Bachelor thesis
Computer Science

Radboud University

Implementing the authenticated cipher MORUS using NEON

Author:
Oussama Danba
s4435591

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

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

June 23, 2017

Abstract

Our work implements the authenticated cipher MORUS, a submission for the CAESAR competition, using ARM NEON. For the MORUS variant with a state of 640 bits we were able to achieve an implementation that is twice as fast as the reference implementation. For the MORUS variant with a state of 1280 bits we were able to achieve equivalent speed to our 640-bit implementation. Our implementation makes MORUS the fastest CAESAR submission on ARM when NEON is available.

Contents

1 Introduction **2**

2 Preliminaries **3**
2.1 The qhasm programming language 3
2.2 MORUS 4
2.2.1 State and State Update 4
2.2.2 Initialization 6
2.2.3 Processing Associated Data 7
2.2.4 Encryption and Decryption 7
2.2.5 Finalization and Verification 8
2.3 ARM Cortex-A8 9
2.3.1 NEON 10
2.3.2 Dual Issue 11
2.3.3 Cycle Counter 11

3 Related Work **13**

4 Implementation **14**
4.1 State Update 15
4.2 Processing Associated Data 18
4.3 Encryption and Decryption 21
4.4 Lower Bound Analysis 21

5 Results **23**

6 Conclusions **26**

A qhasm Extensions **29**

Chapter 1

Introduction

Authenticated encryption schemes are schemes that provide confidentiality, integrity, and authenticity simultaneously [3]. By using authenticated encryption it is no longer necessary to combine a traditional scheme that only provides confidentiality with a message authentication code (MAC) which provides integrity and authenticity. This is important because historically the combination has introduced problems [2, 10]. Additionally, some authenticated encryption schemes work in a single pass which prevents a MAC from being added afterwards [9, 18]. An extension of authenticated encryption is authenticated encryption with associated data. This extension allows the addition of data over which there is integrity and authenticity but no confidentiality.

The `CAESAR` (Competition for Authenticated Encryption: Security, Applicability, and Robustness) competition is a competition to select a set of authenticated ciphers that have been reviewed by the cryptographic community. The outcome is going to be a portfolio of ciphers so that an appropriate cipher can be used for every use case.

In our research we have implemented MORUS [8], which is one of the submissions, on an ARM Cortex-A8. MORUS has the property that it can be implemented efficiently using SIMD instructions which is what we did by using the "NEON" vector unit available on the ARM Cortex-A8. Our objective was to answer the question: "How many cycles per byte does MORUS take after vector optimizations using ARM NEON?".

In Chapter 2 we will introduce our implementation setup, MORUS, and the Cortex-A8. Chapter 3 is about other work done on MORUS and using NEON for cryptography. In Chapter 4 we will be discussing our implementation of MORUS and what optimizations we have done. Chapter 5 shows the results of our implementation. A short reflection of our work can be found in Chapter 6.

Chapter 2

Preliminaries

2.1 The qhasm programming language

Writing in pure assembly requires a lot of extra work such as manually tracking which registers are available and which of those are callee-saved registers (registers whose value must be preserved when returning from a function call). To reduce the amount of extra work we used the qhasm toolkit [4] to serve as a small abstraction layer above pure assembly. By using qhasm we gain flexibility without sacrificing the potential to write efficient software. When we write

```
4x s0_blk unsigned>>= 27
```

it becomes

```
vshr.u32 q1,q0,#27
```

after being processed by qhasm. Every time we reference `s0_blk` in qhasm it will be replaced with the appropriate register and qhasm will keep track of which registers are in use. Note that `s0_blk` is not always in the same register. qhasm can use different source and destination registers but will ensure `s0_blk` always points to the register that is currently holding the data.

Besides the easier register allocation, qhasm will also ensure that callee-saved registers are only used if the value in the register is guaranteed to be the same at the end of the qhasm code. This means that registers such as `r4` must first be written to the stack and restored at the end before qhasm will use it. If there are no free registers and the callee-saved registers are not stored, qhasm will fail to compile the qhasm code. Which registers are callee-saved registers can be found in the Procedure Call Standard for the ARM Architecture [13].

2.2 MORUS

MORUS is a family of authenticated ciphers that provides confidentiality, integrity and authenticity of data. Our research is based on version 2 of the MORUS cipher as it is currently the most recent version. MORUS relies on an internal state that is continuously updated using a state update function for its security. There are two possible sizes for the internal state and two possible key sizes. Any combination of those is allowed but the recommended parameters are a 640-bit state with a 128-bit key or a 1280-bit state with either a 128- or 256-bit key [8]. All of those parameter sets use a 128-bit nonce/IV and a 128-bit tag. As a result all configurations provide 128 bits of security against forgeries. The plaintext and associated data are allowed to be of arbitrary length as long as the length stays below 2^{64} bits.

For MORUS to encrypt and decrypt data it has to perform multiple steps. Every step of this process will be described in the upcoming sections.

