulation together with Linux's `binfmt_misc` capability [169] to run our tests within Docker images that emulate these targets.

Due to the large number of tests being run on every implementation, we have split the CI jobs per implementation, operating system, compiler, and

architecture. Otherwise, we quickly exceed the maximum allowed runtime for each job (5 hours). On pull requests, we only run the tests on the affected scheme, if possible. This keeps testing times and feedback cycles short.

17.5.6 Results

We have integrated over 230 implementations of multiple parameter sets of 17 schemes into PQClean over the course of the project. In almost every scheme we identified "unclean" code, ranging from missing casts to memory safety problems and other forms of undefined behavior. In table 17.1 we provide a summary of the number of schemes affected by some of the more significant categories of problems. Many of these flaws were detected by our automated testing, as described in section 17.5.4, but in the process of integration, we also solved many problems by hand. The symbols in the table correspond to the tests that might have detected the type of flaw.

Many of the flaws are simply detected by enabling compiler warnings. Solving these warnings probably took the most time when integrating schemes into PQClean. Although many of the reported warnings did not immediately mean the code had a security or correctness flaw, we found that enabling all warnings did help find those problems that were flaws, as well as improve the general code quality.

A perhaps surprising issue was the uncertainty around licensing of 4 of the 17 schemes. Although NIST required the submitters of code in submissions to grant "the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process" [270], it is unclear what the scope of "public review and evaluation" is exactly, and NIST did not require any specific open-source license. Several of the included schemes did not include clear licensing information with their implementations. We contacted their authors and found that many had intended to grant permissive licensing, or even CC0 copyright waivers, to their implementations. However, this often resulted in notable delays; one submission team never provided us with a license and we had to abandon including it in PQClean.

17.6 Beyond "cleaning C"

The various difficulties we discussed in the previous sections motivate the question of whether C is the appropriate programming language for a cryptographic standardization project. This is perhaps as much a philosophical discussion as a technical one. We cannot answer this question definitively for future standardization projects but feel it is worth highlighting some issues.

17.6.1 Is C a good fit for specifications?

C has compilers for almost every system under the sun, which makes it very attractive for experiments (implementation-specific behavior notwithstanding). However, to document and explain an implementation, C is perhaps less well suited. More expressive higher-level languages like Python are perhaps better suited for the role of "executable pseudocode". Rust could perhaps have stood in as a low-level language that simply does not allow most of the problems that our testing system was designed to catch. Additionally, allowing implementations in computational algebra systems like SageMath [308] or Magma [74] would permit expressing the mathematical constructions very directly, not distracting a reader with the details of, for example, polynomial multiplication. For specifications, there have also been efforts such as hacspec [253] that aim to not only generate executable code, but also translate specifications to formal-verification frameworks like F* [336], EasyCrypt [26], or Coq [344]. We believe this pathway has the potential for powerful collaborations with the world of computer-aided high-assurance cryptography [20].

17.6.2 Other languages in PQClean

Although PQClean initially only collected cleaned-up reference C implementations of schemes, we now also have optimized C and assembly implementations of schemes that use platform-specific features like AVX2 or Neon. These implementations are subjected to the same tests as the reference implementations. It would be possible to extend PQClean to implementations in other languages as well. The cross-implementation testing of test vectors would grant more confidence that each implementation is correct and interoperable, especially if at least one of the implementations has been formally verified against a machine-readable specification in, e.g., hacspec [253].

17.6.3 Beyond standardization projects

The goal of PQClean could be described as building a repository of highly-tested, high-confidence implementations. We believe cleaned-up implementations are valuable to other projects. The Open Quantum Safe [333] and the pqm4 [204] projects already automatically integrate implementations from PQClean. We argue that there is value in such an approach for more algorithms and cryptographic libraries. Firstly, it allows focusing analysis and testing efforts. It can save developer time and energy to, e.g., implement automated timing side-channel testing centrally instead of in each individual project. Any such efforts would then benefit all the consumers of the implementations. Currently, it seems that often when a vulnerability like a side-channel leak is discovered, it affects many cryptographic libraries and the effort of patching is duplicated many times. A central repository would minimize the maintenance effort required. Thus, we do believe that a well-designed testing framework benefits not only the standardization effort itself, but may reach beyond, into the phase of deployment.

17.7 Conclusions

This chapter presented what we believe NIST and other bodies coordinating cryptography standardization competitions should do to improve the software quality of submitted code. Properly implemented, a set of guidelines together with a testing framework could benefit submitters, the community, and the standardization body itself. It will allow everyone to focus on what the competition is about: evaluating the candidate cryptographic schemes.

We believe that many of the recommendations in this chapter are uncontroversial and should be implemented in any future competition. For example, providing a working example of what is expected from submitters together with a testing framework would be the bare minimum. The scope of the testing framework may be more controversial and one has to be careful to not raise the bar for submissions too high. Limited resources at standardization bodies may also limit the features of such a framework.

More controversially, we question if C is a suitable programming language for reference implementations, especially if the main goal is clarity of the implementation. While as of now there seems to be no consensus on which alternative should be used, standardization entities should revisit this regularly.

Conclusions and outlook

In this last chapter, I will first summarize the conclusions drawn in this thesis, before looking out toward new developments.

Post-quantum TLS

In this thesis, we have examined what is necessary to transition TLS to post-quantum cryptography. In chapter 3, we have discussed how, with minimal changes, we may instantiate the current TLS 1.3 handshake with post-quantum KEM and signature schemes. As post-quantum signature schemes are much larger than most of the proposals for post-quantum key exchange, we have revisited OPTLS in chapter 4, which was an early proposal in the development of TLS 1.3. OPTLS does not use signature schemes to authenticate the server (and optionally the client) in the handshake; instead, the original proposal for OPTLS uses DH key agreement to authenticate [221]. However, OPTLS' handshake protocol relies on the fact that DH key agreement is a *non-interactive* key-exchange scheme (NIKE). Unfortunately, the only somewhat efficient post-quantum NIKE scheme is CSIDH [91], which limits the number of algorithms we can use to instantiate OPTLS with; additionally, there is disagreement about the level of security offered by CSIDH (see, e.g. [71, 97, 289]) and it has heavy computational requirements.

In light of this, we proposed KEMTLS in chapter 5. This is an alternative to the TLS 1.3 handshake protocol which, like OPTLS, avoids (post-quantum) signatures in the TLS handshake and instead uses key exchange. KEMTLS combines an ephemeral KEM key exchange with a key exchange that uses a server's (and optionally client's) long-term KEM key, as held in a TLS certificate. By combining the ephemeral key exchange result with the symmetric key that is encapsulated to the long-term KEM key, we obtain a secret key that only the party that possesses the long-term KEM private key can compute. However, in a naive approach, because KEMs are an *interactive* key exchange

mechanism, a naive integration of KEMs-based authentication in the TLS handshake requires an additional round-trip. To avoid this, we allow the client to use the derived *implicitly* authenticated secret key to encrypt and send its request to the server *before* the server has explicitly confirmed that it was able to derive the same secret key. This allows the client to send its request to the server at the same time as it would be able to in the TLS 1.3 1-RTT handshake. We argue that this makes unilaterally authenticated KEMTLS in most applications as fast as the TLS 1.3 handshake in terms of RTTs, as servers, notably HTTP servers used in web browsing, generally cannot provide much useful information to the client before having received its request.

We note that TLS clients often make many connections to the same server, for example, because they are making many visits to the same website, or because they are an Internet-of-Things (IoT) device or a back-end service that connects to the same TLS server every time. In such scenarios, we argue that the client may cache or be configured with the server long-term KEM public key. To further reduce the size of KEMTLS handshakes and improve handshake performance in such a *pre-distributed key* scenario, we proposed KEMTLS-PDK in chapter 6. In KEMTLS-PDK, the client, which already has the public key of the server, encapsulates a ciphertext to this key and includes its very first message to the server; this abbreviates the handshake and avoids the server having to transmit a full certificate chain. Unlike the traditional TLS resumption or pre-shared key mechanisms, KEMTLS-PDK does not rely on symmetric keys, so it does not require protected storage. It additionally does not require the server to keep a database of valid resumption keys. Finally, KEMTLS-PDK can fall back to the full KEMTLS handshake, for example, if the client's copy of the server's key has expired, which makes it a flexible extension to KEMTLS.

As KEMTLS(-PDK) are new handshake protocols, and especially as KEMTLS relies on implicit authentication, a security property not previously seen in TLS, we justify its design by providing extensive proofs of the KEMTLS and KEMTLS-PDK handshake protocols in chapters 7 and 8. We show that the KEMTLS and KEMTLS-PDK handshakes provide full forward secrecy and retroactive explicit authentication to all traffic keys in every fully completed handshake. In this thesis, we additionally extended the proof for KEMTLS to cover mutually authenticated KEMTLS and unified the KEMTLS and KEMTLS-PDK proofs. These proofs in the computational model have been done in the pen-and-paper approach. To increase the confidence in the security

of KEMTLS(-PDK), we modeled the protocols and the security properties in the symbolic analysis tool Tamarin [27, 251] in chapter 9. We approach the symbolic analysis in two ways: first, by adopting an existing model which was used to analyze TLS 1.3, adapting it to KEMTLS(-PDK) and proving the security properties originally claimed for TLS 1.3 for KEMTLS; and by more directly translating our protocol and pen-and-paper proof to Tamarin. The latter approach can be directly related to the pen-and-paper results and found discrepancies with our original security claims.

As a real-world security protocol, the performance of post-quantum TLS is almost as important as its security. Part III focused on this subject. In chapter 10, we discussed the experimental implementations of the new handshake protocols in Rust, how we integrated high-performance implementations of the post-quantum KEMs and signature schemes, and how we have measured the performance of the handshake protocols in an emulated network setting. In the next chapters, we show which KEMs and post-quantum signature schemes can be used to instantiate the handshake protocols, and their performance. We specifically discuss post-quantum TLS 1.3 (chapter 11), OPTLS (chapter 12), KEMTLS (chapter 13) and KEMTLS-PDK (chapter 14). In chapter 15, we move from a network environment that is simulated on a single computer with a fictional application payload, to a real-world experiment, which examines the performance of post-quantum TLS 1.3 and KEMTLS for connections between two data centers on two continents. Finally, in chapter 16, we examine a KEMTLS client that runs on a microcontroller and examine its performance when connecting to a server over low-bandwidth connections typical for the IoT setting.

In all of these measurements, we see that the size of the handshake contributes non-negligibly to the handshake latency: the time between the client setting up its network connection and first receiving a response from the server. This is especially noticeable if the connection has low bandwidth or if the handshake exceeds the size of the TCP Slow Start algorithm's initial congestion window (`initcwnd`), which has a size of roughly 15 kB [66, 104]. More obviously, the computational overhead of post-quantum algorithms also contributes to the connection establishment time. KEMTLS allows to significantly reduce the amount of data that is required for public-key cryptography in TLS handshakes and avoids (relatively) computation-heavy post-quantum signature schemes Dilithium [241] and Falcon [293] in favor of the relatively lighter KEM Kyber [319]. Furthermore, the size of the trusted code base is

reduced by KEMTLS: the only sensitive computations remaining are KEM computations, and servers and mutually authenticating clients no longer need to produce online signatures. Verification code needs much less protection compared to signing code in hardened platforms. If there is only enough space for one signature algorithm, KEMTLS' lack of online signing also allows picking a signature scheme without concern for its signing performance or sensitivity to side-channel attacks; this makes the Falcon signature scheme a much more widely applicable option.

In mutually authenticated connections, KEMTLS still manages to reduce the size of the handshake, but as it requires an additional round-trip to securely transmit the client's certificate and execute the authenticating key exchange, its performance is worse than that of TLS 1.3 in most experiments. However, Birghan and Van der Merwe have shown that web browsers seldom use mutual authentication [61]; in many other applications, we conjecture that we can avoid the extra round-trip by using KEMTLS-PDK. For unilaterally authenticated handshakes, KEMTLS-PDK only further improves the handshake performance compared to KEMTLS by reducing the size of the handshakes. In mutually authenticated handshakes, KEMTLS-PDK allows the client to transmit its certificate along the initial `ClientHello` message, which enables client-authentication within a 1-RTT handshake. KEMTLS-PDK additionally allows the selection of algorithms such as Classic McEliece for server authentication, which have very small ciphertexts but too-large public keys to generally transfer in TLS handshakes but, if preinstalled, can be used to offer security based on very conservative assumptions and large reductions in handshake bandwidth requirements.

Outlook

While working on this thesis, I have had the opportunity to examine the performance of TLS and network protocols from different angles; specifically, while interacting with the IETF and working with Cloudflare, I learned a lot about the many environments that make use of TLS. The requirements are sometimes very different. TLS is famously used in web browsers, which generally operate on (compared to microcontrollers) high-performance computers and smartphones. Websites are often hosted on similar high-performance computers, and often even on content-delivery networks that deploy hard-

ware and software that is optimized to set up TLS connections as quickly as possible. Websites themselves are also quite big; the median page weight in May 2022 was 2001 kB for mobile web pages [191]. This makes implementing a new handshake protocol to save a few hundred bytes in the TLS handshake perhaps less interesting to those operating web browsers, even though in many parts of the world (mobile) data is expensive: in more than half of middle-income and low-income countries, 1 GB of mobile data costs more than 2 % of the average monthly income [357].

On the other end of the spectrum, TLS is used in small devices and protocols that have smaller amounts of computational power or constraints on code size. In those applications, KEMTLS represents an opportunity for considerable savings, even though it requires investments in implementation. TLS seems to have been the ubiquitous go-to secure communication protocol in many system designs; I think that the transition from very small elliptic-curve-based primitives to much larger post-quantum alternatives may result in more bespoke approaches as trade-offs are made for individual applications.

Finally, this thesis was completed on the cusp of the National Institute of Standards and Technology deadline for the call for new post-quantum signature algorithms [267]. Some new proposals seem promising, offering small signature sizes and small public keys. The security assumptions of some of these schemes have not yet received much scrutiny, however, and their performance characteristics are currently not always clear. Even so, they will influence the discussion on post-quantum authentication for TLS and other protocols.

Development of post-quantum TLS will continue in the IETF TLS working group, where we have contributed to the discussion on post-quantum authentication in TLS by submitting KEMTLS(-PDK) as an Internet-Draft [94]. The large size of post-quantum signatures and the incumbent large migration to post-quantum cryptography has also motivated other proposals to reduce the size of TLS handshakes, for example, by suppressing (intermediate) CA certificates [202]. In March 2023, Benjamin, O'Brien, and Westerbaan of Google and Cloudflare have even proposed a complete redesign of the TLS public-key infrastructure, by replacing the traditional certificates with credentials authenticated by a new system based on the Merkle trees of the current certificate transparency services [37].

The story of post-quantum TLS appears to have only just begun.

Additional papers

A Verifying post-quantum signatures in 8 kB of RAM

In this chapter, we study implementations of post-quantum signature schemes on resource-constrained devices. We focus on verification of signatures and cover NIST PQC round-3 candidates Dilithium, Falcon, Rainbow, GeMSS, and SPHINCS$^+$. We assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code; a practical and widely deployed setup in, for example, the automotive sector. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big; SPHINCS$^+$ signatures do not fit in this memory. To make signature verification work for these schemes, we stream in public keys and signatures. Due to the memory requirements for efficient Dilithium implementations, we stream in the public key to cache more intermediate results. We discuss the suitability of the signature schemes for streaming, adapt existing implementations, and compare performance.

A.1 Introduction

The generally larger keys and signatures of post-quantum signature schemes have enormous impact on cryptography on constrained devices. This is especially important when the payload of the signed message is much smaller than the signature, due to additional transmission overhead required for the signature. Such short messages are for example used in the real-world use case of feature activation in the automotive domain. Feature activation is the remote activation of features that are already implemented in the soft- and hardware of the car. For example, an additional infotainment package. Usually, a short activation code is protected with a signature to prevent unauthorized activation of the feature.

In the automotive sector, it is very common to perform all cryptographic operations on a dedicated hardware security module (HSM) that resembles a

Cortex-M3 processor with a clock frequency of 100 MHz and limited memory resources, e.g., [175]. Typically, the HSM is in the same package as the main processor with its own memory and is connected via an internal bus with a bus speed of about 20 Mbps. A fair estimate for available memory for signature verification on the HSM is under 18 kB of RAM and 10 kB of flash. However, we aim for a lower memory usage of 8 kB of RAM and flash. To allow additional space for other applications and an operating system.

In this scenario signatures are verified in the very constrained environment of an HSM. It may not be able to store large public keys or keep large public keys or signatures in memory. Sometimes even the main processor does not have sufficient memory resources. Then the public key or signature must be provided to the HSM by another device in the vehicle network, like the head unit. In this case, the public key or signature must be streamed in portions over the in-vehicle network to the destination processor. A typical streaming rate over the CAN bus of an in-vehicle network is about 500 kbps, considering a low error transmission rate. Appendix A.5 provides more details on the use case.

Contribution

In this work, we address the challenge of performing signature verification of post-quantum signature schemes with a large public key or signature in a highly memory-constrained environment. Our approach is to stream the public key or the signature.[1] We show that this way signature verification can be done keeping only small data packets in constrained memory. When streaming the public key, the device needs to securely store a hash value of the public key to verify the authenticity of the streamed public key. During signature verification, the public key is incrementally hashed, matching the data flow of the streamed public key. We implemented and benchmarked the proposed public key and signature streaming approach for four different signature schemes (Dilithium, SPHINCS$^+$, Rainbow, and GeMSS). Although for Dilithium streaming the public key is not strictly necessary, the saved bytes allow us to keep more intermediate results in memory. This results in a speed-up.

[1] Appendix A.6 sketches an alternative scheme that relies on symmetric cryptography with device-specific keys. This would fit even more constrained environments, but comes at the expense of the downsides of symmetric key management.

For comparison, we also implemented the lattice-based scheme Falcon for which streaming small data packets is not necessary in our scenario as the entire public key and signature fit into RAM. The source code is published and available at the link found in the data management appendix. We demonstrate that the proposed streaming approach is very well suited for constrained devices with a maximum utilization of 8 kB RAM and 8 kB Flash.

Related work

To the best of our knowledge, this is the first work that addresses signature verification by streaming in the public key or signature. For signature schemes, streaming approaches have been investigated in [187] but the focus of that work was on signature generation (for stateless hash-based signatures). Encryption scheme Classic McEliece was studied for constrained devices, solving the issue of public keys being larger than the available RAM by either streaming [305, 335] or placing them in additional Flash [99, 137].

A.2 Analyzed post-quantum signature schemes

We now briefly discuss the different signature schemes that we considered. Our exposition is focused on signature verification due to limited space. For all schemes we selected parameters that meet at least NIST security level I. Where possible we prioritized verification speed over signature speed as we assume that signatures are created on devices that are significantly more powerful than the ones we consider for signature verification.

A.2.1 Hash-based schemes

For hash-based signature schemes security solely relies on the security properties of the cryptographic hash function(s) used. Hash-based signatures can be split into stateful and stateless schemes. Stateful schemes require that a user keeps a state with the secret key. The stateful schemes LMS and XMSS are already specified as RFCs [181, 249] and standardized by NIST [106]. As these schemes have sufficiently small signatures and keys, we do not consider them in this work.

SPHINCS$^+$

SPHINCS$^+$ is the last remaining stateless hash-based signature scheme in the NIST competition [47]. In the following we give a rough overview of SPHINCS$^+$ signature verification and motivate our parameter choice. For a high-level description of SPHINCS$^+$ see appendix A.7.

SPHINCS$^+$ signature verification consists of four components. First, the message compression, second message mapping functions, third computing hash chains, and fourth verifying authentication paths in binary hash trees. The message mapping functions take negligible time compared to the other operations and also only minimally increase space. Hence, they are ignored in our exposition. Message compression consumes an n-bit randomizer value from the signature in addition to the message which can in theory be streamed in chunks of the internal block size of the used hash function. The resulting message digest is mapped to a set of indices used later to decide the ordering of hash values in the authentication path verifications. Hash chain computation consumes one n-bit hash value from the signature and iterates the hash function a few times on the given value. The results of 67 hash chain computations are compressed using one hash function call. Hence, results have to be kept in memory until one block for the hash function is full. Finally, authentication path computation takes the n-bit result of a previous computation and consumes one authentication path node per tree level. In theory, these computations can be done one-by-one which would allow streaming each n-bit node separately.

SPHINCS$^+$ is defined as a signature framework with a magnitude of different instantiations and parameter sets. SPHINCS$^+$ defines parameters for three different hash functions: SHA-3, SHA-256, and Haraka. We chose a SHA-256 parameter set due to its performance, well understood security, and widely deployed hardware support. Moreover, SPHINCS$^+$ defines *simple* and *robust* parameters. We chose *simple* as it matches the security assumptions of the schemes that we compare to and has better performance. Lastly, the SPHINCS$^+$ specification [184] proposes fast and small parameters, the former optimized for signing speed, the latter for signature size. However, the small parameters have better verification speed. We chose to implement `sphincs-sha256-128s-simple` and `sphincs-sha256-128f-simple` to allow for a comparison and show what is possible when reduced signing speed is not an issue. For these SPHINCS$^+$ parameters, signing speed on a general

purpose CPU is about a factor 16 slower for the s-parameters [47]. All internal hash values in SPHINCS$^+$ have $n = 16$ bytes for the parameters we use. Public keys are $2n = 32$ bytes. Hence, they can easily be stored on the device without any compression.

A.2.2 Multivariate-based schemes

Multivariate signature schemes are based on the hardens of finding solutions to systems of equations in many variables over finite fields, where the degree of the equations is at least two. The first multivariate signature scheme was designed by Matsumoto and Imai [246] and broken by Patarin [282]. Patarin, with several coauthors, went on to design modified schemes [211, 283, 286] which form the basis of modern multivariate signature schemes.

To fix notation, let the system of equations be given by m equations in n variables over a finite field $\mathbb{F}_q$. Most systems use multivariate quadratic (MQ) equations, i.e., equations of total degree two. Then the m polynomials have the form

$$f_k(x_1, x_2, \dots, x_n) = \sum_{1 \le i \le j \le n} a_{i,j}^{(k)} x_i x_j + \sum_{1 \le i \le n} b_i^{(k)} x_i + c^{(k)} \qquad \text{(A.1)}$$

with coefficients $a_{i,j}^{(k)}, b_i^{(k)}, c^{(k)} \in \mathbb{F}_q$.

Let M be a message and let $H : \{0,1\}^* \times \{0,1\}^r \to \mathbb{F}_q^m$ be a hash function. A signature on M is a vector $(X_1, X_2, \dots, X_n) \in \mathbb{F}_q^n$ and a string $R \in \{0,1\}^r$ satisfying for all $1 \le k \le m$ that $f_k(X_1, X_2, \dots, X_n) = h_k$ for $H(M, R) = (h_1, h_2, \dots, h_m)$. The inclusion of R is necessary because not every system has a solution.

Verification is conceptually easy—simply test that all signature equations hold. Signing depends on the type of construction and what information the signer has to permit finding a solution to the system, but this is outside the scope of this chapter.

General considerations for streaming

MQ systems lead to short signatures, but the public keys need to contain the coefficients of (equation (A.1)) and are thus very large, in the range of a few hundred kB. The public keys can be streamed in per blocks of rows or columns, depending on how the public key is represented. At most m elements of $\mathbb{F}_q$ are

needed to hold the partial results of evaluating $f_k(X_1, X_2, \ldots, X_n), 1 \leq k \leq m$ in addition to the n elements for the signature and the m elements for the hash.

Rainbow

Rainbow [122] is a finalist in round 3 of the NIST competition. Rainbow uses two layers of the Oil and Vinegar (OV) scheme [284]. For Rainbow the finite field is $\mathbb{F}_{2^4}$, so signatures require $\lceil m/2 \rceil$ bytes, leading to 66 bytes at NIST security level I. We implement `rainbowI-classic` rather than one of the "circumzenithal" or "compressed" variants. Public keys are 158 kB for `rainbowI-classic`.

In Rainbow, the coefficients b and c are all zero. During verification, we load in columns $a_{i,j}^{(*)}$ corresponding to coefficients of each monomial $x_i x_j$, $i \leq j$. If $0 \neq x_i x_j = k \in \mathbb{F}_{16}$, we accumulate $a_{i,j}$ into a column $\boldsymbol{A}_k$, If $x_i = 0$, we skip all columns involving x_i. The final result is $\sum_{k \in \mathbb{F}_{16}^*} k\boldsymbol{A}_k$.

GeMSS

GeMSS [89] is an alternate in round 3 of the NIST competition. GeMSS is based on the HFEv– scheme [285]. For GeMSS the finite field is $\mathbb{F}_2$, so signatures are very small, starting at 258 bits for NIST security level I, but to achieve security the public key needs to be very large, starting at 350 kB for level I.

It bears mentioning that GeMSS is special among multivariates in that it employs the Patarin-Feistel structure to achieve very short signatures, wherein a public key is used *four* times during the verification. With pubkey f being m equations in n variables, to verify the signature of the message $\boldsymbol{M}$, we do:

1. write the signature as $(\boldsymbol{S}_4, \boldsymbol{X}_4, \boldsymbol{X}_3, \boldsymbol{X}_2, \boldsymbol{X}_1)$ where $\boldsymbol{S}_i$ are size m and the $\boldsymbol{X}_i$ are size $n - m$ (so the actual length of the signature is $4n - 3m$).
2. At stage i, which goes from 4 to 1, we set $\boldsymbol{S}_{i-1} = f(\boldsymbol{S}_i \| \boldsymbol{X}_i) \oplus \boldsymbol{D}_i$, where $\boldsymbol{D}_i$ is the first m bits of $(\mathrm{SHA} - 3)^i(\boldsymbol{M})$.
3. The signature is valid if $\boldsymbol{S}_0$ is the zero vector.

There are three types of GeMSS parameters. "RedGeMSS" uses very aggressive parameters; "BlueGeMSS" uses more conservative parameters. Just "GeMSS"

falls in the middle, and this is what we choose to implement. The parameter set targeting NIST security level I is gemss-128 and has 350 kB public keys.

A.2.3 Lattice-based schemes

Lattice-based cryptographic schemes are promising post-quantum replacements for currently used public-key cryptography since they are asymptotically efficient, have provable security guarantees, and are very versatile, i.e., they offer far more functionality than plain encryption or signature schemes.

Lattice-based signature schemes are constructed using one of two techniques, either the Gentry, Peikert, and Vaikuntanathan framework that is based on the hash-and-sign paradigm [157], or the Fiat-Shamir transformation [240]. The security of lattice-based signature schemes can be proven based on hard lattice problems (usually the LWE problem, the SIS problem, and variants thereof) or the NTRU assumption.

Dilithium

Dilithium was selected for standardization [241]. Signature verification for Dilithium works as follows: The public key $pk = (\rho, \boldsymbol{t}_1)$ consists of a uniform random 256-bit seed ρ, which expands to the matrix of polynomials $\boldsymbol{A}$, and $\boldsymbol{t}_1$. For MLWE samples $\boldsymbol{t} = \boldsymbol{As} + \boldsymbol{e}$, $\boldsymbol{t}_1$ is the first output of the Power2Round procedure [241, Figure 3], and $(\boldsymbol{t}_1, \boldsymbol{t}_0) = \text{Power2Round}_q(\boldsymbol{t}, d)$ is the straightforward bit-wise way to break up an element $r = r_1 \cdot 2^d + r_0$, where $r_0 = r \mod 2^d$ and $r_1 = (r - r_0)/2^d$ with $-2^d/2 \leq r_0 < 2^d/2$. Hence, the coefficients of $\boldsymbol{t}_0$ are the d lower order bits and the coefficients of $\boldsymbol{t}_1$ — the second part of the public key — are the $\lceil \log q \rceil - d$ higher order bits of the coefficients of $\boldsymbol{t}$. To verify a signature $(\boldsymbol{z}, \boldsymbol{h}, c)$ for a message M, one computes $\boldsymbol{w}' = \boldsymbol{Az} - c \cdot \boldsymbol{t_1} \cdot 2^d$, uses the hint vector $\boldsymbol{h}$ to recover $\boldsymbol{w}_1' = \text{UseHint}(\boldsymbol{h}, \boldsymbol{w}')$, and finally verifies that $c = h(h(h(\rho||t_1)||M)||\boldsymbol{w}_1')$. For the details, we refer to [241].

All Dilithium parameter sets use $q = 2^{23} - 2^{13} + 1$ and $d = 13$. Hence, while the coefficients of $\boldsymbol{t}$ need 23 bits, the coefficients of the public key $\boldsymbol{t}_1$ need only 10 bits. We use the smallest instance of Dilithium, which is NIST level II parameter set dilithium2. The public key size of dilithium2 in total is 1312 bytes and a signature needs 2420 bytes.

Falcon

Falcon, too, was selected for standardization [293]. Falcon's signature verification works as follows: A signature for message M, consisting of the tuple (r, s), can be verified given the public key $h = gf^{-1} \pmod{q}$, where $f, g \in \mathbb{Z}_q[x]/(\phi)$ for a modulus q and a cyclotomic polynomial $\phi \in \mathbb{Z}[x]$ of degree n. Firstly r and M are concatenated and hashed into a polynomial c and s is decompressed using a unary code into s_2. Then, $s_1 = c - s_2 h$ is computed and it is verified that (s_1, s_2) has a small enough norm ($\leq \lfloor \beta^2 \rfloor$). Coefficients are compressed one-by-one and hence can be decompressed individually. The embedding norm that is computed in [293, Algorithm 16, line 6] can be computed in linear time and only requires two coefficients at a time. However, the preceding polynomial multiplication requires all coefficients of one operand to be present, preventing coefficient-by-coefficient streaming for both the signature and the public key at the same time. If, however, only the signature or the public key is streamed, the polynomial multiplication could be performed. We use `falcon-512`, targeting NIST level I, which uses dimension $n = 512$ and $q = 12289 \approx 2^{14}$, hence each coefficient of the public key needs 14 bits.

