Bachelor's Thesis Computing Science

Optimizing Elephant v2 for ARMv7-M

Norbert Boudens
s1059405

January 7, 2023

First supervisor:
dr. ir. B.J.M. Mennink (Bart)

Second/daily supervisor:
MSc V. Jahandideh (Vahid)

Second assessor:
prof. dr. J.J.C. Daemen (Joan)

Radboud University

Abstract

The lightweight authenticated encryption scheme Elephant v2 is a NIST competition candidate. The mode of Elephant is a permutation-based encrypt-then-MAC construction with three instances. One of these instances, Delirium, uses the Keccak-f permutation. We optimize Delirium for the ARMv7-M architecture, starting with our own implementation based on the specification. We convert our implementation of Elephant v2 into a 32-bit implementation, which we then try to optimize using ARM instructions. Finally, we look at alternative Keccak implementations, such as Keccak-p. The final implementation has an average speedup of 15 over our initial implementation of Elephant.

Contents

Chapter 1

Introduction

Cryptographic algorithms are key in securing data, therefore it is a goal of certain organizations and institutes to further improve on these algorithms. There are constantly evolving issues in cryptography due to the creation/usage of new technology, such as new architectures which will then require these algorithms to adapt, or to be optimized. Developments in cryptography are often aimed at getting these algorithms to work universally across as many devices as possible. One issue with this lies in resource constraints. Certain devices have less storage, computing power or networking capabilities than other devices.

The National Institute of Standards and Technology (NIST) is a government agency in the United States department of Commerce [13]. NIST has an ongoing project to solicit, evaluate, and standardize newly developed lightweight algorithms for resource constrained devices where their current algorithms are not suited. This project is called the NIST Lightweight Cryptography project (NIST LWC). Lightweight cryptography (LWC) is a type of cryptography with the goal of creating or improving algorithms to be able to run on some of the most resource constrained devices.

In this thesis, we look at optimizing the NIST candidate Elephant v2 [8] for the 32-bit resource constrained architecture ARMv7-M. Elephant consists of three instances, Dumbo, Jumbo, and Delirium. We focus on the Delirium instance, with the goal of making it a more viable option for ARMv7-M.

Firstly, we create our own implementation of Elephant's Delirium instance (Ch. 3.1.2) based on the Elephant specification [8]. We use this implementation as the base for a new 32-bit implementation (Ch. 5), which we use for further optimization.

Secondly, we look at possible optimization points of the 32-bit implementation using ARM data processing instructions [2] (Ch. 6).

Thirdly, we look at existing implementations of the permutation function. Elephant's Delirium uses Keccak-$f[200]$ as its permutation function. We look at three alternative permutations from the eXtended Keccak Code Package [1], which we use to create three new implementations (Ch. 7).

Finally, we measure the performance of the implementations in ARMv7. We discuss the results of the measurements and conclude a final implementation (Ch. 8). With this final implementation and the methods used, we answer the research question: How can we optimize the cryptographic cipher Elephant v2 for the ARMv7-M architecture?

Related Work

Besides a RISC-V implementation of Elephant [12], there are no other Elephant implementations optimized for a specific architecture.

Research has been done on implementations of the PRESENT and Fantomas lightweight block ciphers for an ARMv7-M CPU (Cortex-M3) [11]. There is research on binary field multiplication in ARMv7 [16], but this is not relevant to our work.

Chapter 2

Preliminaries

2.1 Notation

For $i \in \mathbb{N}$, let $\{0,1\}^i$ denote the set of all binary strings with length i and $\{0,1\}^*$ the set of arbitrarily length strings. For $X \in \{0,1\}^*$, we define

$$X_1 \dots X_\ell \xleftarrow{n} X$$

to be the function which partitions X into $\ell = \lceil |X|/n \rceil$ blocks of n bits, with the last block appended by trailing zeros.

We denote $\lfloor x \rfloor_j$ as the j left-most bits of the bit-string x.

A logical left shift is denoted by $x \ll k$, and a logical right shift is denoted by $x \gg k$. In a logical left/right shift, the bits of x are moved to the left/right over k positions.

A circular left shift moves each bit of a bit-string left by a specified number of bits. Every bit that is shifted off one end of the bit-string is reintroduced at the other end. This is denoted by $x \lll l$ where x is rotated to the left over l positions. Respectively, a circular right shift is denoted by $x \ggg l$ with a rotation of x to the right.

2.2 Linear-Feedback Shift Register

A linear-feedback shift register (LFSR) is used as a pseudorandom pattern generator to generate a sequence of bits. An LFSR is a circuit with a row of cells, each containing a bit. This sequence of bits within the cells of the LFSR is also called the state of the LFSR. Cycling an LFSR is done by shifting each bit to the cell right of it, with the right-most bit being the output of the LFSR. The next state of the LFSR is determined by so-called

taps on certain cells, these taps get XOR'd sequentially with the output bit of the LFSR. After a cycle to the right of the LFSR, the left-most bit of the new state is the right-most bit of the previous state. For a cycle to the left, the new left-most bit is the previous state's right-most bit.

An example of an 8-bit LFSR with taps on the cells 1,4 and 5 is shown in Figure 2.1.

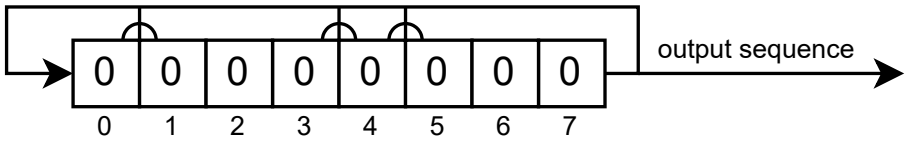

Figure 2.1: An 8-bit LFSR.

2.3 Authenticated Encryption

Authenticated encryption (AE) is a form of encryption which ensures confidentiality and authenticity. Authenticated encryption takes a plaintext, a key and a header. A header is a piece of context or extra information which will not get encrypted, meaning there is no insurance of confidentiality on the header. Encrypting the plaintext results in a ciphertext and authentication tag, which is a string of bits used to check for modifications of the encrypted message. Decryption takes the ciphertext, the key, and tag and header and returns the original plaintext. Associated data (AD) can be used to check the integrity of the ciphertext or message. This type of encryption is called authenticated encryption using associated data (AEAD). Associated data can be any type of information as long as it is present in both encryption and decryption, it is also described as encryption context. If the wrong AD is present in decryption, the decryption process fails.

Encryption (**enc**) takes as input a key $K \in \{0,1\}^k$, a nonce $N \in \{0,1\}^m$, associated data $A \in \{0,1\}^*$, and a message $M \in \{0,1\}^*$. Using these inputs, it outputs a ciphertext $C \in \{0,1\}^{|M|}$ and tag $T \in \{0,1\}^t$.

Decryption (**dec**) takes as input a key $K \in \{0,1\}^k$, a nonce $N \in \{0,1\}^m$, associated data $A \in \{0,1\}^*$, a ciphertext $C \in \{0,1\}^*$, and a tag $T \in \{0,1\}^t$. Using these inputs, it outputs a message $M \in \{0,1\}^{|M|}$ if the corresponding tag is correct, or a $\perp$-sign if it is incorrect.

2.4 Elephant v2

Elephant v2 is a NIST submission by Beyne, Chen, Dobraunig, and Mennink with the purpose of being a lightweight authenticated encryption scheme [8].

The mode of Elephant is a permutation-based encrypt-then-MAC construction with three instances. These instances are: Dumbo, Jumbo, and Delirium. Dumbo and Jumbo use a Spongent permutation, and Delirium uses the Keccak-$f[200]$ permutation [8]. In this thesis, we focus on the Delirium instance of Elephant.