2.2.1 State and State Update

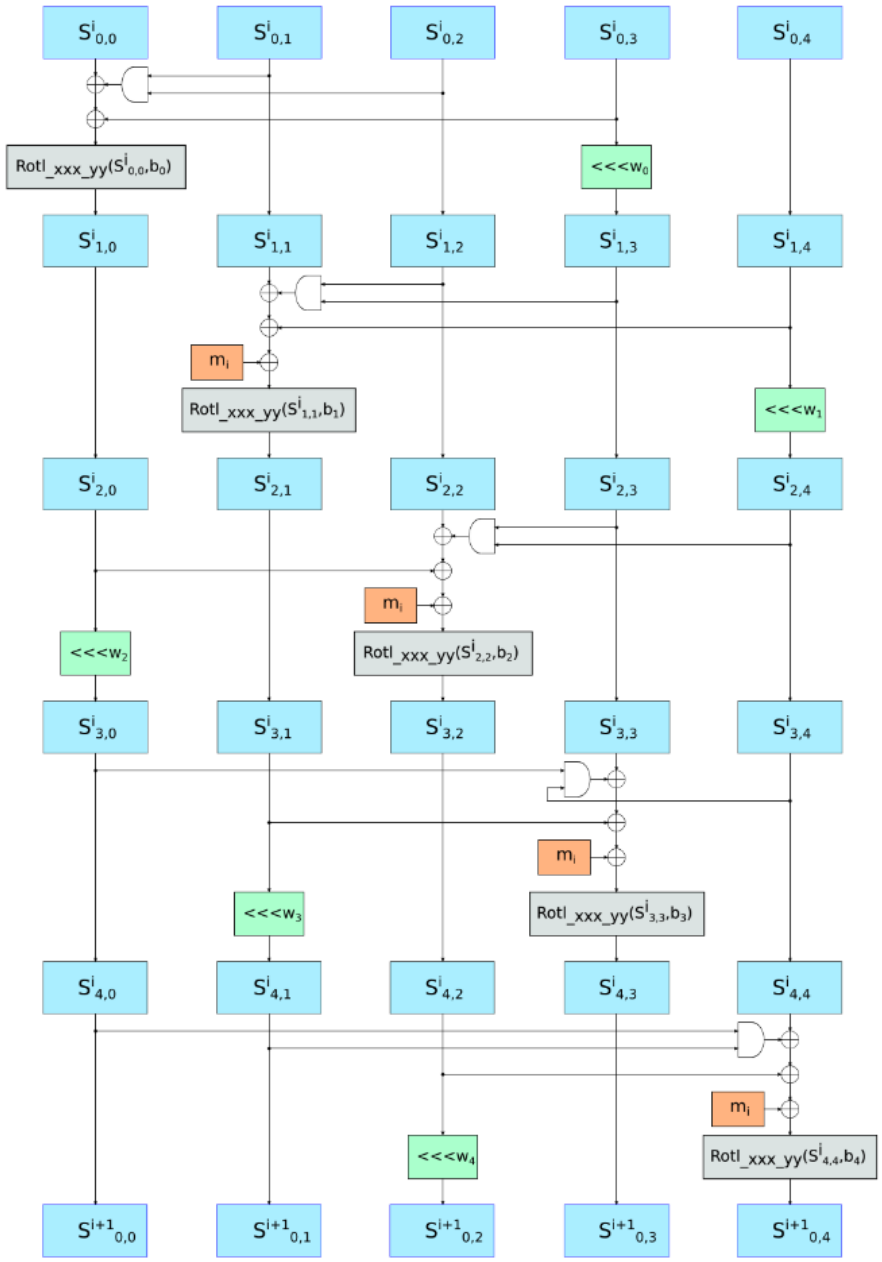
As mentioned before MORUS relies on its internal state for its security. The state is either 640- or 1280-bit so it can be split into five blocks of 128 or 256 bits which corresponds to the typical register sizes for SIMD. These fives blocks are typically denoted as S_0 through S_4.

The initial content of the state is decided by the IV and key. How it is exactly set up can be seen in Section 2.2.2. Once the state is initialized it can be updated by using a `StateUpdate` function. This function is essential to MORUS as every step in MORUS uses it to change the state once it has used the data. By using the `StateUpdate` function the chance of having an internal state collision in normal operation is sufficiently small to never be a problem. This is discussed in [8] under internal state collision. The continuous updating of the state is also what ensures integrity and authenticity. If any part is altered before the data is received, the receiver will get a different state at some point. Due to `StateUpdate` being called continuously this difference will cascade to all later states causing tag verification to fail. Since a change cascades all the way to the end no matter where this change happens, MORUS ensures that tag verification will fail. This allows MORUS to give integrity and authenticity over the associated data, ciphertext and tag.

The `StateUpdate` consist of five rounds where every round alters two out of five blocks. After these five rounds it should be impossible to tell what the internal state was before the `StateUpdate` function was called unless the message blocks are known. A schematic of how the `StateUpdate` works can be found in Figure 2.1. In every round a so called `Rotl_xxx_yy` is used. In MORUS-640 this is `Rotl_128_32` while in MORUS-1280 this is `Rotl_256_64`. This specifies that the block of size 128 or 256 should be split

up in four chunks of 32 or 64 each. Each of these chunks is then rotated to the left over a constant `b`. This `b` is different for every combination of size and round. In every round a $\texttt{<<<}w_n$ is also present. This specifies that this block is rotated to the left as a whole over a constant w_n. All constants can be found in [8]. m_i is the current message block; the meaning of this message block is dependent on the context of where the `StateUpdate` is called.

Figure 2.1: A schematic overview of the `StateUpdate` used in MORUS



2.2.2 Initialization

MORUS starts with state initialization and what it does is using the provided IV and key to set up a state. This state is then mixed by calling the `StateUpdate` function 16 times so that the initial state cannot be known without having the public IV and private key. The initialization for MORUS-640 and MORUS-1280 do the same except for what data is put initially in the state before the mixing happens.

In MORUS-640 block S_0 contains the 128-bit IV. Block S_1 contains the 128-bit key. Block S_2 contains a constant value which is 128 bits set to 1. Block S_3 and S_4 also contain constant values but those are based on the first 32 Fibonacci numbers modulo 256. S_3 contains the first 16 numbers while S_4 contains the other 16 numbers. The Fibonacci numbers were chosen to show that there was no bias in the way these constants were chosen.

MORUS-1280 puts the 128-bit IV concatenated with 128 zeros in S_0. In block S_1 the key is placed. However, MORUS-1280 allows both a 128-bit or a 256-bit key. To accommodate for this, the 128-bit key is concatenated with itself so that the 256-bit block can be filled. This characteristic is exactly the reason why MORUS-1280-128 does not have a separate optimized implementation in our research as this is the only difference from MORUS-1280-256. Block S_2 is filled with 256 bits set to 1. S_3 is similar to S_2 but is filled with 256 bits set to 0 instead. This time block S_4 is filled with the first 32 Fibonacci numbers modulo 256 since all 32 of them fit in the same block.

Once the initial data has been put into the state, the `StateUpdate` function is called 16 times with the message block set to 0 since there is no message. This ensures all initial values are properly mixed. Once the state updates are done, the initialization is finished by XORing block S_0 with the key. From here on the key is never needed again allowing it to be removed from memory.

Function 1 Initialization in MORUS-640

1: **procedure** INITIALIZATION(IV, K, S) ▷ S is 5 blocks of 128 bits
2: $S_0 \leftarrow IV$
3: $S_1 \leftarrow K$
4: $S_2 \leftarrow 1^{128}$
5: $S_3 \leftarrow 00||01||01||02||03||05||08||0d||15||22||37||59||90||e9||79||62$
6: $S_4 \leftarrow db||3d||18||55||6d||c2||2f||f1||20||11||31||42||73||b5||28||dd$
7: **for** `i in 0..15` **do**
8: STATEUPDATE($S, 0$) ▷ Message block set to 0
9: **end for**
10: $S_0 \leftarrow S_0 \oplus K$
11: **end procedure**

2.2.3 Processing Associated Data

Processing the associated data is the first thing that is done after initialization. While nothing is done to the associated data since there is no confidentiality over it, it is processed so that the state is updated. Updating the state gives integrity and authenticity over the associated data.