A.3 Implementation

The following section describes the implementations of the signature schemes for the use case of feature activation described in appendix A.1. The signature verification is performed on a Cortex-M3. The consumption for program flash should be limited to 8 kB. The RAM usage should not exceed 8 kB. The bus speed for streaming is assumed to run at either 500 kbps or 20 Mbps.

A.3.1 Streaming interface

Signed messages and public keys are streamed into the embedded Cortex-M3 device. To avoid performance overhead, our streaming implementation follows a very simple protocol. In a first step, the length of the signed message is transmitted to the embedded device. Then the embedded device initializes streaming by supplying a chunk size to the sender and additionally supplies if signed message or public key is to be streamed first. After every chunk, the embedded device can request a new chunk or return a verification result.

The chunk size may be altered between chunks, but the public key and the signed message are always streamed in-order.The result is a one-bit message, signaling if the verification succeeded or failed, followed by the message in case the verification succeeded.

A.3.2 Public key verification

As the public key is being streamed in from an untrusted source, it is imperative to validate that the key is actually authentic. We assume that a hash of the public key is stored inside the HSM in some integrity-protected area.While the public key is being streamed in, we incrementally compute a hash of it that we eventually compare with the known hash. We use the same hash function as used by the studied scheme, i.e., SHA-256 for `sphincs-sha256` and `rainbowI-classic`, SHAKE-128 for `gemss-128` and SHAKE-256 for `dilithium2` and `falcon-512`. The hash state is kept in memory, occupying additional 200 bytes for SHAKE-128 and 32 bytes for SHA-256. We use the incremental SHA-256 and SHAKE implementations from pqm4 [204].

In the case of `gemss-128`, the public key is needed multiple times; once in every of the four evaluations of the public map. Note that the integrity needs to be verified each time.

A.3.3 Implementation details

In the following, we describe the modifications to existing implementations of the five studied schemes needed to use them with the given platform constraints. Table A.1 lists the public key, signature sizes, and the time needed for streaming them into the device at 500 kbps and 20 Mbps.

SPHINCS$^+$

Our SPHINCS$^+$ implementation is based on the round-3 reference implementation [184]. Preceding work [204] shows that computation time for SPHINCS$^+$ verification on single-core embedded devices is almost exclusively spent in the underlying hash function. We did therefore not investigate further optimization possibilities. Aligning the implementation to a streaming API is fairly straightforward as SPHINCS$^+$ signatures get processed in-order. For both `sphincs-sha256-128f-simple` and `sphincs-sha256-128s-simple` a public key is 32 bytes and hence does not require streaming.

Table A.1: Communication overhead in bytes and ms at 500 kbps and 20 Mbps. GeMSS requires streaming in the public key *nb_ite* times (4 for gemss-128). All other schemes require streaming in the public key and signed message once.

	Streaming data			Streaming time	
	$\|pk\|$	$\|sig\|$	Total	500 kbps	20 Mbps
sphincs-s[a]	32	7 856	7 888	126.2 ms	3.2 ms
sphincs-f[b]	32	17 088	17 120	273.9 ms	6.9 ms
rainbowI-classic	161 600	66	161 666	2 586.7 ms	64.7 ms
gemss-128	352 188	33	1 408 785[c]	22 540.6 ms	563.5 ms
dilithium2	1 312	2 420	3 732	59.7 ms	1.5 ms
falcon-512	897	690	1 587	25.4 ms	0.6 ms

[a] -sha256-128s-simple [b] -sha256-128f-simple [c] $4 \cdot |pk| + |sig|$

For sphincs-sha256-128f-simple, a signature is 17 088 bytes. The selected SPHINCS$^+$ parameter sets use $n = 16$ byte and so a streaming chunk size of 16 bytes is possible. However, such a small chunk is undesirable due to overhead in terms of memory and computation. The leading 16 bytes of the signature make up the randomizer value, followed by the 3696 byte FORS signature, consisting of 33 authentication paths, and the 13 376 bytes for 22 MSS signatures. Our implementation first processes a 3712 byte chunk containing the randomizer value and FORS signature. This is used to compute the message digest and then the FORS root, evaluating the 33 authentication paths. Then, the computed FORS root is verified using the MSS signatures. The overall 22 MSS signatures, each consisting of a WOTS+ signature and an authentication path, are processed in three chunks. Given the memory constraints, the largest available chunk size is 4864 byts containing 8 MSS signatures. MSS signature streaming is therefore done in two 4864 byte chunks and one final 3648 byte chunk. Starting from the FORS root, this data is used to successively reconstruct all the MSS tree roots from the respectively previous root: first computing 67 hash chains using the WOTS+ signature, compressing their end nodes in a single hash, and then evaluating an authentication path. The last (or "highest") MSS tree root is then compared to the root node in the public key. For this to work, the reserved chunk buffer needs

to be 4864 bytes large.

For `sphincs-sha256-128s-simple`, streaming works analogously. The signature size is 7856 bytes and only seven—slightly larger—MSS signatures are used within the scheme. This makes it possible to stream in all 7 MSS signatures in a single 4928 bytes chunk. Streaming therefore consists of one FORS+randomizer-value (2928 bytes) and one MSS (4928 bytes) chunk.

Rainbow

The round-3 submission of Rainbow [122] contains an implementation targeting the Cortex-M4. As it relies only on instructions also available on the Cortex-M3, it is also functional on the Cortex-M3. However, due to the large public key (162 kB), we adapt the implementation for streaming. Rainbow signatures consist of an ℓ bit (128 for `rainbowI-classic`) salt and n (100) variables x_i in a small finite field ($\mathbb{F}_{16}$ for `rainbowI-classic`). Two $\mathbb{F}_{16}$ elements are packed into one byte in the signature and public key. We first unpack the elements of the signature and store one x_i in the lower four bits of a byte. This doubles memory usage from 50 to 100 bytes, but makes look-ups for individual elements easier. After the signature and corresponding x_i are stored in memory, the public key is streamed in. The public key consists of the Macaulay matrix $p_{i,j}^{(k)}$ representing the public map consisting of m (64) equations of the form

$$p^{(k)}(x_1, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=i}^{n} p_{i,j}^{(k)} x_i x_j.$$

with the x_i, x_j corresponding to the variables from the signature. For computational efficiency the public key is represented in the column-major form. The public key's first 32 byte chunk therefore has the form $\left[p_{1,1}^{(1)} \middle| p_{1,1}^{(2)} \middle| \ldots \middle| p_{1,1}^{(m)}\right]$ and the contained coefficients should be multiplied by x_1^2. Subsequent 32 byte chunks have the same form ($\left[p_{1,2}^{(1)} \middle| \ldots \middle| p_{1,2}^{(m)}\right]$ should be multiplied by $x_1 \cdot x_2$ and so forth.) To increase performance, Rainbow implementations delay multiplications. Before the actual multiplication step, coefficient sums are accumulated. Every incoming 32 byte chunk is added to one of 15 accumulators a_k based on the 15 possible values of $y_{i,j} = x_i \cdot x_j$ with $y_{i,j} > 0$. If $y_{i,j}$ is zero, the chunk is discarded. Once all chunks are consumed, every accumulator is multiplied by its corresponding factor $\tilde{a}_k = [k \cdot a_k^{(1)} | k \cdot a_k^{(2)} | \ldots]$ and added

summed up the final result.

One can exploit that if an element x_i is zero, all monomials $x_i x_j$ will be zero and the corresponding columns of the public key will not contribute to the result. As every 16^{th} x_i is expected to be zero, this results in a significant speed-up. As Rainbow is using $\mathbb{F}_{16}$ arithmetic, additions are XOR. For multiplications, we use the bitsliced implementation from the Rainbow Cortex-M4 implementation [122].

The smallest reasonable chunk size for Rainbow is a single column of the Macaulay matrix, i.e., 32 bytes. However, as larger chunk sizes result in lower overhead, we use the largest chunk size which fits in our available memory. Due to the low memory footprint of the Rainbow implementation, we can afford to use chunks of 214 columns, i.e., 6848 bytes. As there are 5050 $(n \cdot (n+1)/2)$ columns, the last chunk is only 128 columns, i.e., 4096 bytes.

GeMSS

To the best of our knowledge, there are no GeMSS implementations available targeting microcontrollers and we, hence, write our own. We base our GeMSS implementation on the reference implementation accompanying the specification [89]. The biggest challenge is that the entirety of the 352 kB public key is needed in each of the evaluations of the public map $\boldsymbol{p}$. Due to the iterative construction of the HFEv- scheme, there appears to be no better approach than streaming in the public key in each iteration, i.e., *nb_ite* (4 for gemss-128) times.

Each application of the public map p requires the computation of

$$p_i = \sum_{i=0}^{n+v} \sum_{j=i}^{n+v} x_i x_j a_{i,j} + a_0.$$

Each column of the Macaulay matrix needs to be multiplied by a product of two variables and then added to the accumulator. The field used by GeMSS is $\mathbb{F}_2$ and, hence, field multiplication is logical AND and field addition is XOR which allows straightforward bitslicing of operations. Unfortunately, since the number of equations (m) is not a multiple of 8 ($m = 162$ for gemss-128), one cannot simply store the Macaulay matrix in column-major form since this would result in the columns not being aligned to byte boundaries. Therefore, GeMSS stores the first $8 \cdot \lfloor m/8 \rfloor$ (160) equations in a column-major

form making up the first $\lfloor m/8 \rfloor \cdot (((n+v)\cdot(n+v+1))/2+1)$ (347 840) bytes of the public key with $n+v$ ($n=174, v=12$) being the number of variables. The last two equations are stored row-wise occupying the last $2\cdot(((n+v)\cdot(n+v+1))/2+1)/8$ (4348) bytes.

We split the computation in two parts: The first $8\cdot\lfloor m/8\rfloor$ equations and the last ($m \mod 8$) equations. For the former, the most important optimization comes from the observation that if either of the two variables x_i or x_j is zero, the corresponding column does not impact the result. Similar to the Rainbow implementation, in the case x_i is zero, the entire inner loop and, hence, $n+v-i$ columns of the public key can be skipped. As half of the x_i are expected to be zero, this results in a vast performance gain. For the last ($m \mod 8$), we first compute the monomials $x_i x_j$ and store them in a vector, then we add this vector to each row of the public key. Lastly, we compute the parity of each row. The smallest reasonable chunk size for the first part of the computation is one column of the public key (20 bytes), while it is one row (2174 bytes) for the second part. However, we use 4560 byte-chunks (285 columns) to achieve lowest overhead with 8 kB RAM.

Dilithium

Our Dilithium implementation is based on previous work targeting the Cortex-M3 and Cortex-M4 [164]. However, this work predates the round 3 Dilithium submission [241] which introduced some algorithm tweaks and parameter changes. Most notably for the performance of `dilithium2` verification, the matrix dimension of $\boldsymbol{A}$ changed from $(k,\ell)=(4,3)$ to $(4,4)$. Therefore, we adapt the existing Cortex-M3 implementation to the new parameters.

For `dilithium2`, the implementation of [164] requires 9 kB of stack in addition to the 2.4 kB signature and 1.3 kB public key in memory. We apply a couple of tricks to fit it within 8 kB: We compute one polynomial of $\boldsymbol{w}'$ at a time, which allows us to stream in the public key $\boldsymbol{t_1}$. Usually, one computes $\boldsymbol{w}' = \boldsymbol{Az} - c\cdot\boldsymbol{t_1}\cdot 2^d$ as $\mathtt{NTT}^{-1}(\hat{\boldsymbol{A}}\cdot\mathtt{NTT}(\boldsymbol{z}) - \mathtt{NTT}(c)\cdot\mathtt{NTT}(\boldsymbol{t_1}\cdot 2^d))$. Hence, it is desirable for performance to keep $\mathtt{NTT}(\boldsymbol{z})$ and $\mathtt{NTT}(c)$ in memory. However, that already occupies 5 kB. We instead keep the compressed forms of $\boldsymbol{z}$ and $\boldsymbol{c}$ in memory, occupying only $\ell\cdot 576 = 2304$ and 32 bytes, respectively, and recompute the $\mathtt{NTT}$ operations.

Previous implementations of Dilithium use 3 temporary polynomials to compute $\mathtt{NTT}^{-1}(\hat{\boldsymbol{A}}\cdot\mathtt{NTT}(\boldsymbol{z}) - \mathtt{NTT}(c)\cdot\mathtt{NTT}(\boldsymbol{t_1}\cdot 2^d))$, one for the ac-

cumulator and two temporary ones for the inputs. We instead compute $\mathtt{NTT}^{-1}(-\mathtt{NTT}(c) \cdot \mathtt{NTT}(\boldsymbol{t_1} \cdot 2^d) + \hat{\boldsymbol{A}} \cdot \mathtt{NTT}(\boldsymbol{z}))$, which can be computed in 2 polynomials by sampling $\hat{\boldsymbol{A}}$ coefficient-wise, as was also proposed for Kyber [75].

The total memory consumption comprises the 2420 byte signature, 2 polynomials of 1024 bytes each, 3 Keccak states of 200 bytes each, and about 600 bytes of other buffers, i.e., approximately 5670 bytes in total. To improve speed, one can cache as much of $\mathtt{NTT}(\boldsymbol{z})$ and $\mathtt{NTT}(\boldsymbol{c})$ as possible. We cache $\mathtt{NTT}(\boldsymbol{c})$ and 3 polynomials of $\mathtt{NTT}(\boldsymbol{z})$ while still remaining below 8 kB of stack.

Falcon

We used a Cortex-M4-optimized implementation which is also part of Falcon's submission [292, 293]. It is compatible with Cortex-M3 processors, but relies on emulated floating point arithmetic. This leads to data-dependent runtimes, which is unproblematic for verification, but may be an issue when considering signing as well. On the Cortex-M3, the implementation submitted to NIST uses around 500 bytes of stack space, public keys of circa 900 bytes, signatures of around 800 bytes, and a 4 kB scratch buffer. The overall memory footprint is about 6.5 kB. Hence, streaming in the data in small packets is not necessary. Our implementation copies the whole public key and signature to RAM before running the unmodified Falcon verification algorithm.

A.4 Results

We chose an ARM Cortex-M3 board with 128 kB RAM and 1 MB Flash, an STM32 Nucleo-F207ZG, as the platform for the implementation of our case study. This board meets most of the specifications of an environment with limited resources of a typical automotive HSM embedded in MCUs. The only mismatch is the non-volatile memory (NVM). A typical limited HSM has much less NVM.

We clock the Cortex-M3 at 30 MHz (rather than the maximum frequency of 120 MHz) to have no Flash wait states. In a practical deployment in an HSM one would use fast ROM instead of Flash and, hence, our cycle counts are close to what we would expect in an automotive HSM.

Table A.2: Cycle count for signature verification for a 33-byte message. Average over 1000 signature verifications. Hashing cycles needed for verification of the streamed in public key (hashing and comparing to embedded hash) are reported separately. We also report the verification time on a practical HSM running at 100 MHz and also the total time including the streaming at 20 Mbps.

	w/o pk vrf.	w/ pk verification			w/ streaming
		pk vrf.	total	time[e]	at 20 Mbps
sphincs-s[a]	8741 k	0	8741 k	87.4 ms	90.6 ms
sphincs-f[b]	26 186 k	0	26 186 k	261.9 ms	268.7 ms
rainbowI-classic	333 k	6850 k[d]	7182 k	71.8 ms	136.5 ms
gemss-128	1619 k	109 938 k[c]	111 557 k	1115.6 ms	1679.1 ms
dilithium2	1990 k	133 k[c]	2123 k	21.2 ms	21.8 ms
falcon-512	581 k	91 k[c]	672 k	6.7 ms	8.2 ms

[a] -sha256-128s-simple [b] -sha256-128f-simple [c] SHA-3/SHAKE [d] SHA-256
[e] At 100 MHz (no wait states)

We base our benchmarking setup on the pqm3 framework and adapt it to support our streaming API. For counting clock cycles, we use the SysTick counter. We stream in the signed message and public key using USART, but disregard the cycles spent waiting for serial communication. We use `arm-none-eabi-gcc` version 10.2.0 with `-O3`. We use a random 33-byte message which resembles the short messages needed for feature activation.

Table A.2 presents the speed results for our implementations. The studied signature schemes rely on either SHA-256 (`rainbowI-classic`, `sphincs-sha256`) or SHA-3/SHAKE (`dilithium2`, `falcon-512`, and `gemss-128`). In a typical HSM-enabled device SHA-256 would be available in hardware and SHA-3/SHAKE will also be available in the future. However, on the Nucleo-F207ZG no hardware accelerators are available. Hence, we resort to software implementations instead. For SHA-256 we use the optimized C implementation from SUPERCOP.[2] For SHA-3/SHAKE, we rely on the ARMv7-M implementation from the XKCP.[3]

[2] https://bench.cr.yp.to/supercop.html
[3] https://github.com/XKCP/XKCP

While GeMSS and Rainbow only compute a (randomized) hash of the message, SPHINCS$^+$, Dilithium, and Falcon use hashing as a core building block of the verification. Consequently, the amount of hashing in multivariate cryptography is minimal (2 % for `rainbowI-classic`, 4 % for `gemss-128`), while it makes up large parts for lattice-based (65 % for `dilithium2`, 36 % for `falcon-512`) and hash-based signatures (90 %for `sphincs-sha256-128s-simple` and 88 % for `sphincs-sha256-128f-simple`). Clearly lattice-based and hash-based schemes would benefit more from hardware accelerated hashing.

Additionally, we need to verify the authenticity of the streamed in public key. We report the time needed for public key verification separately. For hash-based signatures this operation comes virtually for free as the public key itself can be stored in the device, so that no hashing is required. For multivariate cryptography, the public key verification becomes the most dominant operation due to the large public keys and fast arithmetic. This is particularly pronounced for GeMSS as the public key is the largest and needs to be verified 4 times.

Table A.3 presents the memory requirements of our implementations.

Table A.3: Memory and code-size requirements in bytes for our implementations. Memory includes stack needed for computations, global variables stored in the `.bss` section and the buffer required for streaming. Code-size excludes platform and framework code as well as code for `SHA-256` and `SHA-3`.

	Memory				Code
	Total	**Buffer**	**.bss**	**Stack**	**.text**
sphincs-s[a]	6904	4928	780	1196	2724
sphincs-f[b]	7536	4864	780	1892	2586
rainbowI-classic	8168	6848	724	596	2194
gemss-128	8176	4560	496	3120	4740
dilithium2	8048	40	6352	1656	7940
falcon-512	6552	897	5255	400	5784

[a] -sha256-128s-simple [b] -sha256-128f-simple

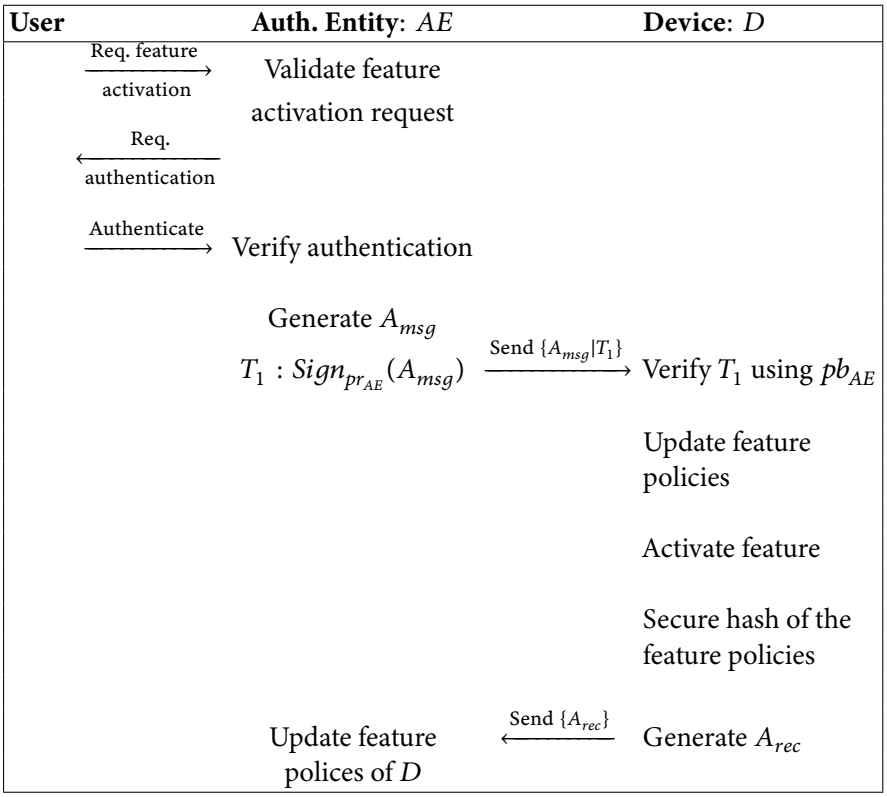
A.5 Appendix: Feature activation

Feature Activation is intended to activate additional functionality on an embedded device that is already deployed and active in the running environment. It differs from a software update because all software and required hardware for the feature's functionality is already included in the device, but not activated.

The feature is activated by an authentic message from an authorized instance. The activation of the feature is device specific, therefore the activation messages must not be portable to other devices. Protocol A.1 describes the actual feature activation process between an embedded device on which the feature is to be activated and an authorized instance, e.g., a back-end system. To authenticate the feature activation request, a signature is part of the message sent from the authorization instance to the embedded device. Nowadays, this signature is implemented, for example, by an ECC signature, which is not a post-quantum algorithm. In the scenario shown, the overall protocol does not take into account any resource constraints on the device, so that, for example, the ECC signature and the public key are stored entirely on the device.

Protocol A.1 can be roughly paraphrased as follows: The user, e.g. the car owner, creates a request to activate a desired feature for a specific vehicle (identified by a vehicle identification number (VIN)). This can be done through an online platform. The authorization instance, which can be represented by a back-end, validates the feature request for the feature policies it stores for the requested vehicle, and requests and verifies the user's authentication. Upon successful authorization, the authorization instance generates a device-specific feature activation request A_{msg} for the device that is part of the vehicle and implements the requested feature. Furthermore, the authorization instance generates a signature T_1 for the message A_{msg} using its private key pr_{AE}. When the device successfully verifies the signature T_1, it updates its feature policies, activates the requested feature, and stores the feature policy hash. Finally, the embedded device confirms the feature activation status in a message A_{rec} to the authorization instance. The authorization instance itself also updates and stores the feature policy for the specific device.

Protocol A.1: Protocol for feature activation



A.6 Appendix: Alternative implementation

For embedded applications it is sometimes attractive to use symmetric cryptography in place of public-key cryptography. Not only is symmetric cryptography a lot faster, it also benefits from already-present hardware acceleration and key sizes are significantly smaller. Of course, the secret keys in symmetric devices are extremely sensitive. We need that a secret key extracted from a particular deployed device does not compromise the entire scheme. This implies the need to provision each device with its own individualised key. However, when deploying hundreds or thousands of devices this means we have a potentially significant key management problem. Fortunately, the many-to-one architecture in this automotive application implies we only need a single key

between each device and the back-end. Furthermore, each deployed device has public identifiers, like the vehicle VIN or a serial number. This allows us to only let the manufacturer store a single key for all deployed devices.

We use these properties to construct an efficient key distribution scheme. Let each device have a unique identifing number n. This could for example be the concatenation of the vehicle VIN and the device's serial number. We let the manufacturer generate a main secret key K_m. Then, we provision at time of manufacturing each device with the following key K_d, such that

$$K_d = \mathrm{KDF}(K_m, n).$$

Here, KDF is an appropriate key-derivation function.

Then, whenever the device needs to use their key K_d in communication with the manufacturer, they send over their identifier n along with the message. For example, if they need to send an authenticated message m, they might send $\{n, m, \mathrm{MAC}_{K_d}(n, m)\}$. The manufacturer can then easily compute K_d based on n and the main secret K_m, and verify the message. As the device does not have access to K_m, they can only have produced this MAC if they were provisioned with K_d at time of manufacture.

Of course, this entire scheme falls down when K_m is compromised. As such, special care needs to be taken to protect it. Although the private keys used in public-key cryptography also need to be protected, we can use revocation mechanisms to recover from a compromise. This is not possible with symmetric cryptography.

A.7 Appendix: Hash-Based Signatures

Hash-based signature schemes are signature schemes for which security solely relies on the security properties of the cryptographic hash function(s) used. In contrast to other proposals for digital signature schemes it does not require an additional complexity theoretic hardness assumption. Given that the security of cryptographic hash functions is well understood and even more, we know that generic attacks using quantum computers cannot significantly harm the security of cryptographic hash functions, hash-based signatures present a conservative choice for post-quantum secure digital signatures. The description in this section is simplified, and we refer to the official specification [184] for a detailed exposition.

One-Time Signature Schemes (OTS)

Hash-based signatures built on the concept of a one-time signature scheme (OTS). This is a signature scheme where a key pair may only be used to sign one message. If two messages are signed with the same secret key, the scheme becomes insecure. Such OTS can be constructed from cryptographic hash functions. The very generic concept is that the secret key consists of random values while the public key contains their hash values. A signature consists of a subset of the values in the secret key, selected by the message. A signature is verified by hashing the values in the signature and comparing the resulting hash values to the respective values in the public key. The OTS commonly used today is the Winternitz OTS (WOTS) or variations thereof which generalizes the above concept to hash chains. WOTS has the important property that a signature is verified by computing a candidate public key by hashing the values in the signature several times (depending on the message) and comparing the result to the public key.

Merkle Signature Schemes (MSS)

Given a OTS, a many-time signature scheme can be constructed following the concept of Merkle Signature Schemes (MSS) [255]. For these, a number (a power of 2) of OTS key pairs is generated and their public keys are authenticated using a binary hash tree, called a Merkle tree. The hashes of the public keys form the leaves of the tree. Inner nodes are the hash of the concatenation of their two child nodes. The root node becomes the MSS public key. Assuming WOTS is used as OTS, a signature consists of the leaf index, a WOTS signature and the so-called authentication path (cf. figure A.1). The authentication path contains the sibling nodes on the path from the used leaf to the root. Verification uses the WOTS signature (and the message) to compute a candidate public key and from that the corresponding leaf. This leaf is then used together with the authentication path to compute a root node: Starting with the leaf, the current buffer is concatenated with the next authentication path node and hashed to obtain the next buffer value. The order of concatenation is determined by the leaf index in the MSS signature. The final buffer value is then compared to the root node in the public key.

In general, this leads a so-called stateful scheme as a signer has to remember which OTS key pairs she already used. This concept is the general idea

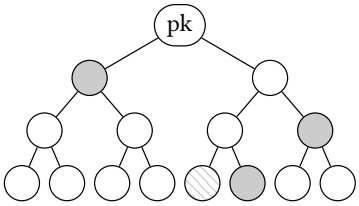


Figure A.1: The authentication path of the fifth leaf (from [47])

underlying the schemes described in recent RFCs [181, 249] LMS and XMSS.

SPHINCS$^+$

The limitation of having to keep a state as signer can be overcome in practice using the SPHINCS construction [45] (previous theoretically efficient proposals by Goldreich go back to the last century but were only of theoretical interest). SPHINCS$^+$ [47] essentially instantiates the SPHINCS construction. The first idea in SPHINCS uses a few-time signature scheme (FTS)—a signature scheme where a key pair can be used to sign a small number of messages without keeping a state before the scheme gets insecure. SPHINCS$^+$ uses a huge number of FTS key-pairs (in the order of 2^{64} depending on the parameters). For every new message, a random FTS key is picked to sign. By making the number of FTS keys large enough, the probability that one key gets used to sign more than a few messages can be made vanishingly small. The public keys of all these FTS key pairs are authenticated using a certification tree of MSS key pairs called the hypertree. The hyper tree is essentially a PKI. The top MSS key works as a root CA and the bottom layer MSS keys certify FTS public keys. The whole structure is deterministically generated using pseudorandom generators. That way, it is not necessary to store which OTS keys where used for an MSS key because the message that a specific OTS key will be used to sign is predetermined.

The FTS in SPHINCS$^+$ is FORS. A FORS secret key consists of several sets of random values the values in each set are authenticated via a Merkle tree. These trees have the hashes of the secret values as leaves. The public key is the hash of the concatenation of all root nodes of these Merkle trees. A signature consists of one secret key value from each set (determined by the message) and the respective authentication path. Verification works by computing the

leaves from the signature values and afterwards computing candidate root nodes as for MSS. This can be done per tree. Afterwards, the roots are used to compute a candidate public key.

A SPHINCS$^+$ signature consists of a randomizer R that is hashed with the message to obtain the message digest, a FORS signature, and the MSS signatures on the path from the FORS keypair to the top tree. Verification computes a message digest using the message and R. The message digest is split into the index of the FORS signature and the indices of the Secret key values in the FORS signature. With this, the FORS signature is used to compute a candidate public key. This candidate FORS public key is used as message to compute a candidate MSS root node with the first MSS signature which is used as message for the next signature, and so on. The final MSS root node is compared to the root node in the public key.

B Practically solving LPN

The best algorithms for the LPN problem require sub-exponential time and memory. This often makes memory, and not time, the limiting factor for practical attacks, which seem to be out of reach even for relatively small parameters. In this chapter, we try to bring the state-of-the-art in solving LPN closer to the practical realm. We improve upon the existing algorithms by modifying the Coded-BKW algorithm to work under various memory constrains. We correct and expand previous analysis and experimentally verify our findings. As a result we were able to mount practical attacks on the largest parameters reported as of the original publication of this work, using only 2^{39} bits of memory.