2.4.1 Mode

The generic mode of Elephant is the authenticated encryption mode. It consists of two algorithms, encryption (enc) and decryption (dec) [8].

A depiction of the authenticated encryption mode of Elephant v2, as described by the official specification [8], is given in Figure 2.2. The encryption part (top) pads the message as $M_1 \dots M_{\ell_M} \xleftarrow{n} M$, and the ciphertext equals $C = \lfloor C_1 \dots C_{\ell_M} \rfloor_{|M|}$.
The authentication part (bottom) has the nonce and associated data padded as $A_1 \dots A_{\ell_A} \xleftarrow{n} N||A||1$, and ciphertext padded as $C_1 \dots C_{\ell_C} \xleftarrow{n} C||1$.

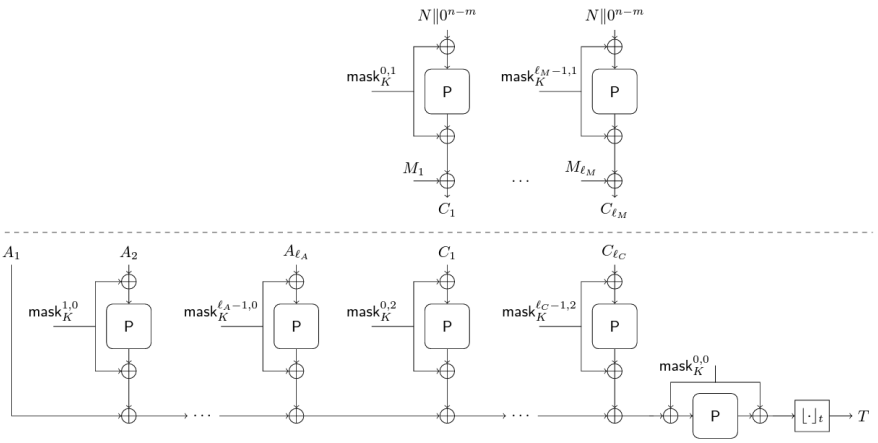

Figure 2.2: Depiction of Elephant v2

2.4.2 Masking Function

Elephant's Delirium uses the LFSR $\varphi_1 : \{0,1\}^n \to \{0,1\}^n$, which is given as a $\mathbb{F}_2$-linear map [8] where the x_i's correspond to 8-bit words:

$$(x_0, \dots, x_{24}) \to (x_1, \dots, x_{24}, x_0 \lll 1 \oplus x_2 \lll 1 \oplus x_{13} \ll 1).$$

For every cycle of φ_1 a circular left shift is done on x_0 and x_2 and a logical left shift on x_{13} of the state. Let φ_2 be defined as $\varphi_2 = \varphi_1 \oplus id$, P the permutation function, K the key, and $i, j \in \mathbb{N}$. The notation φ_1^i or φ_2^j means i cycles of φ_1 and j cycles of φ_2.

We can then give the masking function:

$$mask_K^{i,j} = mask(K, i, j) = \varphi_2^j \circ \varphi_1^i \circ P(K||0^{n-k}).$$

If we were to take $mask_K^{0,1}$ from the encryption part in Figure 2.2, we can define it using φ_1 and φ_2 as follows:

$$mask_K^{0,1} = \varphi_2^1 \circ \varphi_1^0 \circ P(K||0^{n-k}) = \varphi_2 \circ P(K||0^{n-k}).$$

2.4.3 Keccak Permutation

Keccak is a permutation function designed by Bertoni, Daemen, Peeters, and Van Assche [5], and chosen as winner of the NIST SHA-3 competition [14]. The Keccak permutation can be used to construct stream ciphers, cryptographic hash functions, pseudo random number generators, MACs, or AEAD algorithms.

Keccak is a family of hash functions based on the sponge construction. In Keccak a permutation is chosen from a set of seven Keccak-f permutations, denoted by Keccak-$f[b]$, where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of the permutation [5]. Elephant's Delirium instance uses the Keccak-$f[200]$ permutation.

Let X be a 200-bit input, one round of Keccak-$f[200]$ can then be described as follows:

for $i = 1, \ldots, 18$ **do**
  $X \leftarrow \iota \circ \chi \circ \pi \circ \rho \circ \theta(X)$
end for

The five Keccak step functions θ, ρ, π, χ, and ι are defined as:

$$\theta : a[x, y, z] \leftarrow a[x, y, z] \oplus \bigoplus_{y'=0}^{4} a[x-1, y', z] \oplus \bigoplus_{y'=0}^{4} a[x+1, y', z-1],$$
$$\rho : a[x, y, z] \leftarrow a[x, y, z + t[x, y]],$$
$$\pi : a[x, y, z] \leftarrow a[x + 3y, x, z],$$
$$\chi : a[x, y, z] \leftarrow a[x, y, z] \oplus (a[x+1, y, z] \oplus 1)a[x+2, y, z],$$
$$\iota : a[x, y, z] \leftarrow a[x, y, z] \oplus RC[i, x, y, z].$$

Let $P : \{0,1\}^n \to \{0,1\}^n$ be the n-bit Keccak-$f[200]$ permutation used in Delirium.

2.4.4 Encryption

Let $x \in \{0,1\}^n$ and $i \leq n$. We give the encryption algorithm of Elephant, as described by the specification [8], in Algorithm 1 and Figure 2.2.

Algorithm 1 Elephant encryption algorithm enc

Input: $(K, N, A, M) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^*$
Output: $(C, T) \in \{0,1\}^{|M|} \times \{0,1\}^t$

1: $M_1 \dots M_{\ell_M} \xleftarrow{n} M$
2: **for** $i = 1, \dots, \ell_M$ **do**
3: $\quad C_i \leftarrow M_i \oplus P(N||0^{n-m} \oplus \texttt{mask}_K^{i-1,1}) \oplus \texttt{mask}_K^{i-1,1}$
4: $C \leftarrow \lfloor C_1 \dots C_{\ell_M} \rfloor_{|M|}$
5: $A_1 \dots A_{\ell_A} \xleftarrow{n} N||A||1$
6: $C_1 \dots C_{\ell_C} \xleftarrow{n} C||1$
7: $T \leftarrow A_1$
8: **for** $i = 2, \dots, \ell_A$ **do**
9: $\quad T \leftarrow T \oplus P(A_i \oplus \texttt{mask}_K^{i-1,0}) \oplus \texttt{mask}_K^{i-1,0}$
10: **for** $i = 1, \dots, \ell_C$ **do**
11: $\quad T \leftarrow T \oplus P(C_i \oplus \texttt{mask}_K^{i-1,2}) \oplus \texttt{mask}_K^{i-1,2}$
12: $T \leftarrow P(T \oplus \texttt{mask}_K^{0,0}) \oplus \texttt{mask}_K^{0,0}$
13: **return** $(C, \lfloor T \rfloor_t)$

2.4.5 Decryption

Let $x \in \{0,1\}^n$ and $i \leq n$. We give the decryption algorithm of Elephant, as described by the specification [8], in Algorithm 1.

