Bachelor thesis
Computer Science

Radboud University

Implementing CAESAR candidate Prøst on ARM11

Author:
Thom Wiggers
thom@thomwiggers.nl

First supervisor/assessor:
dr. Peter Schwabe
peter@cryptojedi.org

Second assessor:
dr. Lejla Batina
lejla@cs.ru.nl

10th April 2015

Abstract

PRØST is a contestant in the CAESAR competition for Authenticated Encryption. This thesis shows how PRØST was optimised for the ARM11 microprocessor architecture. By implementing PRØST in assembly, a performance gain of 28% to 48% was achieved. We also present a new implementation of `MixSlices`, one of the sub-operations in PRØST's permute function. This new implementation has 33% fewer arithmetic operations than the original version.

Contents

Chapter 1

Introduction

Authenticated encryption (AE) schemes use symmetric keys to encrypt data providing not only confidentiality but also integrity and authenticity [7]. Using these schemes avoids having to combine authentication and traditional, confidentiality-only, encryption, something that has often led to vulnerabilities. Some encryption functions even fail when combined with message authentication codes in certain ways, where that failure might even not be immediately obvious [19].

A variant of AE are *authenticated encryption with associated data* (AEAD) schemes [31]. 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 and (2) are suitable for widespread adoption" [13].

Optimised implementations on various platforms help show that an algorithm is suitable for deployment across a wide range of platforms. While the first things one might think of when considering uses of cryptography might be centred around a PC running the `amd64` architecture, secure algorithms are perhaps even more widely used and needed in embedded platforms, smart cards, and mobile devices.

In this thesis, I will show how I implemented the encryption algorithm PRØST [18] on the ARM11 platform. The ARM11 range of microprocessors powers many devices, from older smartphones to game consoles such as the Nintendo 3DS [6, 16]. These 32-bit processors implement the ARMv6 instruction set. While this architecture has since been replaced by the ARMv7 instruction set, there still are billions of ARM11 microchips deployed. Over the third quarter of 2014 ARM reported that still 3% of the 1.1 billion ARM chips shipped in that quarter were ARM11, while in 2010 and 2011 ARM reported shipment of over half a billion ARM11 microcontrollers per year [5, 3, 4].

In this thesis my goal was to implement PRØST in such a way that it would run as fast as possible, of course still protecting against timing attacks.

In Chapter 2, I will first briefly introduce and explain PRØST and describe some characteristics of the implementation platform. This will provide the building blocks for Chapter 3, in which I will discuss how I optimised PRØST for ARM and found a more efficient way of computing one of the components of the

Prøst permutation function, `MixSlices`.

The resulting implementation can be found via https://thomwiggers.nl/proest.

Acknowledgements I would first like to thank Peter Schwabe for his help and guidance while writing this thesis and for lending me his Raspberry Pi, on which the implementation of Prøst was written. I'd also like to thank Bernard van Gastel for giving me access to "the biggest machine on campus", and the Prøst authors for providing their reference implementation. Finally, I would like to express my gratitude to everyone who listened when I was discussing this project and offered various small but sometimes very useful hints.

Chapter 2

Preliminaries

2.1 Prøst

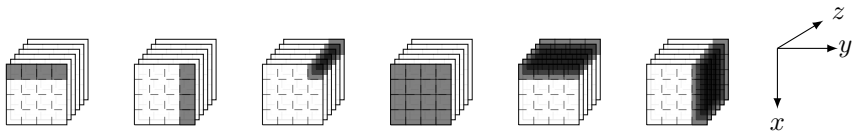
In this section I will briefly summarise PRØST's permutation as described in [17]. I will be describing the PRØST-128 version, which provides 128 bits of security.

PRØST's main operation is the PRØST-permutation. This operation can be combined in various ways to come up with the various modes of operation, such as COPA [2], OTR[1] [28] and APE [1]. My optimisations focused on the permutation, as it is the most expensive operation.

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 register. PRØST's authors adopted the nomenclature of KECCAK [11] and call these registers *lanes*. In this work I will use *register* to refer to a CPU register, and *lane* to refer to a 16-bit $s_{x,y}$ of the state. The terms *row*, *column*, *slice*, *plane* and *sheet* for the other parts of the state are described in Figure 2.1 which was adopted from [17].



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

Figure 2.1: Nomenclature for state parts [17]

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

$$R_i(x) = (\texttt{AddConstants}_i \circ \texttt{ShiftPlanes}_i \circ \texttt{MixSlices} \circ \texttt{SubRows})(x).$$

[1] PRØST-OTR was recently shown to be vulnerable to a related-key forgery attack by Dobraunig, Eichlseder and Mendel [14].

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.

