Implementing CAESAR candidate Prøst on ARM11

Thom Wiggers
Radboud University, Institute for Computing and Information Sciences
thom@thomwiggers.nl

ABSTRACT

PRØST was a contestant in the CAESAR competition for authenticated encryption. I optimised PRØST for the ARM11 microprocessor architecture. By trying to find a provably minimal program for one of the sub-operations, I found a new approach to implementing `MixSlices`, one of the sub-operations in PRØST's permute function. This new implementation has 33% fewer arithmetic operations than the original version. Using this result and by implementing PRØST in assembly and applying micro-optimisations, a performance gain of 28% to 48% was achieved.

INTRODUCTION

Authenticated encryption schemes use symmetric keys to encrypt data providing not only confidentiality but also integrity and authenticity [4]. Using these schemes avoids having to combine authentication and traditional, confidentiality-only, encryption, something that has often led to vulnerabilities [11].

A variant are *authenticated encryption with associated data* schemes [17]. These allow to also include information that does not need to be encrypted but of which the integrity and authenticity needs to still be guaranteed.

The CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) competition was announced in January 2013 to help select a portfolio of ciphers that "(1) offer advantages over AES-GCM[1] and (2) are suitable for widespread adoption" [7]. PRØST was a contestant in this competition.

Optimised implementations on various platforms help show that an algorithm is suitable for widespread deployment. Cryptography is not only used on PCs based on the `amd64` architecture: it is perhaps even more widely used and needed in embedded platforms, smart cards and mobile devices. ARM11 is a prominent example that is used in many of these.

I re-implemented and optimised PRØST in assembly language. Aside from doing "regular" assembly-level micro-optimisations, I also tried to reach a provably minimal algorithm for one of the sub-operations.

All resulting software can be found via **`https://thomwiggers.nl/proest/`**.

[1] AES with the Galois/Counter Mode of operation [13]. This is an authenticated encryption scheme based on AES.

PRELIMINARIES

Prøst

PRØST consists of the PRØST-permutation, which is then combined with various modes of operation. These are COPA [2], OTR[2] [14] and APE [1]. This gives all of the ciphers in the PRØST family: PRØST-COPA, PRØST-OTR and PRØST-APE. My optimisations focused on the PRØST permutation, as it is by far the most expensive operation in each of these modes.

Here I will briefly summarise PRØST's permutation as described in the PRØST v1.1 paper [10]. I will be describing the PRØST-128 version, which provides 128 bits of security. The full description of PRØST, including the modes of operation and PRØST-256, can be found in the PRØST paper.

PRØST-128 has a 256-bit state s which is considered as a $4 \times 4 \times 16$ three-dimensional block

$$\mathrm{S} = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

where each $s_{x,y}$ is a 16-bit value. PRØST's authors call these *lanes*. The terms *row*, *column*, *slice*, *plane* and *sheet* for the other parts of the state are described in Figure 1.

The permutation consists, in the PRØST-128 case, of 16 rounds. The round function $R_i : \mathbb{F}_2^{256} \to \mathbb{F}_2^{256}$ with $0 \leq i < 16$, can be defined as $R_i(x) = (\texttt{AddConstants}_i \circ \texttt{ShiftPlanes}_i \circ \texttt{MixSlices} \circ \texttt{SubRows})(x)$.

In the following I use $\oplus$ to denote the binary exclusive or operation, and $\wedge$ to denote a binary and. "$a \lll n$" and "$a \ggg n$" mean that a is rotated n bits to the left or to the right, respectively.

