BSI invited talk 2024-01-16.

Post-Quantum TLS

Thom Wiggers

Thom Wiggers

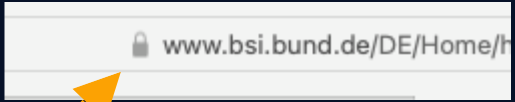
- Cryptography researcher at PQShield
  - Oxford University spin-off
  - We develop and license PQC hardware and software IP
  - Side-channel protected hardware designs
  - FIPS 140-3 validated software
  - We also do fundamental research
- Research interest: applying PQC to real-world systems
  - Post-Quantum TLS
  - Secure messaging
- Ph.D from Radboud University (2024)
  - Dissertation: Post-Quantum TLS

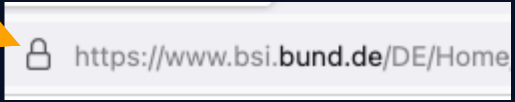
Outline

1. Transport Layer Security
   a. Old TLS
   b. Version 1.3
2. Key Exchange in TLS
   a. Current design
   b. draft-ietf-tls-hybrid-design
   c. Fitting KEMs
3. Public Key Infrastructure
   a. Certificates
   b. OCSP
   c. Too many signatures
   d. Impact of PQC
4. Attempting to fix the WebPKI
   a. Compressing certificates
   b. Merkle Tree Certificates
5. Authentication without signatures
   a. AuthKEM
   b. AuthKEM-PSK

Transport Layer Security

🔒

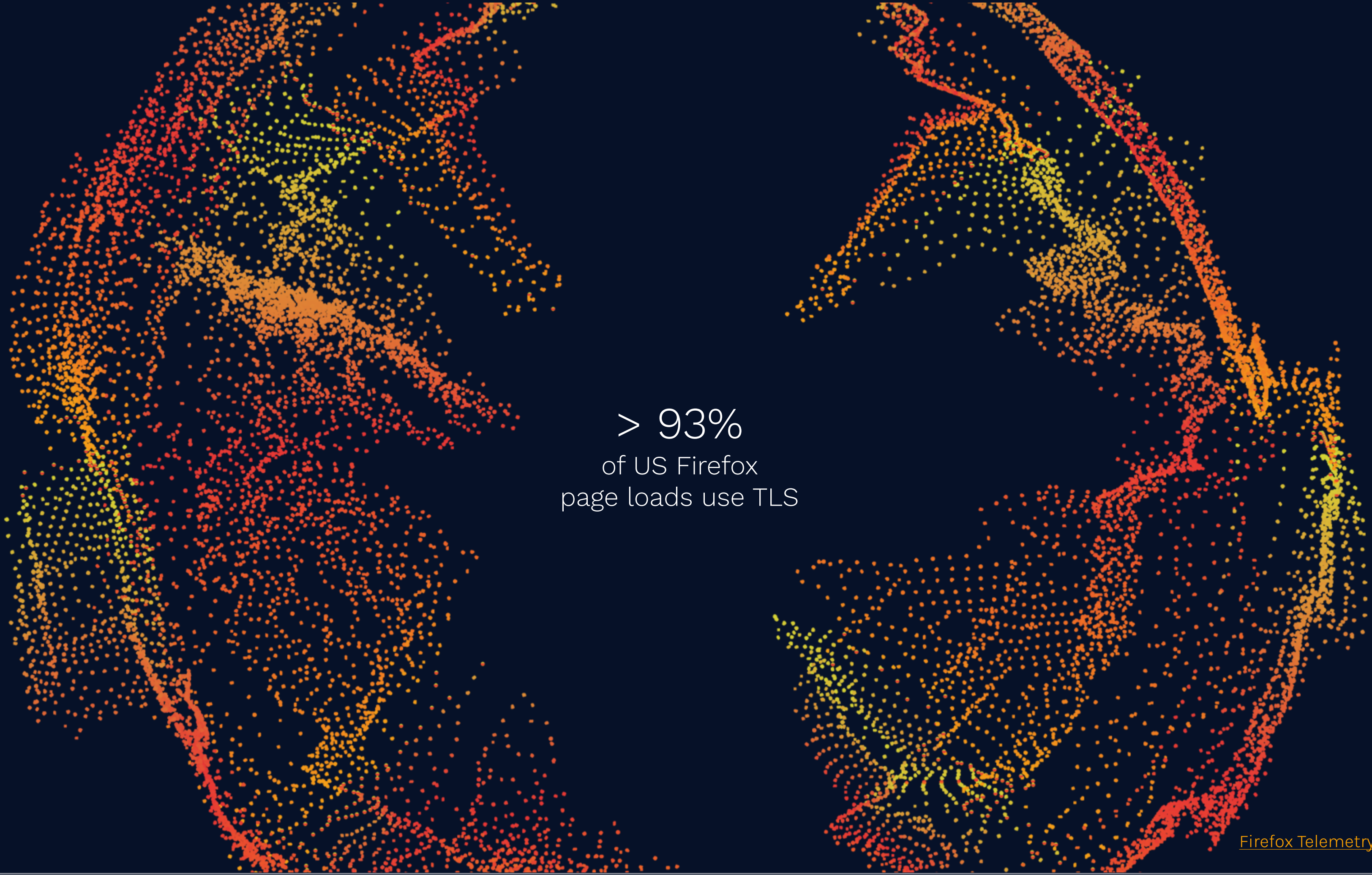

www.bsi.bund.de/DE/Home/h


https://www.bsi.bund.de/DE/Home


bsi.bund.de/DE/Home/home_node.html

bsi.bund.de

Verbinding is beveiligd



> 93%

of US Firefox
page loads use TLS

[Firefox Telemetry, 2023-12-31]

Transport Layer Security

Transport Layer Security

- Colloquially still also known as “SSL”

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`**, but TLS is much more
    - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
    - WPA Enterprise has TLS auth modes,
    - Encrypted email transport (SMTPS),
    - VoIP, RTSP (streaming video), XMPP, ...

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`**, but TLS is much more
  - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
  - WPA Enterprise has TLS auth modes,
  - Encrypted email transport (SMTPS),
  - VoIP, RTSP (streaming video), XMPP, …
- SSL 3.0 (RFC 6101(historic)) -> TLS 1.0 (RFC 2246) (1999)

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`** , but TLS is much more
    - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
    - WPA Enterprise has TLS auth modes,
    - Encrypted email transport (SMTPS),
    - VoIP, RTSP (streaming video), XMPP, …
- SSL 3.0 (RFC 6101(historic)) -> TLS 1.0 (RFC 2246) (1999)
- TLS 1.1 (RFC 4346) (2006)

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`**, but TLS is much more
    - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
    - WPA Enterprise has TLS auth modes,
    - Encrypted email transport (SMTPS),
    - VoIP, RTSP (streaming video), XMPP, ...
- SSL 3.0 (RFC 6101(historic)) -> TLS 1.0 (RFC 2246) (1999)
- TLS 1.1 (RFC 4346) (2006)
- TLS 1.2 (RFC 5246) (2008)

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`**, but TLS is much more
  - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
  - WPA Enterprise has TLS auth modes,
  - Encrypted email transport (SMTPS),
  - VoIP, RTSP (streaming video), XMPP, ...
- SSL 3.0 (RFC 6101(historic)) -> TLS 1.0 (RFC 2246) (1999)
- TLS 1.1 (RFC 4346) (2006)
- TLS 1.2 (RFC 5246) (2008)
- TLS 1.3 (RFC 8446) (2018)

Transport Layer Security

- Colloquially still also known as “SSL”
- Often equated with **`https://`**, but TLS is much more
  - OpenVPN, Cisco AnyConnect, Citrix NetScaler, and more VPNs are based on (D)TLS
  - WPA Enterprise has TLS auth modes,
  - Encrypted email transport (SMTPS),
  - VoIP, RTSP (streaming video), XMPP, ...
- SSL 3.0 (RFC 6101(historic)) -> TLS 1.0 (RFC 2246) (1999)
- TLS 1.1 (RFC 4346) (2006)
- TLS 1.2 (RFC 5246) (2008)
- TLS 1.3 (RFC 8446) (2018)
- See also: DTLS (Datagram TLS), QUIC (RFC 9000)

Strengths of TLS

- Client-to-Server model
- Client does not need the server's keys prior to starting connection
  - Trust is usually from pre-installed PKI plus the server's hostname
- Optional client authentication through certificates
  - Extension: raw public keys are also supported (RFC 7250)
- Security well-studied

Drawbacks of TLS

Drawbacks of TLS

- Cost of connection setup

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures
    - Elliptic-curve cryptography mostly makes this something we ignore

Drawbacks of TLS

- Cost of connection setup
  - Each connection setup transmits a ton of certificates and signatures
  - Elliptic-curve cryptography mostly makes this something we ignore
  - Can be avoided by using session resumption, but isn’t setup-free [MWV23]

Drawbacks of TLS

- Cost of connection setup
  - Each connection setup transmits a ton of certificates and signatures
  - Elliptic-curve cryptography mostly makes this something we ignore
  - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures
    - Elliptic-curve cryptography mostly makes this something we ignore
    - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
    - In particular the discussion on PKI is very focused on websites

Drawbacks of TLS

- Cost of connection setup
  - Each connection setup transmits a ton of certificates and signatures
  - Elliptic-curve cryptography mostly makes this something we ignore
  - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
  - In particular the discussion on PKI is very focused on websites
- Trust model for client authentication is very different from server authentication

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures
    - Elliptic-curve cryptography mostly makes this something we ignore
    - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
    - In particular the discussion on PKI is very focused on websites
- Trust model for client authentication is very different from server authentication
    - Client application trusts server because hostname matches what application specified

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures
    - Elliptic-curve cryptography mostly makes this something we ignore
    - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
    - In particular the discussion on PKI is very focused on websites
- Trust model for client authentication is very different from server authentication
    - Client application trusts server because hostname matches what application specified
    - Server application has to explicitly configure what to do with the incoming client certificate!

Drawbacks of TLS

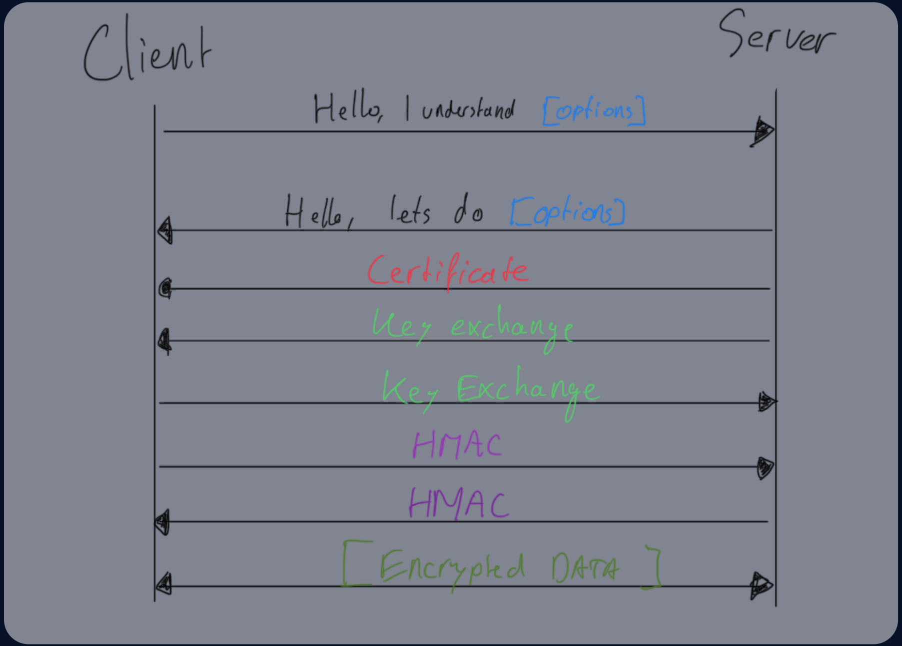
- Cost of connection setup
  - Each connection setup transmits a ton of certificates and signatures
  - Elliptic-curve cryptography mostly makes this something we ignore
  - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
  - In particular the discussion on PKI is very focused on websites
- Trust model for client authentication is very different from server authentication
  - Client application trusts server because hostname matches what application specified
  - Server application has to explicitly configure what to do with the incoming client certificate!
  - Anecdotally, Developers often misunderstand this when setting up mutually-authenticated TLS (mTLS)

Drawbacks of TLS

- Cost of connection setup
    - Each connection setup transmits a ton of certificates and signatures
    - Elliptic-curve cryptography mostly makes this something we ignore
    - Can be avoided by using session resumption, but isn't setup-free [MWV23]
- Development is very focused on web applications (HTTPS)
    - In particular the discussion on PKI is very focused on websites
- Trust model for client authentication is very different from server authentication
    - Client application trusts server because hostname matches what application specified
    - Server application has to explicitly configure what to do with the incoming client certificate!
    - Anecdotally, Developers often misunderstand this when setting up mutually-authenticated TLS (mTLS)

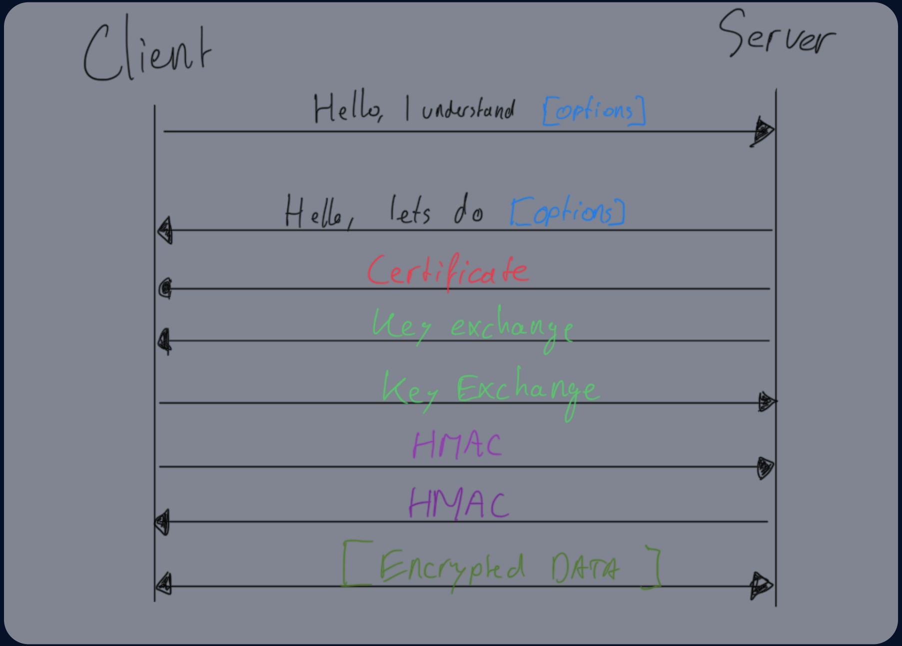
[MWV23]: TLS → Post-Quantum TLS: Inspecting the TLS landscape for PQC adoption on Android: *E.g. Android apps fail to set this up, and sometimes end up doing hundreds of requests to the same hostnames in five minutes*