Algorithm 2 Elephant decryption algorithm dec

Input: $(K, N, A, C, T) \in \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^t$
Output: $M \in \{0,1\}^{|C|}$ or $\perp$

$C_1 \dots C_{\ell_M} \xleftarrow{n} C$
for $i = 1, \dots, \ell_M$ **do**
    $C_i \leftarrow M_i \oplus P(N||0^{n-m} \oplus \texttt{mask}_K^{i-1,1}) \oplus \texttt{mask}_K^{i-1,1}$
$M \leftarrow \lfloor M_1 \dots M_{\ell_M} \rfloor_{|C|}$
$A_1 \dots A_{\ell_A} \xleftarrow{n} N||A||1$
$C_1 \dots C_{\ell_C} \xleftarrow{n} C||1$
$\bar{T} \leftarrow A_1$
for $i = 2, \dots, \ell_A$ **do**
    $\bar{T} \leftarrow \bar{T} \oplus P(A_i \oplus \texttt{mask}_K^{i-1,0}) \oplus \texttt{mask}_K^{i-1,0}$
for $i = 1, \dots, \ell_C$ **do**
    $\bar{T} \leftarrow \bar{T} \oplus P(C_i \oplus \texttt{mask}_K^{i-1,2}) \oplus \texttt{mask}_K^{i-1,2}$
$\bar{T} \leftarrow P(\bar{T} \oplus \texttt{mask}_K^{0,0}) \oplus \texttt{mask}_K^{0,0}$
return $\lfloor \bar{T} \rfloor_t = T \ ? \ M : \perp$

2.5 eXtended Keccak Code Package

The eXtended Keccak Code Package or Xoodoo and Keccak Code Package (both abbreviated as XKCP) is a collection of open-source implementations of the cryptographic schemes defined by the Keccak team [1], this includes the Keccak sponge function family.

The low-level services of XKCP implement the permutations Keccak-f[200 to 1600] and Keccak-p[200 to 1600]. In Keccak-f the number of rounds is fixed while for Keccak-p this is a parameter, this is denoted by Keccak-$p[x, y]$ with x the width of the permutation and y the amount of rounds. We focus on Keccak-f[200] and the closely related Keccak-p[200] where Keccak-p[200, 18] is equivalent to the Keccak-f[200] instance [4].

2.6 Assembly and ARM

Assembly is a low-level programming language where there is a one-to-one relationship between assembly language instructions and the binary opcode executed by the core. ARM is an assembly language first designed by Intel

in 1974 for an 8-bit processor and has been extended to 16, 32, and 64-bit forms [10].

The ARM instruction set has features not found in some processors, while at the same time lacking operations found in others. An example of this is the ROR operation which uses a cyclic shift on a register to the right, which can be found in ARM but not on some standard processors.

The 32-bit processor architecture ARMv7 was introduced for microcontroller implementations, with its strength in high-performance. ARMv7-M is one of the three defined architecture profiles of ARMv7.

Chapter 3

Byte Implementation

3.1 Implementations

3.1.1 Reference Implementation

The specification of Elephant [8] mentions the implementation of Elephant implemented by Beyne [7]. This implementation contains six versions, from which we take elephant200v2 as the reference implementation. The reference implementation is an implementation of Elephant v2's Delirium in C99, using Keccak-$f[200]$ as the permutation function.

3.1.2 Byte Implementation

We created our own implementation of Elephant [9] which we refer to as the byte implementation. The byte implementation is based on the encryption and decryption functions given in Algorithm 1 and Algorithm 2, where the Keccak-f permutation is taken from the reference implementation.

3.2 Comparing Implementations

After writing the byte implementation, we compared it with the reference implementation. We note certain differences in the implementation and design choices:

1. The general structure differs between the byte and reference implementation. This is due to both implementations merging the FOR loops in the algorithm in different ways.

2. Algorithm 1 described the output of the encryption function as $(C, T) \in \{0,1\}^{|M|} \times \{0,1\}^t$.
   The byte implementation outputs two separate buffers for both the

ciphertext and tag. The reference implementation outputs a single buffer, which can be described as $(CT) \in \{0,1\}^{|M|+t}$. CT is a concatenation of the ciphertext and tag.

3. Certain design choices can cause differences in performance between the reference and byte implementation. One of these choices is the way the LFSR cycles are stored. Let $LFSR_x$ with $x \in \{prev, curr, next\}$ denote the last, current, and next LFSR cycle.
   The byte implementation copies the states down from $LFSR_{next}$ to $LFSR_{prev}$, it then cycles the LFSR and stores the result in $LFSR_{next}$. The reference implementation cycles the pointers of the buffers. This results in a decrease in memory copied per cycle, since instead of copying a state (25 bytes) down, it copies a pointer (8 bytes) down.

3.2.1 Performance

We measure the speedup of our byte implementation over the reference implementation using the test vector described in Testing and Measurements (Ch. 4). The result is given in Figure 3.1.

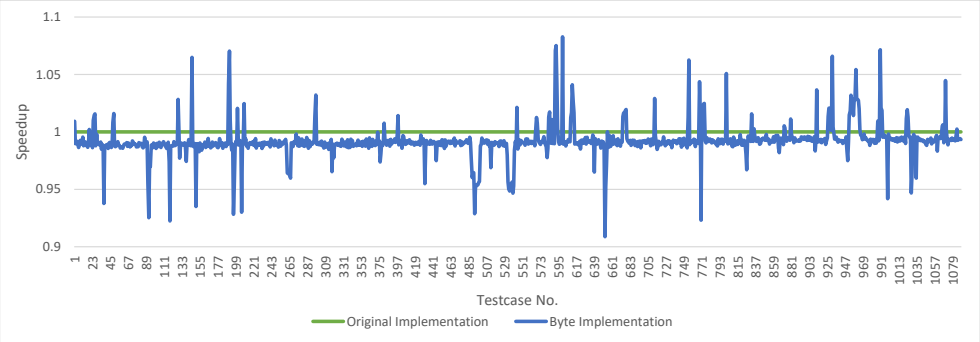

Figure 3.1: Speedup (y-axis) per testcase (x-axis) of the byte implementation over the reference implementation.

Figure 3.1 shows an average speedup of 0.992 for our byte implementation, which is not a significantly worse performance than the official Elephant implementation. This difference in performance is possibly caused by the design differences mentioned (Sec. 3.2).

We continue to use the byte implementation as the base for other implementations and optimizations.

Chapter 4

Testing and Measurements

In this chapter, we discuss how we did our measurements. We discuss what our test vectors are, how we ran our test vectors including a test function, how we dealt with inconsistency in measurements, and in what environment measurements were done. The reference implementation (Sec. 3.1.1) by Beyne contains the *LWC_AEAD_KAT_128_96.txt* file [7]. This file contains a 1089 sample long test vector which we use for our measurements. The message and additional data length in the testcases each range from 0 to 32 bytes, this gives us a total of 1089 testcases in the test vector.

We verify the output of each testcase with the test vector to make sure our implementation gives an output consistent with the reference implementation.

4.1 Running the test vectors

To run the test vector, we designed a function named runTests included in the "my_implementation_test" file, which we refer to as the test function. We describe runTests in the appendix (App. A.1).

Let $count, repeat \in \mathbb{N}$ and $input\_file, output\_file \in \{0,1\}^*$. The test function takes the first *count*-number of tests from the input file ($input\_file$). It then encrypts every testcase in $input\_file$ *repeat*-number of times, meanwhile calculating the average encryption time per testcase in microseconds (μs). Finally, it writes the results to the output file ($output\_file$) together with the average time per encryption, number of correct testcases, and name of the tested implementation.

We use the results of the test function to compare the average encryption time per testcase between implementations.

4.2 Inconsistency in measurements

The average encryption time per testcase is measured in microseconds. Due to this being an incredibly small measurement unit, it can be affected by external factors such as processes running in the background of the testing system. We repeat the encryption on a testcase *repeat*-number of times to decrease the inconsistency in results and get a more precise average encryption time per testcase.