SubRows

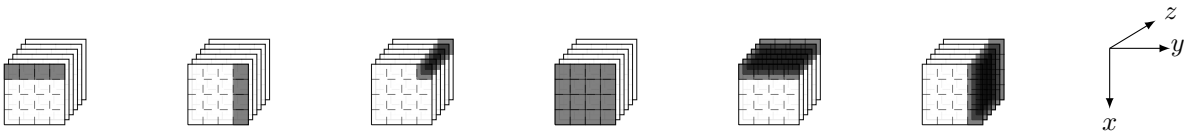
The `SubRows` operation substitutes each row (a, b, c, d) of the state by a new row (a', b', c', d') where

$$\begin{aligned} a' &= c \oplus (a \wedge b), \qquad b' = d \oplus (b \wedge c), \\ c' &= a \oplus (a' \wedge b'), \quad d' = b \oplus (b' \wedge c'). \end{aligned} \tag{1}$$

MixSlices

The `MixSlices` operation mixes the slices of the state by multiplying the vector of lanes

[2] PRØST-OTR was recently shown to be vulnerable to a related-key forgery attack by Dobraunig, Eichlseder and Mendel [8].



(a) Row (b) Column (c) Lane (d) Slice (e) Plane (f) Sheet (g) Axes

Figure 1: Nomenclature for state parts (figure adopted from [10]).

$(s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{1,0}, \ldots s_{3,3})^T$ with matrix M, where

$$M = \begin{pmatrix} 1&0&0&0&1&0&0&1&0&0&1&0&1&0&1&1 \\ 0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&1 \\ 0&0&1&0&0&1&0&0&1&1&0&0&1&0&0&0 \\ 0&0&0&1&0&0&1&0&0&1&1&0&0&1&0&0 \\ 1&0&0&1&1&0&0&0&1&0&1&1&0&0&1&0 \\ 1&0&0&0&0&1&0&0&1&0&0&1&0&0&0&1 \\ 0&1&0&0&0&0&1&0&1&0&0&0&1&1&0&0 \\ 0&0&1&0&0&0&0&1&0&1&0&0&0&1&1&0 \\ 0&0&1&0&1&0&1&1&1&0&0&0&1&0&0&1 \\ 0&0&0&1&1&0&0&1&0&1&0&0&1&0&0&0 \\ 1&1&0&0&1&0&0&0&0&0&1&0&0&1&0&0 \\ 0&1&1&0&0&1&0&0&0&0&0&1&0&0&1&0 \\ 1&0&1&1&0&0&1&0&1&0&0&1&1&0&0&0 \\ 1&0&0&1&0&0&0&1&1&0&0&0&0&1&0&0 \\ 1&0&0&0&1&1&0&0&0&1&0&0&0&0&1&0 \\ 0&1&0&0&0&1&1&0&0&0&1&0&0&0&0&1 \end{pmatrix}.$$

ShiftPlanes$_i$

The operation `ShiftPlanes`$_i$ rotates each of the lanes in a plane of the state by a a given amount. The lanes in each row are rotated by a number of bits given by the *shift vector*, which is different for odd and even rounds.

For round i, if i is even, the lanes in the first row are rotated by zero bits, in the second row they are rotated by one bit, the third row's lanes are rotated by eight bits and the lanes in the last row are rotated by nine bits.

If i is odd, the lanes in the four rows are rotated by zero, two, four and six bits, respectively.

AddConstants$_i$

`AddConstants`$_i$, the last operation in each round i of PRØST, updates the state s by adding a constant to each lane. There are two constants, c_1 = `0x7581` and c_2 = `0xb2c5`. With index j enumerating the lanes from 0 to 16, constant c_1 is applied to even lanes with even j and c_2 is applied to lanes with odd j. Before being applied, the constants are rotated left by the round number i and by index j.

ARM11

In this subsection I will briefly set out the most important characteristics of the ARM11 microarchitecture that are relevant to the rest of this work [3].

ARM11 processors have a 32-bit instruction set. They provide fourteen 32-bit 'work' registers and a stack pointer to the programmer. If the programmer needs more registers, he or she will need to store values to main memory. These operations are expensive and should mostly be avoided.

The architecture provides instructions that allow to rotate or shift registers by an arbitrary amount, spending one computation cycle. Additionally, all arithmetic instructions support having the second input value rotated or shifted by an arbitrary distance. These shifts are essentially free.