TLS 1.2 and before



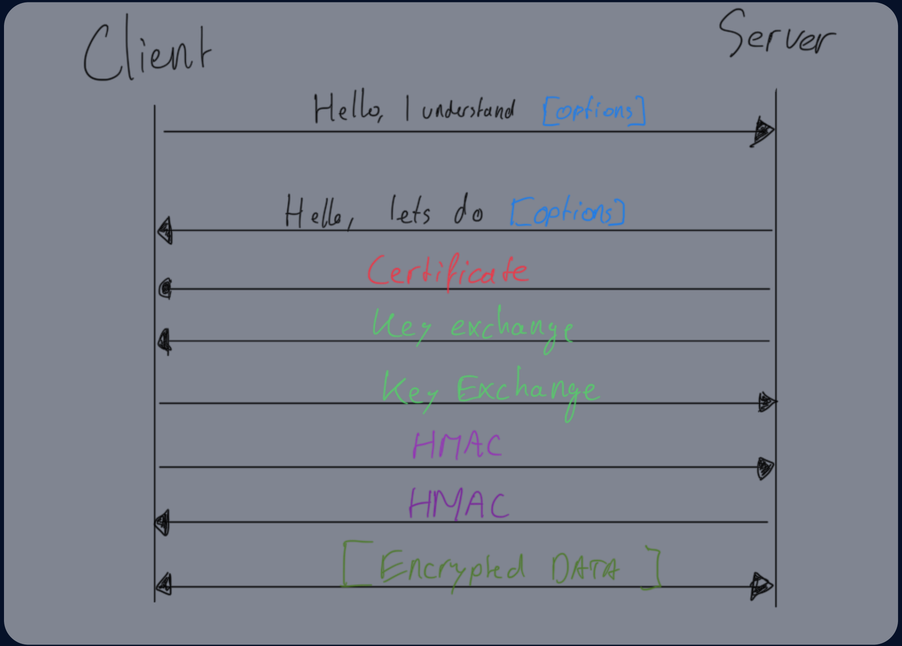
TLS 1.2 and before

- Many round-trips



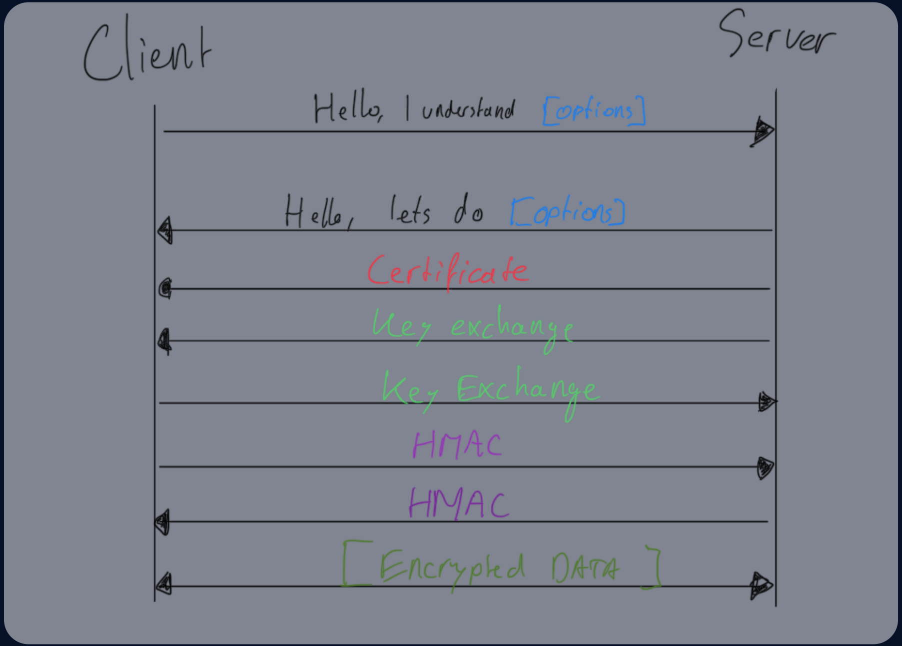
TLS 1.2 and before

- Many round-trips
- Certificates are sent in the clear



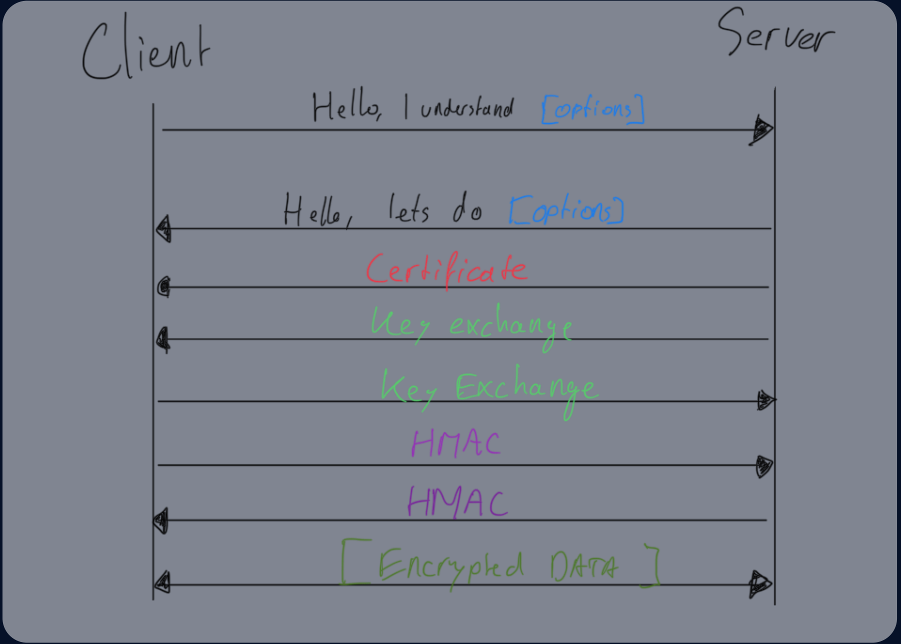
TLS 1.2 and before

- Many round-trips
- Certificates are sent in the clear
    - Everybody can see you’re connecting to bsi.bund.de



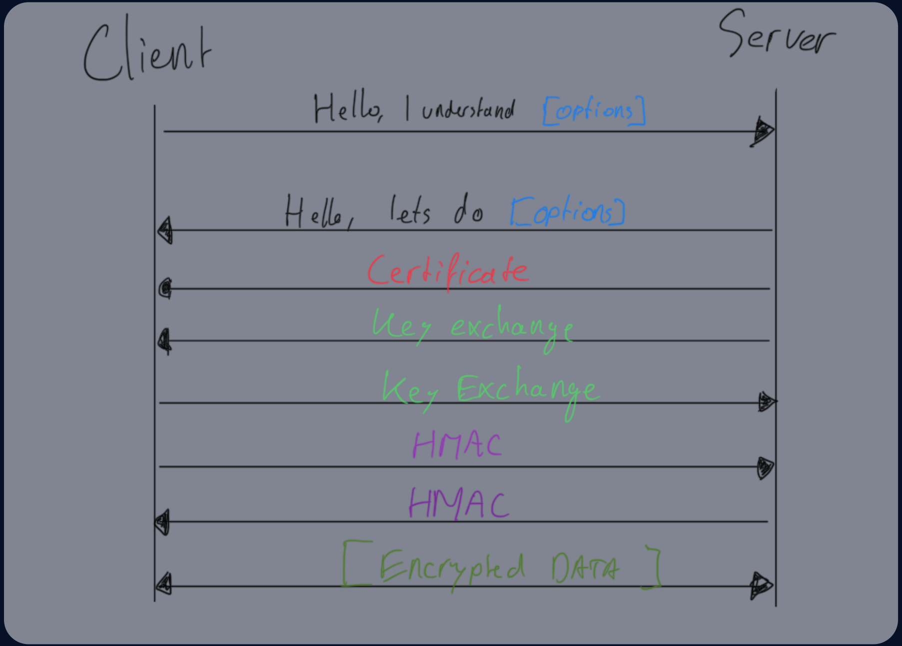
TLS 1.2 and before

- Many round-trips
- Certificates are sent in the clear
    - Everybody can see you're connecting to bsi.bund.de
    - Especially problematic for client authentication



TLS 1.2 and before

- Many round-trips
- Certificates are sent in the clear
  - Everybody can see you’re connecting to bsi.bund.de
  - Especially problematic for client authentication
- A lot of legacy cryptography and patches against attacks



Attacks on TLS (incomplete list)

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
    - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery
- 2015/2016: FREAK / Logjam

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery
- 2015/2016: FREAK / Logjam
  - implementation flaws downgrade to EXPORT cryptography

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)...)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️...)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery
- 2015/2016: FREAK / Logjam
  - implementation flaws downgrade to EXPORT cryptography
- 2016: DROWN: use the server's SSLv2 support to break SSLv3/TLS 1.{0,1,2}

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery
- 2015/2016: FREAK / Logjam
  - implementation flaws downgrade to EXPORT cryptography
- 2016: DROWN: use the server’s SSLv2 support to break SSLv3/TLS 1.{0,1,2}
- 2018: ROBOT: Bleichenbacher’s 1998 attack is still valid on many TLS 1.2 implementations

Attacks on TLS (incomplete list)

- 1998, 2006: Bleichenbacher breaks RSA encryption and RSA signatures using errors as side-channel
- 2011: BEAST: breaks SSL 3.0 and TLS 1.0 (nobody was using TLS 1.1 (2006) or 1.2 (2008)…)
  - avoid attack by using RC4 (but since 2013 RC4 is considered ☠️…)
- 2012/2013: CRIME / BREACH: compression in TLS is bad
- 2013: Lucky Thirteen: timing attack on encrypt-then-MAC
- 2014: POODLE: destroys SSL 3.0
- 2014: Bleichenbacher again (BERserk): signature forgery
- 2015/2016: FREAK / Logjam
  - implementation flaws downgrade to EXPORT cryptography
- 2016: DROWN: use the server's SSLv2 support to break SSLv3/TLS 1.{0,1,2}
- 2018: ROBOT: Bleichenbacher's 1998 attack is still valid on many TLS 1.2 implementations
- 2023: Everlasting ROBOT: Bleichenbacher's 1998 attack is still, still valid on many TLS 1.2 implementations

Common themes

- Attacks on old versions of TLS remain valid for decades

Common themes

- Attacks on old versions of TLS remain valid for decades
    - XP, Vista, Android <5 never supported TLS 1.1, 1.2

Common themes

- Attacks on old versions of TLS remain valid for decades
  - XP, Vista, Android <5 never supported TLS 1.1, 1.2
- Many attacks are possible because legacy algorithms are never turned off by servers

Common themes

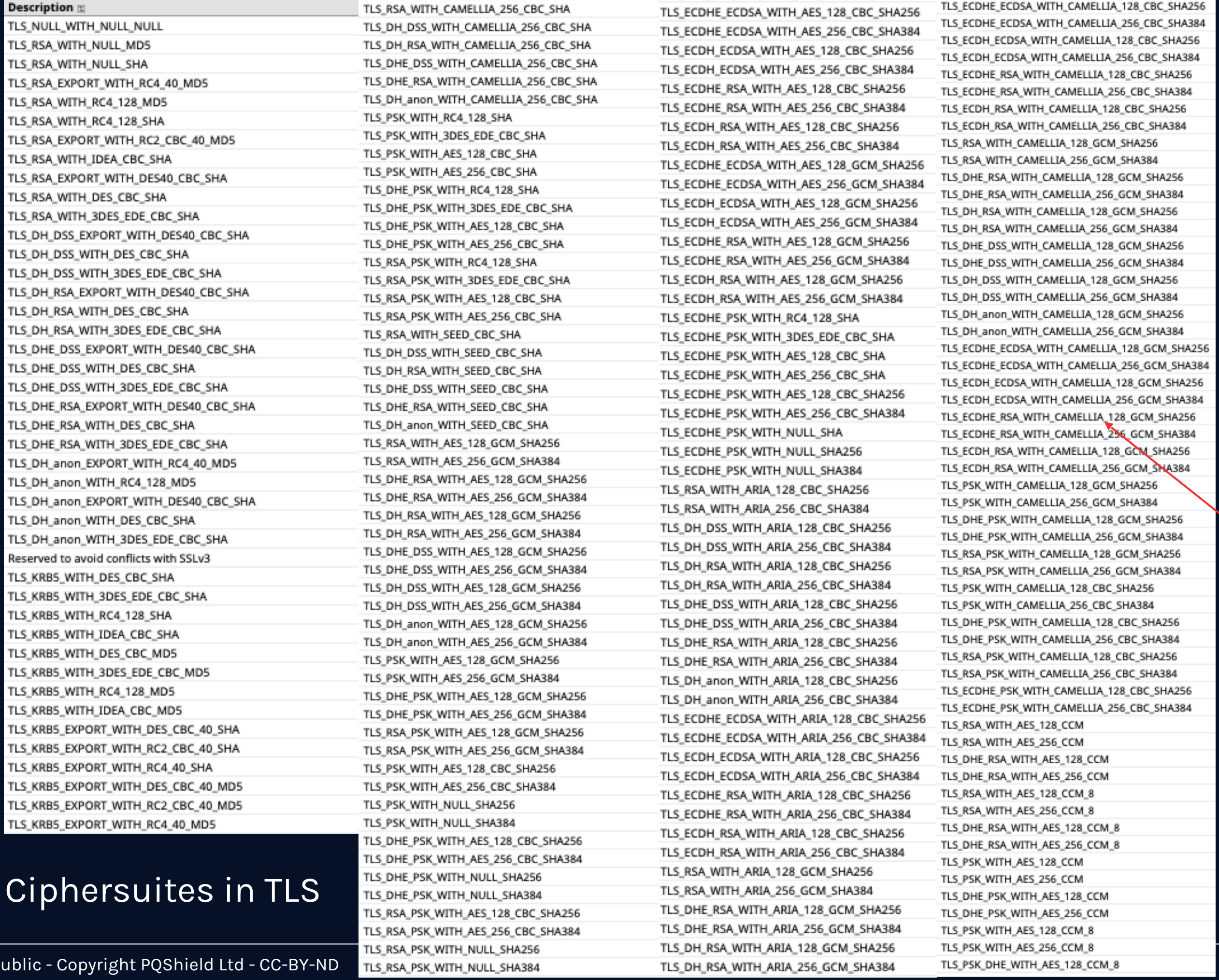
- Attacks on old versions of TLS remain valid for decades
  - XP, Vista, Android <5 never supported TLS 1.1, 1.2
- Many attacks are possible because legacy algorithms are never turned off by servers
  - FREAK/Logjam: 512-bit RSA/Diffie-Hellman (‘Export’ crypto)

