Implementing Prøst on the Cortex A8 using Internal Parallelisation

Joost Rijneveld, s4048911
joostrijneveld@gmail.com

January 2015

Abstract

The CAESAR competition is looking for ciphers that are able to provide authentication and offer advantages over the currently popular AES-GCM. The Prøst permutation provides the basis for a family of AEAD cipher schemes that have been submitted as candidates to this competition. In this paper, an implementation of the Prøst permutation on the Cortex A8 is discussed. The main focus of this implementation is on internal parallelisation, using the NEON vector instruction set included in the ARMv7 architecture. At just under 200 cycles per byte, the implementation is hindered by a suboptimal MixSlices function that has proven difficult to vectorise efficiently.

1 Introduction

In cryptography, block ciphers are encryption algorithms that operate on a block of data of a fixed length, using the same key for both encryption and decryption. This is typically referred to as a 'symmetric key'. Block ciphers are among the fundamental building blocks of larger cryptographic algorithms and protocols, and are used in very diverse applications. In order to provide different features and properties, such a cipher is used in a specific mode of operation. These modes make it possible to apply a block cipher to a bigger amount of data.

Traditionally, block ciphers are used in modes of operation that provide either confidentiality or authenticity of data[1]. Some modes provide an efficient way of concealing the contents of messages, while other modes provide strong guarantees with respect to the authenticity of the data. Modes such as cipher block chaining (CBC), counter (CTR), cipher feedback (CFB) and output feedback (OFB) fall in the former category, while authentication is typically done by using message authentication codes. For various needs and purposes, however, people have sought to establish both of these properties at the same time in a secure manner. This category of modes of operation is called Authenticated Encryption with Associated Data (AEAD). In Section 2, we will examine this in more detail.

[1]Note that block ciphers are used for a variety of other purposes as well (such as random number generation), but those uses are not immediately relevant in this particular context.

In the past, cryptographic research has greatly benefited from so-called "crypto competitions". The AES competition, won by the Rijndael algorithm [1], is the best known example of this. In order to stimulate research towards AEAD schemes, the CAESAR competition has been set up. CAESAR will be discussed in more detail in Section 3. The Prøst [2] algorithm is one of the candidates submitted to CAESAR. Efficiently implementing Prøst on the widely used ARM Cortex A8 microarchitecture is the topic of this paper.

In Section 4, we will go into the basic workings of internal parallelisation and vector instructions. In particular, the vector instruction set provided as part of the ARMv7 architecture, ARM NEON, will be discussed.

As mentioned above, the main contribution of this research consists of an implementation of the Prøst algorithm using the vector instructions provided by the ARMv7 architecture. Section 5 contains a description of the Prøst permutation, and in Section 6, the implementation is discussed. In these sections, the distinct operations that make up the permutation are discussed one by one.

2 AEAD

As mentioned above, Authenticated Encryption with Associated Data (AEAD) is a class of cryptographic algorithms or modes of operations that is able to provide both authentication and encryption of data. While the demand for a combination of authentication and encryption is not particularly new, solutions traditionally relied on a mixture of a scheme for authentication and a different scheme for encryption. A typical solution is achieved by combining cipher block chaining with message authentication codes (CBC-MAC). Combining different schemes to obtain multiple security goals has been done successfully in the past, but has invariably come with inconvenient trade-offs. The user often pays a large penalty in terms of computational efficiency or memory footprint, but may also run risks with respect to the security of the system. The mix of different schemes has proven to be quite delicate to use, and may break down altogether when it is implemented in the wrong conditions or situation. For example, the CBC-MAC construction is only secure when used with fixed-length messages [3]. Further fixes have been suggested that overcome this particular nuisance (such as CBC-MAC-ELB, where the last block is encrypted with a secondary key), but one can imagine that this adds further complications and restrictions. The aim of AEAD is to fundamentally merge both encryption and authentication into a single mode of operation, rather than combine them in a later stage. This approach greatly reduces complexity [4] and also allows for a much more efficiency-focused design (such as an increased level of parallelism [5]).

2.1 CBC-MAC

One of the problems with using a block cipher in confidentiality modes (such as CBC) and applying a message authentication code (MAC) for authentication lies in the fact that the individual blocks are not authenticated, but only the chain of blocks as a whole. This is not a problem when the chains are small, but when considering large files or even streaming data it is very tempting to perform operations on part of the

decrypted data before waiting for the MAC validation. In one of his blog posts [6], Adam Langley illustrates this using a pipe from `gpg` to `tar`, where files are actually being written to the file system before the authenticity of the ciphertext has been verified. Langley does note that an AEAD scheme is not a plug-and-play solution here, but suggests that it may be able to form the basis for a system that provides authenticity for streaming data.