B.1 Preliminaries

We will denote vectors and matrices with bold-face letters, like $\boldsymbol{v}$ or $\boldsymbol{M}$. We write inner product of two vectors as $\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle$ The Hamming weight of $\boldsymbol{v}$ is $wt(\boldsymbol{v})$. We write Ber_τ for a Bernoulli distribution with parameter τ. Bin_τ^n is the binomial distribution with n trails and success rate τ. We write $y \xleftarrow{\$} Y$ when we uniformly sample y from Y.

The LPN Search problem can be defined using the following definition from [69].

Definition B.1. (Search LPN problem). Let $\boldsymbol{s} \xleftarrow{\$} \mathbb{F}_2^k$ be a secret vector of length k and let $0 \leqslant \tau < \frac{1}{2}$ be a constant noise parameter. An LPN oracle $\mathcal{O}_{\boldsymbol{s},\tau}^{\mathsf{LPN}}$ outputs independent random samples $(\boldsymbol{a}, c)$ according to the distribution:

$$\left\{ (\boldsymbol{a}, c) \,\middle|\, \boldsymbol{a} \xleftarrow{\$} \mathbb{F}_2^k, c = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e, e \leftarrow \mathrm{Ber}_\tau \right\}.$$

The Search LPN Problem, denoted by $\mathsf{LPN}_{k,\tau}^n$ is to find the (secret) vector $\boldsymbol{s}$, given access to the LPN oracle.

We will be interested in algorithms that solve $\mathsf{LPN}^n_{k,\tau}$ in time t, using at most n samples and using at most m bits of memory. Such an algorithm may fail with a certain probability θ. Sometimes, instead of the noise parameter τ we will use the *bias* of an LPN instance $\mathsf{LPN}_{k,\tau}$, defined as $\delta = E\left((-1)^X\right) = 1 - 2\tau$ where $X \sim \mathrm{Ber}_\tau$. We will refer to the bias of the secret as δ_s.

B.2 Solving LPN problems

The known algorithms that solve an LPN instance $\mathsf{LPN}_{k,\tau}$ typically follow a common structure. We can usually split them in two phases: a *reduction phase* in which a *reduction* algorithm reduces the problem to a smaller one $\mathsf{LPN}_{k',\tau'}$, $k' \leqslant k$; and a *decoding phase* in which a *decoding* algorithm recovers the secret of the smaller LPN instance. Intuitively, a smaller LPN problem is easier to decode, but a reduction typically increases the level of noise and may change the number of samples.

It is possible to apply a sequence of reduction algorithms before decoding the reduced $\mathsf{LPN}_{k',\tau'}$ instance. This is already implied by the original BKW algorithm. Bogos and Vaudenay [70] proposed using chains of different reduction algorithms before applying a decoding algorithm. We summarize this meta-algorithm in algorithm B.1.

Note that most decoding algorithms recover only part of the secret. However, the algorithm can be repeated to obtain more information. We will, as in the literature, only discuss the first run of the algorithm, since this is the most resource-intensive of recovering the full $\boldsymbol{s}$.

B.2.1 Reduction algorithms

We will now briefly discuss algorithms that reduce an $\mathsf{LPN}^n_{k,\tau}$ problem to an $\mathsf{LPN}^{n'}_{k',\tau'}$ problem. For more details on these algorithms, we refer to the cited works.

`drop-reduce`($\boldsymbol{b}$)

Deletes all samples that do not have b zero bits at the end.

$k' = k - b$; $n' = n2^{-b}$; $\delta' = \delta$; $\delta'_s = \delta_s$;
$t = \mathcal{O}(kn)$; $m = \mathcal{O}(kn)$.

Algorithm B.1: General LPN decoding algorithm

Input: n samples $(\boldsymbol{a}, c)$ from $\mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s},\tau}$, list of reduction algorithms $\mathcal{R}$, and decoding algorithm D
Output: Information on $\boldsymbol{s}$

for $R \in \mathcal{R}$ **do**
  Apply R to obtain $\mathsf{LPN}_{k',\tau'}$, $k' \leqslant k$ and n' samples.
  $k \leftarrow k', n \leftarrow n'$
end for
Use decoding algorithm D, consuming n samples.
return Information on $\boldsymbol{s}$

`xor-reduce`($\boldsymbol{b}$)

Partitions samples by the last b bits and sums all pairs of vectors within each partition [239]. This cancels out the last b bits. The bias of the LPN problem is squared as per the piling-up lemma: the bias of the sum of n Bernoulli variables with bias δ is δ^n.

$$k' = k - b;\ n' = \tfrac{n(n-1)}{2^{b+1}};\ \delta' = \delta^2;\ \delta'_s = \delta_s;$$
$$t = \mathcal{O}\left(k\ \max(n, n')\right);\ m = \mathcal{O}\left(\max(kn, k'n')\right).$$

`sparse-secret`

Transforms the problem so that the secret is Bernoulli-distributed with $\tau < \frac{1}{2}$ instead of uniform [16, 70, 170]. This reduction does not simplify the LPN problem, but is necessary for `code-reduce`.

$$k' = k;\ n' = n - k;\ \delta' = \delta;\ \delta'_s = \delta;\ m = \mathcal{O}\left(kn\right);$$
$$t = \mathcal{O}\left(\min_{\chi \in \mathbb{N}}\left(\frac{n'k^2}{\log_2 k - \log_2 \log_2 k} + k^2, kn'\lceil\tfrac{k}{\chi}\rceil + k^3 + k\chi^{2\chi}\right)\right).$$

`code-reduce`($[\boldsymbol{k}, \boldsymbol{k'}]$ *code*)

Uses the covering property of codes to reduce the LPN problem size [69, 70, 170]. Using a linear $[k, k']$ code, `code-reduce` approximates the samples to the closest codeword of the code. The effect on the bias is called bc and depends on the original δ and the properties of the code. A bigger bc is better, as it will maximize the bias of the reduced LPN instance.

Theorem B.2. *(Upper bound for* bc *[70]). A* $[k, k', D]$ *linear code* C *has for any* $r \in \mathbb{N}$ *and* $\delta_s \in [0, 1]$*:*

$$\mathsf{bc} \leqslant 2^{k'-k} \sum_{w=0}^{r} \binom{k}{w} (\delta_s^w - \delta_s^{r+1}) + \delta_s^{r+1}.$$

Equality for any δ_s *implies that* C *is a* (quasi-)perfect *code, in which case* r *equals the packing radius* $R = \lfloor \frac{D-1}{2} \rfloor$.

In [170], the analysis of code-reduce was done for codes that reach the bound in theorem B.2. This overestimates the efficiency of the reduction. In practice, we know a few small codes that are close to the bound and have efficient decoding. Instead, Bogos and Vaudenay concatenate small codes that either reach or are close to the bound [70]. We use the same approach. As the modified δ_s is hard to quantify, we only allow code-reduce to be applied once.

$$k = k'; n' = n'; \delta' = \delta \cdot \mathsf{bc}; t = \mathcal{O}(kn); m = \mathcal{O}(kn).$$

c-sum-Dissection(b)

It is possible to sum up more than just two samples, such that the last bits add up to 0. This was initially proposed in [362] as LF(4). Esser, Heuer, Kübler, May, and Sohler [139] rephrased it as a time-memory trade-off for solving LPN problems. They use the Dissection technique [123] to solve c-sum problems in lists of samples. Dissection requires that c is one of $c_i \in \left\{\frac{1}{2}(i^2 + 3i + 4) \mid i \in \mathbb{N}\right\}$. The first few c are 2, 4, 7, 11. It also requires that $\log_2(n/c_i) \leqslant b/i$.

$$k' = k - b; n' = \tbinom{n}{c} \cdot 2^{-b}; \delta' = \delta^c; \delta'_s = \delta_s; t = \mathcal{O}\left(2^{c_{i-1}\frac{n}{c_i}}\right);$$
$$m = \mathcal{O}(kn).$$

Note that Delaplace, Esser, and May [114] have suggested improving c-sum-Dissection by using the Van Oorschot–Wiener Parallel Collision Search (PCS) algorithm [349]. We denote this variant as c-sum-PCS(b).

Algorithm B.2: WHT algorithm [239]

Input: A set V of s k'-bit samples $(\boldsymbol{a}, c) \in \mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s}',\tau'}$.
Output: $(\boldsymbol{s}'_1, \ldots, \boldsymbol{s}'_{k'})$ from $\boldsymbol{s}'$
$f(\boldsymbol{x}) = \sum_{(\boldsymbol{a},c)\in V} 1_{V_{1,\ldots,k'}=\boldsymbol{x}}(-1)^c$
$\hat{f}(\boldsymbol{x}) = \sum_x (-1)^{\langle \boldsymbol{a},\boldsymbol{x}\rangle} f(x)$
return $(\boldsymbol{s}'_1, \ldots, \boldsymbol{s}'_{k'}) = \arg\max_{\boldsymbol{a}\in\mathbb{Z}_2^{k'}}(\hat{f}(\boldsymbol{a}))$

B.2.2 Decoding algorithms

The general algorithm from algorithm B.1 for solving LPN reduces $\mathsf{LPN}^n_{k,\tau}$ to a smaller instance $\mathsf{LPN}^{n'}_{k',\tau'}$ through a number of reduction steps. It then solves the final instance using some sort of decoding algorithm. The original BKW used majority decoding [68]. This was improved by using the Walsh–Hadamard transform (WHT) [239] and subsequently used in [70, 170].

$$t = k' \cdot 2^{k'-1}(\log s + 1) + k's;\ m = k'(2^{k'} + s).$$

Esser, Kübler, and May [140] used the folklore Gauss algorithm that performs simple Gaussian eliminations using k' samples, assuming error-freeness. The obtained candidate $\boldsymbol{s}'$ is then tested against s samples to determine whether the error's distribution is closer to Bin^s_τ or $\mathrm{Bin}^s_{\frac{1}{2}}$. The Pooled-Gauss variant randomly selects samples from a re-used pool.

$$t = ({k'}^3 + k's) \cdot \log^2 k' \cdot (1-\tau')^{-k'};\ m = k'(k' + s).$$

The two algorithms are given in algorithm B.2 and algorithm B.3.

B.2.3 Finding the best reduction chain

Bogos and Vaudenay proposed in [70] to search for the most efficient combination of reductions (*reduction chain*) before decoding the problem. They present their algorithm as an automaton that defines all possible reduction paths. They used (concatenated) perfect, quasi-perfect and random codes for the code-reduce reduction and failure probability $\theta = 0.33$. We modify the algorithm to include the Pooled-Gauss decoding algorithm, as well as more reduction techniques. We present the updated automaton in figure B.1.

Algorithm B.3: Gauss algorithm [140]

function GAUSS($\mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s}',\tau'}, \tau'$)
  repeat
    $(\boldsymbol{A}, \boldsymbol{c}) \leftarrow \left(\mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s}',\tau'}\right)^{k'}$ such that $\boldsymbol{A}$ is full rank
    $\boldsymbol{s}' = \boldsymbol{A}^{-1}\boldsymbol{c}$
  until TEST($\boldsymbol{s}', \tau', \frac{1}{2^k}, \left(\frac{1-\tau'}{2}\right)^k$)
  return $\boldsymbol{s}'$
end function

function TEST($\boldsymbol{s}', \tau', \alpha, \beta$)
  $s = \left(\frac{\sqrt{\frac{3}{2}\ln(\frac{1}{\alpha})} + \sqrt{\ln(\frac{1}{\beta})}}{\frac{1}{2}-\tau'}\right)^2$
  $c = \tau' s + \sqrt{3\left(\frac{1}{2}-\tau'\right)\ln\left(\frac{1}{\alpha}\right)s}$
  $(\boldsymbol{A}, \boldsymbol{c}) \leftarrow \left(\mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s}',\tau'}\right)^{s}$
  return $wt(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \leqslant c$
end function

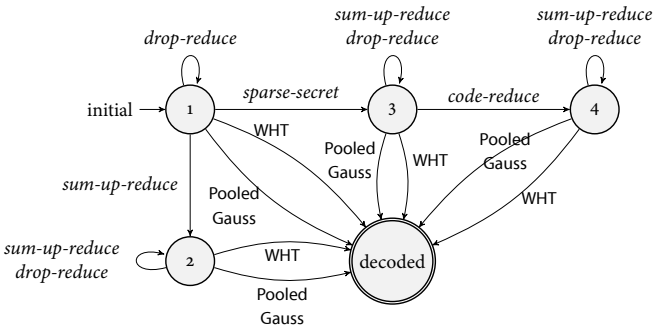


Figure B.1: The automaton accepting valid LPN reduction chains. sum-up-reduce represents any of the reductions combining samples, i.e., xor-reduce, lf4-reduce, c-sum-Dissection or c-sum-PCS.

B.3 Fair comparison between WHT and Gauss

We revisit both WHT and Gauss and provide a unified analysis in order to compare them. Our analysis shows that assuming negligible decoding error $1/2^k$, both algorithms require (almost) the same number of samples. However, their efficiency depends very differently on the size of the problem and the bias. As a consequence, they are suitable for different scenarios. This has several implications.

First, we show that there is no obstacle in obtaining a negligible error in WHT by choosing an appropriate number of samples. This was overlooked in [140].

Second, we provide the basis for a fair comparison between chains of reduction steps ending in Gauss and WHT. As we will see later, this disproves the claim in [140] that Gauss can be combined with various reduction steps and give better results than performing reduction steps and using WHT. This further explains the experimental results from [140] which imply that Gauss almost always performs better without any sum-up-reduce reduction steps.

As a side result, we improve the efficiency of Gauss by obtaining a better bound for the sample complexity.

Proposition 1. *If*

$$s = \left(\frac{4}{(1-2\tau)^2} - 2\right) \ln \frac{1}{\sqrt{2\pi}\gamma}$$

samples are available, where $\gamma \in \left(0, \frac{1}{\sqrt{2\pi e}}\right]$*, the WHT algorithm applied to* $\mathsf{LPN}^n_{k,\tau}$ *outputs the correct solution with probability at least* $1-\gamma$*.*

Proof. We detail the analysis for a positive bias following the approach of [70]. For a negative bias, the analysis is equivalent. WHT outputs the candidate with the largest value of $\hat{f}$. Failure occurs when there exists another $\bar{\boldsymbol{s}} \neq \boldsymbol{s}$ such that $\hat{f}(\bar{\boldsymbol{s}}) > \hat{f}(\boldsymbol{s})$, i.e., when $\mathrm{HW}(\boldsymbol{A}\bar{\boldsymbol{s}} + \boldsymbol{c}) < \mathrm{HW}(\boldsymbol{A}\boldsymbol{s} + \boldsymbol{c})$. Let $\bar{\boldsymbol{y}} = \boldsymbol{A}\bar{\boldsymbol{s}} + \boldsymbol{c}$ and $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{c}$. Then the expectation and variance of the random variables $\boldsymbol{x}_i = \boldsymbol{y}_i - \bar{\boldsymbol{y}}_i$ is $\mathrm{E}(\boldsymbol{x}_i) = \frac{2\tau-1}{2}$ and $\mathrm{Var}(\boldsymbol{x}_i) = \frac{1}{2} - \left(\frac{2\tau-1}{2}\right)^2$. Let

$$Z = \frac{\sqrt{s}(S_s - \mathrm{E}(\boldsymbol{x}_i))}{\sqrt{\mathrm{Var}(\boldsymbol{x}_i)}},$$

where $S_s = \frac{\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_s}{s}$. By the Central Limit Theorem $Z \xrightarrow{d} N(0,1)$. Using

standard upper-tail inequalities for the standard normal distribution $N(0,1)$, we obtain

$$Pr\left[\hat{f}(\bar{\boldsymbol{s}}) > \hat{f}(\boldsymbol{s})\right] = Pr\left[Z \geqslant \frac{(1-2\tau)\sqrt{s}}{\sqrt{2-(1-2\tau)^2}}\right] \leqslant \frac{e^{-\frac{(1-2\tau)^2 s}{2(2-(1-2\tau)^2)}}}{\sqrt{2\pi}} \tag{B.1}$$

Taking $s = \left(\frac{4}{(1-2\tau)^2} - 2\right) \ln \frac{1}{\sqrt{2\pi}\gamma}$, inequality (B.1) becomes

$$Pr\left[\hat{f}(\bar{\boldsymbol{s}}) > \hat{f}(\boldsymbol{s})\right] \leqslant \gamma.$$

We can make the probability of an error in the WHT procedure arbitrarily small if we take $\gamma = \mathsf{negl}(k)$. □

Proposition 2. *If*

$$s = \left(\frac{\sqrt{2\tau(1-\tau)\ln(\frac{1}{\sqrt{2\pi}\alpha})} + \sqrt{\frac{1}{2}\ln(\frac{1}{\sqrt{2\pi}\beta})}}{\frac{1}{2} - \tau} \right)^2$$

samples are available for $\alpha, \beta \in \left(0, \frac{1}{\sqrt{2\pi e}}\right]$, *the Test function from the Gauss algorithm applied on* $LPN^n_{k,\tau}$ *accepts the correct solution with probability at least* $1-\alpha$, *and rejects incorrect solutions with probability at least* $1-\beta$.

Proof. A correct $\boldsymbol{s}'$ input to the Test algorithm, means that $\boldsymbol{e} = \boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}$ follows the Binomial distribution Bin^s_τ, i.e., $\boldsymbol{e}_i \sim \mathrm{Ber}_\tau$, $i \in \{1, \ldots, s\}$. Then $\mathrm{E}(\boldsymbol{e}_i) = \tau$ and $\mathrm{Var}(\boldsymbol{e}_i) = \tau(1-\tau)$. Using the same approach as in proposition 1 for

$$Z = \frac{\sqrt{s}(S_s - \mathrm{E}(\boldsymbol{e}_i))}{\sqrt{\mathrm{Var}(\boldsymbol{e}_i)}},$$

and $S_s = \frac{\boldsymbol{e}_1 + \cdots + \boldsymbol{e}_s}{s}$, and we obtain

$$Pr\left[\mathrm{HW}(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \geqslant c\right] \leqslant \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2s} \cdot \frac{(c - s\tau)^2}{\tau(1-\tau)}\right) \tag{B.2}$$

Taking $c = s\tau + \sqrt{2s\tau(1-\tau)\ln\frac{1}{\sqrt{2\pi}\alpha}}$ (similarly as in [140]), equation (B.2) turns into $Pr\left[\mathrm{HW}(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \geqslant c\right] \leqslant \alpha$. For the chosen c, the probability that a

correct $\boldsymbol{s}'$ will produce an error $\boldsymbol{e}$ of larger weight than c can be made negligible. Therefore we can use this c as a threshold value in the Test algorithm.

We estimate $Pr\left[\mathrm{HW}(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \leqslant c\right]$ similarly,

$$Pr\left[\mathrm{HW}(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \leqslant c\right] \leqslant \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(s-2c)^2}{2s}\right) \tag{B.3}$$

Taking $s = \left(\frac{\sqrt{2\tau(1-\tau)\ln(\frac{1}{\sqrt{2\pi}\alpha})} + \sqrt{\frac{1}{2}\ln(\frac{1}{\sqrt{2\pi}\beta})}}{\frac{1}{2}-\tau}\right)^2$ and the previously found c, equation (B.3) turns into $Pr\left[\mathrm{HW}(\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{c}) \leqslant c\right] \leqslant \beta$. With this we have also estimated the required amount of samples needed for the Test function. □

In order to compare fairly the two decoding algorithms, the errors γ for WHT and $\alpha + \beta$ for Gauss should be approximately the same. For simplicity we take $\alpha = \beta = \gamma = 1/(2^k\sqrt{2\pi})$. Then we get approximately the same amount of needed samples i.e.

$$s_G \approx \frac{8k\ln 2}{(1-2\tau)^2},\ s_{WHT} \approx \frac{4k\ln 2}{(1-2\tau)^2}$$

This shows that we can ignore the sample number s from the time and memory expressions of both decoding algorithms and look at them as functions in k' and τ'. Interestingly, the time complexity of both algorithms is exponential in k', but with different bases: 2 for WHT and $\left((1-\tau)^{-1}\right)$ for Gauss. As we add more reduction steps, $\left((1-\tau)^{-1}\right)$ grows and the Gauss algorithm quickly overruns WHT. Hence, we can expect that having more reduction steps favors WHT instead of Gauss as this reduces the LPN problem, and it becomes more likely that we can fit the WHT algorithm in memory. This observation is shown in figure B.2 and further confirmed in appendix B.2.3.

B.4 Combining code-reduce with Gauss

In [140] it was suggested that the low-memory Gauss decoding algorithms can be combined with various reduction algorithms. The intuitive combination with the code-reduce reduction that uses little memory and does not consume any samples, would appear to make sense. Using Pooled-Gauss,

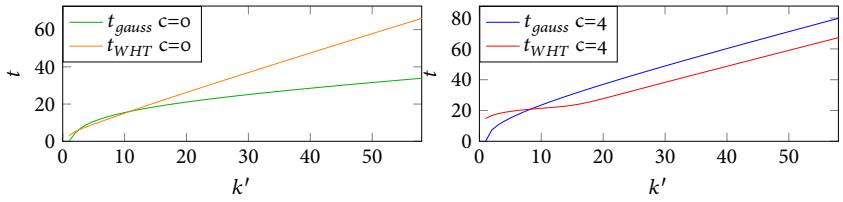


Figure B.2: Comparing WHT and Gauss for different $\delta' = \delta^{2^c}$. c indicates the number of reduction steps.

Algorithm B.4: Coded Pooled Gauss

Input: $n = k + k^2 \log_2^2 k + s$ samples from $\mathcal{O}^{\mathsf{LPN}}_{\boldsymbol{s},\tau}$,
a $[k, k']$ code C with generator matrix G
Output: Linear relations on $\boldsymbol{s}$

`sparse-secret()`
`code-reduce`(k, k', C)
$s \leftarrow$ `Pooled-Gauss`(k')
return $\boldsymbol{s}'$ of size k' such that $\boldsymbol{s}G^T = \boldsymbol{s}'$

a variant that does not regenerate samples, this combination looks like algorithm B.4. However, we will show that this approach is not more viable than just applying `Pooled-Gauss` to the full problem. Even hypothetical codes that reach the Hamming Bound [172] don't have good enough bc that makes Coded (Pooled) Gauss better.

In our analysis we assume that we can decode a sample in insignificant time. We explore whether even under this assumption, Coded Gauss can be competitive. In practice, constant decoding time is only feasible for (concatenations of) small codes. Those are not the best possible covering codes theoretically.

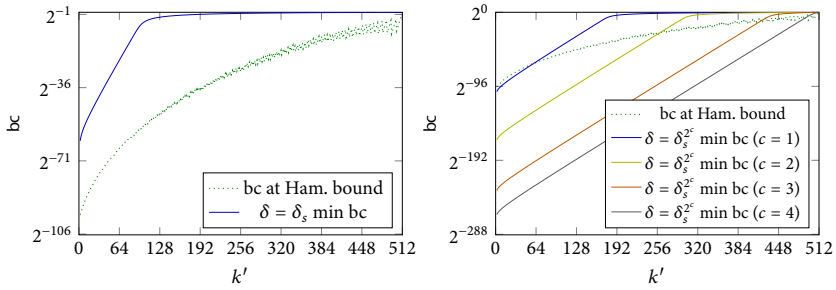
B.4.1 Analysis of the required bias of the code

In order for Coded Pooled Gauss to have advantage over Plain Pooled Gauss, we need the time complexity of Coded Pooled Gauss to be better, i.e.

$$\frac{(k^3 + ks)\log_2^2 k}{(\frac{1}{2} + \frac{1}{2}\delta)^k} \geqslant \frac{(k'^3 + k's)\log_2^2 k'}{(\frac{1}{2} + \frac{1}{2}\delta\mathrm{bc})^{k'}} + s + n. \tag{B.4}$$

Recall that theorem B.2 bounds the bc of any $[k, k']$ code and that the bound is met for perfect or quasi-perfect codes. Combining it with the Hamming bound, reached by perfect codes, $(2^{k'} \geqslant \sum_{w=0}^{R} \binom{k}{w})$, we can compute the upper bound on bc for any $[k, k']$ code. In turn, this gives us the best possible time complexity for Coded (Pooled) Gauss using any $[k, k']$ code. Unfortunately, our calculations show (see figure B.3(a)) that the required bc can not be reached even for codes on the Hamming bound. This implies that Coded (Pooled) Gauss is always worse than immediately applying (Pooled) Gauss.

Note that here, since we only combine code-reduce and Gauss we have $\delta = \delta_s$ (the sparse-secret transformation is performed right before code-reduce). However, in order for the code-reduce step to be worth applying we actually need $\delta < \delta_s$. This corresponds to applying other reduction steps in between sparse-secret and code-reduce. Figure B.3(b) depicts this scenario. As before, c indicates the number of reduction steps. Note that as c increases, so does the possible advantage of applying code-reduce.



(a) $\delta = \delta_s; \tau = \frac{1}{8}$ (b) $\delta = \delta_s^{2^c} < \delta_s = \frac{3}{4}$

Figure B.3: Minimal bc for Coded Gauss to be faster than just applying Gauss and the bc obtained at the Hamming bound. (b) actually requires additional reduction steps before code-reduce.

The previous analysis does not give the full picture. We have neglected the running time of the in-between steps for the sake of argument and to show that the only favorable case involves several reduction steps before Coded Gauss.

B.4.2 Memory Cost

The samples used by Gauss to test if candidate $\boldsymbol{s}'$ is correct greatly contributes to its memory consumption. With small bias, Gauss is not memory-efficient. For quite realistic $\delta \cdot \mathrm{bc} \approx 10^{-6}$ and $k' \gtrapprox 16$, Gauss needs many terabytes of memory. When $\delta \cdot \mathrm{bc} \approx 10^{-7}$, it even crosses into the exabytes. This further limits realistic attacks. We note that relaxing the failure probability reduces the memory consumption, though not by many orders of magnitude. However, this could make the difference for a practical attack to fit in memory.

B.5 Finding memory restricted reduction chains

Our main goal here is to find the best reduction chains in the spirit of [70] but under memory constraints. As a first step, we modified the chain finding algorithm from [70] to only allow branches to be taken if the memory consumed by the reduction or decoding is below a set limit. Although in theory this approach should yield the best chain in the end, it is extremely inefficient, time-consuming, and does not scale well. This was especially visible after adding new reduction steps to the algorithm. However, we noticed that the automaton can be greatly simplified due to many impossible branches and some clear optimization steps due to the memory restrictions.

Proposition 3. *The sequence* `sum-up-reduce` $\rightarrow$ `drop-reduce` *can never occur in the best reduction chain for solving a given* $\mathsf{LPN}_{k_0,\tau_0}$ *search problem under memory constrains.*

Proof. We will prove the claim for `sum-up- reduce=xor-reduce`. The rest can be shown very similarly. Suppose that after a number of reduction steps we need to reduce the problem $\mathsf{LPN}_{k,\tau}$. Using the sequence `xor-reduce` $\rightarrow$ `drop-reduce`, we can reduce it first to $\mathsf{LPN}_{k-b,\tau'}$ using `xor-reduce`, and then to $\mathsf{LPN}_{k',\tau'}$ using `drop-reduce`. Here $\tau' = \left(1-(1-2\tau)^2\right)/2$ and $b \in [0, k-k']$.

Table B.1: Complexities of solving $\mathsf{LPN}_{k,\tau}$ in restricted memory

	k =	128			256			384			512		
τ	m =	40	60	80	40	60	80	40	60	80	40	60	80
0.05	**Our work**	$\mathbf{26}^W$	$\mathbf{26}^W$	$\mathbf{26}^W$	$\mathbf{38}^W$	$\mathbf{38}^W$	$\mathbf{38}^W$	$\mathbf{58}^G$	$\mathbf{49}^W$	$\mathbf{49}^W$	$\mathbf{68}^W$	$\mathbf{58}^W$	$\mathbf{58}^W$
	Hybrid / MMT	37 34	37 34	37 34	54 40	54 40	54 40	70 48	68 48	68 48	87 57	87 57	84 57
0.10	**Our work**	$\mathbf{31}^W$	$\mathbf{31}^W$	$\mathbf{31}^W$	$\mathbf{50}^W$	$\mathbf{46}^W$	$\mathbf{46}^W$	$\mathbf{81}^G$	$\mathbf{60}^W$	$\mathbf{60}^W$	$\mathbf{99}^W$	$\mathbf{92}^W$	$\mathbf{73}^W$
	Hybrid / MMT	41 38	41 38	41 38	76 53	61 53	61 53	106 70	106 70	74 70	136 87	136 87	101 87
0.125	**Our work**	$\mathbf{33}^W$	$\mathbf{33}^W$	$\mathbf{33}^W$	$\mathbf{56}^W$	$\mathbf{49}^W$	$\mathbf{49}^W$	$\mathbf{92}^G$	$\mathbf{71}^W$	$\mathbf{64}^W$	$\mathbf{114}^W$	$\mathbf{105}^W$	$\mathbf{78}^W$
	Hybrid / MMT	41 41	41 41	41 41	86 61	61 61	61 61	121 81	110 81	81 81	157 102	157 102	101 102
0.25	**Our work**	$\mathbf{38}^W$	$\mathbf{38}^W$	$\mathbf{38}^W$	$\mathbf{102}^G$	$\mathbf{58}^W$	$\mathbf{58}^W$	$\mathbf{140}^G$	$\mathbf{92}^W$	$\mathbf{77}^W$	$\mathbf{179}^G$	$\mathbf{186}^G$	$\mathbf{115}^W$
	Hybrid / MMT	47 57	47 57	47 57	113 95	69 95	69 95	175 134	135 134	104 134	230 172	202 172	171 172
0.40	**Our work***	$\mathbf{51}^W$	$\mathbf{48}^W$	$\mathbf{48}^W$	$\mathbf{136}^G$	$\mathbf{84}^W$	$\mathbf{71}^W$	$\mathbf{189}^G$	$\mathbf{176}^G$	$\mathbf{116}^W$	$\mathbf{245}^G$	$\mathbf{241}^G$	$\mathbf{209}^W$
	Hybrid / MMT	62 75	57 75	57 75	129 132	93 132	81 132	197 189	160 189	139 189	264 245	228 245	207 245

^{G}: Gauss decoding method. ^{W}: WHT decoding method.
Hybrid / MMT per [140], generated by a version of their script that contains a bug-fix acknowledged by the authors.
*: 0.40 results do not use random codes from [70].