The `StateUpdate` is called for every 16 or 32 bytes of associated data, depending on the size of the state. If the last part of the associated data is not a full block, it is padded with 0 bits until it is its full size before going through the `StateUpdate`. Note that while the associated data is padded to update the state the associated data itself is not actually altered.

Function 2 Processing Associated Data in MORUS-640

1: **procedure** ProcessAD(ad, S) ▷ Assumes ad is a multiple of 16 bytes
2: StateUpdate(S, ad)
3: **end procedure** ▷ If ad was padded we continue without padding

2.2.4 Encryption and Decryption

After processing the associated data, the plaintext or ciphertext itself is processed. This is the first difference between what the sender and receiver do, albeit a small one.

Just like the associated data the plaintext or ciphertext is processed in blocks of 16 or 32 bytes. If the last block is not a full block it is again padded with 0 bits for only the duration of the encryption and decryption process and the subsequent `StateUpdate`.

The encryption process is: $C_i = P_i \oplus S_0 \oplus (S_1 <<< a) \oplus (S_2 \& S_3)$ where a is 96 in the case of MORUS-640 and 192 in the case of MORUS-1280. After encryption, `StateUpdate` is called with P_i (the padded plaintext) as message block. The decryption process is exactly the same except that C_i and P_i are swapped since we want to go back to the plaintext given the ciphertext. The `StateUpdate` is still called with P_i as message block during decryption otherwise the state would diverge from encryption.

Function 3 Encryption in MORUS-640

1: **procedure** Encryption(m, c, S)▷ Assumes m is a multiple of 16 bytes
2: $c \leftarrow m \oplus S_0 \oplus (S_1 <<< a) \oplus (S_2 \& S_3)$
3: StateUpdate(S, m)
4: **end procedure** ▷ If m was padded we only use the unpadded amount of bytes from c

Function 4 Decryption in MORUS-640

1: **procedure** DECRYPTION(m, c, S) ▷ Assumes tag is stripped from c
2: $m \leftarrow c \oplus S_0 \oplus (S_1 <<< a) \oplus (S_2 \& S_3)$
3: $m' \leftarrow m||0^{16-len(m)}$ ▷ m' is m padded
4: STATEUPDATE(S, m') ▷ m' is only used for StateUpdate
5: **end procedure**

2.2.5 Finalization and Verification

The last step in MORUS is finalization which means generating the tag. Tag generation is done in encryption as well as decryption.

Tag generation starts by doing some preparation. The first step is concatenating the length of the associated data with the length of the message so that they make a 128-bit number. For MORUS-1280, 128 zeros are concatenated as well to obtain a 256-bit number. This number is stored temporarily for later use. The second step is XORing block S_4 with block S_0 and storing the result in S_4. The third and final preparation step is calling the `StateUpdate` ten times with the previously stored value as the message block. By using the length of the message and associated data in the tag generation MORUS ensures that the message and associated data can not be extended without generating a different tag.

Once the preparation is done, the actual tag is generated. The generation is similar to encryption and is: $T' = S_0 \oplus (S_1 <<< a) \oplus (S_2 \& S_3)$ with a being 96 and 192 again. The actual tag is then the appropriate amount of least significant bits of T'. In the case of MORUS-640 this means all bits but in the case of MORUS-1280 this means half of the bits since the tag is 128-bit while the blocks have 256 bits.

Verification of the tag is only done during decryption and simply means checking that the given tag is the same as the generated tag. It is important to mention that a MORUS implementation should not return the generated tag as output if verification fails. If this is done MORUS is known to be vulnerable [8]. Simply returning whether tag verification succeeds or fails is sufficient.

Function 5 Tag Generation in MORUS-640

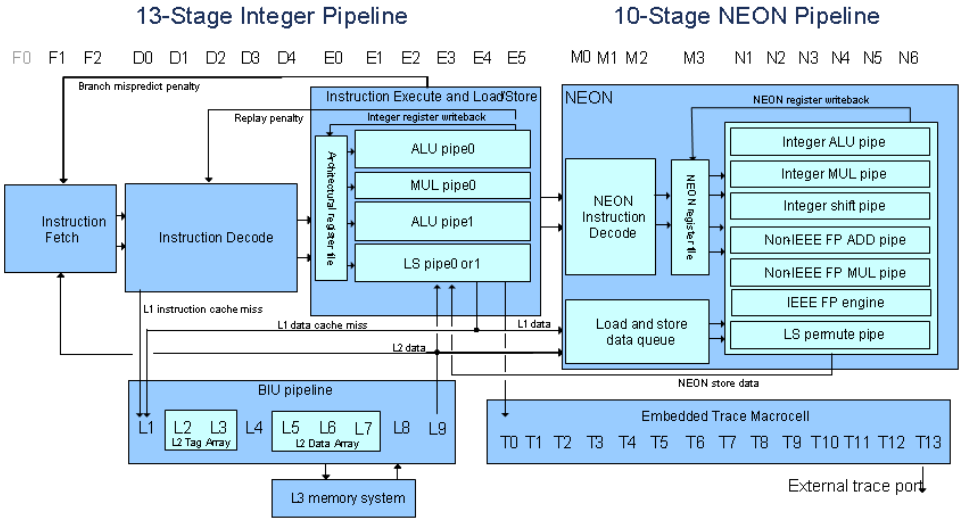
1: **procedure** TAGGENERATION($adlen, mlen, S$)
2: $tmp \leftarrow adlen||msglen$
3: $S_4 \leftarrow S_4 \oplus S_0$
4: **for** `i in 0..9` **do**
5: STATEUPDATE(S, tmp)
6: **end for**
7: $t \leftarrow S_0 \oplus (S_1 <<< a) \oplus (S_2 \& S_3)$
8: **end procedure** ▷ Tag is concatenated with the ciphertext afterwards

2.3 ARM Cortex-A8

Optimization of MORUS will be done on an ARM Cortex-A8 processor. The ARM architecture is a RISC (Reduced Instruction Set Computing) architecture. This architecture allows instructions to take fewer cycles versus a CISC (Complex Instruction Set Computing) architecture. However, it does mean that there are less instructions which in turn means more instructions are required for the same set of computations. One such example is that ARM uses separate load/store instructions to read from and write to memory whereas in a CISC architecture this may be included in an arithmetic instruction.