2.1.1 SubRows

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

$$\begin{aligned} a' &= c \oplus (a \wedge b), \\ b' &= d \oplus (b \wedge c), \\ c' &= a \oplus (a' \wedge b'), \\ d' &= b \oplus (b' \wedge c'). \end{aligned} \tag{2.1}$$

2.1.2 MixSlices

The `MixSlices` operation mixes the slices of the state by xoring lanes according to a matrix M. The result of applying `MixSlices` to a state $s = (s)_{x,y}$ will result in a state $s' = (s')_{x,y}$ as in (2.2).

$$\begin{aligned} s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\ s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\ s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\ s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\ s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\ s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\ s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\ s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\ s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\ s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\ s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\ s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\ s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\ s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\ s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\ s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3} \end{aligned} \tag{2.2}$$

2.1.3 ShiftPlanes$_i$

The operation `ShiftPlanes`$_i$ rotates each of the lanes in a plane of the state by a a given amount. Each row is 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.

2.1.4 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, $0 \leq j < 16$ enumerating the lanes, constant c_1 is applied to even j and c_2 is applied to odd j. Before being applied, the constants are rotated left by the round number i and by j.

Thus, `AddConstants`$_i$ applied to state $s = (s)_{x,y}$ in round i will result in a state $s' = (s')_{x,y}$ like (2.3).

$$\begin{pmatrix} s'_{0,0} \\ s'_{0,1} \\ s'_{0,2} \\ s'_{0,3} \\ s'_{1,0} \\ \vdots \\ s'_{3,3} \end{pmatrix} = \begin{pmatrix} s_{0,0} \oplus (c_1 \lll i \lll 0) \\ s_{0,1} \oplus (c_2 \lll i \lll 1) \\ s_{0,2} \oplus (c_1 \lll i \lll 2) \\ s_{0,3} \oplus (c_2 \lll i \lll 3) \\ s_{1,0} \oplus (c_1 \lll i \lll 4) \\ \vdots \\ s_{3,3} \oplus (c_2 \lll i \lll 15) \end{pmatrix} \tag{2.3}$$

2.2 ARM11

The ARM11 architecture is the only implementation of the ARMv6 instruction set. Four major variants have been released in the ARM11 family: the ARM1136, ARM1156, ARM1176 and the ARM11MPcore variants [6]. All are available with an optional floating-point unit. The technical characteristics, including cycle timings, are described in the ARM11 reference manuals [23, 25, 24, 27, 26, 22]. The instruction set is documented in the ARMv7 architecture reference manual[2] [21]. In this section I will briefly set out the most important characteristics that are relevant to the rest of this work.

2.2.1 Registers

ARM11 processors have a 32-bit instruction set. They provide sixteen 32-bit registers to the programmer, although only fourteen are freely usable: one register is used as the stack pointer and another is used as the program counter.

Registers can be considered as value on the display of an electronic calculator. The CPU can use the values stored in registers immediately for arithmetic operations, writing the result back to a register. In the ARM instruction set, it can write the result back to any register, not just to one of the input registers.

Because there are only fourteen usable registers, a programmer will need to carefully manage these registers. Running out of registers means he or she needs to store values ("spill") to memory. This can be compared to writing the value shown on a calculator display onto a piece of paper: it takes a bit of time and the next time you want to use that value, you will need to look it back up and

[2] ARM don't publish a separate ARMv6 manual anymore except for the ARMv6-M range.

```
x1 = mem16[address_a]
x1 += 10
x2 = mem16[address_b]
x2 += 10
x3 = mem16[address_c]
x3 += 10
```