With regards to the use of CBC-MAC, SSL and TLS have seen a number of attacks over the past few years, often based on padding oracle attacks (most recently POODLE, the last variation of which also impacts a number of TLS 1.2 implementations [7]). In early 2013, the Lucky 13 attack exposed a timing side channel in verification of the MACs [8], only briefly after the BEAST attack against essentially all implementations of TLS with block ciphers in CBC mode (most prominently AES-CBC) in 2011 [9]. For both attacks, often-heard recommendations were to either switch to RC4 or use TLS 1.2 [10], as it has support for AES in an AEAD mode (which will be discussed in the next subsection). Of these options, switching to RC4 is the most widely chosen alternative to AES-CBC when using TLS. However, as AlFardan, Bernstein, Paterson, Poettering and Schuldt have shown [11], RC4 contains serious vulnerabilities that make it unsuitable for use in TLS. While patching was viable for BEAST and Lucky 13, patching RC4 is largely considered to be off the table [10].

2.2 AES-GCM

In 2004, McGrew and Viega proposed Galois/Counter Mode (GCM) [12] as a mode of operation that is able to provide authenticated encryption. Internally, GCM is still a combination of separate parts for encryption and authentication, mixing counter-mode with Galois authentication. However, GCM has considerably better performance properties than previous combined solutions, as the authentication is now highly parallelisable. In addition to authenticated encryption, GCM can be used in stand-alone mode for authentication (referred to as GMAC) when there is no need to preserve confidentiality of the data.

The most common method when constructing an AEAD scheme using GCM is by selecting AES as the block cipher, to form AES-GCM. This is especially attractive on platforms where AES has been implemented in hardware, such as is made available through Intel's AES-NI instructions. Benchmarks by Gueron [13], presented as baselines for CAESAR (see Section 3) show that it can perform at a rate of a minimum of 1 cycle per byte. On the software side, however, the algorithm has proven to be difficult to implement securely while maintaining performance. In [14], Käsper and Schwabe present multiple optimised implementations of AES-GCM. When relying on a lookup-table based approach, they are able to achieve a speed of 10.68 cycles per byte. Note, however, that table lookups are often susceptible to cache-timing attacks [15]. To prevent this, Käsper and Schwabe also present a constant-time implementation that runs at 21.99 cyles per byte that does not have this vulnerability. This significant difference makes it attractive to opt for an implementation that does use table lookups, but is inherently vulnerable to side channel analysis.

In his talk at Real World Crypto 2013 [16], Adam Langley mentioned this as one of the reasons why he believes that AES-GCM should be replaced. He illustrates that many of the current implementations of AES-GCM's core function, GHASH, remind him

of AES implementations of a decade ago. Presenting it as a controversial suggestion, he goes on to argue that it would be preferable to pick a different AEAD that can be conveniently implemented in a secure fashion, instead. While AES-GCM is great on chips with hardware support, it is far from ideal on platforms that have to rely on software implementations.

In [17], Gueron and Krasnov demonstrate a vulnerability that was included in the AES-GCM implementation in the development repository of OpenSSL for a brief period of time. While the specifics of the vulnerability are not immediately relevant in the current context, they make an interesting observation with regards to the implementation of AES-GCM in OpenSSL. While the GCM mode allows for a high level of integration between the encryption and authentication parts (i.e. CTR and GHASH), the implementation in OpenSSL still keeps these quite separate, and treats them as distinct functions. This is intriguing, given the higher-level idea of close integration between authentication and encryption in AEAD schemes, but not necessarily surprising, as AES-GCM still has the look and feel of the 'traditional' AES block cipher in a slightly different role.

3 CAESAR

In the past, competitions in cryptographic research have provided a valuable stimulus to the field. Examples of this include the well-known AES competition, as well as the ECRYPT Stream Cipher competition (eSTREAM) [18], and the SHA-3 competition [19] that resulted in the standardisation of Keccak [20]. These competitions have shown that the combination of defining a common purpose and encouraging a wide range of members from the cryptographic community to submit their ideas is a very efficient and rewarding way of constructing a new direction for research to grow in. The open character of these competitions make them very valuable for the collective understanding of the specific field that is being explored.