The sequence takes time

$$t = k \max\{n, \frac{n(n-1)}{2^{b+1}}\} + (k-b)\frac{n(n-1)}{2^{b+1}}$$

and memory $m = \max\{kn, (k-b)\frac{n(n-1)}{2^{b+1}}\}$. For some constants A, B, C, these can be written as functions in b as $t(b) = A + n(n-1)\frac{Bk-b}{2^{b+1}}$ and $m(b) = A + Cn(n-1)\frac{k-b}{2^{b+1}}$. It is easy to see that both functions are strictly decreasing in b, so the minimum on $[0, k-k']$ is achieved when $b = k-k'$. Note further that the number of remaining samples does not depend on b, so the choice of b does not affect subsequent reduction steps. Summarizing, in the best chain any sequence `xor-reduce` → `drop-reduce` collapses to just `xor-reduce`. □

We also looked into the sequences `code-reduce` → `drop-reduce` and `code-reduce` → `sum-up-reduce`. However, due to the very complex relation between the time complexity and the bias bc of the code, we could not make a compact analysis similar to proposition 3. Instead, we performed an extensive set of experiments where we tested the appearance of these sequences just before a decoding algorithm is applied, i.e., sequences of type `code-reduce` → `reduce` → `decode`. Our experiments showed that such sequences never appear, and that they collapse to `code-reduce` → `decode`. Therefore, we decided to not allow in the automaton any other reduction steps after `code-reduce`.

As a final modification, we put drop-reduce as a first step. This is a logical choice in a memory restricted environment and has been used in previous works as well [140]. Samples can be generated on the fly and discarded immediately if they don't satisfy the requirements of drop-reduce. This creates a time-memory trade-off since only the reduced samples from drop-reduce remain in memory.

We updated the automaton from figure B.1 using our findings, and what we get is depicted in figure B.4.

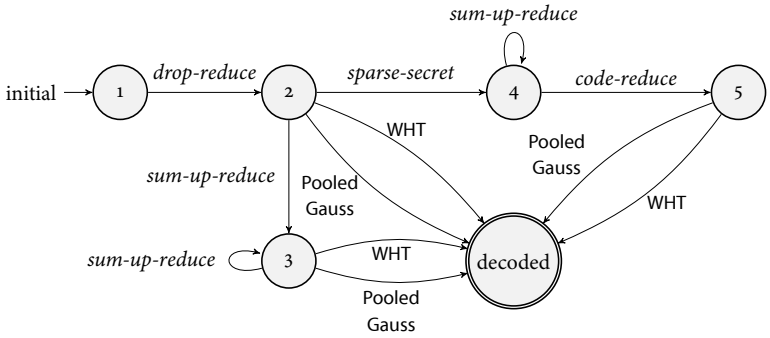


Figure B.4: The updated automaton using the results from appendix B.5. The notation is the same as in figure B.1.

B.5.1 Experimental Results

We applied our algorithm to find reduction chains that fit in 2^{40} (128 GiB), 2^{60} (128 PiB) and 2^{80} (128 ZiB) bits of memory. 2^{40} bits is an amount of memory that is readily available from server vendors in common configurations. 2^{60} bits is a much larger, but not necessarily impractical amount of memory. A top supercomputer, Summit, has over 250 PB of storage [278].[1] Finally, 2^{80} is included to give some safety margin.

In table B.1, we show that solving most LPN instances is fastest using the WHT decoding algorithm. Only when we get severely memory-restricted, does the algorithm find chains with Gauss. This improves upon the results of [140], who were not able to fit any WHT-based algorithm in 2^{60} bits of

[1] While this is networked storage, Summit nodes have over 10 PB of local storage combined. 128 PB of RAM is probably within reach in the near future.

memory. We also see that the found reduction and decoding chain is able to recover $\mathsf{LPN}_{256,0.25}$ in 2^{58} time. This is a significant improvement on the complexity of 2^{63} for their best attack on $\mathsf{LPN}_{256,0.25}$, which involved a quantum algorithm. Going up to $m = 80$ shows that more memory does not necessarily allow for better algorithms. This is probably related to the fact that the most significant factor affecting memory requirements is the number of samples, which in turn affects the required time.

B.6 Practical attack on LPN

Using our results, with memory limit $m = 39$, we have executed several attacks. The results are listed in table B.2. We implemented the reductions and solving algorithms in Rust. We hope these results and memory bounds are meaningful and illustrate what some time complexities mean in practice. Experiment were run on a computer with 192 GB RAM and two Intel Xeon Gold 6230s totaling 80 threads. Their runtime, due to the tight memory restriction, is dominated by drop-reduce, so we also give the number of bits dropped.

Table B.2: Solved $\mathsf{LPN}_{k,\tau}$ instances with $m = 39$

k	τ	Exp. time	Init. samples	drop bits	runtime
190	1/8	$2^{40.9}$	$2^{31.0}$	7	33 minutes
200	1/8	$2^{44.4}$	$2^{31.2}$	12	290 minutes
150	1/4	$2^{44.5}$	$2^{31.4}$	12	281 minutes
154	1/4	$2^{48.4}$	$2^{31.4}$	16	3 741 minutes

We see that our results scale in line with the theoretical complexity. For $k = 512$, we see that for $\tau = \frac{1}{8}$ the theoretical time complexity is $t = 2^{114}$. Extrapolating to this complexity, we would expect to need 2^{77} minutes to run an attack in practice, with our implementation. Of course, both extrapolations assume the exact same hardware and software for attacking a problem of this size. There is potential for acceleration by using GPUs or trivially distributing, e.g., drop-reduce over multiple computers. We leave this for future work.

B.7 Conclusion

In this chapter we focused on practical consideration for solving the LPN problem, in particular the issue of memory consumption. We improved the state-of-the-art by modifying and enhancing the Coded-BKW algorithm to work under various memory constraints. Our analysis of Coded (Pooled) Gauss disproved that this intuitive combination of low-memory algorithms is generally feasible. We further showed that when combined with several reduction steps, Gauss is generally always worse than using WHT, especially for practical parameters. The practicality of our approach was demonstrated by mounting attacks on the largest parameters reported so far, in only 2^{39} bits of memory.

Lists of abbreviations

TLS message abbreviations

CH	ClientHello message
SH	ServerHello message
HRR	HelloRetryRequest message
CCS	ChangeCipherSuite message
CKC	ClientKemCiphertext message
SKC	ServerKemCiphertext message
EE	EncryptedExtensions message
CR	CertificateRequest message
CRT	Certificate message
SCRT	ServerCertificate message
CRTV	CertificateVerify message
SCV	ServerCertificateVerify message
CCRT	ClientCertificate message
CCV	ClientCertificateVerify message
FIN	Finished message
SF	ServerFinished message
CF	ClientFinished message

TLS key abbreviations

AHS	Authenticated Handshake Secret key
CAHTS	Client Authenticated Handshake Secret key
CATS	Client Application Traffic Secret key
CHTS	Client Handshake Secret key
dAHS	Derived Authenticated Handshake Secret key
dES	Derived Early Secret key
dHS	Derived Handshake Secret key
EMS	Exporter Main Secret key
ES	Early Secret key
ETS	Early Traffic Secret key
HS	Handshake Secret key
MS	Main Secret key
RMS	Resumption Main Secret key
SAHTS	Server Authenticated Handshake Secret key
SATS	Server Application Traffic Secret key
SHTS	Server Handshake Secret key

Abbreviations

AEAD	authenticated encryption with associated data
AKE	authenticated key exchange
CA	certificate authority
CSR	certificate-signing request
CT	certificate transparency
DC	delegated credential
DH	Diffie–Hellman key exchange
ECDH	elliptic-curve Diffie–Hellman
FO	Fujisaki–Okamoto
IETF	Internet Engineering Task Force
IoT	Internet-of-Things
IRTF	Internet Research Task Force
KAT	known-answer test
KDF	key-derivation function
KEM	key encapsulation mechanism
LPN	Learning Parity with Noise
MAC	message-authentication code
MLWE	module-learning with errors
MSS	maximum segment size
NIKE	non-interactive key exchange
NIST	United States National Institute of Standards and Technology
OCSP	online certificate status protocol

PKI	public-key infrastructure
PQC	post-quantum cryptography
PSK	pre-shared key
RFC	Request For Comments, an IETF or IRTF publication often containing a standard.
RTT	round-trip time
TLS	Transport Layer Security

Bibliography

[1] 3rd Generation Partnership Project (3GPP). *The Mobile Broadband Standard Specification Release 13*. Tech. rep. 3GPP, Sept. 2015. URL: www.3gpp.org/ftp/Information/WORK_PLAN/Description_Releases/Rel-13_description_20150917.zip (cit. on p. 318).

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice." In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 5–17. DOI: 10.1145/2810103.2813707. URL: weakdh.org/ (cit. on pp. 33, 93).

[3] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. *HQC*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions, archived at https://web.archive.org/web/20230501144806/https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions on May 1, 2023 (cit. on pp. 15, 36, 282).

[4] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NISTIR 8309. National Institute of Standards and Technology, July 2020. DOI: NIST.IR.8309 (cit. on p. 341).

[5] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep.

NISTIR 8240. National Institute of Standards and Technology, Jan. 2019. DOI: 10.6028/NIST.IR.8240 (cit. on p. 341).

[6] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. NISTIR 8413. Updated version. National Institute of Standards and Technology, Sept. 26, 2022. DOI: 10.6028/NIST.IR.8413-upd1 (cit. on pp. 205, 342).

[7] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions, archived at https://web.archive.org/web/20230501144806/https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions on May 1, 2023 (cit. on pp. 15, 35, 208, 249, 282).

[8] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. "Post-quantum Key Exchange - A New Hope." In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 10–12, 2016, pp. 327–343. IACR ePrint: ia.cr/2015/1092. URL: www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim (cit. on p. 49).

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography." In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 31–Nov. 2, 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078 (cit. on p. 338).

[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations." In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 10–12, 2016, pp. 53–70. URL: www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida (cit. on pp. 347, 351).

[11] Ross J. Anderson. "Why Cryptosystems Fail." In: *ACM CCS 93: 1st Conference on Computer and Communications Security*. Ed. by Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby. Fairfax, Virginia, USA: ACM Press, Nov. 3–5, 1993, pp. 215–227. DOI: 10.1145/168588.168615 (cit. on p. 336).

[12] *Another fraudulent certificate raises the same old questions about certificate authorities*. Ars Technica. Aug. 30, 2011. URL: arstechnica.com/information-technology/2011/08/earlier-this-year-an-iranian/ (visited on Oct. 6, 2022), archived at https://web.archive.org/web/20230408001839/https://arstechnica.com/information-technology/2011/08/earlier-this-year-an-iranian/ on Apr. 8, 2023 (cit. on p. 74).

[13] American National Standards Institute, Inc. *ANSI X9.62 Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)*. Nov. 16, 2005. URL: standards.globalspec.com/std/1955141/ANSI%20X9.62 (cit. on p. 27).

[14] *ANSSI views on the post-quantum cryptography transition*. Agence nationale de la sécurité des systèmes d'information. Jan. 4, 2022. URL: www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/ (visited on Mar. 3, 2023), archived at https://web.archive.org/web/20230505121357/https://www.ssi.gouv.fr/en/publication/anssi-views-on-the-post-quantum-cryptography-transition/ on May 5, 2023 (cit. on pp. 227, 268, 293).

[15] *Apple's Certificate Transparency policy*. Apple. Mar. 11, 2021. URL: support.apple.com/en-gb/HT205280 (visited on Oct. 6, 2022), archived at https://web.archive.org/web/20230329153011/https://support.apple.com/en-gb/HT205280 on Mar. 29, 2023 (cit. on pp. 74, 238).

[16] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. "Fast Cryptographic Primitives and Circular-Secure Encryption Based on Hard Learning Problems." In: *Advances in Cryptology – CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 16–20, 2009, pp. 595–618. DOI: 10.1007/978-3-642-03356-8_35 (cit. on p. 391).

[17] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. *BIKE*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-

cryptography/round-4-submissions, archived at https://web.archive.org/web/20230501144806/https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions on May 1, 2023 (cit. on pp. 15, 36, 50, 208, 249, 283).

[18] Nimrod Aviram, Benjamin Dowling, Ilan Komargodski, Kenneth G. Paterson, Eyal Ronen, and Eylon Yogev. *Practical (Post-Quantum) Key Combiners from One-Wayness and Applications to TLS*. Cryptology ePrint Archive, Report 2022/065. Feb. 25, 2022. IACR ePrint: ia.cr/2022/065 (cit. on p. 25).

[19] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. "CTIDH: faster constant-time CSIDH." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), pp. 351–387. DOI: 10.46586/tches.v2021.i4.351-387. IACR ePrint: ia.cr/2021/633 (cit. on p. 241).

[20] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. "SoK: Computer-Aided Cryptography." In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008. IACR ePrint: ia.cr/2019/1393 (cit. on pp. 164, 357).

[21] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. *Hybrid Public Key Encryption*. RFC 9180. Feb. 2022. DOI: 10.17487/RFC9180 (cit. on p. 196).

[22] Richard Barnes and Owen Friel. *Usage of PAKE with TLS 1.3*. Internet-Draft draft-barnes-tls-pake-04. Work in Progress. Internet Engineering Task Force, July 2018. 11 pp. URL: datatracker.ietf.org/doc/draft-barnes-tls-pake/04/ (cit. on p. 42).

[23] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. *Automatic Certificate Management Environment (ACME)*. RFC 8555. Mar. 2019. DOI: 10.17487/RFC8555 (cit. on p. 73).

[24] Richard Barnes, Subodh Iyengar, Nick Sullivan, and Eric Rescorla. *Delegated Credentials for TLS and DTLS*. RFC 9345. July 2023. DOI: 10.17487/RFC9345 (cit. on p. 302).

[25] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. "High-Assurance Cryptography in the Spectre Era." In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 1884–1901. DOI: 10.1109/SP40001.2021.00046. IACR ePrint: ia.cr/2020/1104 (cit. on p. 338).

[26] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. "Computer-Aided Security Proofs for the Working Cryptographer." In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Project website: https://github.com/EasyCrypt/easycrypt. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 14–18, 2011, pp. 71–90. DOI: 10.1007/978-3-642-22792-9_5 (cit. on pp. 180, 357).

[27] David Basin, Cas Cremers, Jannik Dreier, Robert Künneman, Simon Meier, Ralf Sasse, and Benedikt Schmidt. *Tamarin Prover*. 2022. URL: tamarin-prover.github.io, archived at https://web.archive.org/web/20230507185025/https://tamarin-prover.github.io/ on May 7, 2023 (cit. on pp. 9, 164, 165, 361).

[28] David A. Basin, Jannik Dreier, and Ralf Sasse. "Automated Symbolic Proofs of Observational Equivalence." In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 1144–1155. DOI: 10.1145/2810103.2813662. URL: hal.science/hal-01337409/ (cit. on p. 179).

[29] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision-Resistance." In: *Advances in Cryptology – CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 20–24, 2006, pp. 602–619. DOI: 10.1007/11818175_36. IACR ePrint: ia.cr/2006/043 (cit. on pp. 25, 103).

[30] Mihir Bellare. "Practice-Oriented Provable-Security." In: *ISW 97: First International Workshop on Information Security*. Ed. by Ivan Damgård. Vol. 1561. Lecture Notes in Computer Science. Springer, 1998, pp. 221–231. DOI: 10.1007/BFb0030423. URL: cseweb.ucsd.edu/~mihir/papers/isw-pops.pdf (cit. on p. 179).

[31] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols (Extended Abstract)." In: *30th Annual ACM Symposium on Theory of Computing*. Dallas, TX, USA: ACM Press, May 23–26, 1998, pp. 419–428. DOI: 10.1145/276698.276854. IACR ePrint: ia.cr/1998/008 (cit. on p. 56).

[32] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution." In: *Advances in Cryptology – CRYPTO'93*. Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 22–26, 1993, pp. 232–249. DOI: 10.1007/3-540-48329-2_21 (cit. on pp. 91, 98, 99, 110, 133, 144).

[33] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs." In: *Advances in Cryptology – EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. Lecture Notes in Computer Science. St. Petersburg, Russia: Springer, Heidelberg, Germany, May 28–June 1, 2006, pp. 409–426. DOI: 10.1007/11761679_25. IACR ePrint: ia.cr/2004/331 (cit. on pp. 124, 130, 152, 159).

[34] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540 (cit. on p. 67).

[35] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. ""Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way." In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Busan, South Korea: Springer, Heidelberg, Germany, Sept. 23–26, 2014, pp. 75–92. DOI: 10.1007/978-3-662-44709-3_5. IACR ePrint: ia.cr/2014/161 (cit. on p. 279).

[36] David Benjamin. *Additional TLS 1.3 results from Chrome*. Posting on the TLS mailing list. Dec. 18, 2017. URL: www.ietf.org/mail-archive/web/tls/current/msg25168.html (visited on Oct. 18, 2022), archived at https://web.archive.org/web/20230422015956/https://mailarchive.ietf.org/arch/msg/tls/i9blmvG2BEPf1s1OJkenHknRw9c/ on May 22, 2023 (cit. on p. 201).

[37] David Benjamin, Devon O'Brien, and Bas Westerbaan. *Merkle Tree Certificates for TLS*. Internet-Draft draft-davidben-tls-merkle-tree-certs-00. Work in Progress. Internet Engineering Task Force, Mar. 2023. 45 pp. URL: datatracker.ietf.org/doc/draft-davidben-tls-merkle-tree-certs/00/ (cit. on p. 363).

[38] David Benjamin and Eric Rescorla. Presentation before the TLS WG at IETF 100. Sept. 16, 2017. URL: datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/ (visited on Oct. 18, 2022) (cit. on p. 201).

[39] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. "Strengths and Weaknesses of Quantum Computing." In: *SIAM Journal on Computing* 26.5 (1997), pp. 1510–1523. DOI: 10.1137/S0097539796300933. arXiv: quant-ph/9701001 (cit. on p. 34).

[40] Daniel J. Bernstein. *Cryptographic competitions*. Cryptology ePrint Archive, Report 2020/1608. Dec. 27, 2020. IACR ePrint: ia.cr/2020/1608 (cit. on p. 238).

[41] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records." In: *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. New York, NY, USA: Springer, Heidelberg, Germany, Apr. 24–26, 2006, pp. 207–228. DOI: 10.1007/11745853_14 (cit. on pp. 49, 206).

[42] Daniel J. Bernstein. *Re: [pqc-forum] new quantum cryptanalysis of CSIDH*. Posting to the NIST pqc-forum mailing list. June 19, 2019. URL: groups.google.com/a/list.nist.gov/forum/\#!original/pqc-forum/svm1kDy6c54/0gFOLitbAgAJ (visited on May 5, 2023), archived at https://web.archive.org/web/20230505142107/https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/svm1kDy6c54/m/0gFOLitbAgAJ on May 5, 2023 (cit. on pp. 29, 244).

[43] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. "OpenSSLNTRU: Faster post-quantum TLS key exchange." In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 10–12, 2022, pp. 845–862. IACR ePrint: ia.cr/2021/826. URL: www.usenix.org/conference/usenixsecurity22/presentation/bernstein (cit. on p. 343).

[44] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures." In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Nara, Japan: Springer, Heidelberg, Germany, Sept. 28–Oct. 1, 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9. IACR ePrint: ia.cr/2011/368 (cit. on p. 27).

[45] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. "SPHINCS: Practical Stateless Hash-Based Signatures." In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 26–30, 2015, pp. 368–397. DOI: 10.1007/978-3-662-46800-5_15. IACR ePrint: ia.cr/2014/795 (cit. on p. 387).

[46] Daniel J. Bernstein and Andreas Hülsing. "Decisional Second-Preimage Resistance: When Does SPR Imply PRE?" In: *Advances in Cryptology – ASIACRYPT 2019, Part III*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11923. Lecture Notes in Computer Science. Kobe, Japan: Springer, Heidelberg, Germany, Dec. 8–12, 2019, pp. 33–62. DOI: 10.1007/978-3-030-34618-8_2. IACR ePrint: ia.cr/2019/492 (cit. on p. 239).

[47] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. "The SPHINCS$^+$ Signature Framework." In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. London, UK: ACM Press, Nov. 11–15, 2019, pp. 2129–2146. DOI: 10.1145/3319535.3363229. IACR ePrint: ia.cr/2019/1086 (cit. on pp. 370, 371, 387).

[48] Daniel J. Bernstein and Tanja Lange, eds. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. 2018. URL: bench.cr.yp.to/ (visited on May 6, 2023), archived at https://web.archive.org/web/20230506143033/https://bench.cr.yp.to/ on May 6, 2023 (cit. on pp. 335, 340, 352).

[49] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. "Quantum Circuits for the CSIDH: Optimizing Quantum Evaluation of Isogenies." In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Heidelberg, Germany, May 19–23, 2019, pp. 409–441. DOI: 10.1007/978-3-030-17656-3_15. IACR ePrint: ia.cr/2018/1059 (cit. on pp. 29, 244).

[50] Ward Beullens. "Breaking Rainbow Takes a Weekend on a Laptop." In: *Advances in Cryptology – CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 15–18, 2022, pp. 464–479. DOI: 10.1007/978-3-031-15979-4_16. IACR ePrint: ia.cr/2022/214 (cit. on pp. 15, 303).

[51] Ward Beullens. "Improved Cryptanalysis of UOV and Rainbow." In: *Advances in Cryptology – EUROCRYPT 2021, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Heidelberg, Germany, Oct. 17–21, 2021, pp. 348–373. DOI: 10.1007/978-3-030-77870-5_13. IACR ePrint: ia.cr/2020/1343 (cit. on p. 15).

[52] Ward Beullens, Ming-Shing Chen, Jintai Ding, Matthias J. Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, Chengdong Tao, and Bo-Yin Yang. *UOV parameters*. Posting on the NIST pqc-forum mailing list. June 19, 2022. URL: groups.google.com/a/list.nist.gov/g/pqc-forum/c/B1RFy31rH8I/m/yIqAt7dABAAJ (visited on Oct. 12, 2022), archived at https://web.archive.org/web/20230509150802/https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/B1RFy31rH8I/m/yIqAt7dABAAJ on May 9, 2023 (cit. on p. 36).

[53] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. "Oil and Vinegar: Modern Parameters and Implementations." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (2023), pp. 321–365. DOI: 10.46586/tches.v2023.i3.321-365. IACR ePrint: ia.cr/2023/059 (cit. on p. 15).

[54] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercauteren. *LUOV*. Tech. rep. National Institute of Standards and Technology, 2019. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions, archived at https://web.archive.org/web/20230509100012/https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions on May 9, 2023 (cit. on p. 36).

[55] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate." In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 483–502. DOI: 10.1109/SP.2017.26. URL: hal.science/hal-01528752 (cit. on pp. 41, 48, 59, 164).

[56] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella-Béguelin. "Downgrade Resilience in Key-Exchange Protocols." In: *2016 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2016, pp. 506–525. DOI: 10.1109/SP.2016.37. IACR ePrint: ia.cr/2016/072 (cit. on p. 93).

[57] Karthikeyan Bhargavan, Vincent Cheval, and Christopher A. Wood. "A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello." In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 7–11, 2022, pp. 365–379. DOI: 10.1145/3548606.3559360. URL: hal.science/hal-03922516/ (cit. on p. 186).

[58] Jean-François Biasse, Annamaria Iezzi, and Michael J. Jacobson Jr. "A Note on the Security of CSIDH." In: *Progress in Cryptology - INDOCRYPT 2018: 19th International Conference in Cryptology in India*. Ed. by Debrup Chakraborty and Tetsu Iwata. Vol. 11356. Lecture Notes in Computer Science. New Delhi, India: Springer, Heidelberg, Germany, Dec. 9–12, 2018, pp. 153–168. DOI: 10.1007/978-3-030-05378-9_9. arXiv: 1806.03656 (cit. on pp. 29, 244).

[59] Eric W. Biederman and Nicolas Dichtel. *ip-netns(8)*. `man ip netns`. Jan. 2013. URL: man7.org/linux/man-pages/man8/ip-netns.8.html (cit. on p. 202).

[60] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange." In: *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*. Ed. by Jintai Ding and Rainer Steinwandt. Chongqing, China: Springer, Heidelberg, Germany, May 8–10, 2019, pp. 206–226. DOI: 10.1007/978-3-030-25510-7_12. IACR ePrint: ia.cr/2018/903 (cit. on p. 64).

[61] Moritz Birghan and Thyla van der Merwe. "A Client-Side Seat to TLS Deployment." In: *2022 SecWeb Workshop on Designing Security for the Web (2022 IEEE S&P Workshops)*. Genoa, Italy, May 26, 2022, pp. 13–19. DOI: 10.1109/SPW54247.2022.9833861. URL: secweb.work/papers/2022/birghan2022clienttls.pdf (cit. on pp. 73, 208, 250, 362).

[62] Joseph Birr-Pixton. *Rustls: A modern TLS library in Rust*. URL: github.com/rustls/rustls (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230501005643/https://github.com/rustls/rustls on May 1, 2023 (cit. on p. 189).

[63] Joseph Birr-Pixton. *TLS performance: rustls versus OpenSSL*. July 1, 2019. URL: jbp.io/2019/07/01/rustls-vs-openssl-performance.html (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230209030133/https://jbp.io/2019/07/01/rustls-vs-openssl-performance.html on Feb. 9, 2023 (cit. on p. 189).

[64] Bruno Blanchet. "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules." In: *CSFW 14: 14th IEEE Computer Security Foundations Workshop*. Cape Breton, NS, Canada: IEEE Computer Society, June 11–13, 2001, pp. 82–96. DOI: 10.1109/CSFW.2001.930138 (cit. on p. 165).

[65] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software." In: *PLDI '03: ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, California, USA: Association for Computing Machinery, June 7–14, 2003, pp. 196–207. DOI: 10.1145/781131.781153. arXiv: cs/0701193 (cit. on p. 347).

[66] Ethan Blanton, Vern Paxson, and Mark Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681 (cit. on pp. 213, 292, 361).

[67] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. "You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries." arXiv:2107.04940. July 11, 2021. URL: arxiv.org/abs/2107.04940 (cit. on p. 337).

[68] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant learning, the parity problem, and the statistical query model." In: *32nd Annual ACM Symposium on Theory of Computing*. Portland, OR, USA: ACM Press, May 21–23, 2000, pp. 435–440. DOI: 10.1145/335305.335355. arXiv: cs/0010022 (cit. on p. 393).

[69] Sonia Bogos, Florian Tramer, and Serge Vaudenay. "On solving LPN using BKW and variants." In: *Cryptography and Communications* 8.3 (July 1, 2016), pp. 331–369. DOI: 10.1007/s12095-015-0149-2. IACR ePrint: ia.cr/2015/049 (cit. on pp. 389, 391).

[70] Sonia Bogos and Serge Vaudenay. "Optimization of LPN Solving Algorithms." In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, Dec. 4–8, 2016, pp. 703–728. DOI: 10.1007/978-3-662-53887-6_26. IACR ePrint: ia.cr/2016/288 (cit. on pp. 390–393, 395, 400, 401).

[71] Xavier Bonnetain and André Schrottenloher. "Quantum Security Analysis of CSIDH." In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Heidelberg, Germany, May 10–14, 2020, pp. 493–522. DOI: 10.1007/978-3-030-45724-2_17. IACR ePrint: ia.cr/2018/537 (cit. on pp. 29, 59, 244, 359).

[72] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. "CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM." In: *EuroS&P 2018: IEEE European Symposium on Security and Privacy*. IEEE, Apr. 24–26, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032 (cit. on p. 61).

[73] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem." In: *2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 17–21, 2015, pp. 553–570. DOI: 10.1109/SP.2015.40. IACR ePrint: ia.cr/2014/599 (cit. on p. 49).

[74] Wieb Bosma, John Cannon, and Catherine Playoust. "The Magma algebra system. I. The user language." In: *Journal of Symbolic Computation* 24.3-4 (1997), pp. 235–265. DOI: 10.1006/jsco.1996.0125 (cit. on p. 357).

[75] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. "Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4." In: *AFRICACRYPT 19: 11th International Conference on Cryptology in Africa*. Ed. by Johannes Buchmann, Abderrahmane Nitaj, and Tajje-eddine Rachidi. Vol. 11627. Lecture Notes in Computer Science. Rabat, Morocco: Springer,

Heidelberg, Germany, July 9–11, 2019, pp. 209–228. DOI: 10.1007/978-3-030-23696-0_11. IACR ePrint: ia.cr/2019/489 (cit. on p. 380).

[76] Colin Boyd, Yvonne Cliff, Juan M. Gonzalez Nieto, and Kenneth G. Paterson. "One-Round Key Exchange in the Standard Model." In: *International Journal of Applied Cryptology* 1.3 (Feb. 1, 2009), pp. 181–199. DOI: 10.1504/IJACT.2009.023466. IACR ePrint: ia.cr/2008/007. URL: eprints.qut.edu.au/29294/ (cit. on pp. 56, 61).

[77] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for Authentication and Key Establishment*. Second edition. Information Security and Cryptography. Springer, Nov. 20, 2019. ISBN: 978-3-662-58145-2. DOI: 10.1007/978-3-662-58146-9 (cit. on p. 98).