Pipeline

The ARM11 is a pipelined architecture, which means that the processor can work on several instructions at the same time. Instructions take a certain amount of cycles to complete. If their results are not immediately needed, the CPU will work on other instructions. If however the result is immediately needed, the CPU will wait for it to become available. Most arithmetic instructions have a one-cycle latency, meaning the results can be used by the next instruction immediately. Reading from memory has a 3 cycle latency, if the load is from cache, before the result becomes available. This means that careful scheduling to avoid these latencies can drastically reduce execution time.

OPTIMISING PRØST

I will now explain the biggest optimisations in my implementation of PRØST-128.

Loading two lanes into one CPU register

The lanes in the PRØST state are each 16 bits long, while the CPU registers are each 32 bits in size. Considering that the lanes are stored consecutively in memory, it is possible to load two lanes into one register in one load. This obviously saves us from one register and one load that we would need to do if we naively loaded each lane separately into one register. This however does mean that we need to take care how to apply operations to this register. This can be achieved by using the previously described free shifts in arithmetic instructions: rotating allows to selectively apply the correct half of the register.

MixSlices

`MixSlices` is by far the most expensive operation in PRØST's permutation. A straightforward implementation of the system in the matrix M represents 72 exclusive or operations.

The exclusive or is commutative and associative, which allows us to re-order the inputs in any way we like. Several of the output lanes share some of the input lanes they use, meaning there are combinations of lanes that could be reused in several multiplications. For example, $s'_{0,0}$ and $s'_{0,1}$ share the intermediate result $s_{1,0} \oplus s_{3,0} \oplus s_{3,3}$, as illustrated in (2).

$$\begin{aligned} s'_{0,0} &= s_{0,0} \oplus \mathbf{s_{1,0}} \oplus s_{1,3} \oplus s_{2,2} \oplus \mathbf{s_{3,0}} \oplus s_{3,2} \oplus \mathbf{s_{3,3}} \\ s'_{0,1} &= s_{0,1} \oplus \mathbf{s_{1,0}} \oplus s_{2,3} \oplus \mathbf{s_{3,0}} \oplus \mathbf{s_{3,3}} \end{aligned} \tag{2}$$

This of course leads to the question: can we find a way to exploit this feature so that we can find the program with the maximum amount of reuse? Or in other words: what is the shortest implementation of `MixSlices`?

Optimisation problem

We can represent a function like `MixSlices` as a program with a sequence of lines of the shape $u = v \oplus w$ where v and w are either from the set of input values or one of the previous lines of the program. In fact, this notation is exactly how one would implement `MixSlices`: the arithmetic instructions in ARM assembly look exactly like that. Programs of this form over $\mathbb{F}_2$ are known as *linear straight-line programs* [6, 9].

Unfortunately, the problem of finding the shortest linear program (SLP) is known to be NP-hard. Boyar, Matthews and Peralta additionally showed that SLP is MAXSNP-complete [6]. MAXSNP is a class of optimisation problems that can be approximated with some bounded error [15]. In other words, finding the shortest version of `MixColumns` is going to be very computationally expensive.

Fuhs and Schneider-Kamp show in [9] that it is possible to transform SLP to a different kind of problem: the boolean satisfiability problem (SAT).

To define SLP as the decision problem "does a program of k lines exist" in SAT, Fuhs and Schneider-Kamp encode functions such as the one for `MixSlices` with n inputs and m outputs as an $m \times n$ matrix A, where every row represents one of the outputs and every column one of the inputs. $A_{x,y} = 1$ if and only if in output x, input y is used. This matrix is exactly PRØST's M. They then define matrix B as a $k \times n$ matrix, where $b_{x,y} = 1$ if and only if in line x input variable y is used. The $k \times k$ matrix C is defined where $c_{x,y} = 1$ if and only if intermediate result y is reused in line x of the program. Finally, they define a matrix f to map intermediate results to outputs.

The decision problem is to find valuations of B, C and f such that a set of constraints still hold. These constraints are boolean formulae that can only be satisfied by valid programs.