CAESAR ("Competition for Authenticated Encryption: Security, Applicability, and Robustness") aims to join the ranks of the above-mentioned competitions, and has called for submissions that combine authentication and confidentiality. The competition is not an official NIST initiative, but has received wide recognition and positive acclaim, resulting in over fifty submissions for the first round. The main goal of the competition is to establish authenticated ciphers that provide advantages over AES-GCM (see Section 2.2) and are ready for adoption by the field [21].

The initial round of the CAESAR competition focusses on the security of the various algorithms, as well as the suitability for high-performance software implementations. Submissions for this round were accepted until March 15, 2014. Each submission specifies a set of algorithms, referred to as a 'family', that can contain different parametrisations of the same scheme, depending on security settings. With each submission, CAESAR required software reference implementations for the described (parametrisations of) the scheme. These are to be provided to support understanding of the precise details of the algorithm. Hardware implementations are to follow in the second round, in early 2015 [21].

Schemes that provide authentication and encryption can generally be subdivided into two categories; new algorithms designed for the purpose of use in an AEAD scheme,

and modes of operation for existing algorithms. The earlier discussed AES-GCM would fall in the second category. CAESAR accepts both types of submissions, and has received a wide variety across the entire spectrum [22]. A significant number of submissions provide a way of applying a new or existing mode of operation to AES in order to achieve authenticated encryption. Examples include AES-JAMBU [23], Julius [24] and AES-COPA [25]. Common arguments for these schemes include the level of scrutiny that has gone into establishing the security of AES over the years, as well as the performance of AES, especially when hardware implementations are concerned. These schemes greatly benefit from hardware support for an AES round, such as included in a wide range of modern Intel processors. On the other hand, the competition has also attracted a number of submissions proposing newly designed schemes. This set includes schemes where the block cipher and mode of operation are co-designed or entirely integrated, as well as schemes that propose a new block cipher or permutation using an existing mode of operation that allows it to be used for AEAD (such as COPA and APE). In these categories, we find schemes such as NORX [26] and Joltik [27]. Prøst [2] also belongs to the latter category, and will be discussed in detail in Section 5. Arguments for these schemes are often related to software performance, as this is an often discussed aspect of AES. When considering platforms that do not have support for hardware AES, some of these schemes could be able to beat AES-based ciphers. These schemes are frequently designed with cross-platform performance in mind, aiming at straightforward implementations and inherent robustness against side-channel attacks.

4 Vectorisation

Internal parallelism, or more specifically, vectorisation, is a way of computing the same operation for multiple different inputs at once. Programs that use this can be computationally more efficient by literally processing more data in the same time frame. This has been one of the major areas of improvement in modern CPUs. In particular, loop vectorisation (i.e. computing multiple iterations of the same loop in parallel) has lead to significant performance gains.

While vectorisation has been the subject of extensive research when it was first introduced in vector processors, the addition of SIMD ('single instruction multiple data') units [28] to general-purpose processors has sparked a new interest. These SIMD architectures provide very accessible instructions that enable programmers to use internal parallelism for the specific parts of an application where it is most relevant. This has been applied in a wide range of scenarios (including in graphic controllers in the XBOX and PlayStation) and some form of SIMD is now included in most common processor architectures [29].

While vectorisation can be powerful, it is not without constraints. Most notably, it requires data independence. Sequential operations that intuitively result from imperative programs often depend on the result of the previous operation as an input, and can thus not always be vectorised. For automatic vectorisation by compilers, this has been one of the main areas of research. With the introduction of SIMD operations, new approaches have been devised to deal with this. In particular, vectorisation of small blocks of instructions (rather than entire loops) can be sped up for SIMD architectures.

One might wonder why it is necessary or desirable to integrate the parallelism so deeply into the implementation rather than use external parallelisation to simply run multiple instances of the algorithm side by side. While internal parallelisation might provide more efficiency, a large motivation for this lies also in the fact that parallelising internally allows for a much more convenient API. This way, the throughput of the implementation does not depend on whether the user is able to supply it with sufficient data at the same time.

4.1 ARM NEON

In ARMv6, AMD included a number of SIMD instructions that operated on the existing 32-bit general purpose registers. These operations split the registers into bytes or double-byte blocks and were able to perform two or four operations in parallel.

ARM NEON is ARM's current implementation of a SIMD architecture. NEON is not merely an instruction set on the existing registers, but a coprocessor with its own dedicated register set. Described as a 'general-purpose SIMD engine', the NEON instructions are advertised as a way to efficiently process all sorts of digital media. While the ARMv7 architecture contains sixteen 32-bit registers, the NEON instructions operate on 64 or 128 bits of data grouped together in double-word and quad-word vector registers. These registers are provided by a dedicated register bank of 2048 bits that is viewed as a set of sixteen 128-bit registers (`q0` to `q15`) or thirty-two 64-bit registers (`d0` to `d31`), depending on which specific operation is used [30]. The different operations provided by NEON can operate on different sizes of operands, varying from 8-bit to 64-bit values for the Cortex A8. Depending on the relevance to the specific operation, these can be interpreted as signed or unsigned values.