(a) Latencies slow execution down

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x1 += 10
x2 += 10
x3 += 10
```

(b) Hiding load latencies

Program 2.1: Equivalent programs, but different execution times

type it back into the calculator. Doing this impacts performance, as extra cycles need to be spent for the memory operations involved.

2.2.2 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 can drastically reduce execution time. In Program 2.1, two equivalent programs are shown. They retrieve three sixteen-bit values x_1, x_2, x_3 from memory starting at `address`, and then add 10 to each of these values. However in 2.1a, each time when 10 is added to the just loaded value, the register is not available yet: loads have a three-cycle latency before their result is available. This means that before we execute `x1 += 10`, we are already on the fourth cycle! In 2.1b, we spend the cycles we need to wait for x_1 to become available by preloading x_2 and x_3. Cleverly scheduling instructions results in a much faster program: while the naive example costs twelve cycles, the example hiding latencies only costs six cycles.

2.2.3 Multiword load/stores

While individual registers have a size of 32 bits, the CPU and memory are connected through a 64-bit bus. The `ldrd` instruction allows us to load up to 64 bits from memory in the same amount of time it would take to load a single 32-bit value. The bits need to be consecutive in memory, and can only be loaded to two consecutive registers. We can similarly store two registers to 64 bits in memory using the `strd` instruction.

Similarly we can perform even wider loads with the instruction `ldm`. This instruction can load an arbitrary number of registers, from 1 to 14 registers, from memory in one compute cycle on all ARM11 implementations [23, 25, 24, 27, 26, 22].

These operations are more restricted though: while the registers need not be consecutively numbered, they are loaded to in order: the lowest numbered

register is loaded with the first 32 bits from memory, the second lowest numbered register is loaded with bits 33–64, and so on. These extra wide loads also come with an extra latency for the results to become available and lock registers for a number of cycles. The related store operation, with similar restrictions, is `stm`.

The extra wide loads and stores are further limited by not being able to manipulate the address register by adding an arbitrary amount, these instructions only support optionally incrementing it by the width of the load. Regular load/store operations are more flexible and allow for an offset to be added to the value of the register. The result is then used as the address for memory access and can optionally be written back to the register.

2.2.4 Free shifts and rotations

The ARMv6 architecture provides instructions that allow to rotate or shift registers by an arbitrary amount, spending 1 computation cycle.

In addition, however, all arithmetic instructions support having the second input value rotated or shifted an arbitrary distance. These rotations or shifts are essentially free; they only require that the register which is to be rotated is available one cycle earlier than a non-modified input would need to be. This is because the shifter requires the input value to be available one stage earlier in the pipeline.

2.2.5 Cycle counter

As described in [32], the ARM11 CPUs have a cycle counter, which can only be accessed from kernel mode. Bernstein published source code for a Linux kernel module which exposes the cycle counter through a device file [8]. The SUPERCOP [10] benchmarking suite for cryptographic software supports using this device file. SUPERCOP was used for all final benchmarks. For development purposes I took the cycle measurement code from SUPERCOP and put it in my own wrapper.

A more conveniently packaged version of this cycle counter can be found via `https://thomwiggers.nl/proest/`.

2.3 Qhasm

As one experienced with programming assembly might have already noticed from program 2.1, I have not used pure ARM assembly. I have instead used a preprocessor called qhasm. Qhasm is prototype software developed by Bernstein [9]. It has a slightly friendlier syntax than regular assembly, which resembles `C`. More significantly though, qhasm allows a programmer to not worry about register allocation: it features an allocator which can keep track of which registers are used to store what variable. The programmer still needs to take care of not using more variables at a time than there exist registers: qhasm will not automatically spill variables to memory to free registers.

Otherwise qhasm instructions are a one-to-one mapping to assembly instructions. Anything appearing after a `#` is considered a comment and ignored.

I have made a couple of small modifications to qhasm to better support the target platform and fix some bugs. The version I used to compile my

implementation of PRØST can be found via https://thomwiggers.nl/proest.

Chapter 3

Optimising Prøst

In this chapter I will describe how I optimised the different functions and how I finally arrived at my implementation on ARM.

First I am going to explain how I treated every component of the PRØST permutation function. Then I will explain how these were combined to save additional cycles. I will treat them in the order in which they are applied to the state.

3.1 SubRows

The `SubRows` operation is an S-box permutation that can also be described as the linear operation described earlier in (2.1). It applies that operation to every row in the PRØST state s.

This operation was unrolled to get rid of the cycles that would otherwise be spent on jumps and loop bookkeeping (such as incrementing indexes and comparisons). The biggest gain however was obtained by a clever loading strategy as detailed in the next subsection.

3.1.1 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 free shifts previously described in Subsection 2.2.4.

It is possible to shift a register before it is applied as an input. This means to access the lane stored in the upper 16 bits of a register, we can shift it to the right by 16 bits. If we do not shift a register because we want to use the lane stored in the lower 16 bits of the register, we need to take care that the upper 16 bits of the result will contain some part of the lane that was still stored in the upper 16 bits. To get rid of this garbage, 16-bit stores (`strh`) can be used because they do not consider the upper 16 bits of a register and thus store the result modulo 2^{16}.