I developed a Java program that allows to input programs as a matrix A and then tries to solve the SLP-SAT problem for a specified length k. SAT4j [12] was used to transform the problem from predicate logic to a SAT problem, which was then solved also using SAT4j.

Unfortunately, the smallest satisfiable k proved to be out of reach with this implementation of the SLP-SAT problem. The size of the constraints is in $\mathcal{O}(n \cdot k^2)$ [9], but it appeared that the addition of new constraints to the SAT4j encoder got more expensive faster than that. The largest k for which we could transform SLP to SAT was only $k = 26$. It was unable to provide an answer for $k = 26$ even after running the program for over two weeks. This might be because showing a problem is unsatisfiable is much harder than showing it is satisfiable.

Boyar et al. give a heuristic which allows to approximate the shortest straight-line program. A brief summary of this heuristic, described in [6], will be given here. We define matrix S in which we will store previously produced functions. S has n columns. $s_{x,y} = 1$ if and only if the yth input variable is a part of the function defined by row x. As for the SAT program, we provide the input program as a matrix A. In the case of PRØST, A will be initialised as M.

S is then initialised to contain the input variables $x_0, \cdots, x_{n-1}$, so in the case of $n = 3$, $S = ([1,0,0],[0,1,0],[0,0,1])$. Then, we define a distance function that for a given row in A determines the smallest number of additions of rows in S that need to be made to get that row.

The program then generates new rows in S as combinations of rows in S, minimising the sum of the distance function. Some optimisations are used to achieve better performance. Finally, when the sum of the distance function is known, S can be transformed back into a linear straight-line program.

The above heuristic was, after running for four days on a 24-core machine, able to find a much shorter implementation of `MixSlices` using only 48 exclusive ors. This approach can be found in my implementation.

AddConstants

The constants c_1 and c_2 added by `AddConstants` are first rotated by the round number. Because the first time we need c_2 it needs to be rotated by 1, we can instead set the constant c_2 to $c_2 \lll 1$ at compile time and thus save one instruction.

Because we want to load two lanes into one register every time, we need half of the free rotations we can get from the ARM architecture to shift the correct value in place. That means we still need to explicitly rotate one of the constants every time, instead of using free rotations. This means we still need to do nine explicit rotations: two for the initial rotation by the round number, and 7 rotations of c_2 we can not do for free in arithmetic instructions.

Inlining and unrolling the PRØST operations

We can reduce the overhead of calling subroutines by putting the operations consecutively in the same subroutine. Having the operations in the same subroutine also enables us to do some nice things such as keeping intermediate values in registers between operations. This saves us quite a few loads and stores. Operations previously also had to clean up intermediate results that were spilled to memory and then put those back into the PRØST state. In an inlined program, later operations can just retrieve those values from the stack.

Finally, the unrolling allows to hide latencies better. One can start retrieving data needed for the next function and then while waiting for the load latency do final computations of the previous function.

RESULTS AND COMPARISON

Benchmark results

All benchmark results were obtained by using the SUPERCOP [5] benchmarking suite for cryptographic systems running on a Raspberry Pi model B overclocked to run at 800MHz. Frequency scaling was disabled. The cycle counter still reports accurate results even when overclocked. We used the 2014-11-24 release of SUPERCOP, which was the most recent release when we did the experiments. The version of `gcc`

Implementation	APE	COPA	APE
Reference (C)[a]	2,976,123	2,402,577	1,569,582
ARM Assembly	1,900,274[a]	1,648,407[a]	848,100[b]
Improvement	36%	28%	46%

[a] Compiled with `gcc -funroll-loops -fno-schedule-insns -O3 -fomit-frame-pointer`
[b] Compiled with `gcc -O3 -fomit-frame-pointer`

Table 1: Benchmark results in median cycle counts

used was `4.9.3 20141224 (prerelease)`. The cycle counter provided by the ARM architecture was used to facilitate benchmarking.