Many of the same operations that one typically uses on regular registers are also available for vector registers. This includes regular data processing operations as well as memory access and moving data between registers. As mentioned above, instructions can operate on different data types. This functionality is specified by parametrising the different instruction calls with the types and sizes of the input and output data blocks. For example, in order to perform addition on 32 bit signed operands in two quad-word registers, one would call `VADD.S32 q0, q1, q2`. This would add four operands from `q1` to those in `q2`, and store the result in `q0`. Analogous, calling `VSHR.U8 d0, d0, #3` would shift each byte of the double-word `d0` to the right by 3 bits, and fill the individual bytes from the left with zeros.

The NEON instruction set also includes a number of instructions that are especially convenient when working with large vector registers, such as a set of predefined permutations. Calling `VREV32.8 q0, q0` would reverse the 8-bit sub-blocks that make up each of the blocks of 32 bits in `q0`, and two registers can be interleaved by calling `VZIP.16 d1, d2`. A particularly useful instruction in the context of Prøst is `VTRN`, performing transpositions between two registers as if they were a 2x2 matrix. This will be discussed in some detail in Section 5.1. As all these internal permutation instructions can be parametrised to operate on different block sizes, they can be combined to construct intricate operations. For example, Figure 1 shows how to perform a transposition on four vector registers representing a 4x4 matrix using just three parametrised `VTRN` instructions [31].

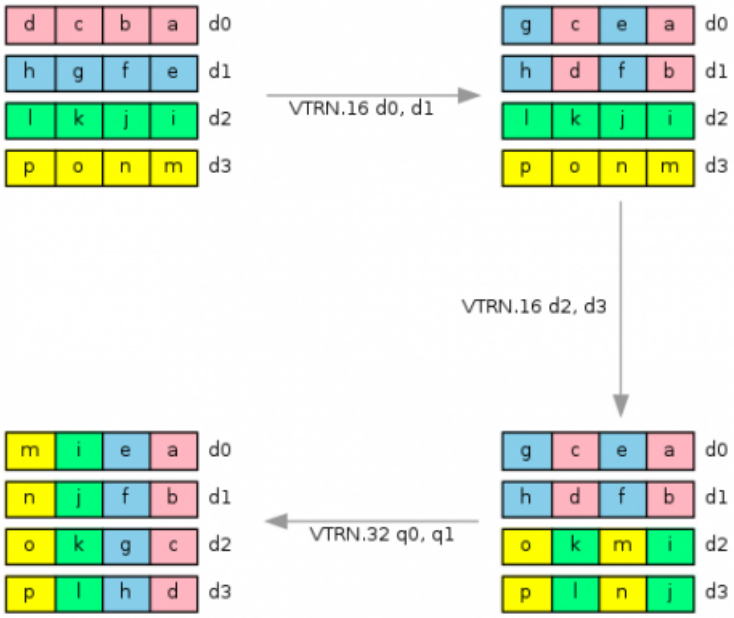

Figure 1: Transposing a 4x4 matrix [31]

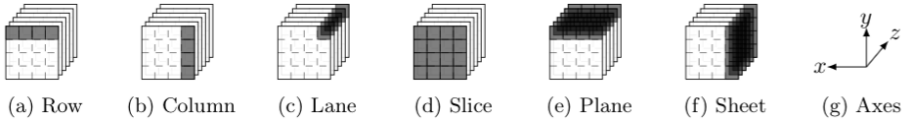

Figure 2: Terminology for parts of the state [2]

5 The Prøst permutation

This section provides an overview of the Prøst permutation, and outlines the operations that make up its construction. Prøst, as presented by Kavun, Lauridsen, Leander, Rechberger, Schwabe and Yalçın in [2], forms the basis for a family of AEAD schemes that have been submitted as candidates for the CAESAR competition (see Section 3). The permutation is used to instantiate the existing COPA [5], OTR [32] and APE [33] modes of operation in order to construct AE schemes.

In [2], the authors specify two different variants of Prøst: Prøst-128 and Prøst-256, and define n to be 128 and 256, respectively.