The Cortex-A8 is a 32-bit ARMv7-A superscalar design [12] meaning that it is capable of executing more than one instruction each cycle. It has 16 32-bit registers of which only 14 are usable due to the stack pointer and program counter being reserved. The processor makes use of pipelines where an instruction can advance one stage every cycle. Most Cortex-A8 processors have a NEON unit included which provides SIMD instructions [7]. The NEON unit sits behind the ARM core pipelines; as a result if a pipeline is stalled within the ARM core the instructions destined for the NEON unit are also stalled. If a stall happens within the NEON unit it does not typically stall the ARM core. One example of where the NEON unit can stall the ARM core is when an ARM instruction is dependent on a NEON store instruction. Other sources for stalling are memory system stalls (such as L1 cache misses) and branch mispredict penalties. See Figure 2.2 for a schematic overview of the ARM Cortex-A8 processor.

Figure 2.2: A schematic overview of the Cortex-A8 by Texas Instruments[1]



[1] http://processors.wiki.ti.com/index.php/File:Cortex-A8Pipeline.png

2.3.1 NEON

The NEON unit is capable of performing SIMD instructions allowing a single instruction to process multiple data elements. The ability to perform SIMD instructions is essential to optimizing MORUS as it is built with the availability of SIMD instructions in mind. For instance, the internal state of MORUS can be either 640- or 1280-bit which is further divided into 128- and 256-bit blocks during operation. NEON provides 128-bit registers and the ability to operate on 128-bit blocks of data which allows us to implement MORUS efficiently using NEON.

NEON provides 32 64-bit registers typically called `d0` through `d31`. Pairs of two adjacent 64-bit registers can be combined to form 128-bit registers which are called `q0` through `q15`. All NEON instructions allow you to operate on 64- and 128-bit registers, they can however take a different amount of cycles.

As stated before, NEON instructions pass through the ARM pipeline before reaching the NEON unit and thus have been fetched and decoded by the ARM core. When these instructions enter the NEON unit they are decoded once again and then scheduled by the NEON unit. This process takes multiple cycles and can stall the ARM core if the NEON unit is stalled. To prevent this from happening frequently, a 16-entry-deep instruction queue is provided. A similar stall can occur when the NEON unit requires data from the ARM core to perform a vector load. Thus, the NEON unit also contains a data queue to prevent this stall from happening frequently. This data queue is between 8 and 12 entries deep depending on the processor revision.

To optimize the MORUS cipher by using NEON it is essential to know how many cycles each instruction takes. This information can be found in Cortex-A8 Technical Reference Manual [12]. In the reference manual each instruction states in which stage it requires its source to be available and in which stage the result is available. While we use qhasm, for this step it is necessary to look at the assembly generated by qhasm since the reference manual lists assembly instructions. An example qhasm instruction such as

```
4x temp = s3_blk << 22
```

which shifts the four chunks of block S_3 to the left needs its source to be available in stage N1 and makes the result available in stage N3. However, due to the convention ARM uses, the result will not be usable in stage N3 but only in N4. When scheduling instructions this should be taken into account. If two of those instructions would have to be scheduled without causing a stall, the second instruction has to begin while the first instruction is in stage N3 so that it reaches stage N1 while the previous instruction reaches N4. If this were not taken into account one would try to begin the second instruction in stage N2 but this would cause a one cycle stall.

2.3.2 Dual Issue

As mentioned before the Cortex-A8 has a superscalar design allowing it to execute more than one instruction per clock cycle. In the case of the Cortex-A8 it is called dual issue due to the fact that it can execute up to two instructions in one cycle. It should be noted that normal ARM instructions can be paired with NEON instructions.

While the processor is able to execute two instructions at the same time there are restrictions. The restrictions for the ARM core are fairly minimal and not of much interest for the optimization of MORUS. For NEON there is only one type of dual issuing allowed and that is pairing a NEON data-processing instruction with an instruction that executes in the NEON load-/store/permute pipeline. This holds true even if two NEON data-processing instructions could theoretically be dual issued due to them being executed in separate pipelines and there being no data hazards.

Even though these restrictions are fairly strict the dual issue capability is useful because some instructions such as vector load and store are multi-cycle instructions (requiring a source/result in multiple stages). For multi-cycle instructions it is possible to dual issue for the first and last cycle. Additionally, some instructions called byte permute instructions execute in the load/store/permute pipeline allowing them to be dual issued.

2.3.3 Cycle Counter

To measure performance of an optimized implementation of MORUS it becomes necessary to know how many cycles a block of assembly takes. To achieve this it is necessary to read the Cycle Counter Register. In qhasm this can be done by using `cyclecount(register)` which translates to the following instruction:

```
mrc p15, 0, r4, c9, c13, 0
```

It reads the value of the cycle counter by telling the internal coprocessor CP15 to read the Cycle Counter Register into a register such as `r4`. Doing this twice, once before a block of assembly and once after it, allows us to know how many cycles have taken place within a block of assembly.

When measuring how many cycles a block of assembly takes the cycle counting may be one cycle off because the `mrc` instruction also takes one cycle. However, the instruction can be dual issued so it is also possible that it does not take an extra cycle. When reading the cycle counter it is thus important to take the context into account, especially if being one cycle off is important.

Another problem that can arise is that instructions are still in the pipeline while the cycle counter is being read. Reading the cycle counter takes only a single instruction while a NEON instruction first has to pass through the

13 stage ARM core and then go through the 10 stage NEON unit. For short sequences of assembly this potential error in measuring may be too large to get accurate readings. This is usually solved by repeating a code fragment multiple times when measuring and getting an average such that the measuring error becomes smaller.

One final problem that can occur when measuring is that a context switch can happen. This becomes more likely if the block of assembly is large. If a context switch happens the result is of very little use. As long as context switches happen infrequently this can be mitigated by measuring multiple times and taking an average.

Chapter 3

Related Work

Little work has been done on implementations of MORUS besides the reference implementation. There is an SSE2 implementation and there is an AVX2 implementation. Both SSE2 and AVX2 provide SIMD instructions but are only available on x86 processors. In those implementations it was shown that for longer messages they were able to reach 1.19 cycles per byte for MORUS-640-128 and 0.69 cycles per byte for MORUS-1280-256 [8] which was a significant improvement over the reference implementation. This lead us to believe that similar performance gains could also be achieved by using NEON.