To get consistent measurements, we determine a set value for *repeat*. As the value of *repeat* increases, the overall time it takes to compute the results increases with it. To find the optimal value for *repeat*, we conduct a test on the accuracy of the results.

4.2.1 Accuracy Test

Let $repeat \in \{10, 10^2, 10^3, 10^4, 10^5\}$ and our testcase be the testcase #1089 of the test vector. This is a test case consisting of a 16 byte key, 12 byte nonce, 32 byte message, and 32 byte additional data. We run a hundred tests with *repeat*-number of encryptions and plotting these as a hundred data points in a graph. We can compare the spread of these data points to deduce the accuracy.

Let for each value of *repeat*: $i \in \{0, \ldots, 99\}, total\_time_i \in \mathbb{R}$, and y_i be the value of the i^{th} data point as average encryption time per byte, where $total\_time_i$ is the sum of the encryption time over *repeat*-number encryptions for data point i. The test vector has varying message and additional data lengths (*mlen* and *adlen* respectively) between testcases, while the key and nonce stay constant throughout all testcases. Due to this we may assume that the *mlen* and *adlen* are the main factors causing a difference in encryption time from testcase to testcase.

Let average encryption time per byte mean the encryption time for a testcase divided by the total of the main factors' lengths ($mlen + adlen$). We calculate the average encryption time per byte (μs) for each data point y_i,

$$y_i = \frac{total\_time}{repeat * (mlen + adlen)}.$$

The results of this test are given in Figure 4.1.

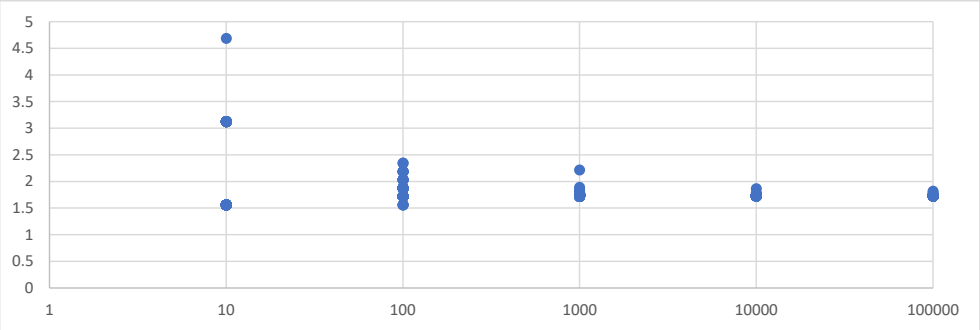

Figure 4.1: Results of the accuracy test (Sec. 4.2.1) with the number of repetitions (x-axis) plotted in the average encryption time per byte in μs (y-axis).

We conclude from Figure 4.1 that the *repeat* values of 10^3 and up have an accuracy of around one decimal. Due to the time it takes to perform the tests we take 10^3 as our *repeat* value, this gives us a sufficiently accurate measurement in a reasonable amount of time. However a single outlier can be seen for this value of *repeat* in the figure, this could explain certain outliers in our results.

4.3 Impact of Inputs

As explained in Accuracy Test (Sec. 4.2.1) the main factors causing the difference in encryption time between testcases are the message length ($mlen$) and additional data length ($adlen$). To explain patterns in the results, we want to know which of these two factors has the biggest effect on the average encryption time per byte. We conduct two tests and compare the results:

4.3.1 Impact Test

Let $i, j \in \{0, \ldots, 99\}$, m_test, ad_test be two test vectors, x_i be the i^{th} testcase in m_test, and y_j be the j^{th} testcase in ad_test.

Then take for each testcase $x_i \in m\_test$ the values: $adlen = 32$ bytes and $mlen = i$ bytes. This means we get 100 testcases where $adlen$ remains constant at 32 bytes while $mlen$ ranges from 0 to 99 bytes.

Similarly, take for each testcase $y_j \in ad\_test$ the values: $mlen = 32$ bytes and $adlen = j$ bytes. This gives us 100 testcases, where $mlen$ remains constant while $adlen$ ranges from 0 to 99 bytes.

We use the runTests function (Sec. 4.1) to run the m_test and ad_test test

vectors, we take $count = 100$ and $repeat = 10^4$. The results of the test vectors are plotted in Figure 4.2 in average encryption time per byte (μs).

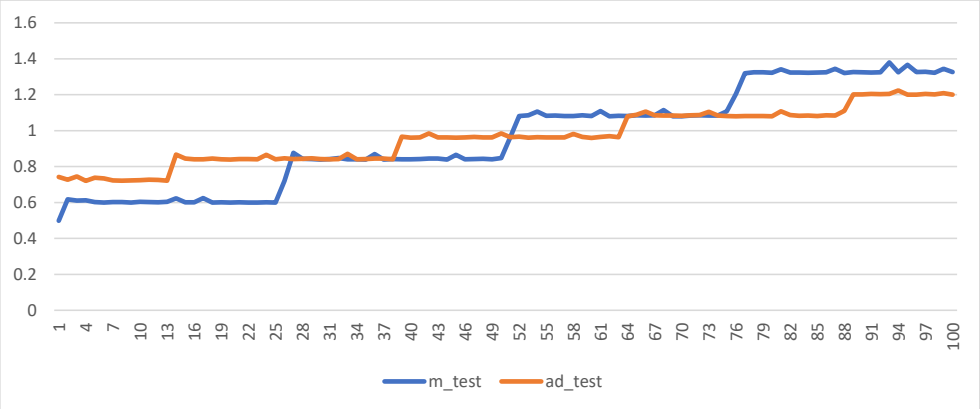

Figure 4.2: Results of the Impact Test (Sec. 4.3.1) with the testcases (x-axis) plotted in the average encryption time per byte in μs (y-axis).

Figure 4.2 shows that *m_test* starts at a lower average encryption time than the *ad_test*, but ends up at a higher encryption time than *ad_test*. From this, we can conclude that *mlen* has a greater effect on the encryption time than *adlen*.

4.4 Testing Environment

The testing environment we chose is the Linux distribution Ubuntu, specifically Ubuntu 64bit version 22.04.01, which we run in a virtual machine. Our virtual machine of choice is VMWare Workstation 16 [17]. The test system's specifications are as follows:

```
Architecture: x86_64
    CPU op-mode(s): 32-bit, 64-bit
Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
16,0 GB of RAM.
```

QEMU is a free and open-source emulator. Setting up a simulator for an ARMv7-M processor, such as an ARM Cortex-M3, to conduct our tests on falls outside our scope. However, we can use the qemu-user-static package to directly run ARM executables in our Linux environment.

We install the qemu-user-static package, this allows us to run ARM executables directly on Linux. After which we use arm-linux-gnueabihf-gcc -march=architecture -static -O2 -o obj_out *.c to compile our implementation directly into a 32-bit executable for ARM, where -O2 tells GCC to try to op-

timize for the target architecture. We run the executable directly on Linux using the previously mentioned qemu-user-static package.

One issue we ran into while using the QEMU package was that running an ARMv7-M specific executable gives us an illegal instruction error. While testing, we use -march=armv7 instead of -march=armv7-m. This means we run ARM executables using ARMv7 instructions, however our results are still representative for ARMv7-M.

Chapter 5

32-Bit Implementation

The first step to optimizing the byte implementation is to transform it into a 32-bit integer implementation, which we refer to as the int implementation. In this chapter, we explain our reasoning for creating this implementation and the changes we made.