Prøst works on a state of $2n$ bits (so 256 and 512 bits, respectively), arranged as a rectangular cuboid. See Figure 2 for the naming convention that is used to refer to parts of the state. For both instances of Prøst, the width of a row and the height of a column is 4. This implies that the depth of a lane is 16 bits for Prøst-128 and 32 bits for Prøst-256 (from here on referred to as d). For the implementation discussed in Section 6, only Prøst-128 is relevant, but the definition provided in this section is applicable to both configurations.

The operations that make up Prøst follow the familiar structure typically found in iterated block ciphers. The permutation consists of a number of rounds (sixteen for Prøst-128 and eighteen for Prøst-256) that themselves consist of the operations `SubRows`, `MixSlices`, `ShiftPlanes` and `AddConstants`. Here the terminology described in Figure 2 can be helpful when interpreting the operation names.

It should be noted that, as two of the operations included in Prøst depend on the round number, the round number is zero-based (i.e. the round number for the first round is 0).

5.1 SubRows

Prøst includes a small 4-bit S-box. As the name of the operation indicates, the substitution is applied to each 4-bit row of the state. The S-box is listed in Figure 3, below.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	0	4	8	F	1	5	E	9	2	7	A	C	B	D	6	3

Figure 3: S-box used in SubRows

The authors stress that the S-box is designed to be efficient to implement and to allow bit-slicing. This is achieved by providing an equivalent and compact formulation of the performed substitution. For (a, b, c, d) the four input bits, the output bits (e, f, g, h) can be defined as follows (note that $\oplus$ represents bitwise XOR and $\odot$ represents bitwise AND):

$$
\begin{aligned}
e &= c \oplus (a \odot b) \\
f &= d \oplus (b \odot c) \\
g &= a \oplus (e \odot f) \\
h &= b \oplus (f \odot g)
\end{aligned}
$$

5.2 MixSlices

The MixSlices operation is effectively a matrix multiplication. Each slice is placed in a column vector by enumerating the bits from the top-left to the bottom-right, one row at a time, and multiplied with the square matrix M as listed in Figure 4

The authors clarify that each of the multiplications with a non-zero value is roughly equivalent to one XOR operation. In fact, the amount of XOR operations increases linearly with the amount of ones in the matrix (as each row consists of at least one bit). The chosen matrix contains a (locally[2]) minimal amount of bits while still satisfying the security requirements. The Hamming weight of the matrix is 88.

[2]The authors note that there is no guarantee that this solution is optimal, as this is generally unknown and exhaustive search is not feasible.

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

Figure 4: Matrix used in MixSlices

5.3 ShiftPlanes

The ShiftPlanes operation is a bitwise rotation to the right. The name is perhaps a bit confusing as to which part of the state is in fact rotated, but this becomes more apparent when examining the reference implementation. Each of the lanes included in a plane is shifted to the right, which is, in light of Figure 2, equivalent to a rotation in the positive z-direction.

The exact number of bits by which the lanes are rotated depends on the specific plane as well as the parity of the round. For Prøst-128, this is a shift of 0, 2, 4 or 6 bits for even rounds and 0, 1, 8 or 9 bits for odd rounds. For Prøst-256, the lanes are shifted 0, 4, 12 or 26 bits in even rounds and 1, 24, 26 or 31 bits in odd rounds.

The authors note that the shifts were chosen in such a way that they would provide maximal diffusion in as few rounds as possible, as well as result in convenient implementations by minimising the total sum (excluding multiples of 8, which are often free or very cheap to perform).

5.4 AddConstants

The AddConstants operation performs a XOR operation between a shifted constant and each lane. This operation is slightly more complex than the typical round-dependent addition, as the constant also depends on the specific lane. Depending on the parity of the lane, constant $c_1 =$ `0x75817b9d` or $c_2 =$ `0xb2c5fef0` is used; c_1 is used for even lanes and c_2 is used for odd lanes. For Prøst-128, these constants are truncated to $c_1 =$ `0x7581` and $c_2 =$ `0xb2c5` to accommodate for the 16 bits lane size.

For each lane, the constant is rotated to the left by the sum of the round number and the lane number, where the lanes are ordered in the same way as was done for the vector in the MixSlices operation (Section 5.2). Like the rounds, the lanes are counted zero-based.

The constants used in Prøst are directly derived from the first 64 decimals of π; the Prøst design document includes C code that can be used to recreate the constants. The rotation of the constants is chosen such that adding the same constant is avoided.

6 Implementing Prøst

In the previous section, we examined the design of Prøst. In this section, we will go over details from the implementation of Prøst on the Cortex A8. The specific platform that was used for this implementation is the "BeagleBone Black" by Texas Instruments. This implementation heavily relies on the ARM NEON instructions, as discussed in Section 4.1.