[78] Alexandre Braga and Ricardo Dahab. "A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software." In: *XV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg) 2015*. 2015, pp. 30–43. DOI: 10.5753/sbseg.2015.20083 (cit. on p. 337).

[79] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. "Towards Post-Quantum Security for Signal's X3DH Handshake." In: *SAC 2020: 27th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn. Vol. 12804. Lecture Notes in Computer Science. Halifax, NS, Canada (Virtual Event): Springer, Heidelberg, Germany, Oct. 21–23, 2020, pp. 404–430. DOI: 10.1007/978-3-030-81652-0_16. IACR ePrint: ia.cr/2019/1356 (cit. on p. 57).

[80] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical." In: *ESORICS 2011: 16th European Symposium on Research in Computer Security*. Ed. by Vijay Atluri and Claudia Díaz. Vol. 6879. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Heidelberg, Germany, Sept. 12–14, 2011, pp. 355–371. DOI: 10.1007/978-3-642-23822-2_20. IACR ePrint: ia.cr/2011/232 (cit. on p. 279).

[81] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. "Key-Schedule Security for the TLS 1.3 Standard." In: *Advances in Cryptology – ASIACRYPT 2022, Part I*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13791. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, Dec. 5–9, 2022, pp. 621–650. DOI: 10.1007/978-3-031-22963-3_21. IACR ePrint: ia.cr/2021/467 (cit. on pp. 48, 59).

[82] Christina Brzuska. "On the foundations of key exchange." PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, June 13, 2013. URL: tuprints.ulb.tu-darmstadt.de/3414/, archived at https://web.archive.org/web/20221203065816/https://tuprints.ulb.tu-darmstadt.de/3414/ on Dec. 3, 2022 (cit. on pp. 98, 109, 143).

[83] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. "Composability of Bellare-Rogaway key exchange protocols." In: *ACM CCS 2011: 18th Conference on Computer and Communications Security*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. Chicago, Illinois, USA: ACM Press, Oct. 17–21, 2011, pp. 51–62. DOI: 10.1145/2046707.2046716. URL: www.cryptoplexity.informatik.tu-darmstadt.de/media/crypt/publications_1/bfww11.pdf (cit. on pp. 98, 109, 143).

[84] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions." In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Ed. by Bo-Yin Yang. Tapei, Taiwan: Springer, Heidelberg, Germany, Nov. 29–Dec. 2, 2011, pp. 117–129. DOI: 10.1007/978-3-642-25405-5_8. IACR ePrint: ia.cr/2011/484 (cit. on p. 37).

[85] Kevin Bürstinghaus-Steinbach, Christoph Krauß, Ruben Niederhagen, and Michael Schneider. "Post-Quantum TLS on Embedded Systems: Integrating and Evaluating Kyber and SPHINCS$^+$ with mbed TLS." In: *ASIACCS 20: 15th ACM Symposium on Information, Computer and Communications Security*. Ed. by Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese. Taipei, Taiwan: ACM Press, Oct. 5–9, 2020, pp. 841–852. DOI: 10.1145/3320269.3384725. IACR ePrint: ia.cr/2020/308 (cit. on pp. 316, 320, 321).

[86] Cristiano Calcagno and Dino Distefano. "Infer: An automatic program verifier for memory safety of C programs." In: *NFM 2011: NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Project website: https://fbinfer.com. Springer, Heidelberg, Germany, 2011, pp. 459–465. DOI: 10.1007/978-3-642-20398-5_33. URL: scholar.archive.org/work/llaphwnfrjaarebob42a43x6ki (cit. on p. 347).

[87] Matt Campagna and Eric Crockett. *Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS)*. Internet-Draft draft-campagna-tls-bike-sike-hybrid-07. Work in Progress. Internet Engineering Task Force, Sept. 2, 2021. 17 pp. URL: datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-07 (cit. on p. 50).

[88] Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. "Optimizations and Practicality of High-Security CSIDH." in submission. 2023. IACR ePrint: ia.cr/2023/793 (cit. on pp. 12, 244–246).

[89] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. *GeMSS*. Tech. rep. National Institute of Standards and Technology, 2020. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, archived at https://web.archive.org/web/20230509100322/https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions on May 9, 2023 (cit. on pp. 36, 372, 378).

[90] Wouter Castryck and Thomas Decru. "An Efficient Key Recovery Attack on SIDH." In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France: Springer, Heidelberg, Germany, Apr. 23–27, 2023, pp. 423–447. DOI: 10.1007/978-3-031-30589-4_15. IACR ePrint: ia.cr/2022/975 (cit. on pp. 15, 36, 50).

[91] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. "CSIDH: An Efficient Post-Quantum Commutative Group Action." In: *Advances in Cryptology – ASIACRYPT 2018, Part III*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11274. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer, Heidelberg, Germany, Dec. 2–6, 2018, pp. 395–427. DOI: 10.1007/978-3-030-03332-3_15. IACR ePrint: ia.cr/2018/383 (cit. on pp. 5, 29, 36, 241, 244, 359).

[92] Sofía Celi, Armando Faz-Hernández, Nick Sullivan, Goutam Tamvada, Luke Valenta, Thom Wiggers, Bas Westerbaan, and Christopher A. Wood. "Implementing and Measuring KEMTLS." In: *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America*. Ed. by Patrick Longa and Carla Ràfols. Vol. 12912. Lecture Notes in Computer Science. Bogotá, Colombia: Springer, Heidelberg, Germany, Oct. 6–8, 2021, pp. 88–107. DOI: 10.1007/978-3-030-88238-9_5. IACR ePrint: ia.cr/2021/1019. URL: wggrs.nl/p/measuring-kemtls (cit. on pp. 10, 480).

[93] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. "A Tale of Two Models: Formal Verification of KEMTLS via Tamarin." In: *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part III*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. Lecture Notes in Computer

Science. Extended version available via URL and IACR ePrint. Copenhagen, Denmark: Springer, Heidelberg, Germany, Sept. 26–30, 2022, pp. 63–83. DOI: 10.1007/978-3-031-17143-7_4. IACR ePrint: ia.cr/2022/1111. URL: wggrs.nl/p/kemtls-tamarin (cit. on pp. 10, 169, 176, 479).

[94] Sofía Celi, Peter Schwabe, Douglas Stebila, Nick Sullivan, and Thom Wiggers. *KEM-based Authentication for TLS 1.3*. Internet-Draft draft-celi-wiggers-tls-authkem-01. Work in Progress. Internet Engineering Task Force, Mar. 7, 2022. 1-29. URL: datatracker.ietf.org/doc/html/draft-celi-wiggers-tls-authkem-01 (cit. on p. 363).

[95] Centraal Bureau voor de Statistiek. *Internetaankopen; persoonskenmerken*. Dutch. CBS StatLine. Oct. 21, 2022. URL: www.cbs.nl/nl-nl/cijfers/detail/84889NED (visited on Apr. 17, 2023), archived at https://web.archive.org/web/20230509131651/https://www.cbs.nl/nl-nl/cijfers/detail/84889NED on May 9, 2023 (cit. on p. 2).

[96] Centraal Bureau voor de Statistiek. *Internettoegang en internetactiviteiten; persoonskenmerken*. Dutch. CBS StatLine. Oct. 21, 2022. URL: www.cbs.nl/nl-nl/cijfers/detail/84888NED (visited on Apr. 17, 2023), archived at https://web.archive.org/web/20230207182935/https://www.cbs.nl/nl-nl/cijfers/detail/84888NED on Feb. 7, 2023 (cit. on p. 2).

[97] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. "The SQALE of CSIDH: sublinear Vélu quantum-resistant isogeny action with low exponents." In: *Journal of Cryptographic Engineering* 12.3 (Sept. 2022), pp. 349–368. DOI: 10.1007/s13389-021-00271-w. IACR ePrint: ia.cr/2020/1520 (cit. on pp. 59, 244, 359).

[98] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. *NTRU*. Tech. rep. National Institute of Standards and Technology, 2020. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, archived at https://web.archive.org/web/20230509100322/https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions on May 9, 2023 (cit. on p. 50).

[99] Ming-Shing Chen and Tung Chou. "Classic McEliece on the ARM Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.3 (2021), pp. 125–148. DOI: 10.46586/tches.v2021.i3.125-148. IACR ePrint: ia.cr/2021/492 (cit. on p. 369).

[100] Shan Chen, Samuel Jero, Matthew Jagielski, Alexandra Boldyreva, and Cristina Nita-Rotaru. "Secure Communication Channel Establishment: TLS 1.3 (over TCP Fast Open) versus QUIC." In: *Journal of Cryptology* 34.3 (July 2021), p. 26. DOI: 10.1007/s00145-021-09389-w. IACR ePrint: ia.cr/2019/433 (cit. on p. 64).

[101] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. *TCP Fast Open*. RFC 7413. Dec. 2014. DOI: 10.17487/RFC7413 (cit. on p. 64).

[102] Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. "Rainbow on Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), pp. 650–675. DOI: 10.46586/tches.v2021.i4.650-675. IACR ePrint: ia.cr/2021/532 (cit. on p. 317).

[103] *Chrome Certificate Transparency Policy*. Google. Mar. 18, 2022. URL: github.com/GoogleChrome/CertificateTransparency/blob/827e20c9e5f6486ae18cfe99e25cd1e67fde1e15/ct_policy.md (visited on Oct. 6, 2022), archived at https://web.archive.org/web/20230509140101/https://github.com/GoogleChrome/CertificateTransparency/blob/827e20c9e5f6486ae18cfe99e25cd1e67fde1e15/ct_policy.md on May 9, 2023 (cit. on pp. 74, 238).

[104] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. *Increasing TCP's Initial Window*. RFC 6928. Apr. 2013. DOI: 10.17487/RFC6928 (cit. on pp. 238, 361).

[105] Connectivity Standards Alliance. *Build with Matter*. 2022. URL: buildwithmatter.com (visited on May 16, 2022), archived at https://web.archive.org/web/20220516074741/https://csa-iot.org/all-solutions/matter/ on May 16, 2022 (cit. on p. 315).

[106] David Cooper, Daniel Apon, Quynh Dang, Michael Davidson, Morris Dworkin, and Carl Miller. *SP800-208: Recommendation for Stateful Hash-Based Signature Schemes*. DOI: 10.6028/NIST.SP.800-208 (cit. on pp. 37, 205, 239, 369).

[107] Jean-Marc Couveignes. *Hard Homogeneous Spaces*. Cryptology ePrint Archive, Report 2006/291. Aug. 24, 2006. IACR ePrint: ia.cr/2006/291 (cit. on p. 36).

[108] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3." In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 31–Nov. 2, 2017, pp. 1773–1788. DOI:

10.1145/3133956.3134063. URL: tls13tamarin.github.io (cit. on pp. 9, 41, 48, 59, 164, 167, 168, 170, 173, 174, 186).

[109] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication." In: *2016 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2016, pp. 470–485. DOI: 10.1109/SP.2016.35 (cit. on pp. 41, 59, 164).

[110] Eric Crockett, Christian Paquin, and Douglas Stebila. *Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH*. Workshop Record of the Second PQC Standardization Conference. 2019. IACR ePrint: ia.cr/2019/858. URL: csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/stebila-prototyping-post-quantum.pdf (cit. on pp. 50, 343).

[111] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. *Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches*. Cryptology ePrint Archive, Report 2020/795. Oct. 13, 2020. IACR ePrint: ia.cr/2020/795 (cit. on p. 343).

[112] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. "On the Concrete Security of TLS 1.3 PSK Mode." In: *Advances in Cryptology – EUROCRYPT 2022, Part II*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Heidelberg, Germany, May 30–June 3, 2022, pp. 876–906. DOI: 10.1007/978-3-031-07085-3_30. IACR ePrint: ia.cr/2022/246 (cit. on p. 92).

[113] Hannah Davis and Felix Günther. "Tighter Proofs for the SIGMA and TLS 1.3 Key Exchange Protocols." In: *ACNS 21: 19th International Conference on Applied Cryptography and Network Security, Part II*. Ed. by Kazue Sako and Nils Ole Tippenhauer. Vol. 12727. Lecture Notes in Computer Science. Kamakura, Japan: Springer, Heidelberg, Germany, June 21–24, 2021, pp. 448–479. DOI: 10.1007/978-3-030-78375-4_18. IACR ePrint: ia.cr/2020/1029 (cit. on pp. 48, 92).

[114] Claire Delaplace, Andre Esser, and Alexander May. "Improved Low-Memory Subset Sum and LPN Algorithms via Multiple Collisions." In: *17th IMA International Conference on Cryptography and Coding*. Ed. by Martin Albrecht. Vol. 11929. Lecture Notes in Computer Science. Oxford, UK: Springer, Heidelberg, Germany, Dec. 16–18, 2019, pp. 178–199. DOI: 10.1007/978-3-030-35199-1_9. IACR ePrint: ia.cr/2019/804 (cit. on p. 392).

[115] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer." In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 463–482. DOI: 10.1109/SP.2017.58. IACR ePrint: ia.cr/2016/1178 (cit. on p. 164).

[116] Cyprien Delpech de Saint Guilhem, Nigel P. Smart, and Bogdan Warinschi. "Generic Forward-Secure Key Agreement Without Signatures." In: *ISC 2017: 20th International Conference on Information Security*. Ed. by Phong Q. Nguyen and Jianying Zhou. Vol. 10599. Lecture Notes in Computer Science. Ho Chi Minh City, Vietnam: Springer, Heidelberg, Germany, Nov. 22–24, 2017, pp. 114–133. IACR ePrint: ia.cr/2017/853 (cit. on p. 61).

[117] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. "Deniable authentication and key exchange." In: *ACM CCS 2006: 13th Conference on Computer and Communications Security*. Ed. by Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati. Alexandria, Virginia, USA: ACM Press, Oct. 30–Nov. 3, 2006, pp. 400–409. DOI: 10.1145/1180405.1180454. IACR ePrint: ia.cr/2006/280 (cit. on pp. 69, 135).

[118] Denis Diemert and Tibor Jager. "On the Tight Security of TLS 1.3: Theoretically Sound Cryptographic Parameters for Real-World Deployments." In: *Journal of Cryptology* 34.3 (July 2021), p. 30. DOI: 10.1007/s00145-021-09388-x. IACR ePrint: ia.cr/2020/726 (cit. on pp. 48, 92).

[119] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography." In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638. URL: scholar.archive.org/work/rb3d72jxjzabre5hr7ajvrtncq (cit. on pp. 1, 5, 28).

[120] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. "Authentication and Authenticated Key Exchanges." In: *Designs, Codes and Cryptography* 2.2 (June 1992), pp. 107–125. DOI: 10.1007/BF00124891. URL: scholar.archive.org/work/gr3oigmairhfllxcoat7dbfdrq (cit. on p. 32).

[121] *Dilithium*. CRYSTALS-Dilithium team. Feb. 16, 2021. URL: pq-crystals.org/dilithium/ (visited on Mar. 6, 2023), archived at https://web.archive.org/web/20230323084032/https://pq-crystals.org/dilithium/ on Mar. 23, 2023 (cit. on pp. 219, 260, 289).

[122] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. *Rainbow*. Tech. rep. National Institute of Standards and Technology, 2020. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-

cryptography-standardization/round-3-submissions, archived at https://web.archive.org/web/20230509100322/https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions on May 9, 2023 (cit. on pp. 15, 36, 372, 377, 378).

[123] Itai Dinur, Orr Dunkelman, Nathan Keller, and Adi Shamir. "Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems." In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2012, pp. 719–740. DOI: 10.1007/978-3-642-32009-5_42. IACR ePrint: ia.cr/2012/217 (cit. on p. 392).

[124] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. "Composability and On-Line Deniability of Authentication." In: *TCC 2009: 6th Theory of Cryptography Conference*. Ed. by Omer Reingold. Vol. 5444. Lecture Notes in Computer Science. Springer, Heidelberg, Germany, Mar. 15–17, 2009, pp. 146–162. DOI: 10.1007/978-3-642-00457-5_10 (cit. on pp. 70, 135).

[125] Dolev Dolev and Andrew Yao. "On the security of public key protocols." In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650 (cit. on p. 166).

[126] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *ISOC Network and Distributed System Security Symposium – NDSS 2017*. San Diego, CA, USA: The Internet Society, Feb. 26–Mar. 1, 2017. DOI: 10.14722/ndss.2017.23160. URL: wireguard.com (cit. on p. 56).

[127] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol." In: *Journal of Cryptology* 34.4 (Oct. 2021), p. 37. DOI: 10.1007/s00145-021-09384-1. IACR ePrint: ia.cr/2020/1044 (cit. on pp. 41, 47, 48, 59, 91, 92, 95, 97, 103, 105, 107, 109, 110, 116, 133, 143, 145, 164, 175).

[128] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates." In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 1197–1210. DOI: 10.1145/2810103.2813653. IACR ePrint: ia.cr/2015/984 (cit. on pp. 41, 59, 91, 92, 95, 97, 103, 105, 107, 109, 110, 116, 133, 143, 145, 164, 175).

[129] Benjamin Dowling and Douglas Stebila. "Modelling Ciphersuite and Version Negotiation in the TLS Protocol." In: *ACISP 15: 20th Australasian Conference on Information Security and Privacy*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. Lecture Notes in Computer Science. Brisbane, QLD, Australia: Springer, Heidelberg, Germany, June 29–July 1, 2015, pp. 270–288. DOI: 10.1007/978-3-319-19962-7_16. IACR ePrint: ia.cr/2015/652 (cit. on pp. 41, 59, 93, 164).

[130] Nir Drucker and Shay Gueron. "A toolbox for software optimization of QC-MDPC code-based cryptosystems." In: *Journal of Cryptographic Engineering* 9.4 (Nov. 2019), pp. 341–357. DOI: 10.1007/s13389-018-00200-4. IACR ePrint: ia.cr/2017/1251 (cit. on p. 343).

[131] Nir Drucker and Shay Gueron. "Selfie: reflections on TLS 1.3 with PSK." In: *Journal of Cryptology* 34.3 (July 2021), p. 27. DOI: 10.1007/s00145-021-09387-y. IACR ePrint: ia.cr/2019/347 (cit. on pp. 82, 181).

[132] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. "An Argument for Increasing TCP's Initial Congestion Window." In: *ACM SIGCOMM Computer Communication Review* 40.3 (June 22, 2010), pp. 26–33. DOI: 10.1145/1823844.1823848. URL: research.google/pubs/pub36640/ (cit. on p. 238).

[133] Julien Duman, Kathrin Hövelmanns, Eike Kiltz, Vadim Lyubashevsky, and Gregor Seiler. "Faster Lattice-Based KEMs via a Generic Fujisaki-Okamoto Transform Using Prefix Hashing." In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 15–19, 2021, pp. 2722–2737. DOI: 10.1145/3460120.3484819. IACR ePrint: ia.cr/2021/1351 (cit. on p. 92).

[134] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. "A Search Engine Backed by Internet-Wide Scanning." In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 542–553. DOI: 10.1145/2810103.2813703 (cit. on p. 33).

[135] Cynthia Dwork, Moni Naor, and Amit Sahai. "Concurrent Zero-Knowledge." In: *30th Annual ACM Symposium on Theory of Computing*. Dallas, TX, USA: ACM Press, May 23–26, 1998, pp. 409–418. DOI: 10.1145/276698.276853. URL: scholar.archive.org/work/gbidq7fyszcznaphg7gx4q3zsa (cit. on p. 69).

[136] *Easy-RSA*. OpenVPN. URL: github.com/OpenVPN/easy-rsa (visited on May 9, 2023), archived at https://web.archive.org/web/20230326200910/https://github.com/OpenVPN/easy-rsa on Mar. 26, 2023 (cit. on p. 199).

[137] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. "MicroEliece: McEliece for Embedded Devices." In: *Cryptographic Hardware and Embedded Systems – CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Lausanne, Switzerland: Springer, Heidelberg, Germany, Sept. 6–9, 2009, pp. 49–64. DOI: 10.1007/978-3-642-04138-9_4 (cit. on p. 369).

[138] Taher Elgamal and Kipp E.B. Hickman. *The SSL Protocol*. Internet-Draft draft-hickman-netscape-ssl-00. Internet Engineering Task Force, Apr. 1995. 31 pp. URL: datatracker.ietf.org/doc/draft-hickman-netscape-ssl/00/ (cit. on p. 2).

[139] Andre Esser, Felix Heuer, Robert Kübler, Alexander May, and Christian Sohler. "Dissection-BKW." In: *Advances in Cryptology – CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 638–666. DOI: 10.1007/978-3-319-96881-0_22. IACR ePrint: ia.cr/2018/569 (cit. on p. 392).

[140] Andre Esser, Robert Kübler, and Alexander May. "LPN Decoded." In: *Advances in Cryptology – CRYPTO 2017, Part II*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10402. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 20–24, 2017, pp. 486–514. DOI: 10.1007/978-3-319-63715-0_17. IACR ePrint: ia.cr/2017/078 (cit. on pp. 393–397, 401, 402).

[141] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Post on the Cloudflare blog. Software available at https://github.com/cloudflare/circl. Cloudflare, June 20, 2019. URL: blog.cloudflare.com/introducing-circl/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230316130746/https://blog.cloudflare.com/introducing-circl/ on Mar. 16, 2023 (cit. on pp. 303, 462).

[142] Richard P. Feynman. "Simulating physics with computers." In: *International Journal of Theoretical Physics* 21.6/7 (1982), pp. 467–488. DOI: 10.1007/BF02650179. URL: scholar.archive.org/work/k6rlf6llkzcgnlawokl5atxkcy (cit. on p. 3).

[143] Marc Fischlin and Felix Günther. "Multi-Stage Key Exchange and the Case of Google's QUIC Protocol." In: *ACM CCS 2014: 21st Conference on Computer and Communications Security*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. Scottsdale, AZ, USA: ACM Press, Nov. 3–7, 2014, pp. 1193–1204. DOI: 10.1145/2660267.2660308. URL: www.cryptoplexity.informatik.tu-darmstadt.de/media/crypt/publications_1/fischlin-guenther-ccs2014.pdf (cit. on pp. 91, 98, 109, 110, 133, 143, 144, 175).

[144] Scott Fluhrer. *Cryptanalysis of ring-LWE based key exchange with key share reuse*. Cryptology ePrint Archive, Report 2016/085. 2016. IACR ePrint: ia.cr/2016/085 (cit. on p. 95).

[145] Scott Fluhrer. *Further Analysis of a Proposed Hash-Based Signature Standard*. Cryptology ePrint Archive, Report 2017/553. June 8, 2017. IACR ePrint: ia.cr/2017/553 (cit. on p. 34).

[146] Paul Ford-Hutchinson. *Securing FTP with TLS*. RFC 4217. Oct. 2005. DOI: 10.17487/RFC4217 (cit. on p. 2).

[147] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. "Non-Interactive Key Exchange." In: *PKC 2013: 16th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. Vol. 7778. Lecture Notes in Computer Science. Nara, Japan: Springer, Heidelberg, Germany, Feb. 26–Mar. 1, 2013, pp. 254–271. DOI: 10.1007/978-3-642-36362-7_17. IACR ePrint: ia.cr/2012/732 (cit. on p. 28).

[148] Stephan Friedl, Andrei Popov, Adam Langley, and Stephan Emile. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension*. RFC 7301. July 2014. DOI: 10.17487/RFC7301 (cit. on p. 68).

[149] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. "Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices." In: *PKC 2012: 15th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Marc Fischlin, Johannes Buchmann, and Mark Manulis. Vol. 7293. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Heidelberg, Germany, May 21–23, 2012, pp. 467–484. DOI: 10.1007/978-3-642-30057-8_28. IACR ePrint: ia.cr/2012/211 (cit. on pp. 56, 61).

[150] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes." In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 15–19, 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34 (cit. on p. 95).

[151] *Future-proof security solution: Infineon launches world's first TPM with a PQC-protected firmware update mechanism*. Infineon Press Release. Infineon, Feb. 15, 2022. URL: www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFCSS202202-051.html (visited on May 9, 2023), archived at https://web.archive.org/web/20230306072437/https://www.infineon.com/cms/en/about-infineon/press/market-news/2022/INFCSS202202-051.html on Mar. 6, 2023 (cit. on p. 344).

[152] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. *Swoosh: Practical Lattice-Based Non-Interactive Key Exchange*. Cryptology ePrint Archive, Report 2023/271. https://eprint.iacr.org/2023/271. 2023 (cit. on p. 245).

[153] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. "Swoosh: Practical Lattice-Based Non-Interactive Key Exchange." In: *USENIX Security 2024: 33rd USENIX Security Symposium*. Philadelphia, PA, USA, Oct. 14–16, 2024. IACR ePrint: ia.cr/2023/271 (cit. on pp. 29, 57, 241).

[154] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. "On the Security of Supersingular Isogeny Cryptosystems." In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Heidelberg, Germany, Dec. 4–8, 2016, pp. 63–91. DOI: 10.1007/978-3-662-53887-6_3. IACR ePrint: ia.cr/2016/859 (cit. on p. 95).

[155] Xinwei Gao, Jintai Ding, Lin Li, Saraswathy RV, and Jiqiang Liu. "Efficient Implementation of Password-based Authenticated Key Exchange from RLWE and Post-Quantum TLS." In: *International Journal of Network Security* 20.5 (2018), pp. 923–930. DOI: 10.6633/IJNS.201809_20(5).14. IACR ePrint: ia.cr/2017/1192 (cit. on p. 50).

[156] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. ""Make Sure DSA Signing Exponentiations Really are Constant-Time"" In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 24–28, 2016, pp. 1639–1650. DOI: 10.1145/2976749.2978420. IACR ePrint: ia.cr/2016/594 (cit. on p. 279).

[157] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. "Trapdoors for hard lattices and new cryptographic constructions." In: *40th Annual ACM Symposium on Theory of Computing*. Ed. by Richard E. Ladner and Cynthia Dwork. Victoria, BC, Canada: ACM Press, May 17–20, 2008, pp. 197–206.

DOI: 10.1145/1374376.1374407. IACR ePrint: ia.cr/2007/432 (cit. on p. 373).

[158] Alessandro Ghedini and Victor Vasiliev. *TLS Certificate Compression*. RFC 7924. RFC Editor, Dec. 2020, pp. 1–8. DOI: 10.17487/RFC8879 (cit. on p. 313).

[159] *GitHub Features – Actions*. GitHub. URL: github.com/features/actions (visited on May 9, 2023), archived at https://web.archive.org/web/20230509121346/https://github.com/features/actions on May 9, 2023 (cit. on p. 355).

[160] Herman H. Goldstine and Adele Goldstine. "The electronic numerical integrator and computer (ENIAC)." In: *Mathematics of Computation (MCOM)* 2.15 (July 1946), pp. 97–110. DOI: 10.1090/S0025-5718-1946-0018977-0 (cit. on p. 3).

[161] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. "Verifying Post-Quantum Signatures in 8 kB of RAM." In: *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Daejeon, South Korea: Springer, Heidelberg, Germany, July 20–22, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12. IACR ePrint: ia.cr/2021/662. URL: wggrs.nl/p/verifying-post-quantum-signatures-in-8kb-of-ram (cit. on pp. 13, 324, 480).

[162] Ruben Gonzalez and Thom Wiggers. "KEMTLS vs. Post-quantum TLS: Performance on Embedded Systems." In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Lejla Batina, Stjepan Picek, and Mainack Mondal. Jaipur, India: Springer Nature Switzerland, Dec. 9–12, 2022, pp. 99–117. DOI: 10.1007/978-3-031-22829-2. URL: wggrs.nl/p/kemtls-embedded (cit. on pp. 11, 479).

[163] Google. *BoringSSL*. URL: boringssl.googlesource.com/boringssl/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230420074945/https://boringssl.googlesource.com/boringssl/ on Apr. 20, 2023 (cit. on p. 189).

[164] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. "Compact Dilithium Implementations on Cortex-M3 and Cortex-M4." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2021), pp. 1–24. DOI: 10.46586/tches.v2021.i1.1-24. IACR ePrint: ia.cr/2020/1278 (cit. on p. 379).

[165] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search." In: *28th Annual ACM Symposium on Theory of Computing*. Philadelphia, PA, USA: ACM Press, May 22–24, 1996, pp. 212–219. DOI: 10.1145/237814.237866 (cit. on p. 34).

[166] Tim Güneysu, Philip W. Hodges, Georg Land, Mike Ounsworth, Douglas Stebila, and Greg Zaverucha. "Proof-of-Possession for KEM Certificates using Verifiable Generation." In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 7–11, 2022, pp. 1337–1351. DOI: 10.1145/3548606.3560560. IACR ePrint: ia.cr/2022/703 (cit. on p. 75).

[167] Felix Günther. "Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols." PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, 2018. URL: tuprints.ulb.tu-darmstadt.de/7162, archived at https://web.archive.org/web/20221203062802/https://tuprints.ulb.tu-darmstadt.de/7162/ on Dec. 3, 2022 (cit. on pp. 102, 117, 138, 146).

[168] Felix Günther, Simon Rastikian, Patrick Towa, and Thom Wiggers. "KEM-TLS with Delayed Forward Identity Protection in (Almost) a Single Round Trip." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Rome, Italy: Springer, Heidelberg, Germany, June 20–23, 2022, pp. 253–272. DOI: 10.1007/978-3-031-09234-3_13. IACR ePrint: ia.cr/2021/725. URL: wggrs.nl/p/kemtls-epoch (cit. on pp. 9, 87, 164, 479).

[169] Richard Günther. *Kernel Support for miscellaneous Binary Formats (binfmt_misc)*. Linux Kernel. URL: www.kernel.org/doc/html/v5.16/admin-guide/binfmt-misc.html (cit. on p. 355).