Common themes

- Attacks on old versions of TLS remain valid for decades
  - XP, Vista, Android <5 never supported TLS 1.1, 1.2
- Many attacks are possible because legacy algorithms are never turned off by servers
  - FREAK/Logjam: 512-bit RSA/Diffie-Hellman (‘Export’ crypto)
- Setting up TLS servers is a massive headache

Common themes

- Attacks on old versions of TLS remain valid for decades
    - XP, Vista, Android <5 never supported TLS 1.1, 1.2
- Many attacks are possible because legacy algorithms are never turned off by servers
    - FREAK/Logjam: 512-bit RSA/Diffie-Hellman (‘Export’ crypto)
- Setting up TLS servers is a massive headache
    - So many ciphersuites, key exchange groups, …


Description
TLS_NULL_WITH_NULL_NULL
TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5
TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_RSA_WITH_IDEA_CBC_SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_DSS_WITH_DES_CBC_SHA
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_RSA_WITH_DES_CBC_SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
TLS_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_anon_WITH_DES_CBC_SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
Reserved to avoid conflicts with SSLv3
TLS_KRB5_WITH_DES_CBC_SHA
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_RC4_128_SHA
TLS_KRB5_WITH_IDEA_CBC_SHA
TLS_KRB5_WITH_DES_CBC_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_KRB5_WITH_RC4_128_MD5
TLS_KRB5_WITH_IDEA_CBC_MD5
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
TLS_PSK_WITH_RC4_128_SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_RC4_128_SHA
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_DHE_PSK_WITH_AES_128_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_PSK_WITH_RC4_128_SHA
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_CBC_SHA
TLS_RSA_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_DH_DSS_WITH_SEED_CBC_SHA
TLS_DH_RSA_WITH_SEED_CBC_SHA
TLS_DHE_DSS_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_DH_anon_WITH_SEED_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_AES_128_GCM_SHA256
TLS_DH_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_WITH_AES_128_GCM_SHA256
TLS_DH_DSS_WITH_AES_256_GCM_SHA384
TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_GCM_SHA256
TLS_PSK_WITH_AES_256_GCM_SHA384
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
TLS_PSK_WITH_AES_128_CBC_SHA256
TLS_PSK_WITH_AES_256_CBC_SHA384
TLS_PSK_WITH_NULL_SHA256
TLS_PSK_WITH_NULL_SHA384
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
TLS_DHE_PSK_WITH_NULL_SHA256
TLS_DHE_PSK_WITH_NULL_SHA384
TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
TLS_RSA_PSK_WITH_NULL_SHA256
TLS_RSA_PSK_WITH_NULL_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_PSK_WITH_RC4_128_SHA
TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_NULL_SHA
TLS_ECDHE_PSK_WITH_NULL_SHA256
TLS_ECDHE_PSK_WITH_NULL_SHA384
TLS_RSA_WITH_ARIA_128_CBC_SHA256
TLS_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256
TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384
TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_DH_anon_WITH_ARIA_128_CBC_SHA256
TLS_DH_anon_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384
TLS_RSA_WITH_ARIA_128_GCM_SHA256
TLS_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256
TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384
TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256
TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384
TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256
TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256
TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_DHE_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_DHE_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384
TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
TLS_RSA_WITH_AES_128_CCM
TLS_RSA_WITH_AES_256_CCM
TLS_DHE_RSA_WITH_AES_128_CCM
TLS_DHE_RSA_WITH_AES_256_CCM
TLS_RSA_WITH_AES_128_CCM_8
TLS_RSA_WITH_AES_256_CCM_8
TLS_DHE_RSA_WITH_AES_128_CCM_8
TLS_DHE_RSA_WITH_AES_256_CCM_8
TLS_PSK_WITH_AES_128_CCM
TLS_PSK_WITH_AES_256_CCM
TLS_DHE_PSK_WITH_AES_128_CCM
TLS_DHE_PSK_WITH_AES_256_CCM
TLS_PSK_WITH_AES_128_CCM_8
TLS_PSK_WITH_AES_256_CCM_8
TLS_PSK_DHE_WITH_AES_128_CCM_8

Ciphersuites in TLS

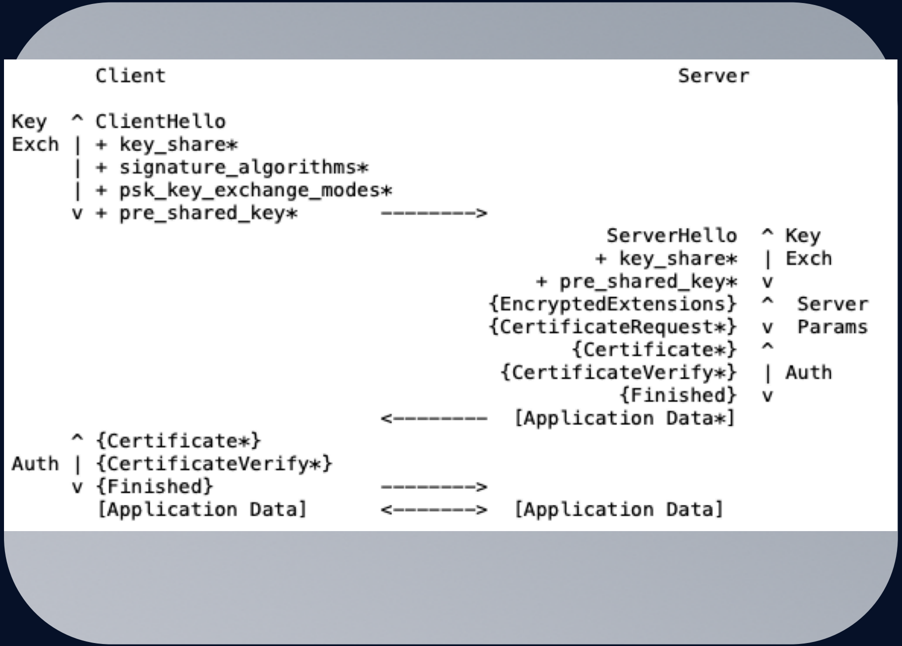
This isn't even all of them!

Japanese cipher: National mandates have external costs!

TLS 1.3 wish list

- Secure handshake
  - More privacy
  - Only forward secret key exchanges
  - Get rid of MD5, SHA1, 3DES, CAMELLIA, EXPORT, NULL, …
- Simplify parameters
- More robust cryptography
- Faster, 1-RTT protocol
- 0-RTT resumption

TLS 1.3

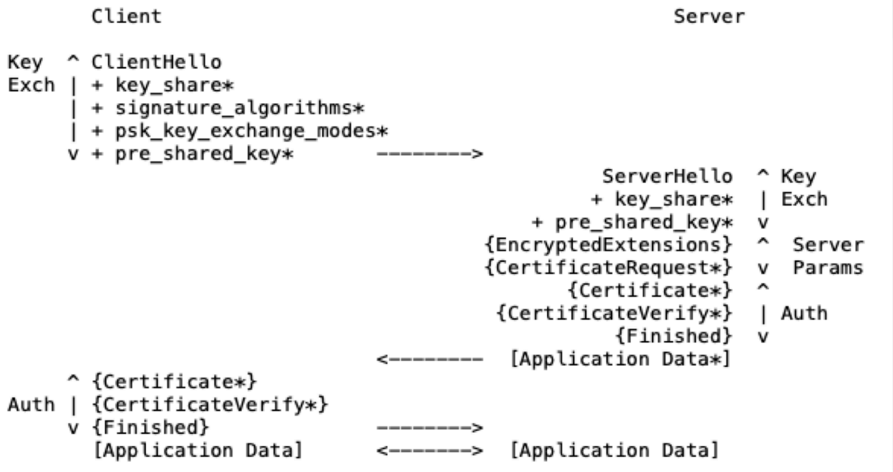


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages

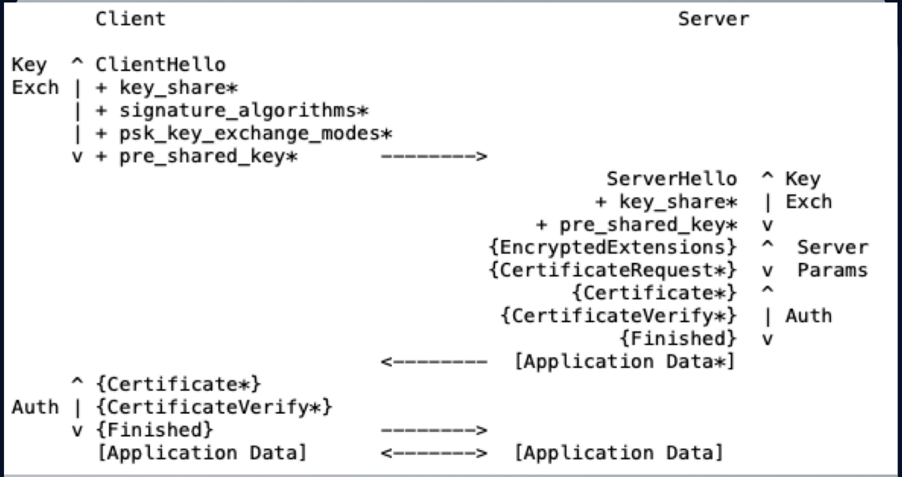


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange

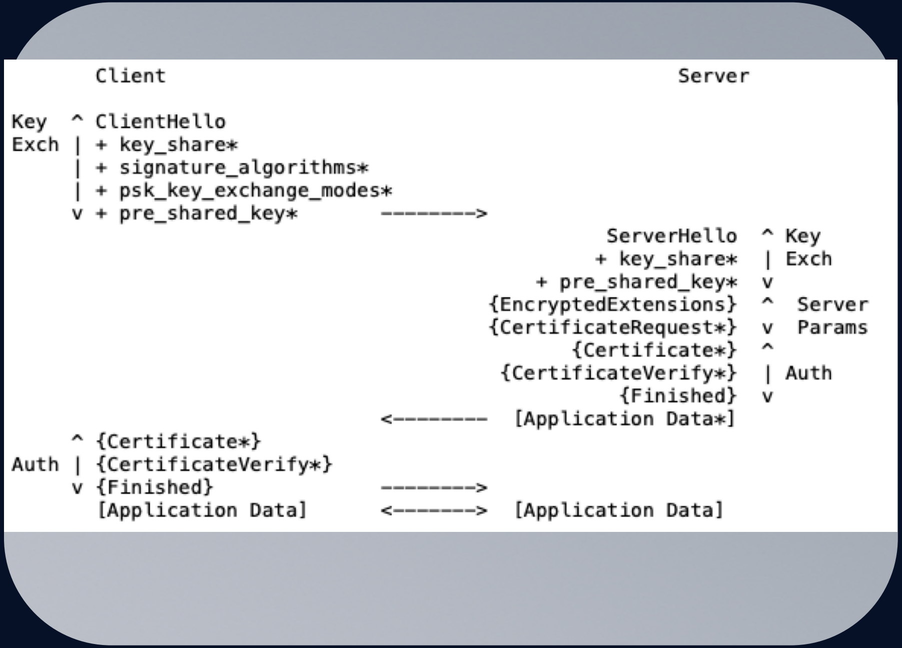


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible

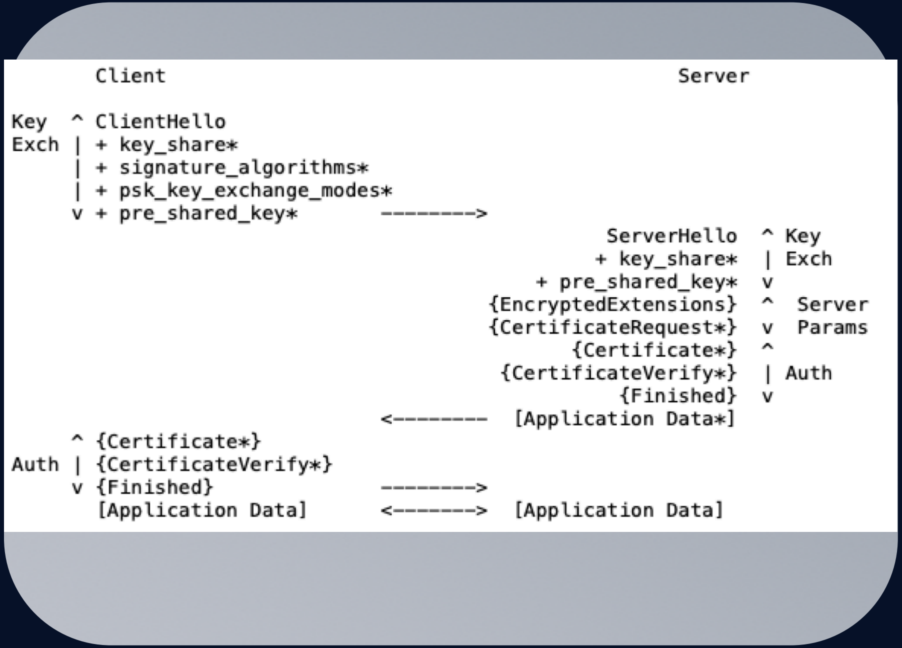


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible

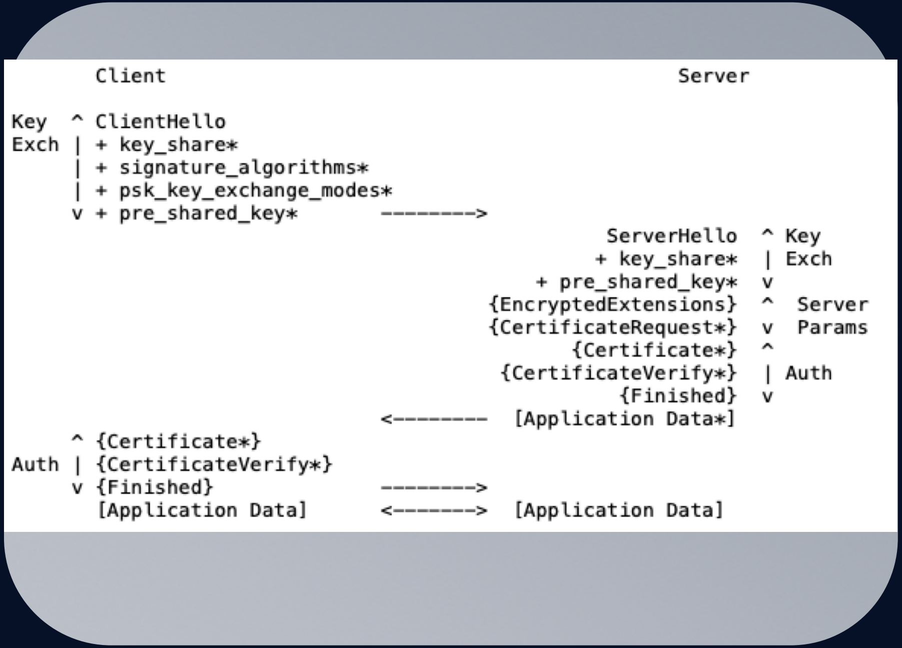


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
  - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible
  - Send certificate, signature and MAC in first response from server