The qhasm language [34] (discussed in more detail in the next subsection) was used to create the implementation described here. The produced qhasm source files can be found on GitHub at `https://github.com/joostrijneveld/proest-cortex-a8`. It was created by incrementally replacing functions from the Prøst reference implementation with qhasm implementations while maintaining unit test satisfaction, and then inlining, combining and unrolling the code to form an integral qhasm implementation.

As the Cortex A8 offers sixteen dedicated quad-word registers, it was possible to keep the entire state in memory throughout the permutation. The state is typically stored in registers `x` and `y` (effectively storing each lane in an individual double-word register).

While the initial intent of this research was to construct a fully optimised implementation of Prøst using internal vectorisation, time constraints have created the need to limit this. While the low-hanging fruit has been taken into account, there is still some potential for further optimisations that could be explored. Where relevant, concrete suggestions are described in the following subsections. In general, the operations are currently still quite distinct. Rescheduling of the different instructions and combining them across operation boundaries may also prove to be valuable.

6.1 qhasm

Qhasm [34] is a language that can be translated directly into assembly instructions, at one line per instruction. This is achieved by using machine description files that specify the mapping from qhasm syntax to actual architecture-specific assembly instructions. By writing code in qhasm rather than directly in the ARM assembly, the program generally becomes much more readable and easier to write and edit. The fact that the qhasm syntax is largely consistent across various platforms also adds to this ease of use, as well as providing re-usability. Additionally, qhasm includes a register allocator. This is a convenient abstraction that prevents having to worry about manually keeping track of register usage.

Because of the way it uses machine descriptions to translate the code into assembly, qhasm is highly extensible. While there was already an elaborate description available for the ARMv7 architecture, it has proven to be convenient to add a number of additional operations for specific NEON assembly instructions.

6.2 SubRows

The SubRows operation is perhaps the most straightforward of all parts of the permutation, as the same operation is to be performed on all rows, allowing for convenient internal parallelisation. The substitution is defined as a sequence of XOR and AND operations in Section 5.1. In contrast to other operations, however, this operation operates on rows rather than lanes. As the state representation is a sequence of lanes, it makes sense to modify this before performing the substitution, and restore this afterwards. Luckily, the required transformation is precisely the transposition described in Figure 4. This results in a total of two sets of three VTRN instructions.

Performing the substitution is now a matter of performing the AND and OR operations on the four double-word registers that contain a bit-sliced representation of the rows. As is suggested by the formalisation in the Prøst paper, it makes sense to create a temporary backup of the first two rows, as the original values are required after their new values have been computed. These bits are conveniently grouped together in the first quad-word register. Further optimisations could include writing to different state registers for alternating rounds in order to prevent this initial copying, as well as exploring a different state representation or incorporating the transposition in other instructions.

6.3 MixSlices

The MixSlices operation has proven difficult to efficiently parallelise internally. This operation remains very much a sequence of XOR operations. It is implemented by shifting a temporarily used masked register along the state registers x and y to retrieve the required lanes, XORing them into this register. As mentioned in Section 5.2, each one-bit in the matrix roughly results in one XOR operation. This means that the complete multiplication requires 72 XOR operations. Because the planes are spread over four double-word registers, lanes that are a multiple of four apart can be combined without requiring individual shifting. The results are stored in temporary quad-word registers r and s that are copied back into x and y in the end.

Minor optimisations (such as allocating different state registers for the results and using those throughout the rest of the round) have been implemented, but this operation could benefit from a more fundamental overhaul. A more intricate optimisation could include a graph-theoretical solution that finds a more optimal combination of XOR operations that allows for the re-use of intermediate results. Alternatively, this could be used to find an ordering that is able to combine the shifts of the temporary register to use it more efficiently.

The qhasm code that implements this operation consists of sixteen very similar sets of instructions, each again consisting of several similar instructions internally as well. It proved convenient to generate this code using a short Python script that takes the code from the reference implementation as input, as this specifies the specific lane numbers that are to be XORed together. This allows for quick testing and editing of the resulting implementation. Specific optimisations have been applied manually after generating the basic building blocks.

6.4 ShiftPlanes

As described in Section 5.3, the ShiftPlanes operation consists of a number of rotations by a constant amount, only influenced by the parity of the round number. As all the rounds are unrolled, the implementation contains two different blocks of instructions that implement the different alternatives for ShiftPlanes, interleaved through the rounds. As all the lanes are grouped into a double-word register per plane, each double-word register needs to be rotated by a single constant amount. Adding qhasm instructions for shifts on 16 bits has improved the performance here. Since the NEON instruction set does not include atomic rotation instructions, bitwise rotation remains a combination of shifts.