The benchmark results can be found in Table 1. The reported figure is the "number of cycles used by a typical cryptographic operation" as reported by SUPERCOP. Also included are the compiler flags used to get the reported figures.

The implementation of PRØST has been submitted to the eBACS project for public benchmarking and has been released as open source software under the New BSD licence.

Comparison and further work

SUPERCOP currently contains no other implementations of PRØST-128 than the reference C implementation. Rijneveld implemented a vectorised version of PRØST for ARMv7 with NEON [16]. A cursory comparison with his reported cycle counts show that my implementation is significantly faster. However, he reported problems with `MixSlices` which perhaps can be addressed with my shorter variant.

PRØST-256 still remains untouched. Further work could try to optimise that version as well. It should also be possible to apply the approach taken to other encryption algorithms, especially those which have an operation similar to `MixSlices`.

ROLE OF THE STUDENT

Peter Schwabe, my supervisor, was involved with the design of PRØST. The designers of PRØST provided a reference implementation which I re-implemented and optimised in ARM assembly. I also wrote additional software, like for the heuristic and the SAT transformation. Peter Schwabe provided valuable feedback along the way.

References

1. Andreeva, E., Bilgin, B., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., and Yasuda, K. APE: authenticated permutation-based encryption for lightweight cryptography. Cryptology ePrint Archive, Report 2013/791. 2013. address: **http://eprint.iacr.org/2013/791**.
2. Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Tischhauser, E., and Yasuda, K. Parallelizable and authenticated online ciphers. Cryptology ePrint Archive, Report 2013/790. 2013. address: **http://eprint.iacr.org/2013/790**.
3. ARM Limited. Arm1176jzf-s technical reference manual. Revision: r0p7. address: **http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/index.html**.
4. Bellare, M., Rogaway, P., and Wagner, D. A conventional authenticated-encryption mode. 2003. Address: **http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/eax/eax-spec.pdf**.
5. D. J. Bernstein and T. Lange, eds. Supercop. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Address: **http://bench.cr.yp.to/supercop.html**.
6. Boyar, J., Matthews, P., and Peralta, R. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26(2), 2013: 280–312.
7. CAESAR: competition for authenticated encryption: security, applicability, and robustness. Address: **http://competitions.cr.yp.to/caesar.html**.
8. Dobraunig, C., Eichlseder, M., and Mendel, F. Related-key forgeries for Prøst-OTR. Cryptology ePrint Archive, Report 2015/091. 2015. address: **http://eprint.iacr.org/2015/091**.
9. Fuhs, C., and Schneider-Kamp, P. Synthesizing shortest linear straight-line programs over GF(2) using SAT. *Proc. SAT'10*, 71–84.
10. Kavun, E. B., Lauridsen, M. M., Leander, G., Rechberger, C., Schwabe, P., and Yalçın, T. Prøst v1.1. 21st June 2014. address: **http://competitions.cr.yp.to/round1/proestv11.pdf**.
11. Krawczyk, H. The order of encryption and authentication for protecting communications (or: how secure is ssl?) *Advances in Cryptology – CRYPTO 2001*. 2001, 310–331.
12. Le Berre, D., and Parrain, A. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7, 2010: 59–64.
13. McGrew, D. A., and Viega, J. The galois/counter mode of operation (GCM). address: **http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf**.
14. Minematsu, K. Parallelizable rate-1 authenticated encryption from pseudorandom functions. Cryptology ePrint Archive, Report 2013/628. 2013. address: **http://eprint.iacr.org/2013/628**.
15. Papadimitriou, C. H., and Yannakakis, M. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3), 1991: 425–440.
16. Rijneveld, J. Implementing Prøst on the Cortex A8 using internal parallelisation. 2015-01. Address: **https://joostrijneveld.nl/papers/20150104_proest_cortexa8.pdf**.
17. Rogaway, P. Authenticated-encryption with associated-data. *Proceedings of the 9th ACM conference on Computer and communications security*. 2002, 98–107.