5.1 Reasoning

Elephant is an authenticated encryption scheme. The Delirium instance works with blocks of 200 bits (25 bytes) [8]. The masking function (Sec. 2.4.2) computes the mask using several operations on these blocks. We are interested in seeing if converting these byte blocks into 32-bit integer blocks gives us a change in performance.

The eXtended Keccak Code Package (XKCP) [1] includes various third-party implementations of Keccak. One of these implementations is Keccak-p optimized for ARMv7-M, which we make use of in the optimization of Keccak (Ch. 7). Keccak-p optimized for ARMv7-M is a 32-bit implementation. This means that it does not fit our byte implementation (Sec. 3.1.2), but it does fit our 32-bit int implementation.

5.2 Implementation Changes

The first step to creating the int implementation, is to transform the byte blocks into 32-bit integer blocks. The byte implementation uses, as mentioned, byte buffers with a length of 25, we convert these into 7 int long buffers.

5.2.1 Converting Buffers

Let $bytebuffer \in \{0,1\}^{200}$, $intbuffer \in \{0\}^{224}$, $i \in \{0,\ldots,24\}$, $j \in \{0,\ldots,6\}$, $byte_i \in \{0,1\}^8$, $int_j \in \{0,1\}^{32}$ where $byte_i$ is the i^{th} byte in *bytebuffer* and int_j is the j^{th} int in *intbuffer*.

We can fit 4 bytes into a single 32-bit integer by putting the bytes in sequence, creating a 32-bit value. Because of this we can fit $\{byte_0, \ldots, byte_{23}\}$ into $\{int_0, \ldots, int_5\}$, this leaves us with $byte_{24}$ which we use as the 8 most significant bits of int_6 followed by 24 trailing zeroes.

A visual representation of this can be seen in Figure 5.1.

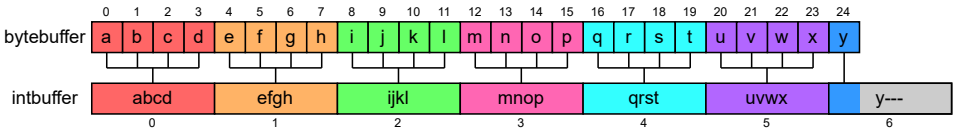

Figure 5.1: Visual representation of converting a 25 long byte buffer to a 7 long 32-bit integer buffer.

Using this method we convert the block, LFSR, tag, and mask buffers to 7 long 32-bit integer arrays.

5.2.2 The XOR function

The byte implementation defines the block_xor function, which takes two byte buffers and XORs their contents, iterating over the 25 bytes in the buffer. We define a similar function xor_int which XORs the contents of two int arrays, iterating over the 7 ints in the buffer.

5.3 Results

We measure the speedup of the int implementation over the byte implementation (Sec. 3.1.2). The methods, environment, and test vector used are described in Testing and Measurements (Ch. 4). The results of the measurements are plotted in Figure 5.2.

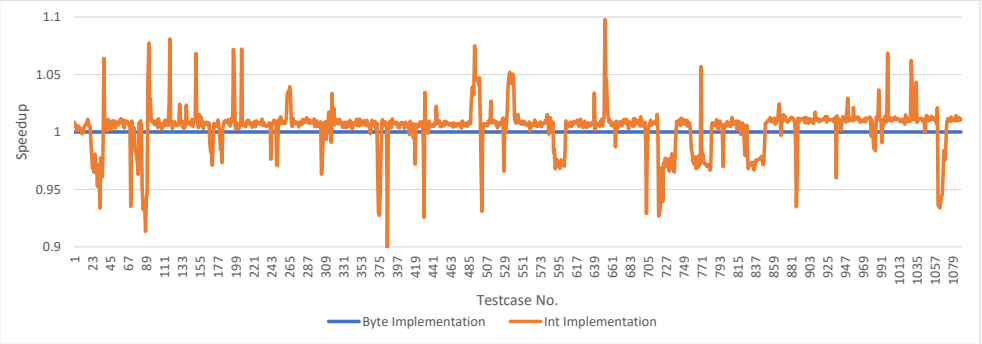

Figure 5.2: Speedup (y-axis) per testcase (x-axis) of the int implementation over the byte implementation of Elephant.

Figure 5.2 shows a minor speedup of the int implementation over the byte implementation. There are obvious spikes both above and below 1, we assume these to be measurements affected by external factors (Sec. 4.2).

There is a visible increase in the speedup of the int implementation as the testcases progress. To find the cause of this increase in speedup, we look at specific testcases in the test vector and their encryption time. We plot the average encryption time (μs) per testcase over 10^3 encryptions in Figure 5.3.

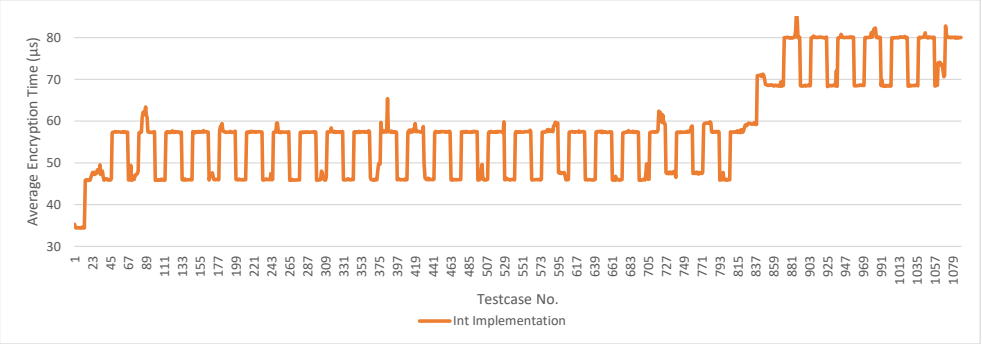

Figure 5.3: Average encryption time per testcase (x-axis) for the int implementation in μs (y-axis).

We look at two specific testcases from the test vector used by the official Elephant implementation [7]. These are the 33^{th} and 859^{th} testcases.

As mentioned before, the Delirium instance of Elephant uses a block size of 200 bits [8]. From the 33^{th} testcase and onwards, the message length is

> 0 bits. This means all testcases after the 33^{th} have at least 1 message block. From the 859^{th} testcase and onwards, the message length is > 200 bits. This gives us at least 2 message blocks for the 859^{th} testcase and up. We know from our Impact Test (Sec. 4.3.1) that the message length has a greater effect on the encryption time than the additional data length.

Figure 5.3 shows a sharp increase in encryption time after both the 33^{th} and 859^{th} testcases. For each message block, the algorithm uses the previously described xor_int function (Sec. 5.2.2) a total of 4 times.

Looking back at the speedup graph (Figure 5.2), we can see that the speedup increases at around the 33^{th} and the 859^{th} testcases. These are the testcases where the amount of message blocks increase, meaning an increase in the usage of the xor_int function. We can now assume that as the message length increases, the speedup of the int implementation from the byte implementation increases.

Chapter 6

Optimization Using ARM Instructions

In this chapter, we discuss ARM-specific optimization methods attempted on the int implementation (Sec. 5). The ARM instruction set contains instructions not found in certain architectures, while at the same time lacking certain other instructions. We try to reduce the amount of instructions in the ARM assembly using ARM data-processing instructions. Data-processing instructions are the fundamental arithmetic and logical operations of the processor operating on values in the registers [2].

The compiler we use to create our 32-bit ARM executables (Sec. 4.4) makes an attempt to translate the C code into ARM assembly, it does so with CPU-specific instructions in mind. We want to know if we can improve this translation of the implementation in ARM assembly.