There were no prior ARM implementations of MORUS. However, the idea of using NEON to optimize cryptography is not new [17]. In the paper "NEON crypto" by Bernstein and Schwabe a set of four cryptographic primitives were implemented using NEON and all of them were found to have a benefit from using NEON [6]. The idea of using byte permute instructions for rotations so that we can dual issue these instructions comes from their paper.

The CAESAR competition currently features 15 third-round candidates. Without NEON enabled on the Cortex-A8, MORUS is already one of the fastest submitted ciphers [1]. In terms of performance NORX [15], Tiaoxin [16], and Keyak [19] are its closest competitors. When NEON is enabled, NORX, which has a NEON implementation, becomes the fastest submission.

Chapter 4

Implementation

Submissions for the CAESAR competition are required to provide a reference implementation of their cipher. The reference implementation exists to support the public understanding of the cipher and to provide a correct implementation which other implementations can verify against. It is meant to be as clear as possible and is thus not optimized [5]. Our implementations of MORUS are based on this reference implementation and produce exactly the same output.

We wrote two implementations, one of these implementations implements MORUS with a 640-bit state and a 128-bit key (MORUS-640-128) and the other one implements MORUS with a 1280-bit state and a 256-bit key (MORUS-1280-256). We do not provide an implementation for MORUS-1280-128 as it is the same as MORUS-1280-256 except for the fact that the key is concatenated with itself during initialization.

Although the goal was to implement an optimized implementation we have not micro-optimized code which already takes a small amount of processor time compared to other parts of the code. Examples of this are the initialization, tag generation, tag verification, and partial block encryption and decryption. All of them incorporate the state update which *is* optimized. The state update costs the majority of processor time even after optimization so optimizing the rest of the code in those sections will have an insignificant effect on the total processor time. The only exception to this was specifying memory alignment. Specifying memory alignment allows us to improve performance of functions that do have an impact by lowering the amount of cycles loads and stores cost. However, to benefit from this we have to do this consistently, even in places where the benefit is small.

The implementation does everything in constant time except processing the associated data and encryption/decryption since they have a variable length. This means that initialization, tag generation, and tag verification takes the same amount of cycles regardless of the input data. This helps against timing attacks.

Since the amount of time required for processing the associated data and encryption/decryption is large, they were good targets for optimization since they are likely to become the dominant factor on longer inputs. As a result we have optimized these functions by implementing them entirely in qhasm. As mentioned before the state update was also optimized since it is the core function in every step of MORUS. How these functions have been optimized in MORUS-640-128 and MORUS-1280-256 can be found in the upcoming sections.

4.1 State Update

The `StateUpdate` function consists of three distinct part. These parts are loading from memory to NEON registers, processing the five rounds, and storing data from NEON registers to memory. This is the case for both MORUS-640-128 and MORUS-1280-256.

The `StateUpdate` functions receives two pointers to memory where the first pointer called `input_0` points to the message block and the second pointer called `input_1` points to an array with the state. Our initial implementation using NEON can be seen in Listing 4.1. This code is not minimal as NEON supports loads up to 256 bits at once. However, qhasm does not support this by default and the performance difference is only two cycles due to needing one more cycle for the second instruction and an extra addition. Additionally other optimizations that will be discussed later can no longer be applied resulting in worse performance than if we were to load 256 bits at once. An important optimization we did do in the loading stage was specifying alignment as can be seen in Listing 4.2. By doing this every load takes one cycle instead of two. Since the `StateUpdate` is used in every step of MORUS the total amount of cycles saved is significant.

Listing 4.1: Normal loading

```
msg_blk = mem128[input_0]
s0_blk = mem128[input_1]
input_1 += 16
s1_blk = mem128[input_1]
input_1 += 16
s2_blk = mem128[input_1]
input_1 += 16
s3_blk = mem128[input_1]
input_1 += 16
s4_blk = mem128[input_1]
```

Listing 4.2: Loading with alignment

```
msg_blk aligned= mem128[input_0]
s0_blk aligned= mem128[input_1]
input_1 += 16
s1_blk aligned= mem128[input_1]
input_1 += 16
s2_blk aligned= mem128[input_1]
input_1 += 16
s3_blk aligned= mem128[input_1]
input_1 += 16
s4_blk aligned= mem128[input_1]
```

The loading code seen so far is for MORUS-640-128, in MORUS-1280-256 we need to load twice the amount. This results in two problems. The first problem is that NEON registers are only 128-bit and the second problem is the we do not have enough caller-saved registers. The first problem was solved by putting every 256-bit block into two 128-bit registers. The code

is similar to Listing 4.2 except that we need two `mem128` for one block. Solving the first problem causes the second problem as there are no longer any caller-saved registers left. We need two more registers which we use as temporary registers during calculations. The most efficient way to solve this is by using callee-saved registers. The use of callee-saved registers costs extra cycles since we first have to store the values of those registers in the memory before we can use them. The code we used to do this can be seen in Listing 4.3. What this does is store `q4` and `q5` on the stack so that we can use them later.

Listing 4.3: Storing callee-saved registers on the stack

```
q4_stack = q4_stack[0]caller_q4[1]
q4_stack = caller_q4[0]q4_stack[1]
q5_stack = q5_stack[0]caller_q5[1]
q5_stack = caller_q5[0]q5_stack[1]
```

After loading is complete the five rounds are next. Since every round is virtually the same except for what registers and constants are used we will be discussing only one round here. The round we will be discussing is round 2 since round 1 is the only round to differ in that it does not use the message block (thus making it a strict subset of the other rounds). The MORUS-640-128 code for round 2 can be seen in Listing 4.4.

Listing 4.4: Round 2 of the state update

```
s1_blk ^= s4_blk
s1_blk ^= msg_blk
temp = s2_blk & s3_blk
s1_blk ^= temp
4x temp = s1_blk << 31
4x s1_blk unsigned>>= 1
s1_blk ^= temp

s4_blk = s4_blk[2,3]s4_blk[0,1]
```

The first 4 lines are simply all the necessary operations in round 2 before we can break up the 128-bit block into 32-bit chunks to rotate. In round 1 the XOR with `msg_blk` would not be present. Once these operations are done we want to rotate every chunk of the 128-bit register. NEON does not have a rotate operation and the rotate is thus rewritten using two shifts and one XOR. It works by first getting all the correct top bits in a temporary register; note that we shift per chunk. Then the original register is used to obtain all the bottom bits; again note that we shift per chunk. Since shifts introduce zeros we can afterwards XOR the temporary register with the original register to combine the top and bottom bits of every chunk to get a rotated original register. We consider the processing of the S_1

block to be optimal in our implementation since every operation takes one instruction except for the rotate. We also do not see any opportunity for dual issuing since all of them are NEON data-processing instructions which can not be dual issued. Reordering the instructions to avoid latencies because of register and pipeline availability is also not possible due to instructions having dependencies on previous instructions.