```
# a = s_{0,0}, b = s_{0,1}
# c = s_{0,3}
# a' = c ^ (a & b)
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
newa  = a_and_b & (a_and_b >>> 16)
newa ^= c_and_d
# only write back the lower 16 bits
mem16[address_of_s] = newa
```

Program 3.1: An example part of `SubRows` using two lanes in one register.

Using this technique will also allow to apply the multiword loads from Section 2.2.3, since they would also load two lanes per register loaded. If we wanted to use these loads, but not have two lanes per register, we would need to do additional operations to extract half of the lanes to their own registers.

In Program 3.1 a part of `SubRows` is shown where $a' = c \oplus (a \wedge b)$ is calculated. The two lanes, one register approach is applied to load lanes a and b into register `a_and_b`. To calculate $a \wedge b$, `a_and_b` is ANDed with itself rotated by 16 bits. This means $a \wedge b$ will be in the lower 16 bits of the register. Then, this result is XORed with `c_and_d` to complete the computation. Finally the result in the lower 16 bits, which now contain $c \oplus (a \wedge b)$, is written back to memory. The upper 16 bits, now containing $d \oplus (b \wedge a)$, are discarded: they are unfortunately not of any use.

3.2 MixSlices

`MixSlices` is by far the most expensive operation in PRØST's permutation. It is a matrix multiplication that can be represented by equation (2.2). This system has 72 XOR operations.

XOR 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 $s_{1,0} \oplus s_{3,0} \oplus s_{3,3}$ (3.1).

$$\begin{aligned} s'_{0,0} &= s_{0,0} \oplus \mathbf{s_{1,0}} \oplus s_{1,3} \oplus s_{2,2} \oplus \mathbf{s_{3,0}} \oplus s_{3,2} \oplus \mathbf{s_{3,3}} \\ s'_{0,1} &= s_{0,1} \oplus \mathbf{s_{1,0}} \oplus s_{2,3} \oplus \mathbf{s_{3,0}} \oplus \mathbf{s_{3,3}} \end{aligned} \tag{3.1}$$

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`?

3.2.1 Optimisation problem

We can also represent a function such as (2.2) 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`, because 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* [15, 12].

Unfortunately, finding the shortest linear program (SLP) is known to be NP-hard. Boyar, Matthews and Peralta additionally show that SLP is MAX-SNP-complete [12]. MAXSNP is a class of optimisation problems that can be approximated with some bounded error [29]. In other words, finding the shortest version of `MixColumns` is going to be very computationally expensive.

3.2.2 Trying to find the shortest program

Fuhs and Schneider-Kamp show in [15] that it is possible to transform the SLP problem to a different kind of problem in MAXSNP: the boolean satisfiability problem (SAT). SAT, the problem of finding a certain valuation that makes a boolean formula true, is NP-complete. However, solving SAT problems is something that scientists have gotten rather good at, as illustrated by various competitions [33].

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 a $m \times n$ matrix A, where every row represents one of the outputs and every column one of the inputs. $A_{x,y} = 1$ iff in output x, input y is used. They then define matrix B as a $k \times n$ matrix, where $b_{x,y} = 1$ iff in line x input variable y is used. A matrix C sized $k \times k$ is defined where $c_{x,y} = 1$ iff intermediate result y is reused in line x of the program. Finally, they define a matrix f to map intermediate results to outputs. f is also encoded as a matrix.

The decision problem is then 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.

The logic encoding Fuhs and Schneider-Kamp give is still in first-order logic. This needed to be transformed into something SAT solvers could understand. To achieve this, I developed a Java program that allows to input programs as a matrix A and try to solve the SLP-SAT problem for a certain length k. SAT4j [20] was used to transform the problem from predicate logic to a SAT problem, which was then solved also using SAT4j. The design of this program is further described in Appendix A.1.

Unfortunately, the smallest k proved to be out of reach with this implementation of the SLP-SAT problem. The smallest k is expected to be somewhere in the upper 40s, based on our results discussed in Section 3.2.3. The size of the constraints is $\mathcal{O}(n \cdot k^2)$ [15], but it appeared that the addition of new constraints to the SAT solver got more expensive faster than that. The biggest k feasible on the fastest machine available[1] was only 26. This machine was unable to provide an answer even after running for over two weeks. This might be because showing a problem is unsatisfiable is much harder than showing it is satisfiable.

[1] This machine had 3 Terabytes of RAM and 120 Intel Xeon E7-4870 v2 cores running at 2.30GHz. Most of SAT4j is however single-threaded.

3.2.3 Approximating the shortest program

Boyar et al. give a heuristic which allows to approximate the shortest SLP problem. The basic idea of this heuristic, described in [12], can be described as follows: define matrix S in which we will store previously produced functions. S has n columns. $s_{x,y} = 1$ iff the yth input variable is a part of the function defined by row x. We also need a matrix A containing the program, similar to matrix A in the SLP-SAT problem described above.

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 consider a distance function that for a given row in A determines how many combinations of rows in S need to at least be made to arrive at the row in M.

The program then will generate 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.

I implemented the above in Python 3 and its implementation is discussed in Appendix A.2. It was, after running for four days on a 24-core machine, able to find a much shorter implementation of `MixSlices` using only 48 XORs. This function can be found in A.3.

It proved infeasible to verify this result using the SAT-SLP program from the previous section, since encoding a problem where $k = 48$ was too expensive.

3.2.4 Implementing the shorter program

In the naive implementation of `MixSlices`, intermediate variables have a fairly short life span. This means that a lot of registers remain free for us to keep more of other things in registers, such as input values and results.

In these shorter straight-line programs, however, intermediate variables have a considerably longer life. After all, we are able to get rid of work we were doing multiple times by reusing those intermediate results. The side effect of having a shorter program is thus a potential increase in the number of loads and stores.

To minimise this effect, I tried to reschedule the instructions of the generated program so that intermediate values are generated as close to where they are first used as possible.

The ARM architecture fortunately proved to have enough registers, if we used the stack pointer as described in Section 3.5, so that the problem of increased register pressure did not negate our gain of 24 XORs too much.

3.3 ShiftPlanes

As described in Section 2.1.3 `ShiftPlanes` rotates each of the lanes in each plane by a specified constant. In even and odd rounds, these constants are different.

Unfortunately, rotating a 16-bit lane on a 32-bit architecture is a bit of a hassle. A normal `ror` rotation would not give us the correct result, because it would rotate the lower bits into the upper 16 bits of the register. It would also shift zeros (or whatever was in the upper half) into the lower half of the register.

Rotation of a 16-bit value can be done by adding the register to itself, shifted by 16 bits, and then rotating the whole register. This is shown in Program 3.2.