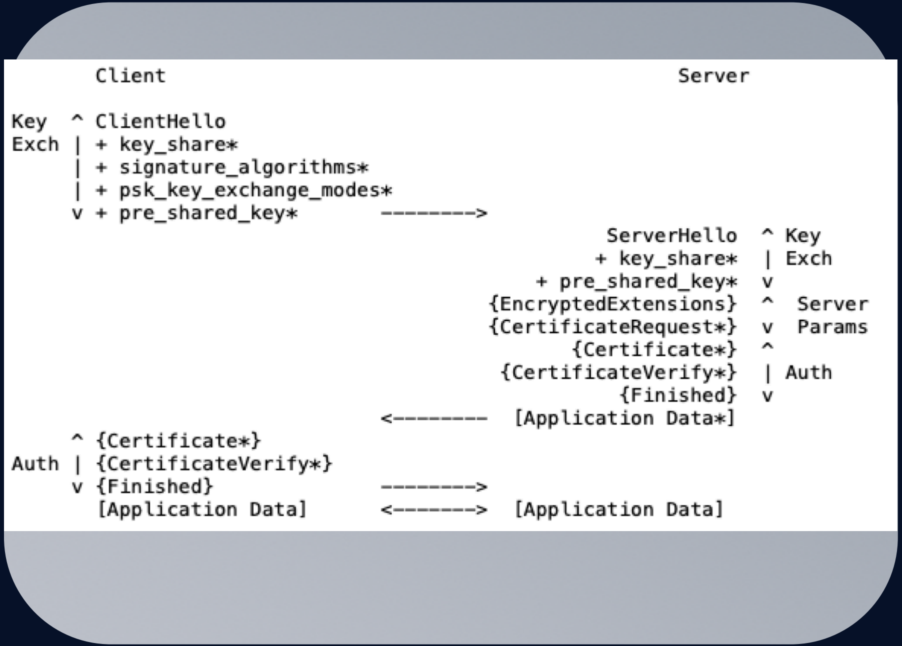


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible
    - Send certificate, signature and MAC in first response from server
- Simplify

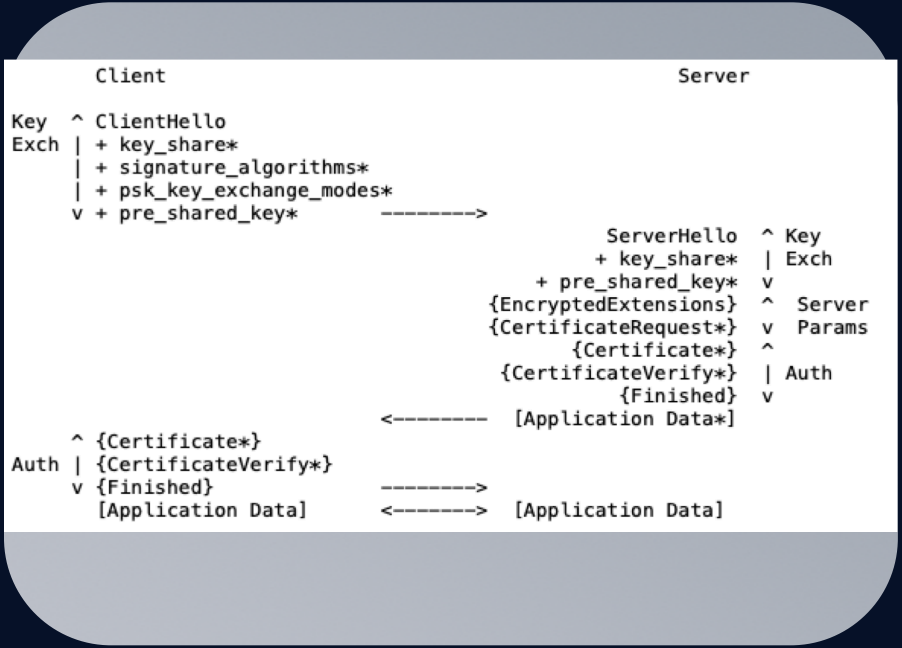


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible
    - Send certificate, signature and MAC in first response from server
- Simplify
    - ECDH: small list of pre-defined groups

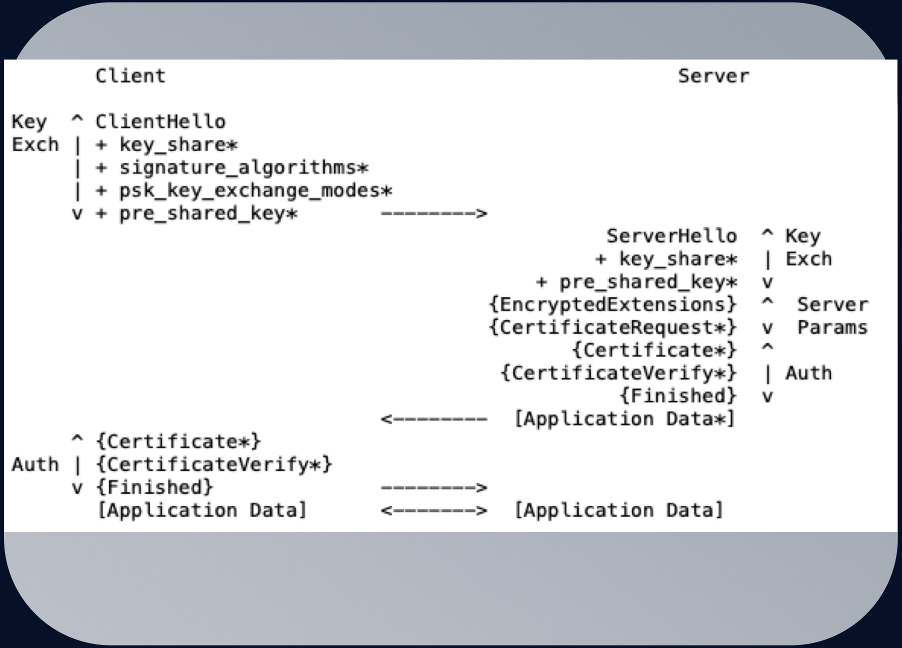


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible
    - Send certificate, signature and MAC in first response from server
- Simplify
    - ECDH: small list of pre-defined groups
    - Almost nobody implements finite-field DH

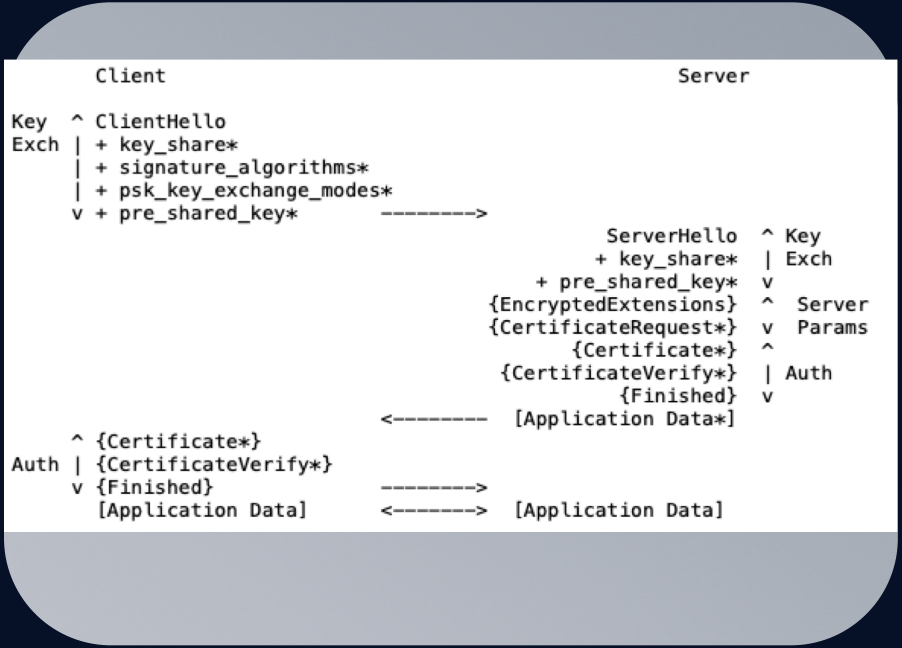


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3

- Move key exchange into the first two messages
    - ECDH ephemeral key exchange
- Encrypt as much as possible
- Be done as soon as possible
    - Send certificate, signature and MAC in first response from server
- Simplify
    - ECDH: small list of pre-defined groups
    - Almost nobody implements finite-field DH
    - Symmetric: AES-GCM, ChaCha20-Poly1305, and HMAC-SHA2

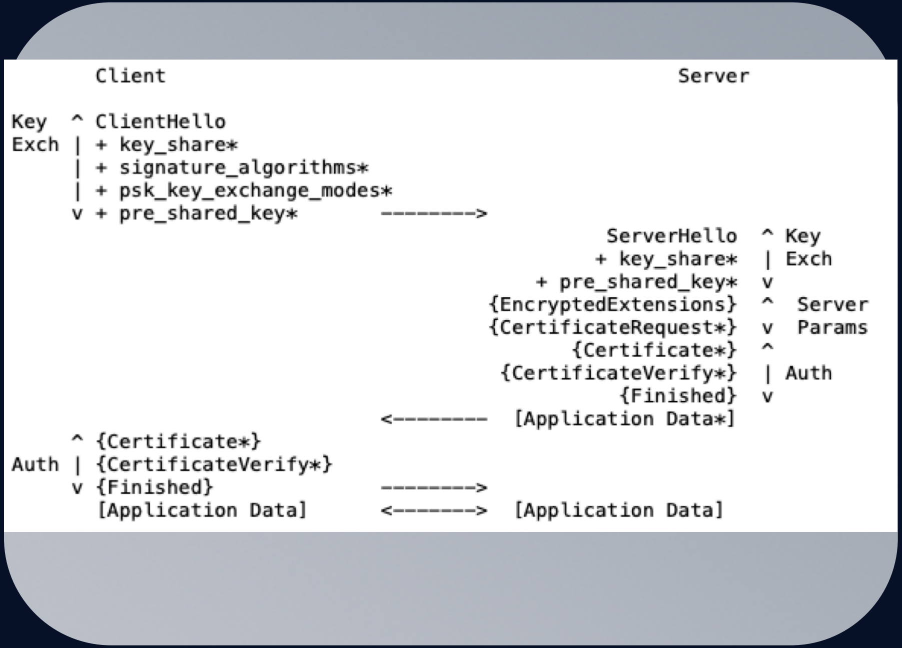


```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

TLS 1.3 Resumption and 0-RTT

- If you have a pre-shared key, you can do a bunch of stuff faster!
- Use PSK to compute traffic secret
- Ephemeral key exchange *optional*
- Use PSK to encrypt “Early Data”



```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]
```

0-RTT caveat

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is **not forward secret**, as it is encrypted solely under keys derived using the offered PSK.

2. There are **no guarantees of non-replay** between connections. Protection against replay for ordinary TLS 1.3 1-RTT data is provided via the server's Random value, but 0-RTT data does not depend on the ServerHello and therefore has weaker guarantees. This is especially relevant if the data is authenticated either with TLS client authentication or inside the application protocol. The same warnings apply to any use of the early_exporter_master_secret.

0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection), and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys). Appendix E.5 contains a description of potential attacks, and Section 8 describes mechanisms which the server can use to limit the impact of replay.

RFC 8446, page 18

Why 0-RTT?

- Siri requests
- GET requests on websites*
- Other stateless stuff

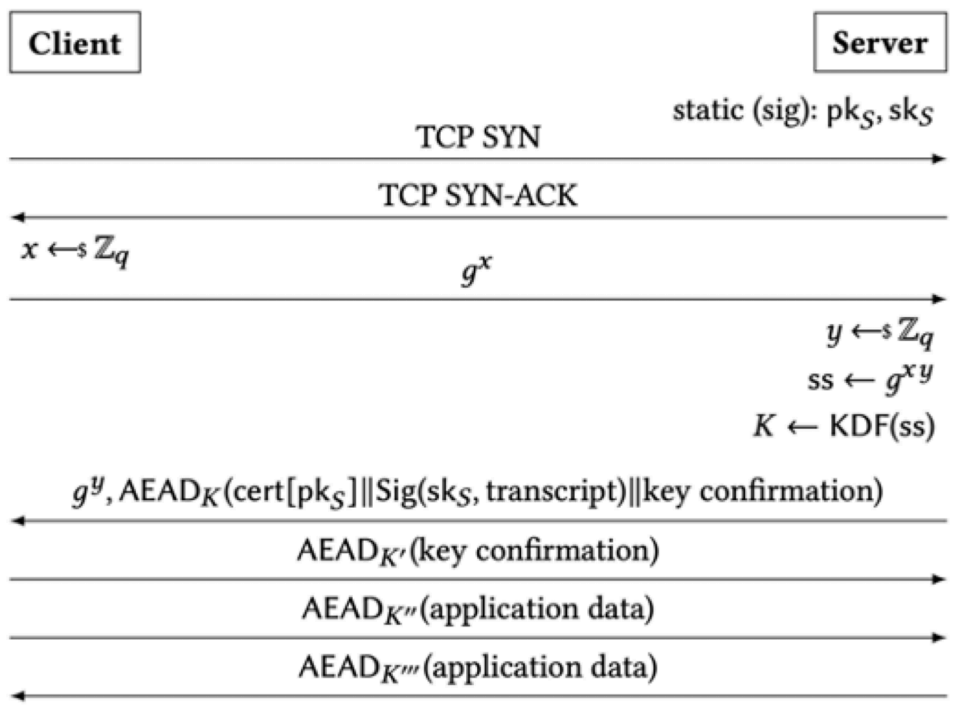
But are you sure that your application is completely robust against replays?