Besides the processing of block S_1, round 2 also processes a second block; in this case block S_4. This is again a rotate which NEON does not have. However, rotates that operate on 128-bit blocks by a distance of 8 can always be done in NEON using a byte-permute instruction [14]. In the case of round 1, 3, and 5 it is a vector extract and in the case of round 2 and 4 it is a vector swap. Since they are byte-permute instructions they happen in the load/store/permute pipeline. As a result they can be dual issued and, due to the way the rounds are set up, do not cause any latencies for the next round. In practice this means that processing the second block has no negative effect on the amount of cycles per byte due to the NEON unit already being occupied with other instructions.

For MORUS-1280-256 the implementation is similar since every instruction except the rotation of the second block does not cross into the other 128-bit register. As a result these instructions can simply be done once for the top 128-bit register and once for the bottom 128-bit register. This does mean that every round costs at least twice as many cycles in MORUS-1280-256. The rotation of the second block in every round can still be done by a byte-permute instruction but is no longer free because in round 1, 3, and 5 we need two subsequent vector extractions instead of one thus causing a delay.

Storing data from NEON registers in memory is very similar to loading it. One notable difference, as can be seen in Listing 4.5, is that `msg_blk` is not written back to memory. This is because the message block is never changed and it is thus not necessary to write back to memory. Storing to memory is also done with alignment since it decreases the amount of cycles needed from two cycles to one cycle.

Listing 4.5: Storing with alignment

```
input_1 -= 64
mem128[input_1] aligned= s0_blk
input_1 += 16
mem128[input_1] aligned= s1_blk
input_1 += 16
mem128[input_1] aligned= s2_blk
input_1 += 16
mem128[input_1] aligned= s3_blk
input_1 += 16
mem128[input_1] aligned= s4_blk
```

Storing in MORUS-1280-256 is, again, similar to loading in MORUS-1280-256. Twice as much data needs to be stored back to memory. During loading the values of registers `q4` and `q5` were put on the stack. During storing we need to retrieve the memory to put the data back in the registers. The code to do this can be seen in Listing 4.6.

Listing 4.6: Loading callee-saved registers from the stack

```
caller_q4 = caller_q4[0]q4_stack[1]
caller_q4 = q4_stack[0]caller_q4[1]
caller_q5 = caller_q5[0]q5_stack[1]
caller_q5 = q5_stack[0]caller_q5[1]
```

One important aspect we have neglected to mention so far is the transitions between the distinct parts. Between loading and the first round it is actually possible to have some overlap since both the message block and block S_4 are not used in round 1. This allows us to dual issue those two instructions saving on the total amount of cycles used by the `StateUpdate`. The way we have implemented this can be found in Listing 4.7. The same optimization is also used in the MORUS-1280-256 implementation.

Listing 4.7: Overlapping loading and round 1

```
s0_blk ^= s3_blk
msg_blk aligned= mem128[input_0]
temp = s1_blk & s2_blk
s4_blk aligned= mem128[input_1]
s0_blk ^= temp
4x temp = s0_blk << 5
4x s0_blk unsigned>>= 27
s0_blk ^= temp

s3_blk = s3_blk[3]s3_blk[0,1,2]
```

A natural thought would be to do the same between round 5 and the storing. Unfortunately this does not work well due to the load/store/permute pipeline already being occupied by the second block at the end of the round.

4.2 Processing Associated Data

Since our implementation is based on the reference implementation we initially processed our associated data in exactly the same way as the reference implementation. The reference implementation has a loop in C code that for every 16 or 32 bytes, depending on the MORUS variant, calls the encryption function which does its encryption routine and then calls the `StateUpdate` function to update the state.

This way of handling the associated data quickly proved to be a bottleneck because it meant that we had the overhead of two function calls and an encryption routine of which the result is unused for every 16 or 32 bytes. Removing the encryption routine for associated data was the first step we took. However, the overhead of two function calls per 16 or 32 bytes was still too much. As a result we implemented processing the associated data in qhasm; including the loop so that we always have at most one function call for the associated data.

Implementing a loop in qhasm is typically not difficult but we did have one problem. MORUS allows the associated data to have a length of up to 2^{64} bits. Combined with the fact that we need to have a counter to know how often the loop needs to be executed we need space for at least 59 bits (since we can store how many blocks of 32 bytes we have). As a result we need to either combine two 32-bit registers or use one 64-bit NEON register. Unfortunately using a NEON register is not an option as NEON does not allow for instructions to set flags (except for instructions shared with the VFP) thus making it infeasible to do conditional execution without constantly having NEON instructions wait on normal instructions causing large delays. We have thus implemented the loop counter using two 32-bit registers. See Listing 4.8 for the implementation. What it does is read the loop counter, which is computed in C code, to `r4` and `r5`. Then it checks if both of them are 0, if that is the case we are done and execution jumps to the storing stage. Otherwise we load the message block, which is the associated data, to `msg_blk` and perform the five rounds of the `StateUpdate`. After the five rounds 1 is subtracted with borrowing from both `r4` and `r5` if `r5` is 0. Otherwise only `r5` is subtracted from. This has the effect of subtracting 1 from the total number of loops left. After that we jump back to the start which checks if we are done or not thus completing our loop. The loop is implemented exactly the same in MORUS-1280-256.

Listing 4.8: Loop implemented in qhasm

```
assign r4 r5 to loop_counter_right loop_counter_left =
    mem64[input_2 + 0]

loop:
=? loop_counter_right - 0
goto start if !=
=? loop_counter_left - 0
goto end if =

start:
msg_blk aligned= mem128[input_0]
input_0 += 16
... # five rounds of StateUpdate

=? loop_counter_right - 0
goto decrement_left if =

loop_counter_right -= 1
goto loop

decrement_left:
loop_counter_left -= 1
loop_counter_right -= 1
goto loop

end:
... # storing data to memory
```

4.3 Encryption and Decryption

Encryption and decryption had the same function call overhead as processing the associated data. As a result both encryption and decryption were implemented in qhasm.