```
a = mem16[address]
a |= (a << 16)
a >>>= x
```

Program 3.2: Rotating a 16-bit value by x

When implementing SHA-3 candidates on ARM, Schwabe, Yang and Yang were able to hide many rotations by making use of the case that arithmetic instructions that look like

$$a = (b \ggg n_1) \oplus (c \ggg n_2)$$

you can instead compute

$$a = b \oplus (c \ggg (n_2 - n_1)).$$

This last instruction can use the free rotations that ARM supports, since n_1 and n_2 are both constants at compile time. You still need to rotate a by n_1 later, but that can often be cancelled out or hidden in other instructions [32].

The operations in PRØST do not immediately look like $a = (b \ggg n_1) \oplus (c \ggg n_2)$. When considering `ShiftPlanes` and `AddConstants` together, however, one sees that the added rotated constant in `AddConstants` resembles the $c \ggg n_2$ part. The rotation in `ShiftPlanes` then forms the $b \ggg 2$ part. We could thus combine these two to not perform the rotations in `ShiftPlanes`.

If we adopt this approach, however, we will need to get rid of the implicit rotation by n_1 that will still be left in the results of these merged rotations. We would need to get rid of those before they get distributed over many other variables.

That means the implicit rotations need to be eliminated the next time `SubRows` is done. This however would cause problems with our "two lanes in one register"-approach from Section 3.1.1. This approach thus could not be used in this implementation of PRØST. It perhaps would be useful in the PRØST-256, which has 32-bit lanes. Those can be more easily rotated.

3.4 AddConstants

`AddConstants` is the final operation in the PRØST permutation function. As described in Equation (2.3), it adds one of two rotated constant to each lane.

These constants c_1 and c_2 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 inline in instructions.

3.5 Obtaining an extra register

A single extra register can have a significant effect on performance, because it allows to keep more intermediate values in registers.

The stack pointer points to a section of memory where values can be stored. Our functions also get such a pointer, pointing to the start of the memory where the PRØST state is stored. If we allocate extra room after the PRØST state, we can use this pointer as a memory reference point to replace the stack pointer. We simply store the stack pointer and any values we would have stored on the stack, in the space behind the PRØST state.

An alternative approach would have been to put the entire PRØST state on stack so we would be able to use the register that would otherwise be holding the pointer to the state. However, this would have cost us more loads and stores.

The extra register was especially useful in the short implementation of `MixSlices`.

In case of interrupts, the stack is normally used to store the program state for when it is resumed. ARM CPUs however use a banked copy of the stack pointer to do this [21]. This allows us to use it without breaking the operating system.

3.6 Inlining the Prøst operations

Calling a function comes with a bit of overhead: the return address is put on the stack, variables of the calling function need to be stored somewhere safely and when the function is done all this needs to be restored. We can reduce this overhead by putting the operations consecutively in the same function.

Having the operations in the same function also enables us to do some nice things because we can keep intermediate values in memory between operations. This saves us quite a few loads and stores. Operations previously also had to retrieve results that were spilled to memory and then put those back into the PRØST state. In an unrolled PRØST, later calls can just retrieve the temporary value from memory.

It also helps us avoid most of the rotations we had to do in `AddConstants`, because we can use the results from `ShiftPlanes` instead of loading new values. Because these intermediate results only contain one lane per register we can use the free rotation in the instructions also for these values.

Finally, the unrolling allows to hide latencies better. One can start retrieving data needed for the next function and then fill up the load latency with the final processing of the result of the previous function. This is especially important with the long result latencies introduced by multi-word loads.