[170] Qian Guo, Thomas Johansson, and Carl Löndahl. "Solving LPN Using Covering Codes." In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Kaoshiung, Taiwan, R.O.C.: Springer, Heidelberg, Germany, Dec. 7–11, 2014, pp. 1–20. DOI: 10.1007/978-3-662-45611-8_1 (cit. on pp. 391–393).

[171] Peter Gutmann. "Lessons Learned in Implementing and Deploying Crypto Software." In: *USENIX Security 2002: 11th USENIX Security Symposium*. Ed. by Dan Boneh. San Francisco, CA, USA: USENIX Association, Aug. 5–9, 2002, pp. 315–325. URL: www.usenix.org/conference/11th-usenix-security-symposium/lessons-learned-implementing-and-deploying-crypto-software (cit. on p. 336).

[172] Richard W. Hamming. "Error detecting and error correcting codes." In: *The Bell System Technical Journal* 29.2 (Apr. 1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x (cit. on p. 398).

[173] Dan Harkins. *Secure Password Ciphersuites for Transport Layer Security (TLS)*. RFC 8492. Feb. 2019. DOI: 10.17487/RFC8492 (cit. on p. 42).

[174] Stephen Hemminger, Fabio Ludovici, and Hagen Paul Pfeiffer. *tc-netem(8)*. man tc netem. Linux foundation. Nov. 2011. URL: man7.org/linux/man-pages/man8/tc-netem.8.html (cit. on pp. 202, 320).

[175] Olaf Henniger, Alastair Ruddle, Hervé Seudié, Benjamin Weyl, Marko Wolf, and Thomas Wollinger. "Securing vehicular on-board IT systems: The EVITA project." In: *25th Joint VDI/VW Automotive Security Conference*. Oct. 2009. URL: evita-project.org/Publications/HRSW09.pdf, archived at https://web.archive.org/web/20230506182834/https://evita-project.org/Publications/HRSW09.pdf on May 6, 2023 (cit. on p. 368).

[176] Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. "Password-Authenticated TLS via OPAQUE and Post-Handshake Authentication." In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France: Springer, Heidelberg, Germany, Apr. 23–27, 2023, pp. 98–127. DOI: 10.1007/978-3-031-30589-4_4. IACR ePrint: ia.cr/2023/220 (cit. on p. 42).

[177] Paul E. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security*. RFC 3207. Feb. 2002. DOI: 10.17487/RFC3207 (cit. on pp. 2, 41).

[178] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. "A Modular Analysis of the Fujisaki-Okamoto Transformation." In: *TCC 2017: 15th Theory of Cryptography Conference, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Baltimore, MD, USA: Springer, Heidelberg, Germany, Nov. 12–15, 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12. IACR ePrint: ia.cr/2017/604 (cit. on p. 30).

[179] Andrew Hopkins. *Post-Quantum TLS now supported in AWS KMS*. Amazon AWS Security Blog. Nov. 4, 2019. URL: aws.amazon.com/blogs/security/post-quantum-tls-now-supported-in-aws-kms/ (visited on May 20, 2022), archived at https://web.archive.org/web/20230505135020/https://aws.amazon.com/blogs/security/post-quantum-tls-now-supported-in-aws-kms/ on May 5, 2023 (cit. on p. 50).

[180] Jonathan Hoyland and Christopher Wood. *TLS 1.3 Extended Key Schedule*. Internet-Draft draft-jhoyla-tls-extended-key-schedule-03. Work in Progress. Internet Engineering Task Force, Dec. 2020. 1-7. URL: datatracker.ietf.org/doc/html/draft-jhoyla-tls-extended-key-schedule-03 (cit. on p. 50).

[181] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391 (cit. on pp. 37, 205, 239, 369, 387).

[182] Loïs Huguenin-Dumittan and Serge Vaudenay. "On IND-qCCA Security in the ROM and Its Applications - CPA Security Is Sufficient for TLS 1.3." In: *Advances in Cryptology – EUROCRYPT 2022, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Heidelberg, Germany, May 30–June 3, 2022, pp. 613–642. DOI: 10.1007/978-3-031-07082-2_22. IACR ePrint: ia.cr/2021/844 (cit. on pp. 31, 48, 95, 103).

[183] Andreas Hülsing. *Public Comments on Draft SP 800-208*. https://csrc.nist.gov/CSRC/media/Publications/sp/800-208/draft/documents/sp800-208-draft-comments-received.pdf, page 7. 2020, archived at https://web.archive.org/web/20220605052334/https://csrc.nist.gov/CSRC/media/Publications/sp/800-208/draft/documents/sp800-208-draft-comments-received.pdf on June 5, 2022 (cit. on p. 239).

[184] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. *SPHINCS$^+$*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, archived at https://web.archive.org/web/20230429160244/https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 on Apr. 29, 2023 (cit. on pp. 15, 37, 207, 248, 370, 375, 385).

[185] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. "Post-quantum WireGuard." In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030. IACR ePrint: ia.cr/2020/379 (cit. on pp. 56, 167, 343).

[186] Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. "High-Speed Key Encapsulation from NTRU." In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi

Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, Sept. 25–28, 2017, pp. 232–252. DOI: 10.1007/978-3-319-66787-4_12. IACR ePrint: ia.cr/2017/667 (cit. on pp. 50, 287).

[187] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS - Computing a 41 KB Signature in 16 KB of RAM." In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9614. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, Mar. 6–9, 2016, pp. 446–470. DOI: 10.1007/978-3-662-49384-7_17. IACR ePrint: ia.cr/2015/1042 (cit. on p. 369).

[188] Andreas Hülsing, Joost Rijneveld, and Fang Song. "Mitigating Multi-target Attacks in Hash-Based Signatures." In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9614. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Heidelberg, Germany, Mar. 6–9, 2016, pp. 387–416. DOI: 10.1007/978-3-662-49384-7_15. IACR ePrint: ia.cr/2015/1256 (cit. on p. 239).

[189] Andreas Hülsing and Florian Weber. "Epochal Signatures for Deniable Group Chats." In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 24–27, 2021, pp. 1677–1695. DOI: 10.1109/SP40001.2021.00058. IACR ePrint: ia.cr/2020/1138 (cit. on pp. 69, 70, 135).

[190] *IBM Unveils 400 Qubit-Plus Quantum Processor and Next-Generation IBM Quantum System Two*. IBM. Nov. 9, 2022. URL: newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generation-IBM-Quantum-System-Two (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230419213748/https://newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generation-IBM-Quantum-System-Two on Apr. 19, 2023 (cit. on p. 4).

[191] Jamie Indigo and Dave Smart. *HTTP Archive Web Almanac: Page Weight*. HTTP Archive. Sept. 26, 2022. URL: almanac.httparchive.org/en/2022/page-weight (visited on Mar. 20, 2023), archived at https://web.archive.org/web/20230406050133/https://almanac.httparchive.org/en/2022/page-weight on Apr. 6, 2023 (cit. on p. 363).

[192] Internet Security Research Group. *Let's encrypt statistics*. URL: letsencrypt.org/stats/ (visited on Oct. 6, 2022), archived at https://web.archive.org/web/20230504131118/https://letsencrypt.org/stats/ on May 4, 2023 (cit. on p. 74).

[193] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000 (cit. on pp. 64, 67).

[194] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. "On the Security of TLS-DHE in the Standard Model." In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2012, pp. 273–293. DOI: 10.1007/978-3-642-32009-5_17. IACR ePrint: ia.cr/2011/219 (cit. on pp. 95, 103).

[195] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. ""They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks." In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 22–26, 2022, pp. 632–649. DOI: 10.1109/SP46214.2022.9833713. IACR ePrint: ia.cr/2021/1650 (cit. on p. 347).

[196] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. "Minerva: The curse of ECDSA nonces." In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.4 (2020), pp. 281–308. DOI: 10.13154/tches.v2020.i4.281-308. IACR ePrint: ia.cr/2020/728 (cit. on p. 279).

[197] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. *SIKE*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions, archived at https://web.archive.org/web/20230501144806/https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions on May 1, 2023 (cit. on pp. 15, 36, 50, 303).

[198] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks." In: *Advances in Cryptology – EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Heidelberg, Germany, Apr. 29–May 3, 2018, pp. 456–486. DOI: 10.1007/978-3-319-78372-7_15. IACR ePrint: ia.cr/2018/163 (cit. on p. 42).

[199] Stephen Jordan. *Quantum Algorithm Zoo*. Oct. 14, 2022. URL: quantumalgorithmzoo . org (visited on May 9, 2023), archived at https : / / web . archive . org / web / 20230413184651 / https : / / quantumalgorithmzoo.org/ on Apr. 13, 2023 (cit. on p. 3).

[200] Simon Josefsson. *Storing Certificates in the Domain Name System (DNS)*. RFC 4398. Mar. 2006. DOI: 10.17487/RFC4398 (cit. on pp. 62, 78).

[201] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC 8032. Jan. 2017. DOI: 10.17487/RFC8032 (cit. on p. 27).

[202] Panos Kampanakis, Cameron Bytheway, Bas Westerbaan, and Martin Thomson. *Suppressing CA Certificates in TLS 1.3*. Internet-Draft draft-kampanakis-tls-scas-latest-03. Work in Progress. Internet Engineering Task Force, Jan. 2023. 10 pp. URL: datatracker . ietf . org / doc / draft - kampanakis - tls - scas-latest/03/ (cit. on pp. 78, 208, 238, 249, 313, 363).

[203] Panos Kampanakis and Michael Kallitsis. "Faster Post-Quantum TLS Handshakes Without Intermediate CA Certificates." In: *CSCML 2022: Cyber Security, Cryptology, and Machine Learning*. Ed. by Shlomi Dolev, Jonathan Katz, and Amnon Meisels. Virtual Event: Springer, Cham, 2022, pp. 337–355. DOI: 10 . 1007 / 978 - 3 - 031 - 07689 - 3 _ 25. URL: www . amazon . science / publications / faster - post - quantum - tls - handshakes - without - intermediate-ca-certificates (cit. on p. 78).

[204] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Post-quantum crypto library for the ARM Cortex-M4*. URL: github.com/mupq/pqm4 (cit. on pp. 317, 319, 322, 327, 335, 358, 375).

[205] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. "Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates." In: *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. Lecture Notes in Computer Science. Bogota, Colombia: Springer, Heidelberg, Germany, June 5–7, 2019, pp. 281–301. DOI: 10.1007/978-3-030-21568-2_14. IACR ePrint: ia.cr/2018/1018 (cit. on p. 343).

[206] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Improving Software Quality in Cryptography Standardization Projects." In: *SSR 2022: Security Standardization Research (2022 IEEE S&P Workshops)*. Genoa, Italy: IEEE Computer Society, June 6–10, 2022, pp. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010. URL: wggrs.nl/p/pqclean (cit. on pp. 12, 479).

[207] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. *PQClean*. URL: github.com/PQClean (cit. on p. 335).

[208] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. Oct. 2014. DOI: 10.17487/RFC7296 (cit. on p. 280).

[209] Auguste Kerckhoffs. "La cryptographie militaire." French. In: *Journal des sciences militaires* IX/X (Jan.–Feb. 1883). Available with translation at https://petitcolas.net/kerckhoffs/index.html, pp. 5–38/161-191, archived at https://web.archive.org/web/20230509140148/https://petitcolas.net/kerckhoffs/index.html on May 9, 2023 (cit. on p. 20).

[210] Franziskus Kiefer and Krzysztof Kwiatkowski. *Hybrid ECDHE-SIDH Key Exchange for TLS*. Internet-Draft draft-kiefer-tls-ecdhe-sidh-00. Work in Progress. Internet Engineering Task Force, Nov. 2018. 13 pp. URL: datatracker.ietf.org/doc/html/draft-kiefer-tls-ecdhe-sidh-00 (cit. on p. 50).

[211] Aviad Kipnis, Jacques Patarin, and Louis Goubin. "Unbalanced Oil and Vinegar Signature Schemes." In: *Advances in Cryptology – EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Prague, Czech Republic: Springer, Heidelberg, Germany, May 2–6, 1999, pp. 206–222. DOI: 10.1007/3-540-48910-X_15 (cit. on p. 371).

[212] Neal Koblitz. "Elliptic Curve Cryptosystems." In: *Mathematics of Computation* 48.177 (Jan. 1987), pp. 203–209. DOI: 10.1090/S0025-5718-1987-0866109-5 (cit. on p. 5).

[213] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. "(De-)Constructing TLS 1.3." In: *Progress in Cryptology - INDOCRYPT 2015: 16th International Conference in Cryptology in India*. Ed. by Alex Biryukov and Vipul Goyal. Vol. 9462. Lecture Notes in Computer Science. Bangalore, India: Springer, Heidelberg, Germany, Dec. 6–9, 2015, pp. 85–102. DOI: 10.1007/978-3-319-26617-6_5. IACR ePrint: ia.cr/2014/020 (cit. on pp. 41, 59, 164).

[214] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme." In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 15–19, 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34. IACR ePrint: ia.cr/2010/264 (cit. on p. 24).

[215] Hugo Krawczyk. "HMQV: A High-Performance Secure Diffie-Hellman Protocol." In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Full version available via IACR ePrint. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 14–18, 2005, pp. 546–566. DOI: 10.1007/11535218_33. IACR ePrint: ia.cr/2005/176 (cit. on pp. 56, 99).

[216] Hugo Krawczyk. "SIGMA: The "SIGn-and-MAc" Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols." In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2003, pp. 400–425. DOI: 10.1007/978-3-540-45146-4_24 (cit. on p. 280).

[217] Hugo Krawczyk. "SKEME: a versatile secure key exchange mechanism for Internet." In: *ISOC Network and Distributed System Security Symposium – NDSS'96*. Ed. by James T. Ellis, B. Clifford Neuman, and David M. Balenson. San Diego, CA, USA: IEEE Computer Society, Feb. 22–23, 1996, pp. 114–127. DOI: 10.1109/NDSS.1996.492418. URL: scholar.archive.org/work/jztebyi24jdn3jr7ptf777lzkq (cit. on pp. 56, 69).

[218] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. DOI: 10.17487/RFC2104 (cit. on p. 24).

[219] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869 (cit. on p. 24).

[220] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis." In: *Advances in Cryptology – CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 18–22, 2013, pp. 429–448. DOI: 10.1007/978-3-642-40041-4_24. IACR ePrint: ia.cr/2013/339 (cit. on pp. 95, 103).

[221] Hugo Krawczyk and Hoeteck Wee. "The OPTLS Protocol and TLS 1.3." In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbruecken, Germany: IEEE, Apr. 21–24, 2016, pp. 81–96. DOI: 10.1109/EuroSP.2016.18. URL: ia.cr/2015/978 (cit. on pp. 41, 53, 70, 163, 164, 359, 465).

[222] Robert Krebbers. "The C standard formalized in Coq." PhD thesis. Nijmegen, The Netherlands: Radboud University, 2015. DOI: 2066/147182. URL: robbertkrebbers.nl/thesis.html (cit. on p. 345).

[223] Holger Krekel and pytest developers. *pytest: helps you write better programs*. 2023. URL: pytest.org/ (visited on May 9, 2023), archived at https://web.archive.org/web/20230507115503/https://docs.pytest.org/en/7.3.x/ on May 7, 2023 (cit. on p. 354).

[224] Wouter Kuhnen. "OPTLS revisited." Master's thesis. Nijmegen, The Netherlands: Radboud University, 2018. URL: www.ru.nl/publish/pages/769526/thesis-final.pdf, archived at https://web.archive.org/web/20230509090915/https://www.ru.nl/publish/pages/769526/thesis-final.pdf on May 9, 2023 (cit. on pp. 55, 57).

[225] Greg Kuperberg. "A Subexponential-Time Quantum Algorithm for the Dihedral Hidden Subgroup Problem." In: *SIAM Journal on Computing* 35.1 (2005), pp. 170–188. DOI: 10.1137/S0097539703436345. arXiv: quant-ph/0302112 (cit. on p. 244).

[226] Kris Kwiatkowski and Luke Valenta. *The TLS Post-Quantum Experiment*. Post on the Cloudflare blog. Cloudflare, 2019. URL: blog.cloudflare.com/the-tls-post-quantum-experiment/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230509090346/https://blog.cloudflare.com/the-tls-post-quantum-experiment/ (cit. on pp. 49, 344).

[227] Krzysztof Kwiatkowski, Nick Sullivan, Adam Langley, Dave Levin, and Alan Mislove. *Measuring TLS key exchange with post-quantum KEM*. Workshop Record of the Second PQC Standardization Conference. 2019. URL: csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kwiatkowski-measuring-tls.pdf, archived at https://web.archive.org/web/20230506193449/https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kwiatkowski-measuring-tls.pdf on May 6, 2023 (cit. on p. 50).

[228] *Kyber*. CRYSTALS-Kyber team. Dec. 23, 2020. URL: pq-crystals.org/kyber/ (visited on Mar. 6, 2023), archived at https://web.archive.org/web/20230425044723/https://pq-crystals.org/kyber/ on Apr. 25, 2023 (cit. on pp. 219, 260, 289).

[229] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. "Stronger Security of Authenticated Key Exchange." In: *ProvSec 2007: 1st International Conference on Provable Security*. Ed. by Willy Susilo, Joseph K. Liu, and Yi Mu. Vol. 4784. Lecture Notes in Computer Science. Wollongong, Australia: Springer, Heidelberg, Germany, Nov. 1–2, 2007, pp. 1–16. IACR ePrint: ia.cr/2006/073 (cit. on p. 56).

[230] Leslie Lamport. *Constructing Digital Signatures from a One-way Function*. Technical Report SRI-CSL-98. SRI International Computer Science Laboratory, Oct. 1979. URL: www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/, archived at https://web.archive.org/web/20230509091713/https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Constructing-Digital-Signatures-from-a-One-Way-Function.pdf on May 9, 2023 (cit. on p. 37).

[231] Adam Langley. *Apple's SSL/TLS bug*. Blog post. Feb. 22, 2014. URL: www.imperialviolet.org/2014/02/22/applebug.html (visited on May 9, 2023), archived at https://web.archive.org/web/20230429002827/https://www.imperialviolet.org/2014/02/22/applebug.html on Apr. 29, 2023 (cit. on p. 337).

[232] Adam Langley. *CECPQ1 results*. Blog post. 2016. URL: www.imperialviolet.org/2016/11/28/cecpq1.html, archived at https://web.archive.org/web/20230509092408/https://www.imperialviolet.org/2016/11/28/cecpq1.html on May 9, 2023 (cit. on pp. 49, 344).

[233] Adam Langley. *CECPQ2*. Blog post. 2018. URL: www.imperialviolet.org/2018/12/12/cecpq2.html, archived at https://web.archive.org/web/20230509092514/https://www.imperialviolet.org/2018/12/12/cecpq2.html on May 9, 2023 (cit. on pp. 49, 344).

[234] Adam Langley. *Commit "Drop HRSS Assembly"—BoringSSL Gerrit*. Jan. 25, 2023. URL: boringssl-review.googlesource.com/q/commit:97873cd1a59b97ced00907e274afaff75edf4a57 (visited on May 9, 2023), archived at https://web.archive.org/web/20230509140931/https://boringssl-review.googlesource.com/c/boringssl/+/55885 on May 9, 2023 (cit. on p. 287).

[235] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. RFC 6962. June 2013. DOI: 10.17487/RFC6962 (cit. on pp. 74, 236).

[236] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. "An Efficient Protocol for Authenticated Key Agreement." In: *Designs, Codes and Cryptography* 28.2 (2003), pp. 119–134. DOI: 10.1023/A:1022595222606 (cit. on p. 56).

[237] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. "Why Does Cryptographic Software Fail? A Case Study and Open Problems." In: *APSys '14: Proceedings of 5th Asia-Pacific Workshop on Systems*. Beijing, China: Association for Computing Machinery, 2014. DOI: 10.1145/2637166.2637237. URL: people.csail.mit.edu/nickolai/papers/lazar-cryptobugs.pdf (cit. on p. 337).

[238] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. "TensorCrypto: High Throughput Acceleration of Lattice-Based Cryptography Using Tensor Core on GPU." In: *IEEE Access* 10 (Feb. 16, 2022), pp. 20616–20632. DOI: 10.1109/ACCESS.2022.3152217 (cit. on p. 343).

[239] Éric Levieil and Pierre-Alain Fouque. "An Improved LPN Algorithm." In: *SCN 06: 5th International Conference on Security in Communication Networks*. Ed. by Roberto De Prisco and Moti Yung. Vol. 4116. Lecture Notes in Computer Science. Maiori, Italy: Springer, Heidelberg, Germany, Sept. 6–8, 2006, pp. 348–359. DOI: 10.1007/11832072_24. URL: scholar.archive.org/work/3m3f3dcevfb5znettjtl4ovrse (cit. on pp. 391, 393).

[240] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures." In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Tokyo, Japan: Springer, Heidelberg, Germany, Dec. 6–10, 2009, pp. 598–616. DOI: 10.1007/978-3-642-10366-7_35 (cit. on p. 373).

[241] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. *CRYSTALS-DILITHIUM*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, archived at https://web.archive.org/web/20230429160244/https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 on Apr. 29, 2023 (cit. on pp. 5, 15, 35, 50, 206, 248, 282, 361, 373, 379).

[242] Luciano Maino, Chloe Martindale, Lorenz Panny, Giacomo Pope, and Benjamin Wesolowski. "A Direct Key Recovery Attack on SIDH." In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France: Springer, Heidelberg, Germany, Apr. 23–27, 2023, pp. 448–471. DOI: 10.1007/978-3-031-30589-4_16. IACR ePrint: ia.cr/2023/640 (cit. on pp. 15, 36, 50).

[243] Varum Maram and Keita Xagawa. "Post-Quantum Anonymity of Kyber." In: *PKC 2023: 26th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. to appear. 2023. URL: eprint.iacr.org/2022/1696.pdf (cit. on p. 135).

[244] Mugur Marculescu. *Introducing gRPC, a new open source HTTP/2 RPC framework*. Post on the Google developers blog. Google, Feb. 26, 2015. URL: developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230206193643/https://developers.googleblog.

com/2015/02/introducing-grpc-new-open-source-http2.html on Feb. 6, 2023 (cit. on p. 305).

[245] Moxie Marlinspike and Trevor Perrin. *The X3DH Key Agreement Protocol*. Specification. Signal Foundation, Nov. 4, 2016. URL: signal.org/docs/specifications/x3dh/ (visited on Aug. 24, 2022), archived at https://web.archive.org/web/20230505124521/https://signal.org/docs/specifications/x3dh/ on May 5, 2023 (cit. on pp. 29, 57).

[246] Tsutomu Matsumoto and Hideki Imai. "Public Quadratic Polynominal-Tuples for Efficient Signature-Verification and Message-Encryption." In: *Advances in Cryptology – EUROCRYPT'88*. Ed. by C. G. Günther. Vol. 330. Lecture Notes in Computer Science. Davos, Switzerland: Springer, Heidelberg, Germany, May 25–27, 1988, pp. 419–453. DOI: 10.1007/3-540-45961-8_39 (cit. on p. 371).

[247] *mbed TLS*. URL: www.trustedfirmware.org/projects/mbed-tls/ (visited on Apr. 9, 2022), archived at https://web.archive.org/web/20230503075354/https://www.trustedfirmware.org/projects/mbed-tls/ on May 9, 2023 (cit. on p. 316).

[248] Robert J. McEliece. *A public-key cryptosystem based on algebraic coding theory*. The Deep Space Network Progress Report 42-44. Jet Propulsion Laboratory, California Institute of Technology, Jan.–Feb. 1978, pp. 114–116. URL: ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (cit. on pp. 35, 298).

[249] David McGrew, Michael Curcio, and Scott Fluhrer. *Leighton–Micali Hash-Based Signatures*. RFC 8554. Apr. 2019. DOI: 10.17487/RFC8554 (cit. on pp. 37, 205, 369, 387).

[250] David A. McGrew, Panos Kampanakis, Scott R. Fluhrer, Stefan-Lukas Gazdag, Denis Butin, and Johannes Buchmann. "State Management for Hash-Based Signatures." In: *Security Standardisation Research (SSR) 2016*. Ed. by Lidong Chen, David A. McGrew, and Chris J. Mitchell. Vol. 10074. Lecture Notes in Computer Science. Gaithersburg, MD, USA: Springer, Heidelberg, Germany, Dec. 5–6, 2016, pp. 244–260. DOI: 2016/357 (cit. on pp. 37, 280).

[251] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols." In: *CAV 2013: Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 2013, pp. 696–701. DOI: 10.1007/978-3-642-39799-8_48 (cit. on pp. 164, 165, 361).

[252] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)." In: *USENIX Security 2021: 30th USENIX Security Symposium*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 11–13, 2021, pp. 213–230. URL: www.usenix.org/conference/usenixsecurity21/presentation/merget (cit. on p. 33).

[253] Denis Merigoux, Franziskus Kiefer, and Karthikeyan Bhargavan. *hacspec: succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust*. Technical Report. Inria, Mar. 22, 2021. URL: hal.science/hal-03176482 (cit. on p. 357).

[254] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function." In: *Advances in Cryptology – CRYPTO'87*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 16–20, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32 (cit. on p. 37).

[255] Ralph C. Merkle. "Fast Software Encryption Functions." In: *Advances in Cryptology – CRYPTO'90*. Ed. by Alfred J. Menezes and Scott A. Vanstone. Vol. 537. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 11–15, 1990, pp. 476–501. DOI: 10.1007/3-540-38424-3_34 (cit. on p. 386).

[256] Victor S. Miller. "Use of Elliptic Curves in Cryptography." In: *Advances in Cryptology – CRYPTO'85*. Ed. by Hugh C. Williams. Vol. 218. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 18–22, 1985, pp. 417–426. DOI: 10.1007/3-540-39799-X_31 (cit. on p. 5).

[257] Tero Mononen, Tomi Kause, Stephen Farrell, and Carlisle Adams. *Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)*. RFC 4210. Sept. 2005. DOI: 10.17487/RFC4210 (cit. on p. 74).

[258] Michele Mosca. *Cybersecurity in an Era with Quantum Computers: Will We Be Ready?* Cryptology ePrint Archive, Report 2015/1075. Nov. 5, 2015. IACR ePrint: ia.cr/2015/1075 (cit. on p. 4).

[259] Michele Mosca and Marco Piani. *Quantum Threat Timeline*. Tech. rep. Global Risk Institute, Oct. 2019. URL: globalriskinstitute.org/publications/quantum-threat-timeline/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230506185422/https://globalriskinstitute.org/publication/quantum-threat-timeline/ on May 6, 2023 (cit. on p. 4).

[260] Nicky Mouha, Mohammad S. Raunak, D. Richard Kuhn, and Raghu Kacker. "Finding Bugs in Cryptographic Hash Function Implementations." In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 870–884. DOI: 10.1109/TR.2018.2847247. IACR ePrint: ia.cr/2017/891 (cit. on p. 337).

[261] National Institute of Standards and Technology. *Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process*. Sept. 6, 2022. URL: csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf (visited on Oct. 5, 2022), archived at https://web.archive.org/web/20221014132635/http://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf on Oct. 14, 2022 (cit. on p. 342).

[262] National Institute of Standards and Technology. *FIPS 140-3: Security Requirements for Cryptographic Modules*. Tech. rep. FIPS 140-3. National Institute of Standards and Technology, 2019. DOI: 10.6028/NIST.FIPS.140-3 (cit. on p. 42).

[263] National Institute of Standards and Technology. *Guidelines for submitting tweaks for Fourth Round Candidates*. July 5, 2022. URL: csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/guidelines-for-submitting-tweaks-fourth-round.pdf (visited on Oct. 5, 2022), archived at https://web.archive.org/web/20220923212902/https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/guidelines-for-submitting-tweaks-fourth-round.pdf on Sept. 23, 2022 (cit. on p. 342).

[264] National Institute of Standards and Technology. *Guidelines for submitting tweaks for Third Round Finalists and Candidates*. July 23, 2020. URL: csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/guidelines-for-sumbitting-tweaks-third-round.pdf, archived at https://web.archive.org/web/20220707231333/https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/guidelines-for-sumbitting-tweaks-third-round.pdf on July 7, 2022 (cit. on p. 341).

[265] National Institute of Standards and Technology. *PQC - API notes*. Dec. 2016. URL: csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/api-notes.pdf, archived at https://web.archive.org/web/20230425024136/https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/api-notes.pdf on Apr. 25, 2023 (cit. on p. 340).

[266] National Institute of Standards and Technology. *PQC – Known Answer Tests and Test Vectors*. Dec. 2016. URL: csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/kat.pdf, archived at https://web.archive.org/web/20230126000238/https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/example-files/kat.pdf on Jan. 26, 2023 (cit. on p. 340).

[267] National Institute of Standards and Technology. *PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates*. July 5, 2022. URL: csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4 (visited on Aug. 25, 2022), archived at https://web.archive.org/web/20230425042709/https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4 on Apr. 25, 2023 (cit. on pp. 15, 338, 363).

[268] National Institute of Standards and Technology. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Tech. rep. FIPS 202. Gaithersburg, MD, USA: National Institute of Standards and Technology, Aug. 4, 2015. DOI: 10.6028/NIST.FIPS.202 (cit. on p. 239).

[269] National Institute of Standards and Technology. *Source Code Files for KATs*. Dec. 2016. URL: csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/example-files/source-code-files-for-kats.zip, archived at https://web.archive.org/web/20230509095158/https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/example-files/source-code-files-for-kats.zip on May 9, 2023 (cit. on p. 340).

[270] National Institute of Standards and Technology. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. Dec. 2016. URL: csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (visited on May 9, 2023), archived at https://web.archive.org/web/20230429211728/https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf on Apr. 29, 2023 (cit. on pp. 37, 338–340, 356).