```
GET /?query=INSERT into payments (to, amount)
                           VALUES (“thom”, 1000);
```

Post-Quantum


Peter Shor
Server operator
g^x
RSA
ECC

Pre-quantum TLS



Client

Server

static (sig): $\mathsf{pk}_S, \mathsf{sk}_S$

TCP SYN

TCP SYN-ACK

$x \leftarrow_{\$} \mathbb{Z}_q$

g^x

$y \leftarrow_{\$} \mathbb{Z}_q$

$\mathsf{ss} \leftarrow g^{xy}$

$K \leftarrow \mathsf{KDF}(\mathsf{ss})$

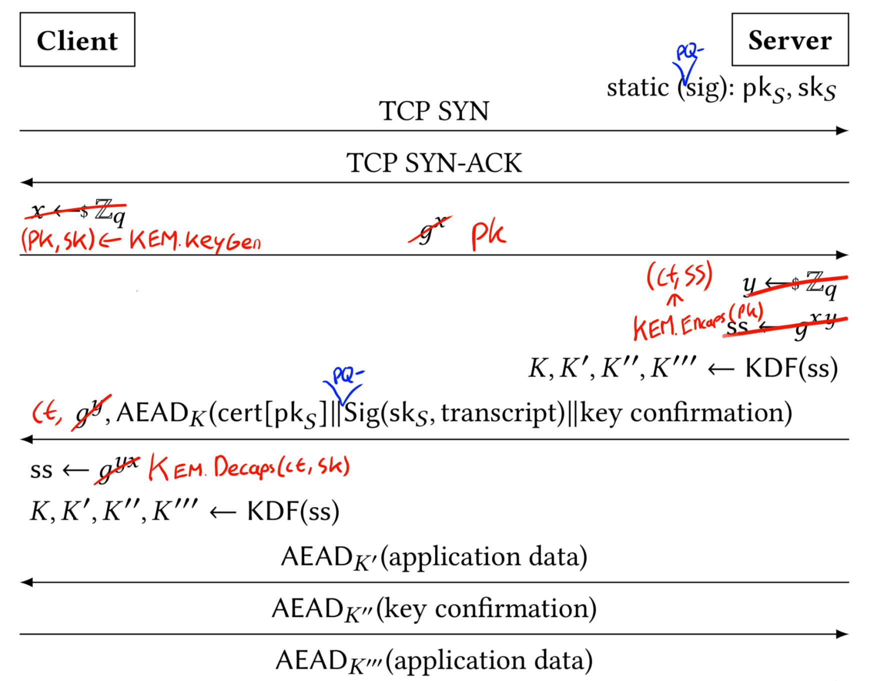
$g^y, \mathsf{AEAD}_K(\mathsf{cert}[\mathsf{pk}_S] \| \mathsf{Sig}(\mathsf{sk}_S, \mathsf{transcript}) \| \mathsf{key\ confirmation})$

$\mathsf{AEAD}_{K'}(\mathsf{key\ confirmation})$

$\mathsf{AEAD}_{K''}(\mathsf{application\ data})$

$\mathsf{AEAD}_{K'''}(\mathsf{application\ data})$

Post-quantum TLS



Client

Server

static (PQ- sig): $\mathrm{pk}_S, \mathrm{sk}_S$

TCP SYN

TCP SYN-ACK

~~$x \leftarrow_\$ \mathbb{Z}_q$~~

(pk, sk) ← KEM.keyGen

~~g^x~~ pk

(ct, ss) ← KEM.Encaps(pk)

~~$y \leftarrow_\$ \mathbb{Z}_q$~~

~~$\mathrm{ss} \leftarrow g^{xy}$~~

$K, K', K'', K''' \leftarrow \mathrm{KDF}(\mathrm{ss})$

(ct, ~~g^y~~, $\mathrm{AEAD}_K(\mathrm{cert}[\mathrm{pk}_S] \| \text{PQ-}\mathrm{Sig}(\mathrm{sk}_S, \mathrm{transcript}) \| \text{key confirmation})$

$\mathrm{ss} \leftarrow$ ~~g^{yx}~~ KEM.Decaps(ct, sk)

$K, K', K'', K''' \leftarrow \mathrm{KDF}(\mathrm{ss})$

$\mathrm{AEAD}_{K'}(\text{application data})$

$\mathrm{AEAD}_{K''}(\text{key confirmation})$

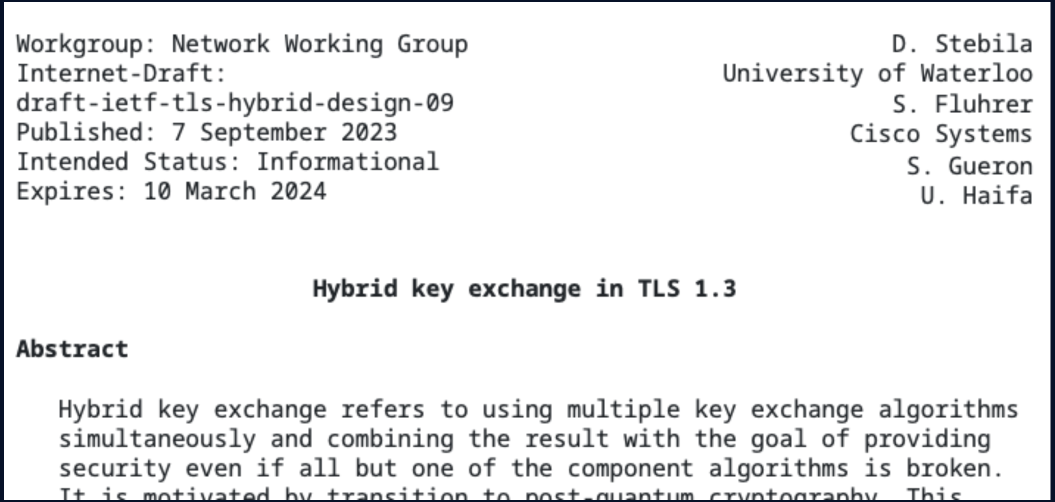
$\mathrm{AEAD}_{K'''}(\text{application data})$

Crossing out g^x

- draft-ietf-tls-hybrid-design
  Hybrid: ECDH + KEM key exchange
- draft-tls-westerbaan-xyber768d00
  Instantiates the above with
  X25519 + Kyber768
- draft-kwiatkowski-tls-ecdhe-kyber
  P256 + Kyber768

Main question not how, but how will clients react?

Cloudflare is reporting on its ongoing experiments



Workgroup: Network Working Group
Internet-Draft:
draft-ietf-tls-hybrid-design-09
Published: 7 September 2023
Intended Status: Informational
Expires: 10 March 2024

D. Stebila
University of Waterloo
S. Fluhrer
Cisco Systems
S. Gueron
U. Haifa

Hybrid key exchange in TLS 1.3

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken. It is motivated by transition to post-quantum cryptography. This

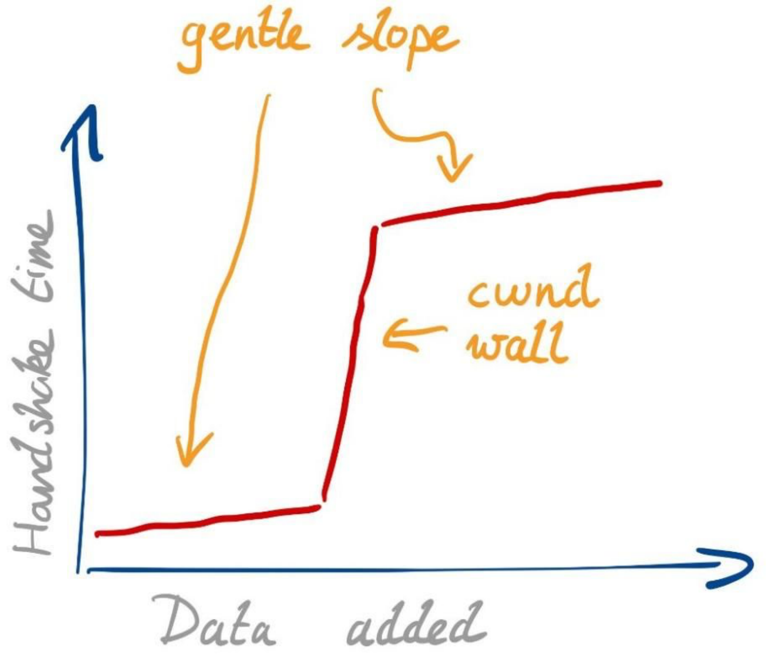
What about BSI's conservative KEMs?

- TLS restricts size of ephemeral **`key_share`** to 65535 bytes
- McEliece public key: doesn't fit
- FrodoKEM: does fit, but is still quite chunky (~15kb for FrodoKEM-976 pk)

But TLS runs over the internet!

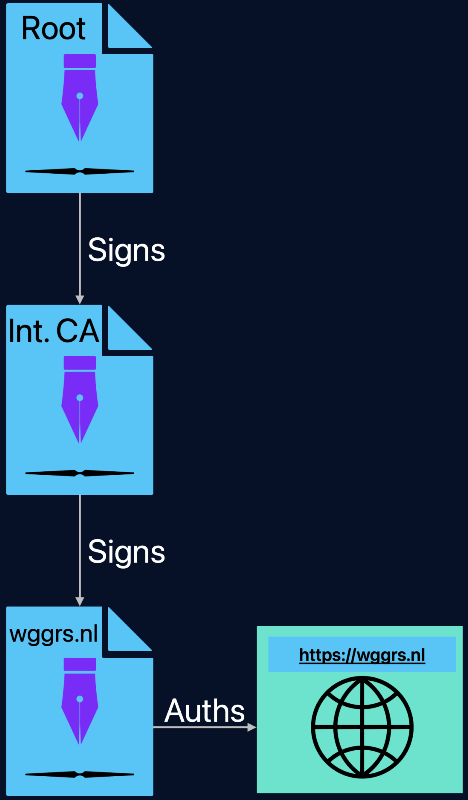
TCP congestion control

- TCP gives us a reliable transport
- Initial congestion window of 10 MSS ≈ 15 KB
- After sending this amount of data, TCP will just wait until it receives confirmation: additional round-trips
- FrodoKEM hits this wall



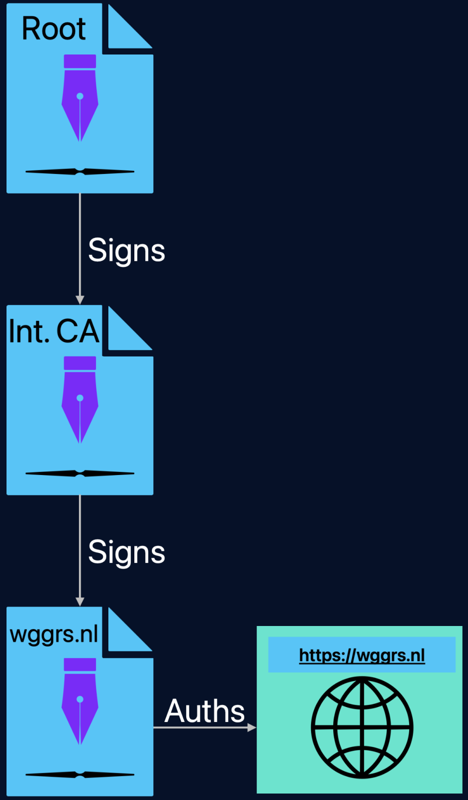
Picture by Bas Westerbaan: Sizing up post-quantum signatures (Cloudflare blog)

Authentication



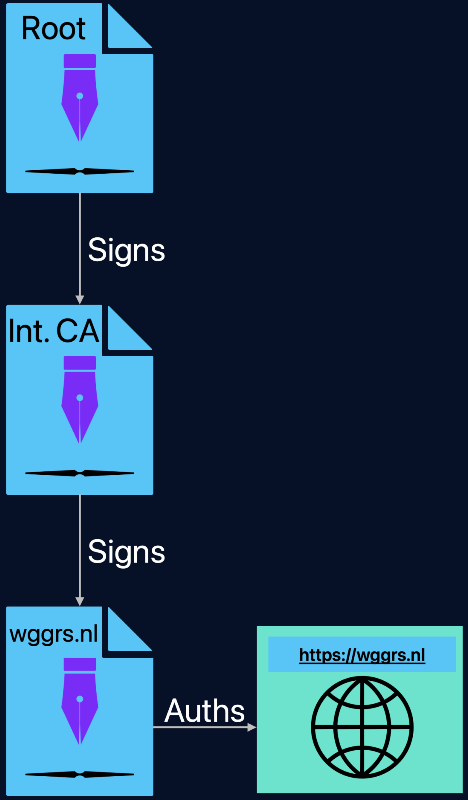
Authentication

- TLS authenticates servers (and clients) through certificates



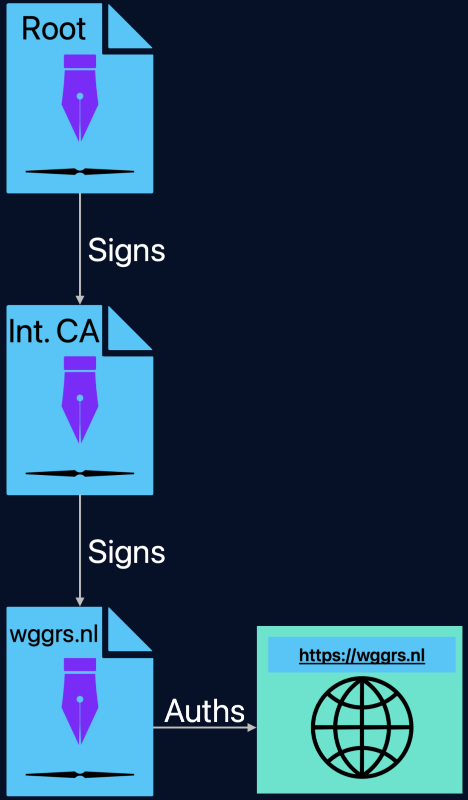
Authentication

- TLS authenticates servers (and clients) through certificates
- Root public key is preinstalled



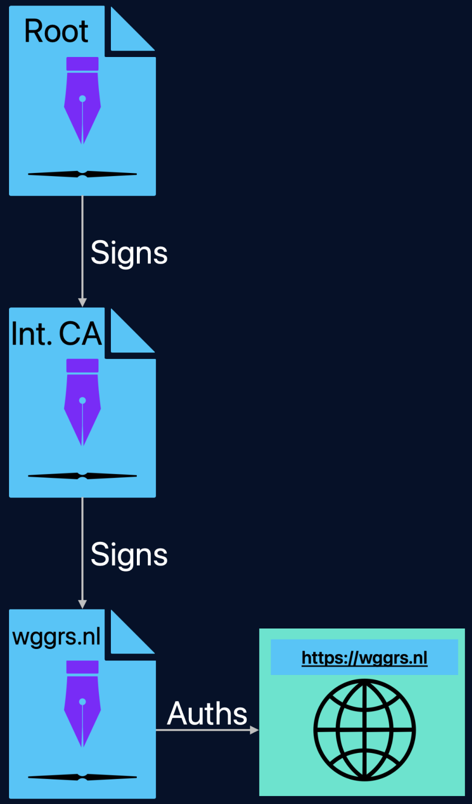
Authentication

- TLS authenticates servers (and clients) through certificates
- Root public key is preinstalled
- TLS traffic requirement:



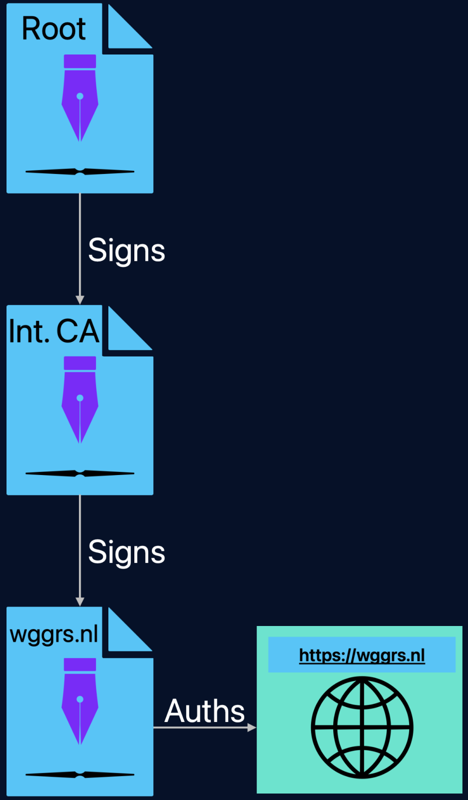
Authentication

- TLS authenticates servers (and clients) through certificates
- Root public key is preinstalled
- TLS traffic requirement:
    - 2 public keys



Authentication

- TLS authenticates servers (and clients) through certificates
- Root public key is preinstalled
- TLS traffic requirement:
  - 2 public keys
  - 3 signatures



Authentication transmission requirements

Signature alg	Public key traffic	Signature traffic	Sum
ML-DSA 44 (Dil2)	2.624	7.260	9.884
ML-DSA 65 (Dil3)	3.904	9.927	13.831
ML-DSA 87 (Dil5)	5.184	13.881	19.065
Falcon-512	1.794	1.998	3.792
Falcon-1024	3.586	3.840	7.426

Evaluating Dilithium and Falcon

Evaluating Dilithium and Falcon

- Even Dilithium2 already pushes us very close to additional round-trips

Evaluating Dilithium and Falcon

- Even Dilithium2 already pushes us very close to additional round-trips
- Falcon seems nice, but…
  - Signing uses floating-point arithmetic
  - Implementing Falcon without timing side-channels is extremely difficult
  - Verification is possible though

Evaluating Dilithium and Falcon

- Even Dilithium2 already pushes us very close to additional round-trips
- Falcon seems nice, but…
  - Signing uses floating-point arithmetic
  - Implementing Falcon without timing side-channels is extremely difficult
  - Verification is possible though

But there are many more signatures in web TLS!

Additional signatures

Additional signatures

- Certificate revocation:
  - Online Certificate Status Protocol
  - Staple OCSP status to certificate: another signature

Additional signatures

- Certificate revocation:
    - Online Certificate Status Protocol
    - Staple OCSP status to certificate: another signature
- Certificate Transparency
    - Started after Diginotar incident
    - Keeps CAs honest
    - Chrome and Safari require two CT inclusion proofs: two additional signatures

Additional signatures

- Certificate revocation:
  - Online Certificate Status Protocol
  - Staple OCSP status to certificate: another signature
- Certificate Transparency
  - Started after Diginotar incident
  - Keeps CAs honest
  - Chrome and Safari require two CT inclusion proofs: two additional signatures

Typical TLS handshake: 2 public keys and 7 signatures!

Two kinds of signature

Two kinds of signature

- Only one signature needs to be produced on-the-fly

Two kinds of signature

- Only one signature needs to be produced on-the-fly
- The remainder (certificate chain, SCT, OCSP) are produced out-of-band
  - “Offline”

Two kinds of signature

- Only one signature needs to be produced on-the-fly
- The remainder (certificate chain, SCT, OCSP) are produced out-of-band
  - “Offline”
- Can we use Falcon for CAs?
  - Protect against side-channels by hiding the HSM deep in a vault?
  - Complicates PKI management a bit

Two kinds of signature

- Only one signature needs to be produced on-the-fly
- The remainder (certificate chain, SCT, OCSP) are produced out-of-band
    - “Offline”
- Can we use Falcon for CAs?
    - Protect against side-channels by hiding the HSM deep in a vault?
    - Complicates PKI management a bit
- Hash-based signatures for CAs?
    - Few-times, not-level-5 XMSS?
    - Specially tuned SPHINCS+ with additional compression tricks?

Part 2

- Reducing the impact of authentication
- KEMTLS

Break time…

Public 4.0 Wikimedia Commons

Dealing with signatures

Signatures in TLS

- OCSP / Certificate revocation
- Certificate signatures
- Certificate transparency
- Handshake signature

Certificate revocation

Certificate revocation

- Certificate Revocation Lists were annoying to download: huge, slow

Certificate revocation

- Certificate Revocation Lists were annoying to download: huge, slow
- OCSP: make client check if certificate is currently non-revoked

Certificate revocation

- Certificate Revocation Lists were annoying to download: huge, slow
- OCSP: make client check if certificate is currently non-revoked
    - Privacy leak

Certificate revocation

- Certificate Revocation Lists were annoying to download: huge, slow
- OCSP: make client check if certificate is currently non-revoked
  - Privacy leak
- OCSP Stapling: have server include a recent proof of non-revocation along certificate

Certificate revocation

- Certificate Revocation Lists were annoying to download: huge, slow
- OCSP: make client check if certificate is currently non-revoked
  - Privacy leak
- OCSP Stapling: have server include a recent proof of non-revocation along certificate
- What do you do if an attacker blocks the OCSP query?

Returning to CRLs

Returning to CRLs

- Centrally process *all* CRLs, compress them, and push to users daily

Returning to CRLs

- Centrally process *all* CRLs, compress them, and push to users daily
- Larisch *et al.* (2017) CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers
  - Compress CRLs using Bloom filters
  - Implemented and deployed by Mozilla, circa 2020

Returning to CRLs

- Centrally process *all* CRLs, compress them, and push to users daily
- Larisch *et al.* (2017) CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers
  - Compress CRLs using Bloom filters
  - Implemented and deployed by Mozilla, circa 2020
- Similarly, Chrome implements CRLSets

Returning to CRLs

- Centrally process *all* CRLs, compress them, and push to users daily
- Larisch *et al.* (2017) CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers
    - Compress CRLs using Bloom filters
    - Implemented and deployed by Mozilla, circa 2020
- Similarly, Chrome implements CRLSets
- Since October 2022, Apple and Mozilla require CAs to publish CRLs

Returning to CRLs

- Centrally process *all* CRLs, compress them, and push to users daily
- Larisch *et al.* (2017) CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers
  - Compress CRLs using Bloom filters
  - Implemented and deployed by Mozilla, circa 2020
- Similarly, Chrome implements CRLSets
- Since October 2022, Apple and Mozilla require CAs to publish CRLs
- Only feasible for large browser vendors and the like

Certificates

- Use Falcon?
- Use MAYO?
- SQI-sign?

Abridged certificate chains

Abridged certificate chains

- Browsers already ship intermediate certificates
  - We've been transmitting them mostly for out-of-date clients
  - … so leave them out

Abridged certificate chains

- Browsers already ship intermediate certificates
  - We've been transmitting them mostly for out-of-date clients
  - … so leave them out
- **`draft-ietf-tls-cert-abridge`**: Abridged Compression for WebPKI Certificates
  - Collect intermediate certificates and include them in browsers
  - Assign an (incrementing) identifier to a particular list of intermediate certificates
  - Have client indicate which version of the list it has
  - If client's identifier indicates server's intermediate certificate is on the list, do not transmit intermediate certificate

Abridged certificate chains

- Browsers already ship intermediate certificates
    - We've been transmitting them mostly for out-of-date clients
    - … so leave them out
- **`draft-ietf-tls-cert-abridge`**: Abridged Compression for WebPKI Certificates
    - Collect intermediate certificates and include them in browsers
    - Assign an (incrementing) identifier to a particular list of intermediate certificates
    - Have client indicate which version of the list it has
    - If client's identifier indicates server's intermediate certificate is on the list, do not transmit intermediate certificate
- Leaving out intermediate certificates saves us 1 signature and 1 public key

Compressed certificate chains

Compressed certificate chains

- Certificates contain a lot of duplicate information
  - Policy information, Revocation URLs
  - Information about which CT logs have been used
  - Algorithm identifiers, …

Compressed certificate chains

- Certificates contain a lot of duplicate information
  - Policy information, Revocation URLs
  - Information about which CT logs have been used
  - Algorithm identifiers, …
- Just compress it (RFC8879)
  - Because it’s a fixed string, compression attacks like CRIME/BREACH doesn’t apply

Compressed certificate chains

- Certificates contain a lot of duplicate information
  - Policy information, Revocation URLs
  - Information about which CT logs have been used
  - Algorithm identifiers, …
- Just compress it (RFC8879)
  - Because it's a fixed string, compression attacks like CRIME/BREACH doesn't apply
- **`draft-ietf-tls-cert-abridge`**: Abridged Compression for WebPKI Certificates
  - Extends RFC8879 by sampling certificates for every CA from Certificate Transparency to pre-train a Zstd compression dictionary also included with browsers
  - Combined with intermediate suppression, saves on average 3 KB with pre-quantum certificates

Compressed certificate chains

- Certificates contain a lot of duplicate information
  - Policy information, Revocation URLs
  - Information about which CT logs have been used
  - Algorithm identifiers, …
- Just compress it (RFC8879)
  - Because it's a fixed string, compression attacks like CRIME/BREACH doesn't apply
- **`draft-ietf-tls-cert-abridge`**: Abridged Compression for WebPKI Certificates
  - Extends RFC8879 by sampling certificates for every CA from Certificate Transparency to pre-train a Zstd compression dictionary also included with browsers
  - Combined with intermediate suppression, saves on average 3 KB with pre-quantum certificates
- You can make Dilithium fit

Compressed certificate chains

- Certificates contain a lot of duplicate information
    - Policy information, Revocation URLs
    - Information about which CT logs have been used
    - Algorithm identifiers, …
- Just compress it (RFC8879)
    - Because it's a fixed string, compression attacks like CRIME/BREACH doesn't apply
- **`draft-ietf-tls-cert-abridge`**: Abridged Compression for WebPKI Certificates
    - Extends RFC8879 by sampling certificates for every CA from Certificate Transparency to pre-train a Zstd compression dictionary also included with browsers
    - Combined with intermediate suppression, saves on average 3 KB with pre-quantum certificates
- You can make Dilithium fit
- … but only for WebPKI

Building on Certificate Transparency

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
  - Collect all currently valid certificates from certificate transparency logs, every hour

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
    - Collect all currently valid certificates from certificate transparency logs, every hour
    - Build a Merkle tree

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
    - Collect all currently valid certificates from certificate transparency logs, every hour
    - Build a Merkle tree
    - Ship the new tree head to browsers every hour

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
    - Collect all currently valid certificates from certificate transparency logs, every hour
    - Build a Merkle tree
    - Ship the new tree head to browsers every hour
    - Server replaces certificates by public key plus authentication path

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
  - Collect all currently valid certificates from certificate transparency logs, every hour
  - Build a Merkle tree
  - Ship the new tree head to browsers every hour
  - Server replaces certificates by public key plus authentication path
  - Authenticates server public key in under 1000 bytes

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
    - Collect all currently valid certificates from certificate transparency logs, every hour
    - Build a Merkle tree
    - Ship the new tree head to browsers every hour
    - Server replaces certificates by public key plus authentication path
    - Authenticates server public key in under 1000 bytes
    - Server still needs to authenticate itself to the client though

Building on Certificate Transparency

- Chrome and Safari require proof of inclusion in certificate transparency logs
- … thus, the CA certificates can be made pointless
- Certificate transparency: Merkle tree of logged certificates
- Merkle Tree Certificates for TLS draft-davidben-tls-merkle-tree-certs-01
  - Collect all currently valid certificates from certificate transparency logs, every hour
  - Build a Merkle tree
  - Ship the new tree head to browsers every hour
  - Server replaces certificates by public key plus authentication path
  - Authenticates server public key in under 1000 bytes
  - Server still needs to authenticate itself to the client though
- Probably only suitable for WebPKI

The matter of the server signature

The matter of the server signature

- Both Abridged Certs and Merkle Tree Certs still use handshake signatures

The matter of the server signature

- Both Abridged Certs and Merkle Tree Certs still use handshake signatures
- Dilithium is very large

The matter of the server signature

- Both Abridged Certs and Merkle Tree Certs still use handshake signatures
- Dilithium is very large
- Falcon is probably not safe or too slow to use

The matter of the server signature

- Both Abridged Certs and Merkle Tree Certs still use handshake signatures
- Dilithium is very large
- Falcon is probably not safe or too slow to use
- What is the function of the signature in the TLS handshake?

The matter of the server signature

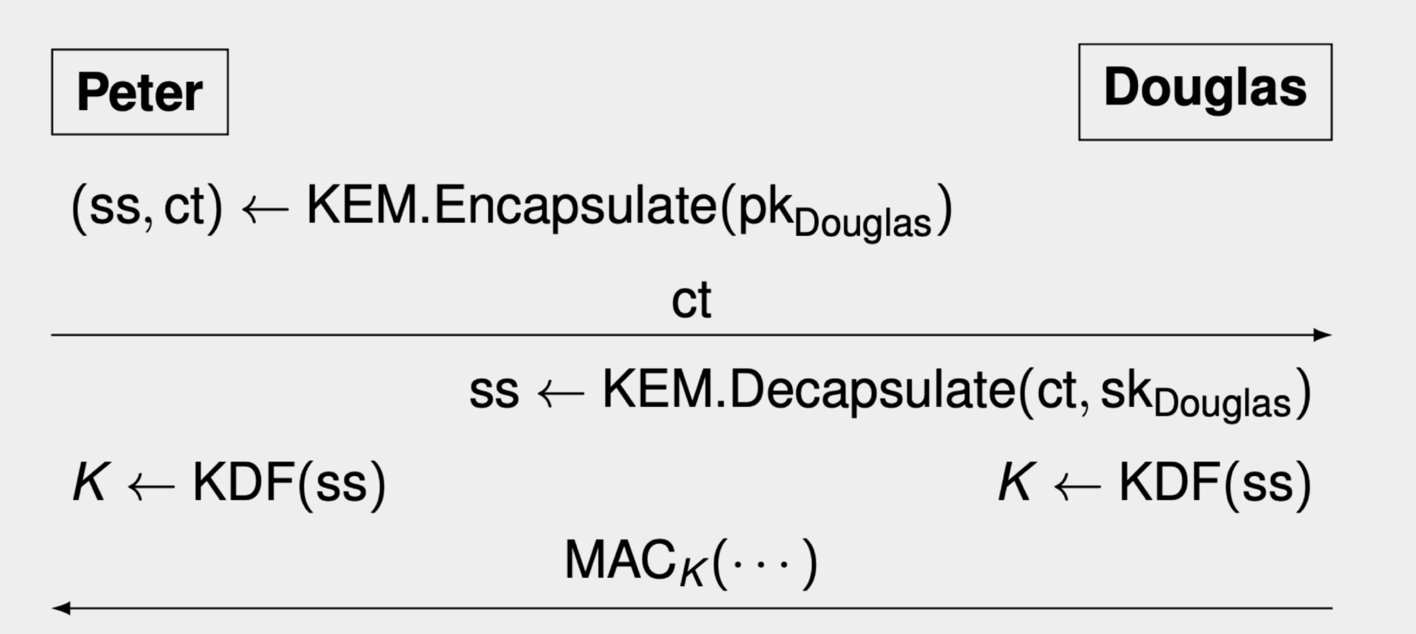
- Both Abridged Certs and Merkle Tree Certs still use handshake signatures
- Dilithium is very large
- Falcon is probably not safe or too slow to use
- What is the function of the signature in the TLS handshake?
  - Proves access to the private key that corresponds to the certificate’s public key

AuthKEM

Authentication via key exchange

- The signature in TLS proves that the server has access to the private signing key
- If I send you Enc(k, m), and you can show me m, you must know k
    - You have access to the secret key

Authenticated Key Exchange with KEM



Peter **Douglas**

$(\mathsf{ss}, \mathsf{ct}) \leftarrow \mathsf{KEM.Encapsulate}(\mathsf{pk}_{\mathsf{Douglas}})$

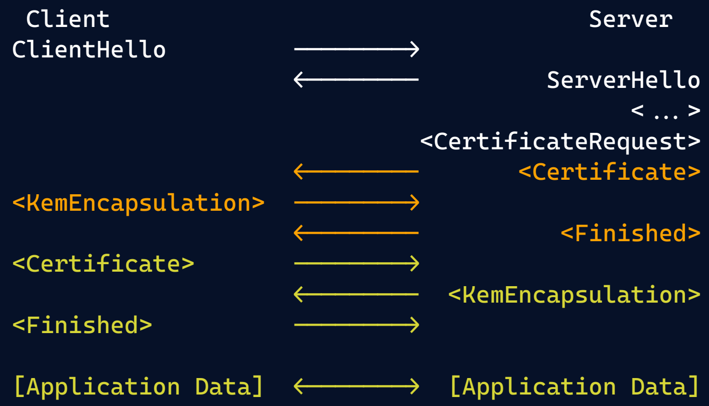
$$\xrightarrow{\mathsf{ct}}$$

$\mathsf{ss} \leftarrow \mathsf{KEM.Decapsulate}(\mathsf{ct}, \mathsf{sk}_{\mathsf{Douglas}})$

$K \leftarrow \mathsf{KDF}(\mathsf{ss})$ $K \leftarrow \mathsf{KDF}(\mathsf{ss})$

$$\xleftarrow{\mathsf{MAC}_K(\cdots)}$$

TLS authentication via KEM (naively)



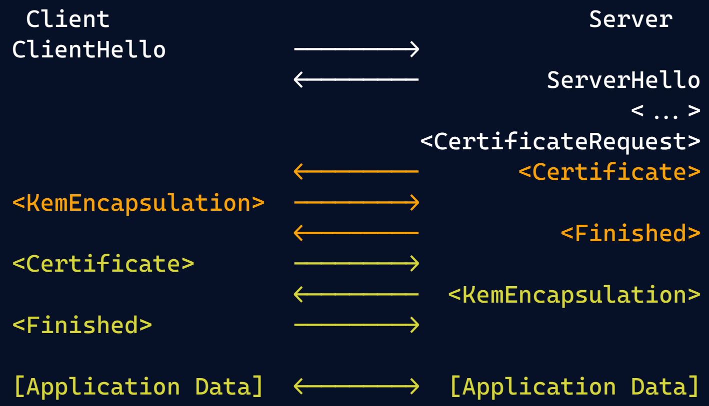
```
 Client                                         Server
ClientHello          ———————————>
                     <———————————          ServerHello
                                               < ... >
                                  <CertificateRequest>
                     <———————————        <Certificate>
<KemEncapsulation>   ———————————>
                     <———————————           <Finished>
<Certificate>        ———————————>
                     <———————————   <KemEncapsulation>
<Finished>           ———————————>