Chapter 4

Results and comparison

4.1 Benchmark results

All benchmark results were obtained by using the SUPERCOP [10] 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 at the time of writing. It was tweaked to not attempt to benchmark using options that are irrelevant on the ARM11, such as 64-bit modes and PowerPC and x86-only optimisations and tunings. `distcc` was used to offload compilation to a more powerful computer, an Intel x64 machine set up to cross-compile for ARMv6. The version of `gcc` used was `4.9.3 20141224 (prerelease)`. The cycle counter previously described in Section 2.2.5 was loaded to facilitate benchmarking.

The benchmark results can be found in Tables 4.1, 4.2 and 4.3. The reported figure is the "number of cycles used by a typical cryptographic operation" as reported by SUPERCOP. Also included in each table is the compiler flags used to get the reported figures.

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

Implementation	**Median cycle count**
Reference Implementation (`C`)[a]	2,976,123
My implementation (ARM qhasm)[a]	1,900,274
Improvement	36%

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

Table 4.1: Benchmark results for PRØST-APE

Implementation	**Median cycle count**
Reference Implementation (`C`)[a]	2,402,577
My implementation (ARM qhasm)[a]	1,648,407
Improvement	28%

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

Table 4.2: Benchmark results for PRØST-COPA

Implementation	**Median cycle count**
Reference Implementation (`C`)[a]	1,569,582
My implementation (ARM qhasm)[b]	848,100
Improvement	46%

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

Table 4.3: Benchmark results for PRØST-OTR

4.2 Comparison

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 [30]. 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. Most of my optimisations could also be backported to a more efficient C implementation, as well.

Bibliography

[1] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha and Kan Yasuda. *APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography*. Cryptology ePrint Archive, Report 2013/791. http://eprint.iacr.org/2013/791. 2013 (cit. on p. 4).

[2] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser and Kan Yasuda. *Parallelizable and Authenticated Online Ciphers*. Cryptology ePrint Archive, Report 2013/790. http://eprint.iacr.org/2013/790. 2013 (cit. on p. 4).

[3] *ARM holdings PLC reports results for the third quarter and nine months ended 30 September 2010*. URL: http://arm.com/about/newsroom/arm-holdings-plc-reports-results-for-the-third-quarter-and-nine-months-ended-30-september-2010.php (cit. on p. 2).

[4] *ARM holdings PLC reports results for the third quarter and nine months ended 30 September 2011*. URL: http://arm.com/about/newsroom/arm-holdings-plc-reports-results-for-the-third-quarter-and-nine-months-ended-30-september-2011.php (cit. on p. 2).

[5] *ARM holdings PLC reports results for the third quarter and nine months ended 30 September 2014*. URL: http://arm.com/about/newsroom/arm-holdings-plc-reports-results-for-the-third-quarter-and-nine-months-ended-30-september-2014.php (cit. on p. 2).

[6] *ARM11 Processor Family*. URL: http://www.arm.com/products/processors/classic/arm11/index.php (cit. on pp. 2, 6).

[7] Mihir Bellare, Phillip Rogaway and David Wagner. 'A conventional authenticated-encryption mode'. 2003. URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/eax/eax-spec.pdf (cit. on p. 2).

[8] D. J. Bernstein. *Counting cycles on the Nokia N810*. 24th Dec. 2014. URL: http://bench.cr.yp.to/cpucycles/n810.html (cit. on p. 8).

[9] D. J. Bernstein. *qhasm: tools to help write high-speed software*. 24th Dec. 2014. URL: http://cr.yp.to/qhasm.html (cit. on p. 8).

[10] D. J. Bernstein and Tanja Lange, eds. *SUPERCOP*. eBACS: ECRYPT Benchmarking of Cryptographic Systems. 5th Apr. 2015. URL: http://bench.cr.yp.to/supercop.html (cit. on pp. 8, 16).

[11] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. 'Keccak specifications'. In: *Submission to NIST* 42 (2008) (cit. on p. 4).

[12] Joan Boyar, Philip Matthews and René Peralta. 'Logic Minimization Techniques with Applications to Cryptology'. English. In: *Journal of Cryptology* 26.2 (2013), pp. 280–312. ISSN: 0933-2790. DOI: 10.1007/s00145-012-9124-7. URL: http://dx.doi.org/10.1007/s00145-012-9124-7 (cit. on pp. 12, 13, 21, 23).