Since processing the associated data is exactly same except that there is no encryption routine it was used as the basis for the encryption. See Listing 4.9 for the implementation. The optimization that was in the generic `StateUpdate` function that allowed `msg_blk` to be loaded later had to be undone since the message block, which is the plaintext here, is necessary to compute the ciphertext. The ciphertext block is also immediately written back to memory as the register needs to be reused for the next ciphertext block. Once the encryption routine is over the five rounds of the state update are executed and then the loop continues to the next iteration. Decryption is equivalent to encryption except that `msg_blk` and `cipher_blk` are swapped. The implementation for MORUS-1280-256 is similar except that we need twice as many instructions.

Listing 4.9: Encryption routine

```
start:
msg_blk aligned= mem128[input_0]
input_0 += 16

temp = s2_blk & s3_blk
temp2 = s1_blk[1,2,3]s1_blk[0]
temp ^= temp2
temp ^= s0_blk
cipher_blk = msg_blk ^ temp

mem128[input_1] aligned= cipher_blk
input_1 += 16

... # five rounds of StateUpdate
```

4.4 Lower Bound Analysis

When optimizing we want to know how well the current implementation performs in comparison to the best possible implementation. To do this we made an approximation of the lower bound for MORUS-640 by initially assuming perfect instruction scheduling.

The natural place to start is the five rounds in the `StateUpdate` since every byte processed by MORUS goes through these five rounds. Note that we excluded the loading and storing from memory since they can be (partially) avoided in certain cases such as processing the associated data and will thus skew the calculation of the lower bound.

The five rounds contain the following instructions: 19 XORs, 5 ANDs, 5 left shifts, 5 right shifts, 3 vector extracts, and 2 vector swaps. Assuming perfect instruction scheduling we can leave out the vector extracts and vector swaps since they can always happen while doing other operations. The XORs and bitwise ANDs are one cycle each whereas the shifts are two cycles each. This gives us a minimum of 44 cycles for the five rounds. This is $\frac{44}{16} = 2.75$ cycles per byte. We can make this number slightly more accurate because we know that the shifts *always* happen as the last change to a block and must thus wait on an XOR or AND to finish. There are no other instructions left that could be scheduled in between. By using that fact we can state that for every round there is a one cycle wait that has to occur. This adds five cycles to the total making the new approximation $\frac{49}{16} = 3.06$ cycles per byte.

We expected that in practice we would not reach this minimum as there are quite a few situations within the five rounds where perfect instruction scheduling is *extremely* difficulty, if not impossible. Additionally the estimate does not take overheads into account that always exist such as loading and storing, cache misses, function call overhead, and loop overhead. Also, this lower bound approximation is only taking the `StateUpdate` into account whereas in practice we need to use the data from the state to encrypt, decrypt, and generate a tag.

Chapter 5

Results

Benchmarking was done on a BeagleBone Black. The model used features a 1GHz AM3358 ARM Cortex-A8 processor by Texas Instruments. To obtain our results we used the `SUPERCOP` benchmarking toolkit. It is part of eBACS [11] which is a project for benchmarking cryptographic systems. `SUPERCOP` contains all submissions for the `CAESAR` competition and allows us to compare our results against other implementations; especially because `SUPERCOP` also includes results for the BeagleBone Black. The cycle counter used is the cycle counter integrated in `SUPERCOP` which reads the Cycle Counter Register on ARM. The `SUPERCOP` version used was the 2017-02-28 release which is the most recent version.

Benchmark results can be found in Table 5.1. `gcc 4.9.2` was used for compilation. The compiler flags are chosen by `SUPERCOP`. For MORUS-640-128 they were `-funroll-loops`, `-fno-schedule-insns`, `-O3`, and `-fomit-frame-pointer` while for MORUS-1280-256 they were `-mfloat-abi=hard`, `-mfpu=neon`, `-O3`, and `-fomit-frame-pointer`. The benchmarking results are directly compared to the results of the reference implementation. Every benchmark entry consists of a left side, a + sign, and a right side. The left side is the size of the plaintext while the right side is the size of the associated data. The term "long" means 2048 bytes. We chose for this notation to match the notation used in the eBACS project. All values were computed by taking the median cycle count and then dividing it by the amount of bytes processed.

Our implementations have been submitted to the eBACS project. While we submitted a MORUS-1280-128 implementation for completeness we do not benchmark it here as the implementation is equivalent to MORUS-1280-256. Our submission is currently pending approval and is thus available elsewhere[1].

From our benchmarking results we can see that our MORUS-640-128 implementation is typically around twice as fast as the reference imple-

[1] `https://github.com/OussamaDanba/MORUS_on_NEON`

Table 5.1: Benchmarking results

		REF-640	MORUS-640	REF-1280	MORUS-1280
long+0	encrypt	9.89	5.26	15.64	5.42
	decrypt	10.39	5.29	16.10	5.39
	forgery	10.40	5.28	16.11	5.41
long+long	encrypt	10.41	4.72	15.77	4.41
	decrypt	10.30	4.75	16.09	4.39
	forgery	10.29	4.74	16.09	4.40
0+long	encrypt	10.95	5.85	15.92	6.04
	decrypt	10.20	5.86	16.07	6.02
	forgery	10.20	5.85	16.07	6.03
1536+0	encrypt	12.37	5.81	22.76	6.12
	decrypt	12.92	5.84	23.32	6.22
	forgery	12.92	5.83	23.31	6.26
1536+1536	encrypt	11.68	5.03	19.41	4.83
	decrypt	11.58	5.03	19.77	4.84
	forgery	11.58	5.03	19.76	4.85
0+1536	encrypt	13.45	6.40	23.06	6.94
	decrypt	12.73	6.42	23.29	6.91
	forgery	12.73	6.40	23.29	6.93
64+0	encrypt	69.30	87.45	186.39	108.55
	decrypt	70.61	89.27	189.11	111.09
	forgery	70.59	89.23	189.08	111.66
64+64	encrypt	40.73	29.66	102.84	44.60
	decrypt	40.97	29.92	104.20	45.69
	forgery	40.96	29.82	104.23	46.53
0+64	encrypt	70.69	55.55	187.06	84.84
	decrypt	70.62	56.03	189.16	87.53
	forgery	70.69	55.85	189.16	88.08

mentation. Our MORUS-1280-256 implementation shows even more performance gain. Perhaps surprisingly, the difference between MORUS-640-128 and MORUS-1280-256 is little even though NEON has no 256-bit registers or instructions. We believe that this is the case because despite taking at least twice as much instructions we operate on twice the amount of data (32 versus 16 bytes). This also has the benefit of avoiding the overhead the loop introduces by having to loop less often.