To look for improvements, we need to be able to read the ARM executable code. We can create a readable ARM assembly file from this executable using `arm-none-eabi-objdump`, `objdump` displays the information of object files [15]. We choose to set the `-D` flag, this disassembles the executable into readable ARM assembly.

6.1 Optimization Options

We discuss certain options used to try and optimize the int implementation, specifically the masking function, in ARM assembly using ARM data processing instructions [2]. To implement these instructions in our C code, we use inline assembly. Using the `asm` keyword, we can embed assembler instructions within the C implementation. GCC allows for two types of `asm` statements, `asm` with no operands and `extended asm` with one or more

operands. For example: Let $a \in \{0,1\}^{32}$ be a 32-bit integer, we can implement the ARM Logical Shift Left (LSL) instruction using inline assembly. We give an example, in which we shift the integer a over 7 bits to the left:

```
__asm__ ("LSL %0, %0, #7" :: "r" (a));.
```

The documentation for Cortex-M3 (An ARMv7-M CPU) specifies a time per operation given in clock cycles [3]. It notes that a shift, bit-field, or data operation takes a single clock cycle to execute. Data operations take 2 clock cycles if the destination register is the PC register, or 1 clock cycle for other registers. In ARM, the Program Counter (PC) is a register which stores the memory address of the next instruction to be executed.

6.1.1 Shift Instructions

The masking function of Elephant (Sec. 2.4.2) uses an LFSR φ_1. When φ_1 calculates the least-significant bit of the next state, it uses two circular left shifts and a single logical left shift. While C defines the $\ll$ operator for a logical left shift, it does not define an operator for a circular left shift $\lll$. This is why the int implementation defines its own function Rotate Left (ROTL):

```
uint8_t ROTL(uint8_t a)
{
    return (a<<1) | (a>>7);
}.
```

Looking at the ARM assembly code, we can see that $(a \ll 1 | a \gg 7)$ results in three instructions. These instructions are a Logical Left Shift (LSLS), Bitwise OR (ORRS), and a Logical Right Shift (LSRS), all of these instructions fall into the data processing operations category [3]. Since the destination register of the operations is not the PC register, we may assume that each of these operations takes 1 clock cycle to execute.

Another operation included in this list of data processing operations is Rotate Right (ROR), which for a Cortex-M3 CPU takes a single clock cycle as well. We tried using this ROR operation to replace each use of ROTL using the following asm inline statement:

```
uint8_t a;
__asm__ ("LSL %0, %0, #7" :: "r" (a));/
```

We use a cyclic right shift of 7 bits, since on a byte this would equal a cyclic left shift of 1 bit. However, due to ARM using 32-bit registers, this cyclic right shift is not applied on just the byte but on the whole register. A visual representation of a 1 bit rotate right (ROR) on a 32-bit register is shown in

Figure 6.1. As shown in Figure 6.1 the right-most bit of the byte is rotated out of the 8 bit representing the byte.

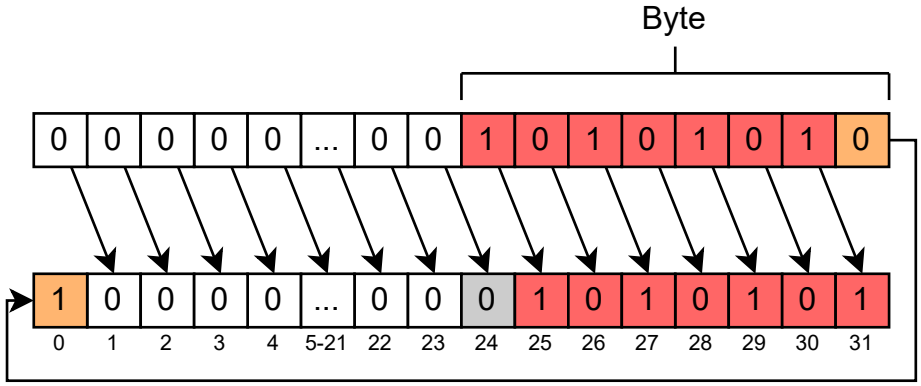

Figure 6.1: A rotate right (ROR) operation used on a 32-bit register containing a byte in little-endian byte order.

ARM does not define a byte-wise rotation, and thus this optimization attempt can not be applied on our current implementation.

6.1.2 Parallel Instructions

Another category of data-processing instructions for ARM is parallel instructions [2]. We specifically look at parallel add and subtract instructions, such as byte-wise addition ADD8 or byte-wise subtraction SUB8. Parallel instructions have variants such as SHADD8, which first uses byte-wise addition and then halves the result. In certain cases, using these parallel instructions, it is possible to combine two instructions into a single operation.

Looking at the ARM assembly code, we can see several mentions of UADD8 and UQSUB8 instructions, variants of ADD8 and SUB8. This means the compiler (Sec. 4.4) already makes use of these instructions when compiling for ARMv7 where possible.

6.2 ARM Conclusion

We are thus able to see that the compiler (Sec. 4.4) makes use of ARM specific instructions where possible in the implementation. This means we are not able to optimize the masking function more than already done by the compiler using just ARM instructions.

Chapter 7

Optimization of Keccak

In the previous chapter (Ch. 6) we discussed potential optimizations using the ARM assembly, mostly aimed at the masking function. We now take a look at the permutation function, which in the case of the int implementation (Ch. 5) is Keccak-f[200].

Instead of optimizing Keccak ourselves, we consider existing implementations and introduce them into the int implementation of Elephant. The official Keccak site [6] lists several software implementations of Keccak. It is here that we can also find the eXtended Keccak Code Package (XKCP).

We look for alternatives to Keccak-f[200], one of these is the Keccak-p permutation. The XKCP contains a reference, compact, and an ARMv7-M-optimized implementation of Keccak-p[200]. We look into these three implementations, making comparisons and implementing them into our int implementation of Elephant. Both the ARMv7-M-optimized and compact Keccak-p implementations are implemented by Van Keer.

The Keccak-p[200] implementations in the XKCP each define the KeccakP200_Permute_18rounds function, the output of this function is equivalent to the permutation function included in the original Keccak-f[200] implementation.

7.1 Keccak Comparison

7.1.1 Original Keccak-f and Reference Keccak-p

First we compare the original Keccak-f implementation, used in the official Elephant implementation [7], with the reference Keccak-p implementation from the XKCP [1].

Both the original Keccak-f and reference Keccak-p use the same functions

and variable naming. Elephant's Keccak-f implementation is based on the Keccak Team's Keccak-f implementation. This explains why the Keccak Team's Keccak-p implementation has the same design and variables as Elephant's Keccak-f implementation.

We thus expect both implementations to have a similar performance.

7.1.2 Reference Keccak-p and Compact Keccak-p

Second, we compare the reference and compact Keccak-p implementations from the XKCP. In this section, we refer to reference Keccak-p as reference and compact Keccak-p as compact.