[Application Data]   <——————————>   [Application Data]
```

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)

TLS authentication via KEM (naively)

- KEMs require interaction



```
 Client                                        Server
ClientHello
                                          ServerHello
                                              < ... >
                                 <CertificateRequest>
                                        <Certificate>
<KemEncapsulation>
                                           <Finished>
<Certificate>
                                   <KemEncapsulation>
<Finished>

[Application Data]                 [Application Data]
```

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)

TLS authentication via KEM (naively)

- KEMs require interaction
- Unlike signatures, which can authenticate immediately

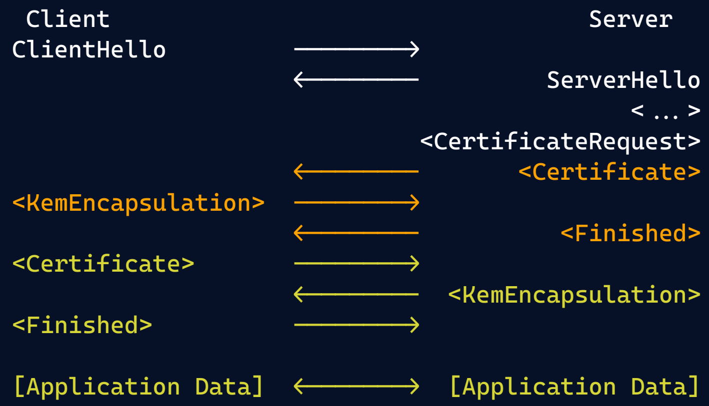
```
 Client                                               Server
ClientHello              ⟶
                         ⟵                        ServerHello
                                                      < ... >
                                           <CertificateRequest>
                         ⟵                        <Certificate>
<KemEncapsulation>       ⟶
                         ⟵                           <Finished>
<Certificate>            ⟶
                         ⟵                   <KemEncapsulation>
<Finished>               ⟶

[Application Data]       ⟷                   [Application Data]
```

<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)

TLS authentication via KEM (naively)

- KEMs require interaction
- Unlike signatures, which can authenticate immediately
    - (pk, m, sig(m)) in one message



```
Client                                                Server
ClientHello                 ⟶
                            ⟵                    ServerHello
                                                     < … >
                                        <CertificateRequest>
                            ⟵                  <Certificate>
<KemEncapsulation>          ⟶
                            ⟵                     <Finished>
<Certificate>               ⟶
                            ⟵             <KemEncapsulation>
<Finished>                  ⟶

[Application Data]          ⟷             [Application Data]
```

```
<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)
```

TLS authentication via KEM (naively)

- KEMs require interaction
- Unlike signatures, which can authenticate immediately
  - (pk, m, sig(m)) in one message

This means that the naive integration of KEMs in authentication requires an additional round-trip!

```
 Client                                            Server
ClientHello              ------------>
                         <------------          ServerHello
                                                   < ... >
                                         <CertificateRequest>
                         <------------          <Certificate>
<KemEncapsulation>       ------------>
                         <------------             <Finished>
<Certificate>            ------------>
                         <------------     <KemEncapsulation>
<Finished>               ------------>

[Application Data]       <----------->     [Application Data]
```

```
<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)
```

TLS authentication via KEM (naively)

- KEMs require interaction
- Unlike signatures, which can authenticate immediately
    - (pk, m, sig(m)) in one message

This means that the naive integration of KEMs in authentication requires an additional round-trip!

Exercise for at home: see how doing this with Diffie-Hellman's non-interactive key exchange property is possible in a single round-trip (see: Krawczyk & Wee's OPTLS)

```
 Client                                            Server
 ClientHello            -------->
                        <--------             ServerHello
                                                  < ... >
                                     <CertificateRequest>
                        <--------           <Certificate>
<KemEncapsulation>      -------->
                        <--------              <Finished>
<Certificate>           -------->
                        <--------      <KemEncapsulation>
<Finished>              -------->

[Application Data]      <------->      [Application Data]
```

```
<msg>: enc. w/ keys derived from ephemeral KEX (HS)
[msg]: enc. w/ keys derived from HS (MS)
```

Implicit authentication

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)
- I do not know if the owner of sk is actually participating in the conversation

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)
- I do not know if the owner of sk is actually participating in the conversation
  - Someone could just have copy/pasted the public key

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)
- I do not know if the owner of sk is actually participating in the conversation
    - Someone could just have copy/pasted the public key
    - But they will not be able to read message

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)
- I do not know if the owner of sk is actually participating in the conversation
  - Someone could just have copy/pasted the public key
  - But they will not be able to read message
- E.g. appears in Signal, Wireguard, Noise Protocols

Implicit authentication

- If I generate (ss, ct) <- KEM.Encapsulate(pk) I know that only the owner of sk can read AEAD(ss, message)
- I do not know if the owner of sk is actually participating in the conversation
  - Someone could just have copy/pasted the public key
  - But they will not be able to read message
- E.g. appears in Signal, Wireguard, Noise Protocols
- If we make the owner of sk use ss, we get explicit authentication

AuthKEM

- Use authentication key to send implicitly authenticated request immediately
- Avoids additional round-trip
- Does require non-trivial implementation changes

draft-celi-wiggers-tls-authkem:
KEM-based Authentication for TLS 1.3

```
Client                                          Server

ClientHello
+ key_share
+ signature_algorithms
                     --------->
                                           ServerHello
                                           + key_share
                                 <EncryptedExtensions>
                     <--------- <Certificate: kem pk>

<KEMEncapsulation>   --------->
{Finished}           --------->
[Application Data]   --------->
  GET /cat.gif HTTP/1.1