6.5 AddConstants

Adding constants is generally not a difficult operation. In Prøst, however, it is complicated somewhat by requiring different rotations for each lane and each round (see Section 5.4). Fortunately, we are able to pre-compute the rotations beforehand. Recall that the original constants are $c_1 = \texttt{0x7581}$ and $c_2 = \texttt{0xb2c5}$, truncated to the lane size of 16 bits. For this implementation, the sixteen lane-dependent rotations have been precomputed, requiring 256 bits of storage for the constants instead of the original 32 bits. The precomputed constants (see C_x and C_y, below) fit precisely in two quad-word registers. The constants are arranged in such a way that the positions of the 16-bit constants match the positions of the lanes with the correct parity in the state registers.

$$C_x = \texttt{0x 7581 658b d605 962d 5817 58b6 605d 62d9}$$
$$C_y = \texttt{0x 8175 8b65 05d6 2d96 1758 b658 5d60 d962}$$

During each round, a vectorised rotation is performed on each of these quad-word registers, effectively rotating eight constants at once. Adding the constants is also highly parallelisable, as this amounts to xoring each of the constants registers into the respective state register. As was the case with the ShiftPlanes operation, a more efficient way of rotating could improve the performance here.

6.6 Performance

While this implementation of Prøst has been optimised to some extent, the sections above mention a few open suggestions that remain. With this in mind, we examine the performance of the described implementation.

In Table 1, the cycle count for each operation as well as for the complete permutation is listed. In order to establish a baseline, the 'empty function' was also benchmarked. This shows how many cycles are lost in overhead. Albeit necessary, it is interesting to see what happens after removing this constant overhead from the measurements, especially when examining benchmarks of a single execution of the component operations (SubRows, MixSlices, ShiftPlanes and AddConstants). In the table below, the reduced measurements are listed, and the measurements including overhead are included in parentheses.

	1st quartile	median	3rd quartile
Empty function	792	792	806
SubRows	12 (804)	12 (804)	12 (818)
MixSlices	272 (1064)	272 (1064)	260 (1066)
ShiftPlanes	8 (800)	10 (802)	10 (816)
AddConstants	8 (800)	10 (802)	8 (814)
Prøst-128	5510 (6302)	**5510 (6302)**	5498 (6304)

Table 1: Cycle count for the individual operations

From this, we can compute the amount of cycles per byte. As Prøst-128 operates on a state of 256 bits, the cycle count per byte can be computed by dividing the median by 32. This results in $6302/32 \approx 196.94$ cycles per byte.

7 Conclusion

By means of the implementation presented in this paper, we have seen that it is viable to construct an efficient implementation of the Prøst permutation, and that the permutation can greatly benefit from internal parallelisation. However, we have seen that in particular the MixSlices step is non-trivial to implement in a way that it benefits from vectorisation. As seen in table 1, this function is by and large the most significant contributor to the total cycle count. This is no particular surprise, as it is by far the most abstract function to translate into assembly instructions, and the current implementation is still very sequential in nature. Further research and optimisation towards implementing the MixSlices operation in a vectorised way could prove to be worthwhile.

References

[1] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf. 2

[2] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. Prøst v1.1. *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/proestv11.pdf. 2, 5, 7

[3] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, 2000. https://cseweb.ucsd.edu/~mihir/papers/cbc.pdf. 2

[4] Charanjit S. Jutla. Encryption modes with almost free message integrity. In *Advances in Cryptology – EUROCRYPT 2001*, pages 529–544. Springer, 2001. http://researcher.ibm.com/researcher/files/us-csjutla/iapmproof_2012.ps. 2

[5] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In *Advances in Cryptology – ASIACRYPT 2013*, pages 424–443. Springer, 2013. http://homes.esat.kuleuven.be/~eandreev/COPA.pdf. 2, 7

[6] Adam Langley. Encrypting streams. 2014. https://www.imperialviolet.org/2014/06/27/streamingencryption.html [accessed 04-12-2014]. 3

[7] Adam Langley. The POODLE bites again. 2014. https://www.imperialviolet.org/2014/12/08/poodleagain.html [accessed 11-12-2014]. 3

[8] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013. http://www.isg.rhul.ac.uk/tls/TLStiming.pdf. 3