We list the following differences:

- The first difference is the amount of function calls used. The reference's KeccakP200_Permute_18rounds calls KeccakP200OnWords, which then calls KeccakP200Round a total of 18 times. Compact immediately calls the KeccakP200_Permute_Nrounds function, which in the function itself computes n-rounds. For each round of the reference the Keccak steps (Theta, Rho, Pi, Chi, Iota) are called as separate functions, whereas compact makes no function calls and has the primitives included in the function itself.
- The second difference is the usage of the index(x, y) function. Reference uses the index(x, y) function to compute the index of a certain byte in the state. The index(x, y) function is defined as follows:

  $$\mathsf{index(x,\ y)} = (((\mathsf{x})\%5)+5*((\mathsf{y})\%5))$$

  Compact uses a different structure to reference when computing its rounds. Compact uses a buffer of 5 bytes called BC which is used to store 5 bytes from the state, and applies all Keccak steps on these bytes before putting them back into the state. The BC buffer makes it possible for compact to get rid of the index(x, y) function, reducing both computations and function calls.
- The third difference is compact's combination of the Keccak steps Rho and Pi into Rhopi. Reference's Rho and Pi use a combined total of three loops, compact's Rhopi makes use of the previously mentioned BC buffer and reduces the number of loops to one.
- The fourth difference is the use of pre-computed values. Compact uses buffers containing pre-computed values to reduce computations done in the Keccak steps. An example of this is the KeccakP200_Mod5 buffer. Let $x \in \{0, \ldots, 9\}$, KeccakP200_Mod5 contains the MOD 5 value for each value of x.

Reference makes regular use of the %5 or MOD 5 operation. By using the KeccakP200_Mod5 buffer, compact reduces the amount of computations done in the Keccak steps.

Because of these differences, the overall number of address calls and computations is reduced in the compact Keccak-p implementation compared to the Keccak-p implementation. This gives us reason to expect compact Keccak-p to have a higher performance than reference Keccak-p in our results (Ch. 8).

7.1.3 Reference Keccak-p and ARMv7-M Keccak-p

Third, we compare the reference and ARMv7-M-optimized Keccak-p, we refer to the latter as ARMv7-M Keccak-p.

Because the ARMv7-M Keccak-p implementation is written in assembly code, we only mention certain observations we made. One of the differences observed is that the Keccak steps Rho and Pi are combined into Rhopi, this is similar to how the compact Keccak-p implementation handles the Keccak steps (Sec. 7.1.2).

Another difference observed is the size of the values used in the state buffer. Both the reference and compact implementation convert the state to a buffer of 8-bit values, whereas the ARMv7-M implementation uses a buffer of 32-bit values. This means that operations can be done on 32 bits of the state at once, possibly allowing the calculation of the 200-bit state in fewer steps.

Keeping in mind that the ARMv7-M implementation is specifically optimized for our target architecture, we can expect ARMv7-M Keccak-p to have a higher performance than reference Keccak-p in our results. Due to the similarity with the compact implementation, the usage of a 32-bit value buffer, and usage of ARM instructions, we can expect the ARMv7-M Keccak-p to have a higher performance than compact Keccak-p in our results.

7.2 Implementing Keccak-p

We implement each of the three previously mentioned Keccak-p implementations (reference, compact, ARMv7-M) into our int implementation of Elephant (Ch. 5).

The XKCP contains the files needed to build each implementation in the repository [1]. In the case of the reference, compact, and ARMv7-M Keccak-p[200] implementations, this is the header file named KeccakP-200-SnP.h.

The contents of this file differ between implementations, which means we have three separate versions of KeccakP-200-SnP.h.

In addition to this, the compact and reference implementations use the brg_endian.h header file designed by Gladman (Acknowledgments in the XKCP [1]), which determines the byte order of the platform. The brg_endian.h header file is not needed in the ARMv7-M implementation, since it uses the predetermined little-endian byte order.

We replace the permutation function from the original Keccak-f[200] with the Keccak-p[200] function KeccakP200_Permute_18rounds. These two functions give an equivalent output when given the same state.

We give three implementations of Elephant, each using a different Keccak-p implementation as the permutation function. All three of these implementations are based on the 32-bit int implementation (Ch. 5).

7.2.1 Reference Keccak-p implementation

We create a new implementation based on our int implementation of Elephant, which we refer to as the reference Keccak-p implementation. This implementation of Elephant uses the reference Keccak-p permutation function. The reference implementation of Keccak-p is similar to the Keccak-f[200] implementation included in Elephant v2 [7].

7.2.2 Compact Keccak-p implementation

We create a new implementation based on our int implementation of Elephant, which we refer to as the compact Keccak-p implementation. This implementation of Elephant uses the compact Keccak-p permutation function. The compact implementation of Keccak-p uses certain methods to reduce the size, amount of address calls, and computations done in the implementation.

7.2.3 ARMv7-M Keccak-p implementation

We create a new implementation based on our int implementation of Elephant, which we name the ARMv7-M Keccak-p implementation. This implementation of Elephant uses the ARMv7-M Keccak-p permutation function. The ARMv7-M implementation of Keccak-p is designed using ARM instructions and techniques.

7.3 Keccak-p Measurements

Using the method and test vector described in Testing Environment (Sec. 4.4), we measure the average encryption time of the reference (Sec. 7.2.1),

compact (Sec. 7.2.2), and ARMv7-M (Sec. 7.2.3) Keccak-p implementations of Elephant.

We measure the average encryption time per testcase in microseconds (μs) and include the measurements of the int implementation (Ch. 5) for comparison. These results can be seen in Figure 7.1.

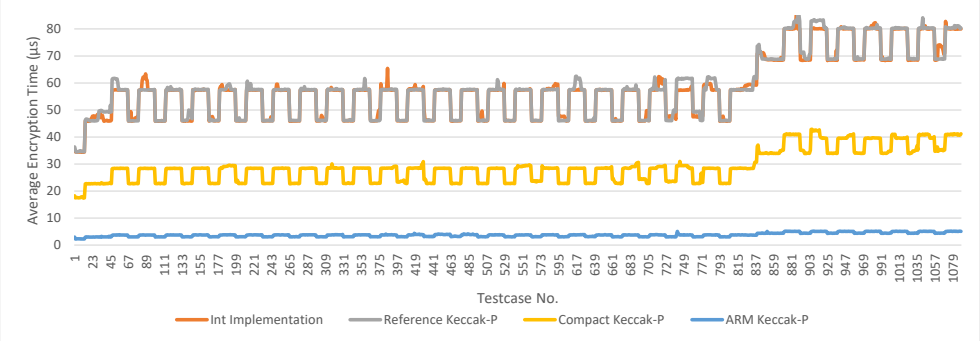

Figure 7.1: Average encryption time per testcase (x-axis) for the reference (Sec. 7.2.1), compact (Sec. 7.2.2), and ARMv7-M (Sec. 7.2.3) Keccak-p implementations of Elephant in μs (y-axis).

By observing the results in Figure 7.1 we can deduce that both the compact and ARMv7-M Keccak-p implementations have a lower encryption time than both the int Keccak-f and reference Keccak-p implementations. The ARMv7-M Keccak-p implementation has the highest performance, as expected from our comparisons (Sec. 7.1).

Chapter 8

Results

In this chapter, we compare the performance of all the implementations mentioned. These are the byte (Sec. 3.1.2), int (Ch. 5), Keccak-p Reference (Sec. 7.2.1), Keccak-p Compact (Sec. 7.2.2), and Keccak-p ARMv7-M (Sec. 7.2.3) implementations.

Using the methods and test vector described in Testing and Measurements (Sec. 4), we measure the encryption time per testcase in microseconds (μs) for all previously mentioned implementations. The results of these measurements are given in Figure 8.1.