[13] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. URL: http://competitions.cr.yp.to/caesar.html (cit. on p. 2).

[14] Christoph Dobraunig, Maria Eichlseder and Florian Mendel. *Related-Key Forgeries for Prøst-OTR*. Cryptology ePrint Archive, Report 2015/091. http://eprint.iacr.org/2015/091. 2015 (cit. on p. 4).

[15] Carsten Fuhs and Peter Schneider-Kamp. 'Synthesizing Shortest Linear Straight-line Programs over GF(2) Using SAT'. In: *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*. SAT'10. Edinburgh, UK: Springer-Verlag, 2010, pp. 71–84. ISBN: 3-642-14185-4, 978-3-642-14185-0. DOI: 10.1007/978-3-642-14186-7_8. URL: http://dx.doi.org/10.1007/978-3-642-14186-7_8 (cit. on pp. 12, 21).

[16] *Hardware – 3DBrew*. 23rd Nov. 2014. URL: http://3dbrew.org/wiki/Hardware (cit. on p. 2).

[17] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe and Tolga Yalçın. *Prø st v1.1*. 21st June 2014. URL: http://competitions.cr.yp.to/round1/proestv11.pdf (cit. on p. 4).

[18] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe and Tolga Yalçın. *Prøst v1*. 16th Mar. 2014. URL: http://competitions.cr.yp.to/round1/proestv1.pdf (cit. on p. 2).

[19] Hugo Krawczyk. 'The order of encryption and authentication for protecting communications (or: How secure is SSL?)' In: *Advances in Cryptology—CRYPTO 2001*. Springer. 2001, pp. 310–331. URL: http://www.iacr.org/archive/crypto2001/21390309.pdf (cit. on p. 2).

[20] Daniel Le Berre and Anne Parrain. 'The Sat4j library, release 2.2 system description'. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64 (cit. on p. 12).

[21] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. 24th July 2012 (cit. on pp. 6, 15).

[22] ARM Limited. *ARM11 MPCore Processor Technical Reference Manual*. Revision: r2p0. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/index.html (cit. on pp. 6, 7).

[23] ARM Limited. *ARM1136JF-S and ARM1136J-S Technical Reference Manual*. Revision: r1p5. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/index.html (cit. on pp. 6, 7).

[24] ARM Limited. *ARM1156T2F-S Technical Reference Manual*. Revision: r0p4. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0290g/index.html (cit. on pp. 6, 7).

[25] ARM Limited. *ARM1156T2-S Technical Reference Manual*. Revision: r0p4. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0338g/index.html (cit. on pp. 6, 7).

[26] ARM Limited. *ARM1176JZF-S Technical Reference Manual*. Revision: r0p7. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/index.html (cit. on pp. 6, 7).

[27] ARM Limited. *ARM1176JZ-S Technical Reference Manual*. Revision: r0p7. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0333h/index.html (cit. on pp. 6, 7).

[28] Kazuhiko Minematsu. *Parallelizable Rate-1 Authenticated Encryption from Pseudorandom Functions*. Cryptology ePrint Archive, Report 2013/628. http://eprint.iacr.org/2013/628. 2013 (cit. on p. 4).

[29] Christos H. Papadimitriou and Mihalis Yannakakis. 'Optimization, approximation, and complexity classes'. In: *Journal of Computer and System Sciences* 43.3 (1991), pp. 425–440. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(91)90023-X. URL: http://www.sciencedirect.com/science/article/pii/002200009190023X (cit. on p. 12).

[30] Joost Rijneveld. 'Implementing Prøst on the Cortex A8 using Internal Parallelisation'. 2015-01. URL: https://joostrijneveld.nl/papers/20150104_proest_cortexa8.pdf (cit. on p. 17).

[31] Phillip Rogaway. 'Authenticated-Encryption with Associated-Data'. In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 98–107 (cit. on p. 2).

[32] Peter Schwabe, Bo-Yin Yang and Shang-Yi Yang. 'SHA-3 on ARM11 processors'. In: *Progress in Cryptology – AFRICACRYPT 2012*. Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. Springer-Verlag Berlin Heidelberg, 2012, pp. 324–341. URL: http://cryptojedi.org/papers/#sha3arm (cit. on pp. 8, 14).

[33] *The international SAT Competitions web page*. 3rd Apr. 2015. URL: http://www.satcompetition.org/ (cit. on p. 12).

Appendix A

Appendix

A.1 Finding the shortest Straight-Line Program with SAT4j

My implementation of Fuhs and Schneider-Kamp's method of finding the shortest linear straight-line program by transforming it to a boolean satisfiability problem [15] can be found via `https://thomwiggers.nl/proest/`.

I implemented the program in Java. The program takes as input a target number of lines k and a matrix A that represents the original program. It will try to prove that a straight-line program equivalent to A of length k exist.