As far as we are aware this is the first time MORUS has been implemented on ARM besides the reference implementation so we are not able to compare against other implementations. However, we can compare it against other submissions. We mentioned before that MORUS is the fastest submission when NEON in not used; when NEON is used NORX became the fastest. The online `SUPERCOP` results only benchmark the Cortex-A9 with NEON [1], which is a slightly faster processor than the Cortex-A8 due to the having a shorter pipeline and allowing out-of-order execution. Nevertheless, we are still able to say our MORUS-640-128 implementation is faster than the fastest NORX implementation. This is because our cycles per byte count on the Cortex-A8 is lower than the NORX implementation on the Cortex-A9. Our MORUS-1280-256 implementation also outperforms the fastest NORX implementation except when short messages are processed.

Chapter 6

Conclusions

Our work has resulted in two optimized implementations of MORUS using NEON. By doing this we have shown that MORUS can be implemented efficiently using NEON.

We do not claim our implementation is the most optimized implementation possible, but we do believe that on the Cortex-A8 not much more can be done.

Future work that can be done is porting our implementations to other ARM processors which have NEON capabilities. Since NEON itself does not change significantly on other processors we believe that this would not be extremely difficult. One such processor would be the Cortex-A9 so that we can more accurately compare other submission for the `CAESAR` competition against MORUS.

Bibliography

[1] Measurements of CAESAR candidates, indexed by machine, 2017. Accessed: 2017-06-23. URL: https://bench.cr.yp.to/results-caesar.html.

[2] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: provably fixing the SSH binary packet protocol. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 1–11. ACM, 2002. URL: http://cseweb.ucsd.edu/~mihir/papers/ssh.pdf.

[3] Mihir Bellare, Phillip Rogaway, and David Wagner. A conventional authenticated-encryption mode. *manuscript, April*, 2003. URL: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/eax/eax-spc.pdf.

[4] Daniel J. Bernstein. qhasm: tools to help write high-speed software. Accessed: 2017-06-23. URL: https://cr.yp.to/qhasm.html.

[5] Daniel J. Bernstein. CAESAR call for submissions, final (2014.01.27), 2014. Accessed: 2017-06-23. URL: https://competitions.cr.yp.to/caesar-call.html.

[6] Daniel J Bernstein and Peter Schwabe. NEON crypto. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 320–339. Springer, 2012. URL: https://cryptojedi.org/papers/neoncrypto-20120320.pdf.

[7] ARM Holdings. Cortex-A8 processor. Accessed: 2017-06-23. URL: https://www.arm.com/products/processors/cortex-a/cortex-a8.php.

[8] Hongjun Wu, Tao Huang. The authenticated cipher MORUS (v2), September 2016. URL: https://competitions.cr.yp.to/round3/morusv2.pdf.

[9] Charanjit S Jutla. Encryption modes with almost free message integrity. In *International Conference on the Theory and Applications*

of Cryptographic Techniques, pages 529–544. Springer, 2001. URL: https://eprint.iacr.org/2000/039.pdf.

[10] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Annual International Cryptology Conference*, pages 310–331. Springer, 2001. URL: https://www.iacr.org/archive/crypto2001/21390309.pdf.

[11] Daniel J. Bernstein, Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. Accessed: 2017-06-23. URL: https://bench.cr.yp.to/.

[12] ARM Limited. Cortex-A8 technical reference manual, revision r3p2, 2010. Accessed: 2017-06-23. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf.

[13] ARM Limited. Procedure call standard for the ARM architecture, 2015. Accessed: 2017-06-23. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf.

[14] Martyn. Coding for NEON - part 5: Rearranging vectors, 2013. Accessed: 2017-06-23. URL: https://community.arm.com/processors/b/blog/posts/coding-for-neon---part-5-rearranging-vectors.

[15] Jean-Philippe Aumasson, Philipp Jovanovic, Samuel Neves. NORX v3.0, September 2016. URL: https://competitions.cr.yp.to/round3/norxv30.pdf.

[16] Ivica Nikolić. Tiaoxin, September 2016. URL: https://competitions.cr.yp.to/round3/tiaoxinv21.pdf.

[17] Joost Rijneveld. Implementing Prøst on the cortex A8 using internal parallelisation, January 2015. URL: https://joostrijneveld.nl/papers/20150104_proest_cortexa8.pdf.

[18] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):365–403, 2003. URL: http://web.cs.ucdavis.edu/~rogaway/papers/ocb-full.pdf.

[19] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles van Assche, Ronny van Keer. CAESAR submission: Keyak v2, September 2016. URL: https://competitions.cr.yp.to/round3/keyakv22.pdf.

Appendix A

qhasm Extensions

During our research we stumbled upon a few instructions that the Cortex-A8 supported but were not present in the qhasm syntax. Since qhasm is easily extendable we have added these ourselves.

The first instruction we missed was the ability to do a vector swap operation on two 128-bit registers instead of two 64-bit registers. In MORUS-1280-256 this is used to rotate 128 to the left. While it is possible to do this in two vector swap operations it costs one more cycle due to having to fetch two instructions. When trying to dual issue it is easier to only have to use one instruction. The syntax we introduced to do this was `>=<` and is implemented as:

```
r >=< s:<r=reg128:<s=reg128:asm/vswp <r,<s:
```

The second extension of qhasm we ended up not using. It gives qhasm the ability to load and store 256 bits of aligned memory to and from two 128-bit registers. This was supposed to be used in MORUS-1280-256 to do our loading and storing since it saves two cycles. Loading was implemented as

```
mem256[t] aligned= r s:<r=reg128:<s=reg128:<t=int32:asm/vst1.64
    {<r%bot-<r%top,<s%bot-<s%top},[<t,!colon 256]
```

while storing was implemented as

```
r s aligned= mem256[t]:>r=reg128:>s=reg128:<t=int32:asm/vld1.64
    {>r%bot->r%top,>s%bot->s%top},[<t,!colon 256]
```

This implementation is functional but poses problems when qhasm decides that these registers are not adjacent. Normally, registers not being adjacent would not be a problem but since the assembly generated contains a list of registers there is a restriction on the so called register stride. The list of registers must not have a space of more than 2. This means that {`d0-d1,d2-d3`} is valid but {`d0-d1,d6-d7`} is not. qhasm does not support a way to enforce the adjacency of registers so we left this small optimization out.