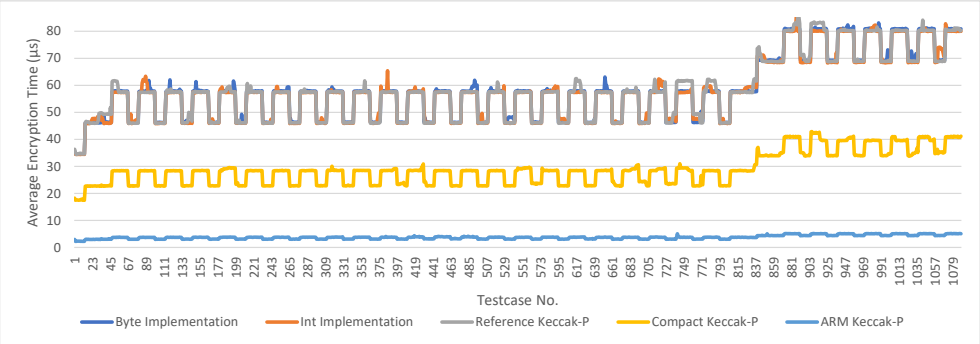

Figure 8.1: Average encryption time per testcase (x-axis) for all implementations in μs (y-axis).

We can put the results from Figure 8.1 into a speedup graph, where the speedup over the byte implementation is given for each implementation mentioned. The speedup for each testcase can be seen in Figure 8.2.

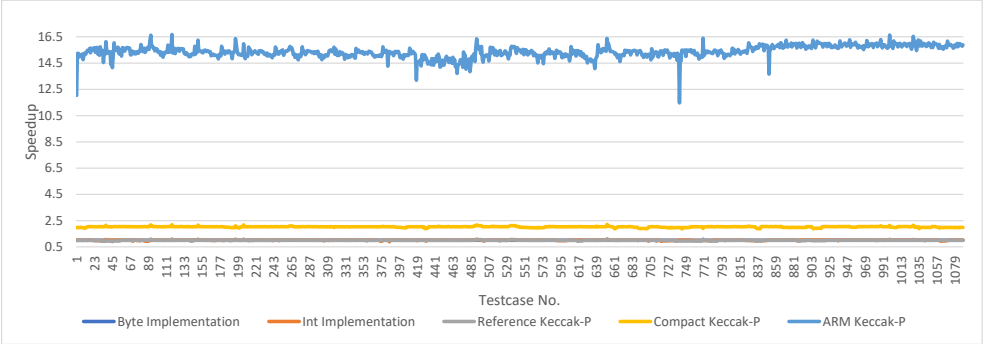

Figure 8.2: Speedup (y-axis) per testcase (x-axis) of each implementation over the byte implementation.

In Table 8.1, we give the average encryption time and speedup of the implementations over all testcases.

Implementation	**Average Encryption Time (μs)**	**Average Speedup**
Byte Implementation	58.2847	1 (reference)
Int Implementation	58.0452	1.0039
Reference Keccak-p	58.3005	0.9999
Compact Keccak-p	28.8663	2.0203
ARMv7-M Keccak-p	3.7815	15.3696

Table 8.1: Average encryption time and speedup per implementation over all testcases in the test vector.

Although Table 8.1 shows a similar performance for both the byte and int implementations, we expect the int implementation to have a higher performance for longer messages or additional data (Sec. 5.3).

We can conclude from the results in Table 8.1 that the ARMv7-M Keccak-p implementation is the best performing implementation, with an average speedup of around 15.37 over our initial implementation (the byte implementation).

8.1 Final Implementation

Based on the results of each implementation, we take the ARMv7-M Keccak-p implementation (Sec. 7.2.3) as our final implementation.

Chapter 9

Conclusions

We conclude that our final implementation of Elephant v2 (Sec. 8.1) is a successful optimization from the original Elephant v2 implementation. Our final implementation has a speedup of around 15 over our initial implementation on the ARMv7 architecture.

Changing Elephant from an 8-bit into a 32-bit implementation is a successful optimization. The 32-bit implementation has a higher performance for large input lengths than an 8-bit or byte implementation (Sec. 5.3).

We can now conclude that both the conversion to 32-bit and the use of an alternative permutation function cause a speedup of around 15 over the initial implementation. This answers our research question: How can we optimize the cryptographic cipher Elephant v2 for the ARMv7-M architecture?

This concludes our research. Further research could be done by optimizing the LFSR to work with a 32-bit state, optimizing the other instances of Elephant, or by porting the implementation to the 64-bit ARMv8 architecture.

In our research, we measured the performance of our implementations as encryption time in microseconds. In further research, the performance can be measured in CPU cycles to decrease the amount of outliers in the results.

The project can be found on GitHub, which includes the final implementation: `https://github.com/noboud/Elephant`.

Bibliography

[1] *XKCP/XKCP: eXtended Keccak Code Package*. https://github.com/XKCP/XKCP.

[2] ARMDeveloper. *Data processing instructions*. Documentation: https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/The-ARMv7-M-Instruction-Set/Data-processing-instructions.

[3] ARMDeveloper. *Processor instruction timings*. For Cortex-M3 (ARMv7-M): https://developer.arm.com/documentation/ddi0337/e/Instruction-Timing/Processor-instruction-timings.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak-p permutations*. https://keccak.team/keccakp.html.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak reference*, January 2011. http://keccak.noekeon.org/.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Software resources*. https://keccak.team/software.html.

[7] Tim Beyne. *Elephant AEAD Reference implementation*. https://github.com/TimBeyne/Elephant.

[8] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf, 2021.

[9] Norbert Boudens. *Elephant AEAD optimized for ARMv7*. https://github.com/noboud/Elephant.

[10] Cark Burch. *Introducing ARM assembly language*, October 2011.

[11] Rafael Cruz, Tiago Reis, Diego Aranha, Julio López, and Harsh Kupwade Patil. Lightweight cryptography on arm, 01 2017.

[12] Mauk Lemmen. *Optimizing Elephant for RISC-V*, March 2020. https://www.cs.ru.nl/bachelors-theses/2020/Mauk_Lemmen___4798937___Optimizing_Elephant_for_RISC-V.pdf.

[13] NIST. *About NIST*, October 2012. https://www.nist.gov/about-nist.

[14] NIST. *NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition*, October 2012. https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition.

[15] GNU Development Tools. *OBJDUMP*. https://manpages.debian.org/unstable/binutils-arm-none-eabi/arm-none-eabi-objdump.1.en.html.

[16] Pham Van Luc, Vo Tung Linh, Hoang Dang Hai, and Leu Duc Tan. Fast binary field mutiplication on armv7 embedded processors. In *2020 4th International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, pages 6–10, 2020.

[17] VMWare. *VMware Workstation Player Documentation*. Documentation: https://docs.vmware.com/en/VMware-Workstation-Player/index.html.

Appendix A

Appendix

A.1 Testing Algorithm

To describe the runTests algorithm, we define the functions:
read_from(i, IF) is a function that reads the key, nonce, message, additional data, ciphertext, and tag of the i^{th} testcase from the file IF.
write_to(i, enc_time, OF) is a function that writes the encryption time in microseconds for testcase i to the file OF.
clock() is a function that returns the current time in microseconds.
encryption(M, AD, N, K) is the authenticated encryption function taking a message, additional data, public nonce, and key resulting in a ciphertext and tag.

Using these functions, we give the runTests algorithm (Alg. 3).

Algorithm 3 The testing algorithm runTests

Input: $count \in \mathbb{N}, repeat \in \mathbb{N}, IF \in \{0,1\}^*, OF \in \{0,1\}^*$

```
1:  test_errors ← 0
2:  for i = 1, ..., count do
3:      K, N, M, AD, C, T ← read_from(i, IF)
4:      start ← clock()
5:      for j = 1, ..., repeat do
6:          C', T' ← encrypt(M, AD, N, K)
7:      stop ← clock()
8:      if (C ≠ C') ∨ (T ≠ T') then
9:          test_errors ← test_errors + 1
10:     enc_time ← (stop − start)/repeat
11:     write_to(i, enc_time, OF)
```