As described in Section A.1, Fuhs and Schneider-Kamp define a number of first-order-logic formulae that will only only be satisfiable by valid programs.

SAT4j only accepts input in Conjunctive Normal Form. It however has a class, `GateTranslator`, that allows to transform simple statements such as $\wedge, \vee, \Rightarrow, \Leftrightarrow$ and add them to a SAT4j Solver instance.

I created a hierarchy of classes that represent statements such as `And`, `Or`, `Xor`. These classes know how to add themselves to a SAT4j Solver through `GateTranslator`.

I then built the formulae given by Fuhs and Schneider-Kamp from these classes. To not use too much RAM, I implemented them as classes similar to the simpler statements and only construct the full statement with my representation of propositional logic when adding them to the Solver. It is important to allow this syntax tree to get garbage collected as soon as possible, because otherwise it will eat hundreds of Gigabytes of RAM.

A.2 Approximating the shortest Straight-Line Program

Boyar, Matthews and Peralta define in [12] a heuristic for finding an approximation of the shortest linear program. I have already briefly described this heuristic in Section 3.2.3. My implementation of this heuristic in Python 3 can also be found via `https://thomwiggers.nl/proest`.

The input of the program should be provided by configuring the lists `M` and S to the values of A and the initial value of S.

The meat of the heuristic is selecting the next function to append to the matrix of functions S. Determining the new distance of S to A with every new candidate row is very expensive, because it needs to combine all rows in S with each other until it has computed every row in A.

The distance function is implemented as a recursive algorithm. To improve performance when calculating the distance to a row, I have implemented a cutoff: if we have tried more steps than what we know is the current distance, the distance function stops trying that branch and returns a high value. Additionally, parallel programming was used to calculate the distances for multiple candidate rows at the same time.

A.3 Shorter MixSlices

Equation (A.1) is the implementation of `MixSlices` that was found using the heuristic by Boyar et al. [12].

In the following equation, $x_0, \cdots, x_{15}$ represent the inputs to `MixSlices`, $s_{0,0}, s_{0,1}, \cdots, s_{3,3}$. The outputs $s'_{0,0}, s'_{0,1}, \cdots, s'_{3,3}$ are represented by $y_0, \cdots, y_{15}$. $t_1, \cdots, t_{34}$ represent intermediate variables. The lines have been manually sorted so that intermediates are defined as close to their first use as possible. Nevertheless, it is easy to see that some variables, e.g. t_{14}, have a very long life span and are used both near the start and near the end.

$$
\begin{array}{lcl@{\qquad}lcl}
t_1 &=& x_0 \oplus x_{14} & y_{11} &=& t_5 \oplus t_{24} \\
t_3 &=& t_1 \oplus x_{14} & t_{25} &=& x_0 \oplus t_{13} \\
t_5 &=& x_9 \oplus x_5 & t_{15} &=& x_5 \oplus x_{15} \\
y_{14} &=& t_3 \oplus t_5 & y_5 &=& t_{15} \oplus t_{25} \\
t_{12} &=& x_{10} \oplus t_3 & t_{17} &=& x_3 \oplus x_9 \\
t_2 &=& x_{12} \oplus x_8 & t_{26} &=& x_{12} \oplus t_{26} \\
t_4 &=& t_2 \oplus x_2 & t_{18} &=& x_4 \oplus x_7 \\
y_2 &=& t_4 \oplus t_5 & y_9 &=& t_{18} \oplus t_{26} \\
t_{14} &=& x_6 \oplus t_4 & t_{27} &=& t_2 \oplus t_{22} \\
t_{10} &=& x_1 \oplus x_{11} & t_{16} &=& x_6 \oplus x_{10} \\
t_{19} &=& x_4 \oplus t_{10} & y_6 &=& t_{16} \oplus t_{27} \\
t_{11} &=& x_{12} \oplus x_{15} & t_{28} &=& x_7 \oplus t_{11} \\
y_1 &=& t_{19} \oplus t_{11} & y_0 &=& t_{12} \oplus t_{28} \\
t_{21} &=& x_3 \oplus t_{12} & t_{30} &=& x_8 \oplus t_8 \\
t_{13} &=& x_8 \oplus x_{11} & t_7 &=& x_0 \oplus x_3 \\
y_4 &=& t_{13} \oplus t_{21} & y_{13} &=& t_7 \oplus t_{30} \\
t_6 &=& x_1 \oplus x_{13} & t_{31} &=& x_{13} \oplus t_{17} \\
t_{22} &=& x_{10} \oplus t_6 & y_3 &=& t_{16} \oplus t_{31} \\
y_{10} &=& t_1 \oplus t_{22} & t_{32} &=& x_1 \oplus t_{16} \\
t_9 &=& x_2 \oplus x_{14} & y_{15} &=& t_{15} \oplus t_{32} \\
t_{23} &=& x_9 \oplus t_9 & t_{33} &=& x_{15} \oplus t_{14} \\
t_8 &=& x_7 \oplus x_{13} & y_8 &=& t_{18} \oplus t_{33} \\
y_7 &=& t_8 \oplus t_{23} & t_{34} &=& x_{11} \oplus t_{14} \\
t_{24} &=& t_{10} \oplus t_{23} & y_{12} &=& t_7 \oplus t_{34}
\end{array}
\tag{A.1}
$$