                     <---------             {Finished}

[Application Data]   <-------->     [Application Data]
```

Why AuthKEM?

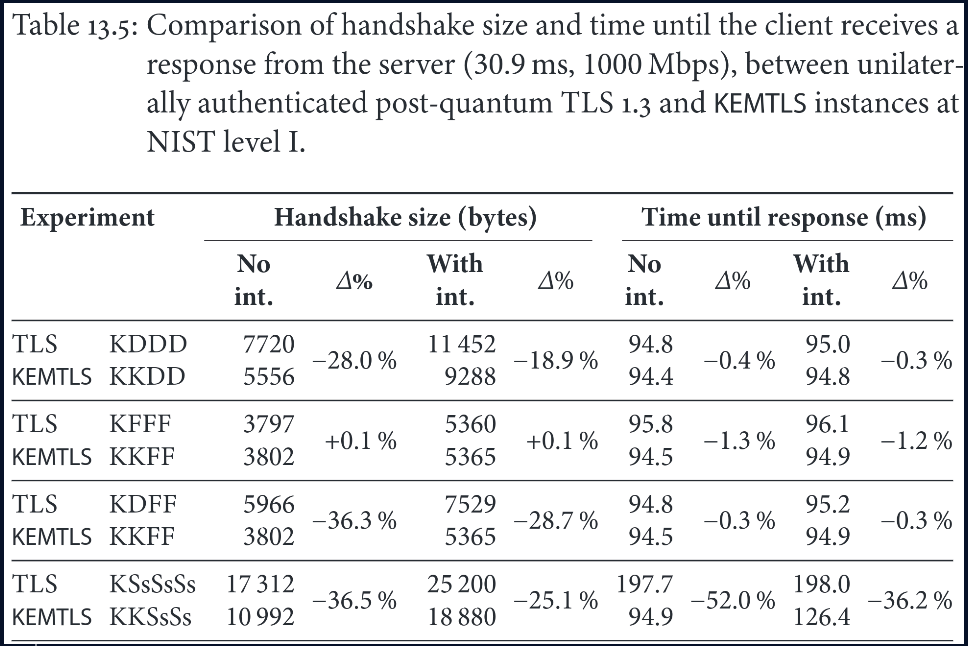


Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

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	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	

Why AuthKEM?

- Save significant amounts of handshake data
  - e.g. replace Dilithium-2 by Kyber-768: 3732 → 2272 bytes (-39%) for handshake authentication

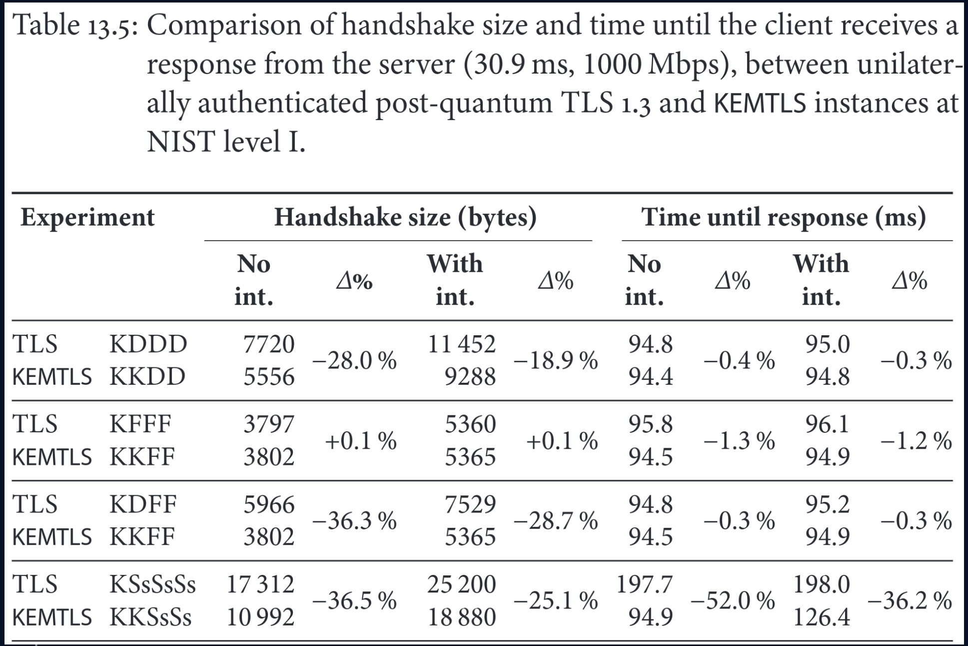


Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

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	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	

Why AuthKEM?

- Save significant amounts of handshake data
  - e.g. replace Dilithium-2 by Kyber-768: 3732 → 2272 bytes (-39%) for handshake authentication
- Kyber is cheaper to compute

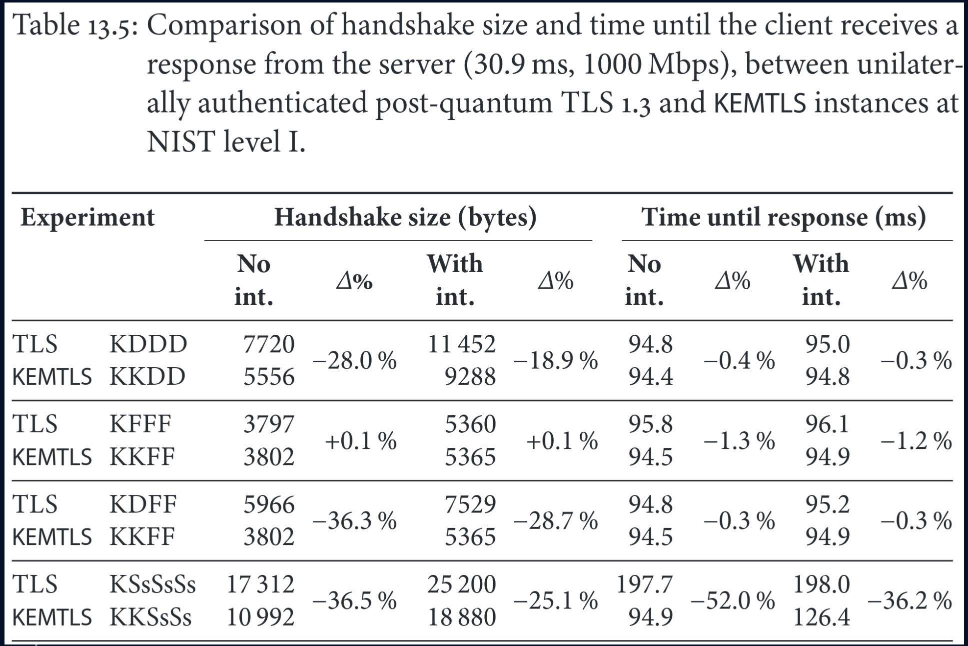


Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

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	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	

Why AuthKEM?

- Save significant amounts of handshake data
    - e.g. replace Dilithium-2 by Kyber-768: 3732 → 2272 bytes (-39%) for handshake authentication
- Kyber is cheaper to compute
- Combining AuthKEM with Falcon for offline signatures is possible
    - Using AuthKEM can reuse the KEM implementation from key exchange
    - don't need Kyber AND Dilithium AND Falcon implementations → reduces code size/ complexity

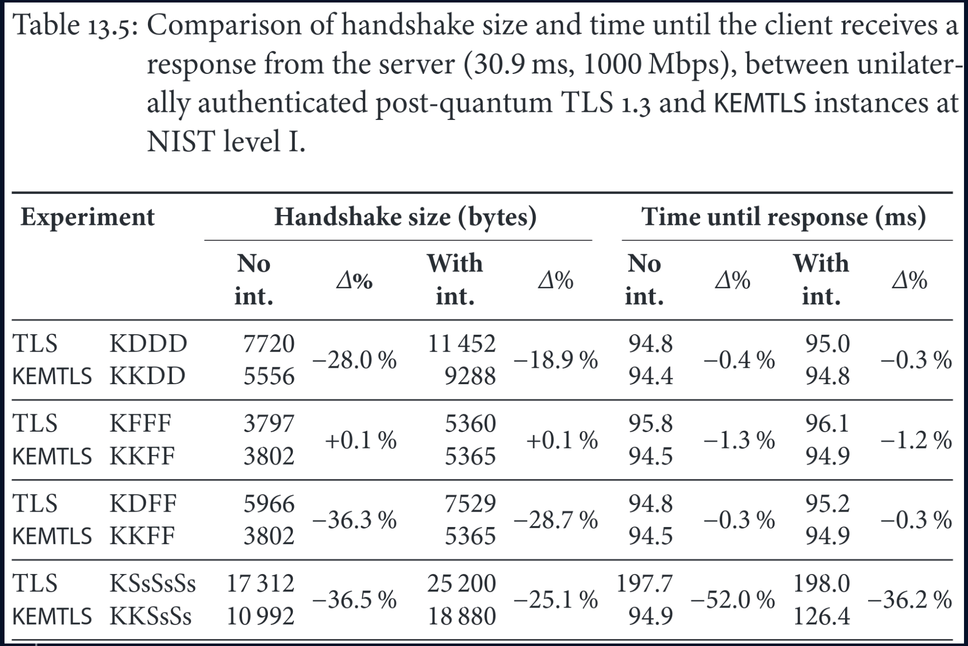


Table 13.5: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilaterally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level I.

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	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	

Level V

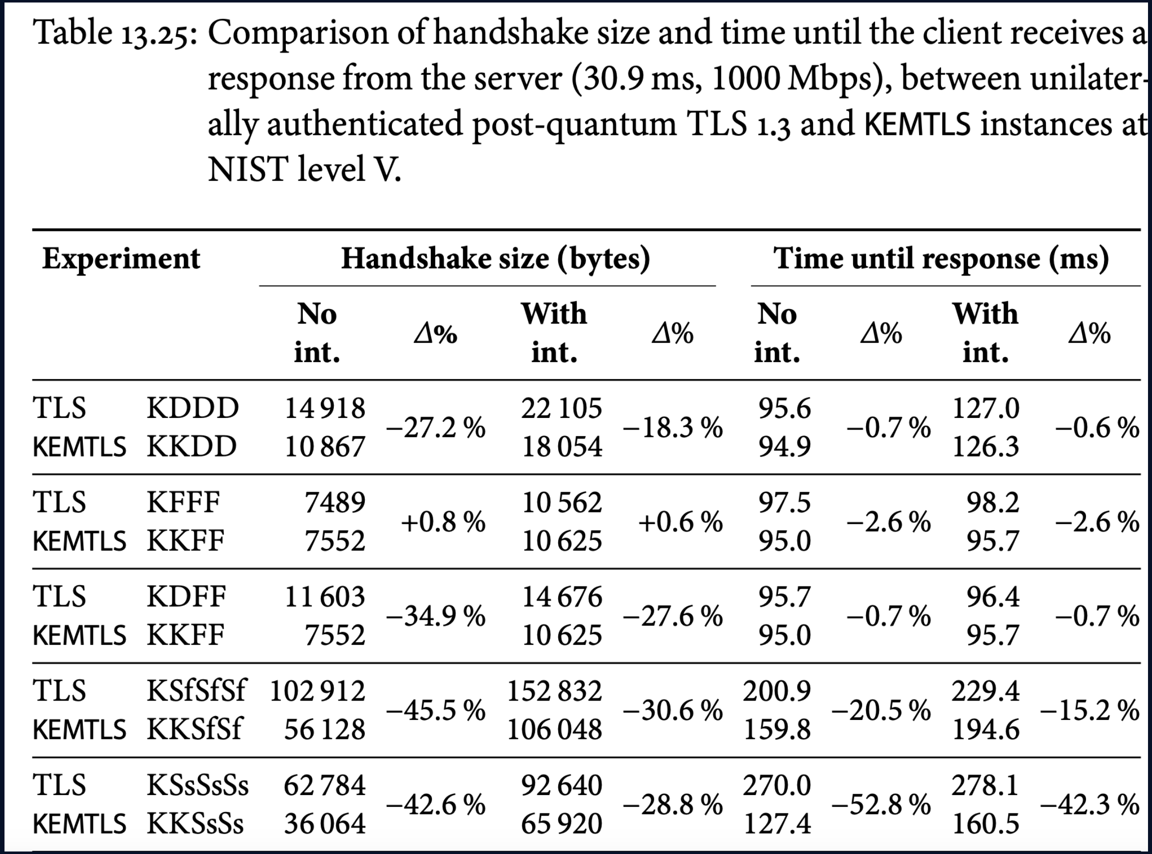


Table 13.25: Comparison of handshake size and time until the client receives a response from the server (30.9 ms, 1000 Mbps), between unilater- ally authenticated post-quantum TLS 1.3 and KEMTLS instances at NIST level V.

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	

What about the SCT/OCSP signatures?

What about the SCT/OCSP signatures?

- AuthKEM is fully compatible with Merkle Tree Certificates or Abridged Certificates

What about the SCT/OCSP signatures?

- AuthKEM is fully compatible with Merkle Tree Certificates or Abridged Certificates
    - Using AuthKEM further pushes down the communication costs

What about the SCT/OCSP signatures?

- AuthKEM is fully compatible with Merkle Tree Certificates or Abridged Certificates
    - Using AuthKEM further pushes down the communication costs
- SCT/OCSP signatures are very “web” problems

What about the SCT/OCSP signatures?

- AuthKEM is fully compatible with Merkle Tree Certificates or Abridged Certificates
    - Using AuthKEM further pushes down the communication costs
- SCT/OCSP signatures are very “web” problems
    - The proposed solutions only work in WebPKI context

What about the SCT/OCSP signatures?

- AuthKEM is fully compatible with Merkle Tree Certificates or Abridged Certificates
  - Using AuthKEM further pushes down the communication costs
- SCT/OCSP signatures are very “web” problems
  - The proposed solutions only work in WebPKI context
- AuthKEM is especially effective in constrained environments (i.e. not using phone or laptop CPUs)

Extensions

Client Authentication

Requires additional round-trip: We need to encrypt the certificate and can't do it earlier.

Pre-shared KEM keys

- E.g. cache or pre-install server KEM key
- Send ciphertext in first client message
- Abbreviate handshake further
- Easy fall-back to AuthKEM

=> "AuthKEM-PSK"

Post-Quantum TLS

Post-Quantum TLS

TLS confidentiality

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier
- Impact of Dilithium is very large

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier
- Impact of Dilithium is very large
- Several proposals in development for reducing impact of authentication

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier
- Impact of Dilithium is very large
- Several proposals in development for reducing impact of authentication
    - Abridged Certificates uses clever compression

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier
- Impact of Dilithium is very large
- Several proposals in development for reducing impact of authentication
    - Abridged Certificates uses clever compression
    - Merkle Tree Certificates fundamentally changes the trust model

Post-Quantum TLS

TLS confidentiality

- Transitioning TLS kex to PQ is in progress
- ~~Kyber~~ ML-KEM is the only plausible candidate for key exchange
- Everyone I talk to seems in favor of hybrids
- Some questions remain surrounding UDP-based protocols (DTLS, QUIC) vs multi-packet ClientHello

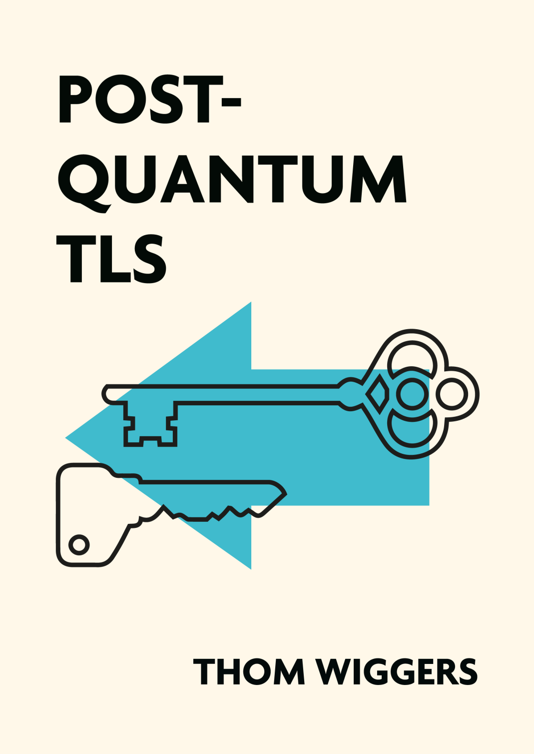
TLS authentication

- WebPKI authentication is more than just certificates
- TCP initial congestion window is a barrier
- Impact of Dilithium is very large
- Several proposals in development for reducing impact of authentication
  - Abridged Certificates uses clever compression
  - Merkle Tree Certificates fundamentally changes the trust model
- AuthKEM swaps signature auth for KEMs

More on Post-Quantum TLS

- Discussion of how to make post-quantum TLS, OPTLS (with CSIDH) and KEMTLS
- Proofs of KEMTLS by pen-and-paper and using Tamarin
- Loads of benchmark measurements for TLS/KEMTLS instances at NIST level I, III, V

- wggrs.nl/p/thesis



POST-QUANTUM TLS

THOM WIGGERS