[9] Thai Duong and Juliano Rizzo. Here come the ⊕ ninjas. *Unpublished manuscript*, 2011. urlhttp://www.hpcc.ecs.soton.ac.uk/ dan/talks/bullrun/Beast.pdf. 3

[10] Kenny Paterson. Authenticated encryption in TLS. In *DIAC 2013*, 2013. http://2013.diac.cr.yp.to/slides/paterson.pdf. 3

[11] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob CN Schuldt. On the security of RC4 in TLS. In *USENIX Security*, pages 305–320, 2013. http://www.isg.rhul.ac.uk/tls/RC4biases.pdf. 3

[12] David McGrew and John Viega. The Galois/Counter mode of operation (GCM). *Submission to NIST*, 2004. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf. 3

[13] Shay Gueron. AES-GCM software performance on the current high end CPUs as a performance baseline for caesar competition. In *DIAC 2013*, 2013. http://2013.diac.cr.yp.to/slides/gueron.pdf. 3

[14] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag Berlin Heidelberg, 2009. Document ID: cc3a43763e7c5016ddc9cfd5d06f8218, http://cryptojedi.org/papers/#aesbs. 3

[15] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 3

[16] Adam Langley. HTTPS: things that bit us, things we fixed and things that are waiting in the grass. 2013. https://www.imperialviolet.org/2013/01/13/rwc03.html [accessed 04-12-2014]. 3

[17] Shay Gueron and Vlad Krasnov. The fragility of AES-GCM authentication algorithm. In *Information Technology: New Generations (ITNG), 2014 11th International Conference on*, pages 333–337. IEEE, 2014. https://eprint.iacr.org/2013/157.pdf. 4

[18] eSTREAM: the ECRYPT stream cipher project, 2008. http://www.ecrypt.eu.org/stream/ [accessed 11-12-2014]. 4

[19] NIST. SHA-3 competition, 2012. http://csrc.nist.gov/groups/ST/hash/sha-3/ [accessed 11-12-2014]. 4

[20] Guido Bertoni, Joan Daemen, Michaël Peeters, and GV Assche. The Keccak reference. *Submission to NIST (Round 3)*, 13, 2011. http://keccak.noekeon.org/Keccak-submission-3.pdf. 4

[21] Daniel J. Bernstein. Cryptographic competitions. 2013. http://competitions.cr.yp.to/caesar-call.html [accessed 21-09-2014]. 4

[22] Farzaneh Abed, Christian Forler, and Stefan Lucks. Classification of the CAESAR candidates. *IACR Cryptology ePrint Archive*, 2014:792, 2014. http://eprint.iacr.org/2014/792. 5

[23] Hongjun Wu and Tao Huang. JAMBU lightweight authenticated encryption mode and AES-JAMBU (v1). *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/aesjambuv1.pdf. 5

[24] Lear Bahack. Julius: Secure mode of operation for authenticated encryption based on ECB and finite field multiplications. *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/juliusv10.pdf. 5

[25] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. AES-COPA v.1. *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/aescopav1.pdf. 5

[26] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. NORX v1. *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/norxv1.pdf. 5

[27] Jérémy Jean, Ivica Nikolic, and Thomas Peyrin. Joltik v1. *Submission to CAESAR*, 2014. http://competitions.cr.yp.to/round1/joltikv1.pdf. 5

[28] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. 5

[29] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *ACM SIGPLAN Notices*, volume 39, pages 82–93. ACM, 2004. http://researcher.watson.ibm.com/files/us-alexe/paper-eichen-pldi04.pdf. 5

[30] ARM Limited. Introducing NEON - Development Article. 2009. Document ID: DHT0002A http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/DHT0002A_introducing_neon.pdf. 6

[31] ARM Limited. Coding for NEON - Part 5: Rearranging vectors. *ARM Connected Community*, 2012. http://community.arm.com/groups/processors/blog/2012/03/13/coding-for-neon--part-5-rearranging-vectors [accessed 06-12-2014]. 6, 7

[32] Kazuhiko Minematsu. Parallelizable rate-1 authenticated encryption from pseudorandom functions. In *Advances in Cryptology – EUROCRYPT 2014*, pages 275–292. Springer, 2014. http://eprint.iacr.org/2013/628.pdf. 7

[33] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. Ape: Authenticated permutation-based encryption for lightweight cryptography. *IACR Cryptology ePrint Archive*, 2013:791, 2013. http://homes.esat.kuleuven.be/~eandreev/APE.pdf. 7

[34] Daniel J. Bernstein. qhasm: tools to help write high-speed software. 2005. http://cr.yp.to/qhasm.html [accessed 27-09-2014]. 10