[271] National Security Agency. *Announcing the Commercial National Security Algorithm Suite 2.0*. Sept. 7, 2022. URL: media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF (visited on Mar. 3, 2023), archived at https://web.archive.org/web/20230315013839/https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF on Mar. 15, 2023 (cit. on pp. 227, 268, 293).

[272] *Nederlander steeds vaker online voor medische diensten*. Dutch. Centraal Bureau voor de Statistiek. Feb. 17, 2023. URL: www.cbs.nl/nl-nl/nieuws/2023/07/nederlander-steeds-vaker-online-voor-medische-diensten (visited on Apr. 17, 2023), archived at https://web.archive.org/web/20230315082317/https://www.cbs.nl/nl-nl/nieuws/2023/07/nederlander-steeds-vaker-online-voor-medische-diensten on Mar. 15, 2023 (cit. on p. 2).

[273] Moritz Neikes. *TIMECOP: Automated dynamic analysis for timing side-channels*. 2020. URL: www.post-apocalyptic-crypto.org/timecop/ (visited on May 9, 2023), archived at https://web.archive.org/web/20230509124818/https://www.post-apocalyptic-crypto.org/timecop/ on May 9, 2023 (cit. on pp. 338, 348).

[274] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation." In: *PLDI '07: 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, California, USA: Association for Computing Machinery, June 10–13, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746. URL: valgrind.org/docs/valgrind2007.pdf (cit. on p. 348).

[275] Chris Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. June 1999. DOI: 10.17487/RFC2595 (cit. on pp. 2, 41).

[276] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439 (cit. on p. 22).

[277] Magnus Nystrom and Burt Kaliski. *PKCS #10: Certification Request Syntax Specification Version 1.7*. RFC 2986. Nov. 2000. DOI: 10.17487/RFC2986 (cit. on p. 73).

[278] Oak Ridge National Laboratory. *Summit FAQs*. URL: www.olcf.ornl.gov/olcf-resources/compute-systems/summit/summit-faqs/ (visited on Jan. 25, 2021), archived at https://web.archive.org/web/20221214234825/https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/summit-faqs/ on Dec. 14, 2022 (cit. on p. 402).

[279] *OpenSSL: The Open Source toolkit for SSL/TLS*. OpenSSL project. URL: www.openssl.org/ (visited on May 9, 2023), archived at https://web.archive.org/web/20230508044427/https://www.openssl.org/ on May 8, 2023 (cit. on pp. 50, 353).

[280] *OpenVPN Protocol*. URL: openvpn.net/community-resources/openvpn-protocol/ (visited on July 20, 2022), archived at https://web.archive.org/web/20230413064334/https://openvpn.net/community-resources/openvpn-protocol/ on Apr. 13, 2023 (cit. on pp. 2, 41).

[281] Christian Paquin, Douglas Stebila, and Goutam Tamvada. "Benchmarking Post-quantum Cryptography in TLS." In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Paris, France: Springer, Heidelberg, Germany, Apr. 15–17, 2020, pp. 72–91. DOI: 10.1007/978-3-030-44223-1_5. IACR ePrint: ia.cr/2019/1447 (cit. on pp. 50, 202, 203, 236).

[282] Jacques Patarin. "Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt'88." In: *Advances in Cryptology – CRYPTO'95*. Ed. by Don Coppersmith. Vol. 963. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 27–31, 1995, pp. 248–261. DOI: 10.1007/3-540-44750-4_20 (cit. on p. 371).

[283] Jacques Patarin. "Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms." In: *Advances in Cryptology – EUROCRYPT'96*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Saragossa, Spain: Springer, Heidelberg, Germany, May 12–16, 1996, pp. 33–48. DOI: 10.1007/3-540-68339-9_4 (cit. on p. 371).

[284] Jacques Patarin. "The Oil and Vinegar Signature Scheme." In: *Dagstuhl Workshop on Cryptography*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Sept. 1997 (cit. on p. 372).

[285] Jacques Patarin, Nicolas Courtois, and Louis Goubin. "QUARTZ, 128–Bit Long Digital Signatures." In: *Topics in Cryptology – CT-RSA 2001*. Ed. by David Naccache. Vol. 2020. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Heidelberg, Germany, Apr. 8–12, 2001, pp. 282–297. DOI: 10.1007/3-540-45353-9_21. URL: www.goubin.fr/papers/rsa2001b.pdf (cit. on p. 372).

[286] Jacques Patarin, Louis Goubin, and Nicolas Courtois. "C^*_{-+} and HM: Variations Around Two Schemes of T. Matsumoto and H. Imai." In: *Advances in Cryptology – ASIACRYPT'98*. Ed. by Kazuo Ohta and Dingyi Pei. Vol. 1514. Lecture Notes in Computer Science. Beijing, China: Springer, Heidelberg, Germany, Oct. 18–22, 1998, pp. 35–49. DOI: 10.1007/3-540-49649-1_4 (cit. on p. 371).

[287] Kenny Paterson and Thyla van der Merwe. "Reactive and Proactive Standardisation of TLS." In: *Security Standardisation Research (SSR) 2016*. Ed. by Lidong Chen, David A. McGrew, and Chris Mitchell. Vol. 10074. Lecture Notes in Computer Science. Gaithersburg, MD, USA: Springer, Dec. 5–6, 2016, pp. 160–186. DOI: 10.1007/978-3-319-49100-4_7. URL: core.ac.uk/reader/81671740 (cit. on pp. 41, 53, 164).

[288] Sebastian Paul, Yulia Kuzovkova, Norman Lahr, and Ruben Niederhagen. "Mixed Certificate Chains for the Transition to Post-Quantum Authentication in TLS 1.3." In: *ASIACCS 22: 17th ACM Symposium on Information, Computer and Communications Security*. Ed. by Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako. Nagasaki, Japan: ACM Press, May 30–June 3, 2022, pp. 727–740. DOI: 10.1145/3488932.3497755. IACR ePrint: ia.cr/2021/1447 (cit. on p. 318).

[289] Chris Peikert. "He Gives C-Sieves on the CSIDH." In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Heidelberg, Germany, May 10–14, 2020, pp. 463–492. DOI: 10.1007/978-3-030-45724-2_16. IACR ePrint: ia.cr/2019/725 (cit. on pp. 29, 59, 244, 359).

[290] Tervor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. Specification. Signal Foundation, Nov. 20, 2016. URL: signal.org/docs/specifications/doubleratchet/ (visited on Aug. 24, 2022), archived at https://web.archive.org/web/20230505114634/https://signal.org/docs/specifications/doubleratchet/ on May 11, 2023 (cit. on p. 56).

[291] Trevor Perrin. *Noise Protocol Framework*. Specification. July 11, 2018. URL: noiseprotocol.org/noise.html (visited on May 9, 2023), archived at https://web.archive.org/web/20230425040355/https://noiseprotocol.org/noise.html on Apr. 25, 2023 (cit. on pp. 56, 100).

[292] Thomas Pornin. *New Efficient, Constant-Time Implementations of Falcon*. Cryptology ePrint Archive, Report 2019/893. Sept. 18, 2019. IACR ePrint: ia.cr/2019/893 (cit. on p. 380).

[293] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *FALCON*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, archived at https://web.archive.org/web/20230429160244/https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 on Apr. 29, 2023 (cit. on pp. 15, 35, 50, 207, 248, 283, 361, 374, 380).

[294] *Ready for tomorrow: Infineon demonstrates first post-quantum cryptography on a contactless security chip*. Infineon Press Release. Infineon, 2017. URL: www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html (visited on May 6, 2023), archived at https://web.archive.org/web/20230506192645/https://www.infineon.com/cms/en/about-infineon/press/press-releases/2017/INFCCS201705-056.html on May 6, 2023 (cit. on p. 344).

[295] Eric Rescorla. *HTTP Over TLS*. RFC 2818. May 2000. DOI: 10.17487/RFC2818 (cit. on p. 41).

[296] Eric Rescorla. *More compatibility measurement results*. Posting on the TLS mailing list. Dec. 22, 2017. URL: mailarchive.ietf.org/arch/msg/tls/6pGGT-wm5vSkacMFPEPvFMEnj-M/# (visited on Oct. 18, 2022), archived at https://web.archive.org/web/20230422015920/https://mailarchive.ietf.org/arch/msg/tls/6pGGT-wm5vSkacMFPEPvFMEnj-M/ on Apr. 22, 2023 (cit. on p. 201).

[297] Eric Rescorla. *Preliminary data on Firefox TLS 1.3 Middlebox experiment*. Posting on the TLS mailing list. Dec. 5, 2017. URL: mailarchive.ietf.org/arch/msg/tls/RBp0X-OWNuWXugFJRV7c_hIU0dI/ (visited on Oct. 18, 2022), archived at https://web.archive.org/web/20230422015931/https://mailarchive.ietf.org/arch/msg/tls/RBp0X-OWNuWXugFJRV7c_hIU0dI/ on Apr. 22, 2023 (cit. on p. 201).

[298] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446 (cit. on pp. 2, 23, 41, 42, 69, 72, 79, 82, 84, 92, 167, 195, 199, 201, 318).

[299] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-rfc8446bis-07. Work in Progress. Internet Engineering Task Force, Mar. 2023. 158 pp. URL: datatracker.ietf.org/doc/draft-ietf-tls-rfc8446bis/07/ (cit. on p. 45).

[300] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246 (cit. on p. 23).

[301] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. *TLS Encrypted Client Hello*. Internet-Draft draft-ietf-tls-esni-16. Work in Progress. Internet Engineering Task Force, Apr. 2023. 48 pp. URL: datatracker.ietf.org/doc/draft-ietf-tls-esni/16/ (cit. on pp. 69, 186).

[302] Eric Rescorla, Nick Sullivan, and Christopher A. Wood. *Semi-Static Diffie–Hellman Key Establishment for TLS 1.3*. Internet-Draft draft-ietf-tls-semistatic-dh-01. Work in Progress. Internet Engineering Task Force, Mar. 7, 2020. 7 pp. URL: datatracker.ietf.org/doc/draft-ietf-tls-semistatic-dh/01/ (cit. on pp. 54, 55, 57, 58).

[303] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Communications of the Association for Computing Machinery* 21.2 (Feb. 1, 1978), pp. 120–126. DOI: 10.1145/359340.359342 (cit. on pp. 2, 5, 27, 206).

[304] Damien Robert. "Breaking SIDH in Polynomial Time." In: *Advances in Cryptology – EUROCRYPT 2023, Part V*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14008. Lecture Notes in Computer Science. Lyon, France: Springer, Heidelberg, Germany, Apr. 23–27, 2023, pp. 472–503. DOI: 10.1007/978-3-031-30589-4_17. IACR ePrint: ia.cr/2022/1038 (cit. on pp. 15, 36, 50).

[305] Johannes Roth, Evangelos G. Karatsiolis, and Juliane Krämer. "Classic McEliece Implementation with Low Memory Footprint." In: *CARDIS*. Vol. 12609. Lecture Notes in Computer Science. Springer, Heidelberg, Germany, 2020, pp. 34–49. DOI: 10.1007/978-3-030-68487-7_3. IACR ePrint: ia.cr/2021/138 (cit. on p. 369).

[306] Jan Rüth, Christian Bormann, and Oliver Hohlfeld. "Large-Scale Scanning of TCP's Initial Window." In: *IMC 2017: 2017 Internet Measurement Conference*. London, United Kingdom: ACM Press, 2017, pp. 304–310. DOI: 10.1145/3131365.3131370. URL: conferences.sigcomm.org/imc/2017/papers/imc17-final43.pdf (cit. on p. 238).

[307] Markku-Juhani O. Saarinen. *API for PQC algorithms*. Posting on the NIST pqc-forum mailing list. Nov. 5, 2016. URL: groups.google.com/a/list.nist.gov/g/pqc-forum/c/QufjmfhuRcs/m/RYLC3pnwBAAJ (visited on Apr. 18, 2023), archived at https://web.archive.org/web/20230509143934/https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/QufjmfhuRcs/m/RYLC3pnwBAAJ on May 9, 2023 (cit. on p. 345).

[308] The Sage Developers. *SageMath, the Sage Mathematics Software System*. 2022. DOI: 10.5281/zenodo.593563. URL: www.sagemath.org (cit. on p. 357).

[309] Joseph A. Salowey, David McGrew, and Abhijit Choudhury. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. RFC 5288. Aug. 2008. DOI: 10.17487/RFC5288 (cit. on p. 22).

[310] Simona Samardjiska, Ming-Shing Chen, Andreas Hulsing, Joost Rijneveld, and Peter Schwabe. *MQDSS*. Tech. rep. National Institute of Standards and Technology, 2019. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions, archived at https://web.archive.org/web/20230509100012/https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions on May 9, 2023 (cit. on p. 36).

[311] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. RFC 6960. June 2013. DOI: 10.17487/RFC6960 (cit. on p. 238).

[312] Stefan Santesson and Hannes Tschofenig. *Transport Layer Security (TLS) Cached Information Extension*. RFC 7924. July 2016. DOI: 10.17487/RFC7924 (cit. on pp. 78, 193, 304).

[313] Jim Schaad. *Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)*. RFC 4211. Sept. 2005. DOI: 10.17487/RFC4211 (cit. on p. 74).

[314] John M. Schanck, Andreas Hulsing, Joost Rijneveld, and Peter Schwabe. *NTRU-HRSS-KEM*. Tech. rep. National Institute of Standards and Technology, 2017. URL: csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions, archived at https://web.archive.org/web/20230509100150/https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions on May 9, 2023 (cit. on p. 50).

[315] John M. Schanck and Douglas Stebila. *A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret*. Internet-Draft draft-schanck-tls-additional-keyshare-00. Work in Progress. Internet Engineering Task Force, Apr. 2017. 1-10. URL: datatracker.ietf.org/doc/html/draft-schanck-tls-additional-keyshare-00 (cit. on p. 50).

[316] John M. Schanck, William Whyte, and Zhenfei Zhang. *Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2*. Internet-Draft draft-whyte-qsh-tls12-02. Work in Progress. Internet Engineering Task Force, Jan. 2017. 1-19. URL: datatracker.ietf.org/doc/html/draft-whyte-qsh-tls12-02 (cit. on p. 50).

[317] Bruce Schneier. "Cryptographic design vulnerabilities." In: *Computer* 31.9 (1998), pp. 29–33. DOI: 10.1109/2.708447. URL: scholar.archive.org/work/d2o4hgd6ofc3phdrwstd7sqtt4 (cit. on p. 336).

[318] Peter Schwabe. *NIST's views on the cost of memory*. Posting on the NIST pqc-forum mailing list. URL: groups.google.com/a/list.nist.gov/g/pqc-forum/c/UFxDg9TenNE/m/uPtzLxTAAgAJ (visited on Oct. 13, 2022), archived at https://web.archive.org/web/20230509132907/https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/UFxDg9TenNE/m/uPtzLxTAAgAJ on May 9, 2023 (cit. on p. 37).

[319] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. *CRYSTALS-KYBER*. Tech. rep. National Institute of Standards and Technology, 2022. URL: csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, archived at https://web.archive.org/web/20230429160244/https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022 on Apr. 29, 2023 (cit. on pp. 5, 15, 35, 206, 248, 282, 361).

[320] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys." In: *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part I*. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Updated version available via url and IACR ePrint. Darmstadt, Germany: Springer, Heidelberg, Germany, Oct. 4–8, 2021, pp. 3–22. DOI: 10.1007/978-3-030-88418-5_1. IACR ePrint: ia.cr/2021/779. URL: wggrs.nl/p/kemtlspdk (cit. on pp. 8, 9, 91, 105, 164, 165, 175, 177, 181, 186, 480).

[321] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures." In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. Updated version available via url and IACR ePrint. Virtual Event, USA: ACM Press, Nov. 9–13, 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350. IACR ePrint: ia.cr/2020/534. URL: wggrs.nl/p/kemtls (cit. on pp. 7, 91, 94, 105, 143, 164, 165, 175, 177, 181, 186, 343, 481).

[322] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *ATEC '12: Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Boston, MA, USA: USENIX Association, 2012. URL: www.usenix.org/system/files/conference/atc12/atc12-final39.pdf (cit. on pp. 348, 354).

[323] Julian Seward and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision." In: *ATEC '05: 2005 USENIX Annual Technical Conference*. Anaheim, CA, USA: USENIX Association, Apr. 10–15,

2005, pp. 17–30. URL: www.usenix.org/conference/2005-usenix-annual-technical-conference/using-valgrind-detect-undefined-value-errors-bit (cit. on pp. 348, 354).

[324] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *35th Annual Symposium on Foundations of Computer Science*. Santa Fe, NM, USA: IEEE Computer Society Press, Nov. 20–22, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. arXiv: quant-ph/9508027 (cit. on pp. 4, 34).

[325] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. "Assessing the Overhead of Post-Quantum Cryptography in TLS 1.3 and SSH." In: *CoNEXT '20: 16th International Conference on Emerging Networking EXperiments and Technologies*. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 149–156. DOI: 10.1145/3386367.3431305. URL: www.researchgate.net/publication/346646724_Assessing_the_Overhead_of_Post-Quantum_Cryptography_in_TLS_13_and_SSH (cit. on p. 236).

[326] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. "Post-Quantum Authentication in TLS 1.3: A Performance Study." In: *ISOC Network and Distributed System Security Symposium – NDSS 2020*. Updated version available on IACR ePrint. San Diego, CA, USA: The Internet Society, Feb. 23–26, 2020. IACR ePrint: ia.cr/2020/071 (cit. on pp. 50, 236, 238, 239, 322, 327, 343).

[327] Brian Smith. *Ring*. URL: github.com/briansmith/ring (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230428144416/https://github.com/briansmith/ring on Apr. 28, 2023 (cit. on p. 189).

[328] Brian Smith. *WebPKI*. URL: github.com/briansmith/webpki (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230227221705/https://github.com/briansmith/webpki on Feb. 27, 2023 (cit. on p. 189).

[329] *SonarQube*. URL: sonarqube.com (visited on May 9, 2023), archived at https://web.archive.org/web/20230501002547/https://www.sonarsource.com/products/sonarqube/ on May 1, 2023 (cit. on p. 347).

[330] Fang Song. "A Note on Quantum Security for Post-Quantum Cryptography." In: *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014*. Ed. by Michele Mosca. Waterloo, Ontario, Canada: Springer, Heidelberg, Germany, Oct. 1–3, 2014, pp. 246–265. DOI: 10.1007/978-3-319-11659-4_15. IACR ePrint: ia.cr/2014/709 (cit. on p. 93).

[331] Drew Springall, Zakir Durumeric, and J. Alex Halderman. "Measuring the Security Harm of TLS Crypto Shortcuts." In: *IMC 2016: 2016 Internet Measurement Conference*. Santa Monica, California, USA: ACM Press, 2016, pp. 33–47. DOI: 10.1145/2987443.2987480 (cit. on p. 33).

[332] Douglas Stebila, Scott Fluhrer, and Shay Gueron. *Hybrid key exchange in TLS 1.3*. Internet-Draft draft-ietf-tls-hybrid-design-06. Work in Progress. Internet Engineering Task Force, Feb. 2023. 22 pp. URL: datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/06/ (cit. on p. 50).

[333] Douglas Stebila and Michele Mosca. "Post-quantum Key Exchange for the Internet and the Open Quantum Safe Project." In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. St. John's, NL, Canada: Springer, Heidelberg, Germany, Aug. 10–12, 2016, pp. 14–37. DOI: 10.1007/978-3-319-69453-5__2. IACR ePrint: ia.cr/2016/1017. URL: github.com/open-quantum-safe/ (cit. on pp. 50, 236, 335, 358, 457).

[334] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++." In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. San Francisco, CA, USA, 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186. URL: research.google/pubs/pub43308/ (cit. on p. 354).

[335] Falko Strenzke. *How to implement the public Key Operations in Code-based Cryptography on Memory-constrained Devices*. Cryptology ePrint Archive, Report 2010/465. Apr. 20, 2012. IACR ePrint: ia.cr/2010/465 (cit. on p. 369).

[336] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*." In: *POPL 2016: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Jan. 20–23, 2016, pp. 256–270. DOI: 10.1145/2914770.2837655. URL: www.fstar-lang.org/papers/mumon/ (cit. on p. 357).

[337] Erik Sy, Christian Burkert, Hannes Federrath, and Mathias Fischer. "Tracking Users across the Web via TLS Session Resumption." In: *ACSAC '18: Proceedings of the 34th Annual Computer Security Applications Conference*. San Juan, PR, USA: ACM Press, Dec. 3, 2018, pp. 289–299. DOI: 10.1145/3274694.3274708. arXiv: 1810.07304 (cit. on p. 79).

[338] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris-Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. "Scalable Bias-Resistant Distributed Randomness." In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 444–460. DOI: 10.1109/SP.2017.45. IACR ePrint: ia.cr/2016/1067 (cit. on p. 305).

[339] George Tasopoulos, Jinhui Li, Apostolos P. Fournaris, Raymond K. Zhao, Amin Sakzad, and Ron Steinfeld. "Performance Evaluation of Post-Quantum TLS 1.3 on Resource-Constrained Embedded Systems." In: *ISPEC 2022: Information Security Practice and Experience*. Ed. by Chunhua Su, Dimitris Gritzalis, and Vincenzo Piuri. Taipei, Taiwan: Springer, Cham, 2022, pp. 432–451. IACR ePrint: ia.cr/2021/1553 (cit. on pp. 316, 320).

[340] David Taylor, Trevor Perrin, Thomas Wu, and Nikos Mavrogiannopoulos. *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*. RFC 5054. Nov. 2007. DOI: 10.17487/RFC5054 (cit. on p. 42).

[341] Edlyn Teske. "An Elliptic Curve Trapdoor System." In: *Journal of Cryptology* 19.1 (Jan. 2006), pp. 115–133. DOI: 10.1007/s00145-004-0328-3. IACR ePrint: ia.cr/2003/058 (cit. on p. 36).

[342] The Clang Team. *clang-tidy – Extra Clang Tools 17.0.0git documentation*. URL: clang.llvm.org/extra/clang-tidy/ (visited on May 9, 2023), archived at https://web.archive.org/web/20230422185417/https://clang.llvm.org/extra/clang-tidy/ on Apr. 22, 2023 (cit. on pp. 347, 354).

[343] The Clang Team. *UndefinedBehaviorSanitizer – Clang 17.0.0git documentation*. URL: clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on May 9, 2023), archived at https://web.archive.org/web/20230502092551/https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html on May 2, 2023 (cit. on pp. 348, 354).

[344] The Coq Development Team. *The Coq Proof Assistant*. Jan. 2022. DOI: 10.5281/zenodo.5846982. URL: coq.inria.fr (cit. on p. 357).

[345] The Open Quantum Safe project. *Open Quantum Safe*. See also [333]. URL: openquantumsafe.org (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230323084054/https://openquantumsafe.org/ on Mar. 25, 2023 (cit. on pp. 191, 320).

[346] The QEMU Project Developers. *QEMU: A generic and open source machine emulator and virtualizer*. URL: www.qemu.org (visited on May 9, 2023), archived at https://web.archive.org/web/20230508124536/https://www.qemu.org/ on May 8, 2023 (cit. on p. 355).

[347] Gaius Suetonius Tranquillus. *De vita Caesarum*. Latin. Original text and translation by J.C. Rolfe available at https://penelope.uchicago.edu/Thayer/E/Roman/Texts/Suetonius/12Caesars/home.html. 121 (cit. on p. 1).

[348] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. "Measuring small subgroup attacks against Diffie-Hellman." In: *ISOC Network and Distributed System Security Symposium – NDSS 2017*. San Diego, CA, USA: The Internet Society, Feb. 26–Mar. 1, 2017. DOI: 10.14722/ndss.2017.23171. IACR ePrint: ia.cr/2016/995 (cit. on p. 33).

[349] Paul C. van Oorschot and Michael J. Wiener. "Parallel Collision Search with Cryptanalytic Applications." In: *Journal of Cryptology* 12.1 (Jan. 1999), pp. 1–28. DOI: 10.1007/PL00003816 (cit. on p. 392).

[350] Victor Vasiliev. *Application-Layer Protocol Settings*. Posting on the TLS mailing list. July 6, 2020. URL: mailarchive.ietf.org/arch/msg/tls/S_9C4TyKT5wQB5XU2MAkTb6pIW0/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20220926054341/https://mailarchive.ietf.org/arch/msg/tls/S_9C4TyKT5wQB5XU2MAkTb6pIW0/ on Sept. 26, 2022 (cit. on p. 68).

[351] Bas Westerbaan. *Sizing up post-quantum signatures*. Post on the Cloudflare blog. Cloudflare, Nov. 2021. URL: blog.cloudflare.com/sizing-up-post-quantum-signatures/ (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20230425041856/https://blog.cloudflare.com/sizing-up-post-quantum-signatures/ on Apr. 25, 2023 (cit. on pp. 50, 236).

[352] William Whyte, Zhenfei Zhang, Scott Fluhrer, and Oscar Garcia-Morchon. *Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3*. Internet-Draft draft-whyte-qsh-tls13-06. Work in Progress. Internet Engineering Task Force, Oct. 2017. 19 pp. URL: datatracker.ietf.org/doc/html/draft-whyte-qsh-tls13-06 (cit. on p. 50).

[353] Thom Wiggers. "Energy-Efficient ARM64 Cluster with Cryptanalytic Applications - 80 Cores That Do Not Cost You an ARM and a Leg." In: *Progress in Cryptology - LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America*. Ed. by Tanja Lange and Orr Dunkelman. Vol. 11368. Lecture Notes in Computer Science. Havana, Cuba: Springer, Heidelberg, Germany, Sept. 20–22, 2017, pp. 175–188. DOI: 10.1007/978-3-030-25283-0_10. IACR ePrint: ia.cr/2018/888. URL: wggrs.nl/p/armcluster (cit. on pp. 481, 483).

[354] Thom Wiggers and Simona Samardjiska. "Practically Solving LPN." In: *2021 IEEE International Symposium on Information Theory (ISIT)*. Melbourne, Australia: IEEE Information Theory Society, July 12–20, 2021, pp. 2399–2404. DOI: 10.1109/ISIT45174.2021.9518109. IACR ePrint: ia.cr/2021/962. URL: wggrs.nl/p/lpn (cit. on pp. 14, 480).

[355] Wikipedia contributors. *List of public corporations by market capitalization*. Wikipedia. URL: en.wikipedia.org/w/index.php?title=List_of_public_corporations_by_market_capitalization&oldid=1148885066 (visited on Apr. 17, 2023) (cit. on p. 2).

[356] Wireshark developers. *Wireshark*. URL: www.wireshark.org (visited on May 9, 2023) (cit. on p. 203).

[357] United Nations International Telecommunication Union Telecom World. *Ministerial Roundtable — "Cutting the cost: can affordable access accelerate digital transformation?"* Oct. 13, 2021. URL: digital-world.itu.int/ministerial-roundtable-cutting-the-cost-can-affordable-access-accelerate-digital-transformation/ (visited on Mar. 20, 2023), archived at https://web.archive.org/web/20230203010139/https://digital-world.itu.int/ministerial-roundtable-cutting-the-cost-can-affordable-access-accelerate-digital-transformation/ on Feb. 3, 2023 (cit. on p. 363).

[358] Paul Wouters, Hannes Tschofenig, John Gilmore, Samuel Weiler, and Tero Kivinen. *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7250. June 2014. DOI: 10.17487/RFC7250 (cit. on p. 279).

[359] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA." In: *Journal of Cryptographic Engineering* 7.2 (June 2017), pp. 99–112. DOI: 10.1007/s13389-017-0152-y. IACR ePrint: ia.cr/2016/224 (cit. on p. 279).

[360] Daniel Zelle, Christoph Krauß, Hubert Strauß, and Karsten Schmidt. "On Using TLS to Secure In-Vehicle Networks." In: *ARES '17: Proceedings of the 12th International Conference on Availability, Reliability and Security*. Reggio Calabria, Italy: Association for Computing Machinery, 2017. DOI: 10.1145/3098954.3105824 (cit. on p. 2).

[361] Zephyr Project. *Zephyr Project*. URL: www.zephyrproject.org (visited on Dec. 22, 2022), archived at https://web.archive.org/web/20221219045635/https://www.zephyrproject.org/ on Dec. 19, 2022 (cit. on p. 320).

[362] Bin Zhang, Lin Jiao, and Mingsheng Wang. "Faster Algorithms for Solving LPN." In: *Advances in Cryptology – EUROCRYPT 2016, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Lecture Notes in Computer Science. Vienna, Austria: Springer, Heidelberg, Germany, May 8–12, 2016, pp. 168–195. DOI: 10.1007/978-3-662-49890-3_7. IACR ePrint: ia.cr/2016/275 (cit. on p. 392).

[363] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library." In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 31–Nov. 2, 2017, pp. 1789–1806. DOI: 10.1145/3133956.3134043. IACR ePrint: ia.cr/2017/536 (cit. on p. 338).

Availability of software and experimental results

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.

In this appendix, we list the software implementations that have been used to obtain the results in different chapters, as well as where to find the raw measurement data obtained in our experiments. The latest version of any software, documentation, and recorded experimental data will be available via https://thomwiggers.nl/publication/thesis/.

Formally analyzing KEMTLS in Tamarin

The two Tamarin models, including the Tamarin-generated proofs, are available under the CC-BY 4.0 license at:

> Thom Wiggers. (2022). KEMTLS-TLS13Tamarin. Archived at Zenodo. DOI: 10.5281/zenodo.7844627. *Model described in section 9.3.*
>
> Douglas Stebila. (2022). Tamarin-multi-stage-model. Archived at Zenodo. DOI: 10.5281/zenodo.7844620. *Model described in section 9.5.*

Experiments with post-quantum TLS 1.3, OPTLS, KEMTLS, and KEMTLS-PDK on simulated networks

For the experiments in chapters 11 to 14 we used and modified open-source cryptographic software and TLS libraries. In addition, we wrote new software to facilitate our experiments and to create certificates. Our modifications and the new software are described in chapter 10. All software we modified is under permissive open-source licenses; we place any newly-developed code into the public domain (CC0).

The software used in the experiments for this thesis as well as the experimental results can be found at:

Thom Wiggers. (2023). thomwiggers/kemtls-experiment: thesis-deposit. Archived at Zenodo. DOI: 10.5281/zenodo.10143416

Full measurement results from all experiments reported in chapters 11 to 14 can be found in the `measuring/archived-results` folder.

For OPTLS, we report additional preliminary results with higher-security parameters for CSIDH. These benchmarks were done independently, and the software and measurement results can be found at:

Fabio Campos and Thom Wiggers. (2023). kemtls-secsidh/code. Archived at Zenodo. DOI: 10.5281/zenodo.10142359
Thom Wiggers. (2023). kemtls-secsidh/secsidh. Archived at Zenodo. DOI: 10.5281/zenodo.10142361

Measuring the performance of KEMTLS in embedded systems

The code for the embedded client application, as well as the server software based on an earlier version of the Rustls implementation of post-quantum TLS and KEMTLS used in the experiments on simulated networks, is available under permissive licensing at:

Ruben Gonzalez. (2022). wolfssl-kemtls-experiments. Archived at Zenodo. DOI: 10.5281/zenodo.7844877

Measuring the performance of KEMTLS over the internet

In chapter 15, we describe an implementation of post-quantum TLS 1.3 and KEMTLS(-PDK) in the Go standard library. This version integrates CIRCL [141] and can be used as a replacement for the standard Go compiler to compile other Go programs. Hence, anyone wanting to experiment with post-quantum algorithms or the new handshake protocols can compile programs with our modified Go compiler.

Sofía Celi and Armando Faz-Hernandéz. (2021). Go-kemtls. Archived at Zenodo. DOI: 10.5281/zenodo.7844629

The source code of the client application used in experiments is available via:

Sofía Celi and Armando Faz-Hernandéz. (2021). KEMTLS-local-measurements. Archived at Zenodo.
DOI: 10.5281/zenodo.7844865

Improving software quality in standardization projects
Chapter 17 discusses PQClean. This ongoing project is available on GitHub at https://github.com/PQClean/PQClean/, but an archived copy of the state of the project at the end of round 3 of the NIST PQC standardization project can be found at:

> Thom Wiggers, Douglas Stebila, Matthias J. Kannwischer, and Peter Schwabe. (2022). PQClean (tag round3). Archived at Zenodo. DOI: 10.5281/zenodo.7844947

Verifying post-quantum signatures in 8 kB of RAM
The implementations of the signature schemes, including scaffolding for executing the benchmarks, can be found at:

> Ruben Gonzalez and Matthias J. Kannwischer. (2021). Streaming Post-Quantum Public Keys and Signatures. Archived at Zenodo. DOI: 10.5281/zenodo.7863702

Practically solving LPN
The software that allows experimenting with and execution of attacks on LPN problems can be found at:

> Thom Wiggers. (2021). thomwiggers/lpn. Archived at Zenodo. DOI: 10.5281/zenodo.4655955

Summary

The Transport Layer Security (TLS) protocol is the most-used secure communication protocol on the internet. In this thesis, we examine the challenges and trade-offs when protecting TLS against attacks using large-scale quantum computers that can break the cryptography used in the current versions.

In chapter 1, we introduce cryptography in general and our subject in particular. Chapter 2 establishes common notions, notation, and definitions that we use in other chapters. The remainder is organized into three parts.

Part I: Post-Quantum TLS

This part introduces the protocols and the changes that need to be made to transition to post-quantum cryptography.

Chapter 3: The TLS protocol

In this background chapter, we examine the current state-of-the-art version of the TLS protocol, version 1.3. TLS 1.3 is based on Diffie–Hellman (DH) key exchange, which has no post-quantum equivalent. We examine existing proposals to integrate post-quantum key encapsulation mechanism (KEM) and signature schemes in the protocol with small changes. In the conclusion of this chapter, we briefly review the literature on experiments with post-quantum TLS, showing that especially the large sizes and/or computational requirements of post-quantum signature schemes are challenging for the deployment of post-quantum TLS.

Chapter 4: Post-quantum OPTLS

Chapter 4 revisits OPTLS, which was an early proposal by Krawczyk and Wee for the TLS 1.3 handshake protocol which was later abandoned. This protocol uses a different authentication mechanism than the signed-key-exchange

approach that is common across TLS versions that are used today: OPTLS authenticates by completing a DH key exchange with a long-term DH public key in certificates. This key exchange requires the authenticated party to use the secret key corresponding to that public key to compute a shared secret key; the shared secret key is in turn used to compute a message-authentication code (MAC). This MAC proves that the authenticated party has access to the secret key and thus must be who they claim to be.

The OPTLS handshake protocol seems a promising approach to avoid the downsides of post-quantum signature schemes that we identified in the previous chapter. Unfortunately, the way OPTLS uses DH for authentication relies on the non-interactive key exchange (NIKE) properties of DH. This is a problem for constructing a post-quantum version of OPTLS, as there are few post-quantum NIKEs: the only scheme that fits in the OPTLS key-exchange messages, CSIDH, requires large amounts of computation time and its security is the subject of debate.

Chapter 5: Post-quantum KEMTLS

This chapter introduces KEMTLS, the main contribution of this thesis. KEMTLS takes the idea from OPTLS to construct a TLS handshake without handshake signatures but uses KEMs instead of NIKE. To get around the additional round-trip that KEM-based authentication would impose, KEMTLS changes the TLS handshake so that the client can use an implicitly-authenticated key to send its request to the server in the same place as it would in TLS 1.3. In TLS 1.3, the server can send data before the client has sent its request, but we argue that the server often has to wait for the client's request before it can send anything useful; this certainly is the case in HTTP, the protocol used in web browsing. In this way, KEMTLS preserves the ability of the client to send its request after one round-trip time (RTT), while at the same time eliminating handshake signatures. This saves significant amounts of bandwidth and computation.

This chapter also discusses a variant of KEMTLS with client authentication. Unfortunately, the client can not send its client certificate before it has (implicitly) authenticated the server, so the handshake protocol of mutually authenticated KEMTLS has an additional round-trip before the client can send its request. Finally, deploying KEMTLS does require obtaining certificates that have KEM public keys, and we briefly discuss the way certificates are currently issued and how this could be done for post-quantum KEMs.

Chapter 6: More efficient KEMTLS with pre-distributed keys

KEMTLS assumes that the client has no prior knowledge of the server's long-term keys. However, in many cases the client might have such knowledge: clients often connect many times to the same server. In such cases, the long-term public key might be cached or otherwise be known to the client before the connection is set up. Under this assumption, this chapter describes a more efficient key-exchange protocol which we call KEMTLS-PDK, KEMTLS with pre-distributed keys. By using the prior knowledge of the server's long-term KEM public key, we allow the client to submit a ciphertext to the server in the first handshake message. This significantly abbreviates the handshake protocol and avoids the exchange of certificates entirely, reducing the amount of data transmitted. Because the ciphertext that is encapsulated to the server's long-term KEM public key is implicitly authenticated, we can use this key to encrypt the client's certificate. This allows us to construct a mutually authenticated KEMTLS-PDK handshake that has the same number of round-trip times as unilaterally authenticated KEMTLS-PDK, saving a full round-trip compared to the mutually authenticated KEMTLS handshake. Finally, compared to pre-shared key variants of TLS 1.3, KEMTLS-PDK avoids symmetric keys and thus does not need to distribute or protect sensitive shared secret keys.

Part II: Security of KEMTLS

In part I, we proposed two new handshake protocols for TLS: KEMTLS and KEMTLS-PDK. In this part of the thesis, we discuss their security properties.

Chapter 7: Security of KEMTLS

In chapter 7, we model and prove the security of unilaterally and mutually authenticated KEMTLS. We provide a granular definition of forward secrecy that captures the nuances of the authentication properties at different stages in the KEMTLS handshake. Our proof is in the reductionist security model: this quantifies the security of the protocol by the security properties of the underlying primitives, such as the security of KEMs, MACs, and hash functions. Specifically, we prove match security and multi-stage security of KEMTLS. A surprising result is that the ephemeral key exchange of KEMTLS requires IND-1CCA security and not IND-CPA security as one might expect.

Chapter 8: Security of KEMTLS-PDK

This chapter instantiates KEMTLS-PDK in the same model as KEMTLS. Using the same approach as in the previous chapter, we give the security properties of KEMTLS-PDK in the reductionist security model and prove match and multi-stage security for the protocol. We take careful consideration to correctly model the first client message in KEMTLS-PDK, which is replayable. Finally, we briefly discuss the security of the two protocols together.

Chapter 9: Formally analyzing KEMTLS in Tamarin

Proving the security of a protocol using pen-and-paper methods is tedious and error-prone. When reading and writing these proofs, it is easy to miss details, or indeed fill in details by reading "between the lines". Computer-assisted proofs fill this gap: by rigorously specifying the model and every step of the proof in a computer program, we can use computers to check our work. In this chapter, we use the Tamarin symbolic analysis tool to examine the security of KEMTLS and KEMTLS-PDK. We approach the security of KEMTLS(-PDK) in two ways: by adapting an existing model of TLS 1.3 to represent our proposals, and by transforming our pen-and-paper properties to a Tamarin model. We show that KEMTLS and KEMTLS-PDK is secure in both of our models. The first approach has the benefit of using the fully-featured TLS 1.3 model, which includes details like handshake encryption, and reuses security lemmas that have already been battle-tested in the other model. This model closely resembles what an implementation would look like. The second approach sticks much closer to the pen-and-paper models from chapters 7 and 8 and does not feature details such as handshake encryption or the full key-derivation sequence. However, it can specify and verify the granular security properties from our reductionist security analyses and allowed us to find some mistakes in the original versions of our proofs. These mistakes have been fixed in the versions of the proofs that appear in this thesis.

Part III: Implementing and measuring post-quantum TLS

This final part of the main body of our thesis instantiates the protocol designs discussed in part I with post-quantum primitives and measures their performance. This allows us to compare the performance of these variants.

Chapter 10: Implementing and measuring post-quantum TLS in Rust

This chapter discusses how we implemented post-quantum TLS 1.3, OPTLS, KEMTLS, and KEMTLS-PDK for the experiments in chapters 11 to 14. We show how we modified the TLS handshake state machine in Rustls, how we generate the certificates for our experiments, and how we integrated the post-quantum primitives. We also discuss our emulated network setup in this chapter.

Chapter 11: Performance of post-quantum TLS

In this chapter, we discuss instantiations of post-quantum TLS 1.3 with post-quantum primitives at NIST PQC security levels I, III and V. We choose many different combinations of algorithms for the post-quantum handshakes. This allows us to make comparisons across the many different schemes that are available. Our results show that, on our server-grade hardware, the performance of most primitives is fast enough for use in TLS 1.3. Hash-based signature schemes, however, perform poorly due to their large sizes and significant computational requirements. Handshake sizes in general, which are in large part determined by the post-quantum signature schemes used, significantly affect the handshake performance; especially if the size exceeds the roughly 15 kB initial congestion window. If this size is exceeded, an extra round-trip is introduced to the handshake by the TCP congestion control algorithms. Our results mirror those of other experiments with post-quantum instantiations of TLS, in Kyber, Dilithium, and Falcon seem the most promising algorithms for use in post-quantum TLS 1.3. The results also suggest that using different signature algorithms for handshake authentication and offline signatures may be necessary to balance handshake size and requirements for side-channel protections and special hardware support.

Chapter 12: Performance of post-quantum OPTLS

This chapter instantiates OPTLS with the post-quantum NIKE CSIDH. We discuss why CSIDH is currently the only scheme that is suitable for use in post-quantum OPTLS, but also cover the discussion on the security of this primitive. Next, we show that although CSIDH has very small public keys its runtime performance is too poor for it to be a strong candidate for post-quantum TLS. This problem is exacerbated if, as we discuss, we need to size up the parameters of CSIDH from 512 bits to 5000 bits or beyond.

Chapter 13: Performance of KEMTLS

This chapter covers the performance of KEMTLS. We instantiate the KEMTLS handshake with post-quantum primitives at NIST PQC security levels I, III, and V in similar ways as we have in chapter 11. This allows us to make direct comparisons between equivalent TLS 1.3 and KEMTLS handshakes. Our results show that compared to post-quantum TLS 1.3, unilaterally authenticated KEMTLS saves significant amounts of handshake data and computation time. Mutually authenticated KEMTLS however suffers from the extra round-trip required to protect the client certificate message, although it still saves at least 30 % of data in most of our experiments.

Chapter 14: Performance of KEMTLS-PDK

The performance of KEMTLS-PDK is discussed in this chapter. Similar to the previous chapters, we show how it can be instantiated with post-quantum primitives at NIST PQC security levels I, III, and V. Because KEMTLS-PDK assumes that the client already has the server's long-term KEM public key and we do not need to transmit it, KEMTLS-PDK additionally allows us to use the conservative Classic McEliece algorithm for handshake authentication. This algorithm has very large public keys, but very small ciphertexts. We compare the performance of the KEMTLS-PDK handshakes in this chapter to those of a variant of TLS 1.3. This variant avoids transmission of certificates by allowing the client to indicate that the server can omit them: this is similar to our pre-distributed-key scenario and allows for apples-to-apples comparisons with TLS. KEMTLS-PDK performs slightly better than this cached-TLS variant and saves handshake data in most instantiations. We also compare the performance of KEMTLS-PDK with KEMTLS: by avoiding the transmission of server public keys we (expectedly) save significant amounts of data.

Chapter 15: Measuring the performance of KEMTLS over the internet

The previous chapters discussed the performance of post-quantum TLS variants in simulated network environments. In this chapter, we show the results of an experiment that measured the performance of post-quantum TLS and KEMTLS(-PDK) in a more realistic scenario. We implemented the protocols in the Go standard library and used this for connections between two continents. Over these connections, we ran a real-world application. We also show

how to experiment in production systems, by delegating trust from existing, CA-issued certificates to not-yet-standardized post-quantum schemes. Our results show that the impact of post-quantum primitives is noticeable on handshake completion times, but that the impact is manageable. Handshakes using KEMTLS complete faster than those using post-quantum TLS 1.3, but mutually authenticated KEMTLS suffers from the additional round-trip times.

Chapter 16: Measuring the performance of KEMTLS in embedded systems

This chapter gives a first look at the performance of KEMTLS when implemented on a device that has significantly less power than the servers used in previous experiments. We implemented post-quantum TLS and KEMTLS clients on an ARM Cortex-M4-based microcontroller and measure the handshakes for different network environments. We focus on network settings that are relevant to the embedded space, namely Narrowband-IoT and LTE Machine-Type communication. Our results show that KEMTLS uses less memory than TLS 1.3, but that for unilaterally authenticated handshakes there was no significant difference in code size. Our run times show that in both protocols post-quantum cryptography (PQC) primitives require a significant amount of computational time during the handshake, sometimes requiring over 50 % of the entire handshake time. Even in the LTE-M setting, the percentage of cycles spent in PQC computations is considerable. However, in the bandwidth-constrained NB-IoT setting, handshake times are mostly driven by handshake size. As KEMTLS handshakes are often smaller, using this protocol may help the performance.

Chapter 17: Improving software quality in standardization projects

This last chapter of the main body takes a step back from the post-quantum protocols and instead looks at the state of the software that has been used in all of our experiments. The NIST PQC standardization project submissions were accompanied by reference implementations of the schemes, but these were often of poor quality and not immediately suitable for use in experiments. In this chapter, we delve into this problem. We argue that the NIST PQC standardization effort—and future public cryptography standardization initiatives—could be improved by having a more extensive software

framework prepared in advance by the organizers for submitters, relying on modern continuous integration and testing tools. If this is done, the large number of bugs that we found could have largely been avoided. We attempt to lay out the requirements for a testing framework, based on our experience in the PQClean project, where we assembled a collection of standalone C implementations of NIST PQC submissions and developed a continuous integration testing approach to improve the software we assembled. Finally, we look beyond PQClean's central focus on "cleaning" C implementations and discuss alternatives to C for representing specifications as well as extensions beyond testing frameworks for cryptographic standardization processes.

Additional papers

Finally, we include two additional papers that are slightly outside of the main subject of this thesis.

Appendix A: Verifying post-quantum signatures in 8 kB of RAM

In this chapter, we discuss how post-quatum signatures can be verified on devices with very small memory. Motivated by a use case from the automotive sectory, we assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big and SPHINCS$^+$ signatures do not fit in this memory. To make signature verification work for these schemes, we show how to stream in public keys and signatures to the verification subroutine. Due to the memory requirements for efficient Dilithium implementations, we use streaming for the public key to give us more space to cache more intermediate results.

Appendix B: Practically solving LPN

In this chapter, we discuss the Learning Parity with Noise (LPN) problem. The best algorithms for the LPN problem require sub-exponential time and memory. This often makes memory, and not time, the limiting factor for practical attacks, which seem to be out of reach even for relatively small parameters. In this chapter, we try to bring the state-of-the-art in solving LPN closer to the practical realm. We improve previous algorithms that find

combinations of attacks to solve particular LPN problems, by reducing the search space. Then, we show how memory constraints can be added to the search algorithm, which in turn helps find more practical attacks. We show how to, and execute, attacks on the largest LPN parameters as of when the paper was published.

Samenvatting

Dit proefschrift gaat over het Transport Layer Security (TLS) protocol, het meestgebruikte beveiligde communicatieprotocol is op het internet. Specifieker bekijken we wat de uitdagingen en afwegingen zijn om het protocol resistent te maken tegen aanvallen met *kwantumcomputers*; dit zijn computers die kwantumeffecten gebruiken in plaats van de 'gebruikelijke' nullen en enen. Kwantumcomputers kunnen gebruikt worden om de veiligheid van huidige versies van TLS te breken; cryptografische systemen die bestand zijn tegen aanvallen met kwantumcomputers noemen we *post-kwantum*. In de hoofdstukken 1 en 2 wordt dit probleem verder uitgewerkt en de context geïntroduceerd. De rest van het proefschrift behandelt de verschillende aspecten van dit onderweg in drie delen.

Deel I: Post-kwantum TLS

In dit deel wordt eerst het originele TLS protocol (versie 1.3) en bestaande voorstellen voor post-kwantum TLS besproken. Daarna bespreken we eerst een ouder voorstel voor het TLS 1.3 protocol, OPTLS. Dit alternatieve protocol maakt geen gebruik van digitale handtekeningen: in de post-kwantumsetting zijn die namelijk kostbaar in termen van grootte en rekentijd. Helaas werkt OPTLS niet zonder zogenoemde niet-interactieve sleuteluitwisselingsalgoritmen, en daarvoor hebben we weinig geschikte post-kwantum kandidaten. De rest van dit hoofdstuk introduceert KEMTLS, de belangrijkste bijdrage van dit werk, en de variant KEMTLS-PDK, welke gebaseerd zijn op het idee achter OPTLS, maar door een slimme truc niet een extra ronde heen-en-weer-communicatie nodig hebben. KEMTLS maakt gebruik van zogenoemde KEMs, een vorm van sleuteluitwisselingsalgoritmen dat op dit moment gestandaardiseerd wordt door het Amerikaanse NIST. KEMs zijn efficiënter dan post-kwantumalgoritmen voor het maken van digitale handtekening, waardoor we KEMTLS voorstellen als een efficiënter alternatief dan eenvoudigweg TLS 1.3 met post-kwantumprimitieven te instantiëren.

Deel II: Veiligheid van KEMTLS

In dit deel bewijzen we de veiligheid van de nieuwe protocollen die we voorgesteld hebben. We bewijzen de veiligheidseigenschappen van KEMTLS en KEMTLS-PDK in het reductionistische model met een bewijs van zogenoemde *match* en *multi-stage security* gebaseerd op stapsgewijze reducties in de vorm van spellen. Omdat het maken van bewijzen met pen en papier een foutgevoelige aangelegenheid is, geven we ook een bewijs van de eigenschappen van KEMTLS(-PDK) dat gebruik maakt van Tamarin, een computerprogramma dat bewijzen van symbolische modellen van beveiligingsprotocollen kan controleren.

Deel III: Implementeren en meten van post-kwantum TLS

Dit deel bespreekt hoe de besproken post-kwantum TLS-protocollen geïmplementeerd kunnen worden. We laten in de hoofdstukken 11 tot 14 zien hoe snel de protocollen een beveiligde verbinding kunnen opzetten wanneer ze gebruik maken van verschillende cryptografische primitieven met NIST veiligheidsniveaus I, III en V. We vergelijken zo de prestaties van post-kwantum TLS 1.3, OPTLS, KEMTLS en KEMTLS-PDK. Voor OPTLS gebruiken we CSIDH, een specifiek post-kwantum niet-interactief sleuteluitwisselingsalgoritme, maar het blijkt dat dit algoritme te langzaam is voor gebruik in TLS. Het gebruiken van KEMTLS kan daarentegen voor besparingen in de tijd en uitgewisselde data die nodig is voor het opzetten van een verbinding zorgen. We bekijken in opvolgende hoofdstukken de prestaties van post-kwantum TLS en KEMTLS in een realistischer setting, namelijk voor verbindingen van een applicatie die over het internet werkt. We vergelijken ook post-kwantum TLS en KEMTLS op computers met beperkte rekenkracht en geheugen.

Tot slot bespreken we de kwaliteit van de software in het NIST standaardiseringsproject; deze software hebben we veel gebruikt in onze experimenten, maar was niet meteen geschikt hiervoor. Er zaten veel fouten in de software, en door gebrek aan zaken als *namespacing* was het moeilijk om verschillende implementaties te integreren in een programma. We geven aanbevelingen hoe een volgend standaardiseringsproject georganiseerd zou kunnen worden, zodat dit soort problemen vermeden kunnen worden.

Zie de Engelstalige samenvatting voor een uitgebreider overzicht per hoofdstuk.

Aanvullende hoofdstukken

Na het hoofdonderwerp van dit proefschrift volgen in de appendices nog twee hoofdstukken die minder direct gerelateerd zijn aan het hoofdonderwerp. Het eerste hoofdstuk beschrijft hoe op computers (microcontrollers) met zeer beperkte hoeveelheden werkgeheugen en opslagruimte voor code toch post-kwantum digitale handtekeningen kunnen worden geverifieerd, zelfs als deze handtekeningen of de voor verificatie benodigde publieke sleutels veel groter zijn dan het werkgeheugen. Hiervoor wordt een aanpak voorgesteld op basis van streamen: door de publieke sleutel of handtekening in delen aan te bieden aan de microcontroller kan de berekening voor de verificatie op een incrementele manier worden gedaan.

In het tweede hoofdstuk bespreken we het LPN probleem. Dit is een theoretisch probleem dat aan de basis ligt van verschillende post-kwantum systemen. In dit hoofdstuk bekijken we hoe verschillende (bestaande) aanvallen op dit probleem efficiënt gecombineerd kunnen worden om grotere LPN-problemen op te lossen. We leggen de nadruk op *praktische* aanvallen: bestaande aanvallen gebruiken vaak subexponentiële hoeveelheden tijd *en* geheugen, terwijl het gebruiken van grote hoeveelheden geheugen typisch een grotere barrière vormt voor het daadwerkelijk kunnen uitvoeren van een aanval. We laten zien hoe combinaties van aanvallen gevonden kunnen worden die passen binnen een vooraf bepaalde hoeveelheid geheugen, en voeren ook enkele van deze aanvallen uit.

List of publications

The following is a list of academic publications that Thom co-authored.

- Ruben Gonzalez and Thom Wiggers. "KEMTLS vs. Post-quantum TLS: Performance on Embedded Systems." In: *Security, Privacy, and Applied Cryptography Engineering*. Ed. by Lejla Batina, Stjepan Picek, and Mainack Mondal. Jaipur, India: Springer Nature Switzerland, Dec. 9–12, 2022, pp. 99–117. DOI: 10.1007/978-3-031-22829-2. URL: wggrs.nl/p/kemtls-embedded

- Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. "A Tale of Two Models: Formal Verification of KEMTLS via Tamarin." In: *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part III*. ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. Lecture Notes in Computer Science. Extended version available via URL and IACR ePrint. Copenhagen, Denmark: Springer, Heidelberg, Germany, Sept. 26–30, 2022, pp. 63–83. DOI: 10.1007/978-3-031-17143-7_4. IACR ePrint: ia.cr/2022/1111. URL: wggrs.nl/p/kemtls-tamarin

- Felix Günther, Simon Rastikian, Patrick Towa, and Thom Wiggers. "KEMTLS with Delayed Forward Identity Protection in (Almost) a Single Round Trip." In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Rome, Italy: Springer, Heidelberg, Germany, June 20–23, 2022, pp. 253–272. DOI: 10.1007/978-3-031-09234-3_13. IACR ePrint: ia.cr/2021/725. URL: wggrs.nl/p/kemtls-epoch

- Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Improving Software Quality in Cryptography Standardization Projects." In: *SSR 2022: Security Standardization Research (2022*

IEEE S&P Workshops). Genoa, Italy: IEEE Computer Society, June 6–10, 2022, pp. 19–30. DOI: 10.1109/EuroSPW55150.2022.00010. URL: wggrs.nl/p/pqclean

- Sofía Celi, Armando Faz-Hernández, Nick Sullivan, Goutam Tamvada, Luke Valenta, Thom Wiggers, Bas Westerbaan, and Christopher A. Wood. "Implementing and Measuring KEMTLS." in: *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America*. Ed. by Patrick Longa and Carla Ràfols. Vol. 12912. Lecture Notes in Computer Science. Bogotá, Colombia: Springer, Heidelberg, Germany, Oct. 6–8, 2021, pp. 88–107. DOI: 10.1007/978-3-030-88238-9_5. IACR ePrint: ia.cr/2021/1019. URL: wggrs.nl/p/measuring-kemtls

- Peter Schwabe, Douglas Stebila, and Thom Wiggers. "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys." In: *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part I*. ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Updated version available via url and IACR ePrint. Darmstadt, Germany: Springer, Heidelberg, Germany, Oct. 4–8, 2021, pp. 3–22. DOI: 10.1007/978-3-030-88418-5_1. IACR ePrint: ia.cr/2021/779. URL: wggrs.nl/p/kemtlspdk

- Thom Wiggers and Simona Samardjiska. "Practically Solving LPN." in: *2021 IEEE International Symposium on Information Theory (ISIT)*. Melbourne, Australia: IEEE Information Theory Society, July 12–20, 2021, pp. 2399–2404. DOI: 10.1109/ISIT45174.2021.9518109. IACR ePrint: ia.cr/2021/962. URL: wggrs.nl/p/lpn

- Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. "Verifying Post-Quantum Signatures in 8 kB of RAM." in: *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Daejeon, South Korea: Springer, Heidelberg, Germany, July 20–22, 2021, pp. 215–233. DOI: 10.1007/978-3-030-81293-5_12. IACR ePrint:

ia.cr/2021/662. URL: wggrs.nl/p/verifying-post-quantum-signatures-in-8kb-of-ram

- Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures." In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. Updated version available via url and IACR ePrint. Virtual Event, USA: ACM Press, Nov. 9–13, 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350. IACR ePrint: ia.cr/2020/534. URL: wggrs.nl/p/kemtls

- Thom Wiggers. "Energy-Efficient ARM64 Cluster with Cryptanalytic Applications - 80 Cores That Do Not Cost You an ARM and a Leg." In: *Progress in Cryptology - LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America*. Ed. by Tanja Lange and Orr Dunkelman. Vol. 11368. Lecture Notes in Computer Science. Havana, Cuba: Springer, Heidelberg, Germany, Sept. 20–22, 2017, pp. 175–188. DOI: 10.1007/978-3-030-25283-0_10. IACR ePrint: ia.cr/2018/888. URL: wggrs.nl/p/armcluster

About the author

Thom Wiggers was born in Vleuten-De Meern, The Netherlands on 2 March 1993. After moving a couple of times, he graduated with a cum laude *gymnasium* diploma from Veluws College Walterbosch in Apeldoorn, The Netherlands. Thom moved to Radboud University (RU) in Nijmegen, where he completed two Bachelor's degrees in Computing Science (bene meritum) and Information Sciences (cum laude). His Bachelor's thesis, which was supervised by Peter Schwabe, was titled *"Implementing CAESAR candidate Prøst on ARM11"*. Thom also completed his Master's degree cum laude in Computing Science in Nijmegen, taking courses at the Eindhoven University of Technology (TU/e) as part of the joint Master's program in computer security *TRU/e*. The Master's thesis project, which was supervised by Simona Samardjiska, was titled *"Solving LPN using large covering codes"*. Before this thesis, Thom completed a research internship project, again supervised by Peter Schwabe. This lead to Thom's first publication *"Energy-Efficient ARM64 Cluster with Cryptanalytic Applications - 80 Cores That Do Not Cost You an ARM and a Leg"* and a conference trip to Cuba; there, Peter asked if Thom would be up for taking a Ph.D. position described in the project proposal—which was partially written on an infamous Cuban Airbnb rooftop.

The project, *engineering post-quantum cryptography*, got funded by the European Union research council, and Thom started his Ph.D. in October 2018. From 17 February to 16 March 2020, Thom visited Douglas Stebila at the University of Waterloo. From September 2021 to February 2022, Thom was an intern in Cloudflare's research team. This thesis is the culmination of the work from October 2018 to April 2023. Since June 2023, Thom has continued his cryptography research with PQShield.

Outside of work, Thom enjoys bouldering and playing video games.