Bachelor thesis
Computing Science

Radboud University

Optimizing Elephant for RISC-V

Author:
Mauk Lemmen
S4798937

First supervisor:
Bart Mennink

Second supervisor:
Joan Daemen

March 12, 2020

Abstract

Elephant is an authenticated encryption scheme competing in the NIST lightweight cryptography competition. The mode of Elephant is a nonce-based encrypt-then-MAC construction. One of the instances of Elephant is Elephant-Keccak-f[200], also called Delirium, which uses Keccak as its permutation primitive. We optimize Delirium for a RISC-V RV32IMAC architecture by exploiting Elephant's possibility for parallelization by using a technique called bit-interleaving. This results in an average speedup of 237%.

Contents

1 Introduction **3**

2 Preliminaries **4**
2.1 Notation . . . 4
2.2 Authenticated encryption . . . 4
2.3 Linear-feedback shift register . . . 5
2.4 RISC-V . . . 5

3 Elephant **7**
3.1 Elephant construction . . . 7
3.2 Delirium instance . . . 9
3.2.1 Permutation . . . 9
3.2.2 LFSR . . . 10

4 Research **11**
4.1 Research angle . . . 11
4.2 Optimization . . . 11
4.2.1 Bit interleaving . . . 11
4.3 Implementation . . . 13
4.3.1 State representation . . . 13
4.3.2 State transformation . . . 13
4.3.3 Keccak-f[200] . . . 14
4.3.4 LFSR . . . 14
4.4 Benchmarks and results . . . 15
4.4.1 Permutation . . . 15
4.4.2 Varying message lengths . . . 16
4.5 Lower bound . . . 18
4.5.1 Theoretical amount of cycles . . . 18
4.5.2 Discussion . . . 19

5 Related Work **20**

6 Conclusions **21**

A Appendix **23**
A.1 How to set up the compiler 23
A.2 How to download and compile and run optimized Elephant . 24

Chapter 1

Introduction

Lately there has been great progress in the development of devices for the Internet of Things, smart devices and other electronic gadgets capable of communicating via the internet. One thing that lacks in those devices is a standardized cryptographic algorithm, capable of guaranteeing secure communication. Many of the current standardized cryptographic algorithms were not designed for the restricted environment of these devices, and thus often perform poorly with the processing power on board. This is why the US National Institute of Standards and Technology (NIST) has launched a competition to find and standardize lightweight algorithms suitable for these devices.

One of the algorithms participating in this competition is the authenticated encryption scheme Elephant. We implement and optimize Elephant for RISC-V. RISC-V is an open source instruction set architecture, meaning it can be freely examined, used and customized by anyone, including the manufacturers of Internet of Things and smart devices. RISC-V is based on the Reduced Instruction Set Principles, making it an ideal architecture for smaller devices.

A general implementation of Elephant exists, but not yet one optimized for a (32-bit) RISC-V based board. Having an optimized implementation is important, not only for faster-running code, but for reduced consumption of power, CPU and memory as well. This is an important property for devices with limited available resources.

In this paper we optimize Elephant, more specifically the Delirium instance of Elephant, on RISC-V.

Chapter 2

Preliminaries

In this chapter, we will explain several concepts that are necessary to fully understand this manuscript.

2.1 Notation

We will use the following notation in order to describe functions and definitions.
For $n \in \mathbb{N}$, we let $\{0,1\}^n$ denote the set of n-bit strings and $\{0,1\}^*$ the set of arbitrary length strings. For $X \in \{0,1\}^*$ we define

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

to be the function that partitions X into $\ell = \lceil |X|/n \rceil$ blocks of size n bits, where the last block is padded with 0s. The expression "$A?B : C$" equals B if A is true, and equals C if A is false. For $x \in \{0,1\}^n$ and $i \leq n$, we denote by $x \ll i$ (resp., $x \gg i$) a shift of x to the left (resp., right) over i positions. We likewise denote by $x \cap \quad i$ (resp., $\circ \quad i$) a rotation of x to the left (resp., right) over i positions. We denote by $\lfloor x \rfloor_i$ the i left-most bits of x.

2.2 Authenticated encryption

Authenticated encryption is a way of encryption that allows the recipient of the encrypted message to verify that the ciphertext has not been tampered with. By calculating a message authentication code (MAC) or tag over a key and input string, one can later verify the authenticity and integrity of that input string. This is done by recalculating the tag and checking if it matches the original tag. Authenticated encryption with associated data (AEAD) provides this property, but also allows for the verification of additional authenticated data sent along with the message. This additional data can be used to send along more information about the message, e.g.

the context in which the message is used. This prevents an attacker from replaying the encrypted message in a different context. An authenticated encryption scheme may have the following parameters as documented in the submission requirements for the Lightweight Cryptography competition [1]:

- Encryption:
  - input: key K, nonce N, additional data A, and message M.
  - output: ciphertext C and a tag T.
- Decryption:
  - input: key K, nonce N, additional data A, ciphertext C.
  - output: message M if tag T is correct and an error message otherwise.

The public message number should only be used once as it may be used to prevent replay attacks or as a nonce in the underlying encryption scheme. The main purpose of a nonce is to ensure confidentiality, for example by applying it to a ciphertext to add randomness. Nonces can also be used as a way to make sure previously sent ciphertexts cannot be used again, as it will become clear to the recipient of the messages that an already used nonce has been used again.

2.3 Linear-feedback shift register

A linear-feedback shift register (LFSR) is a circuit represented as a state of bits, where every cycle each bit value shifts to the right and some cells, also known as taps, get XORed with the other taps. The output bit is the rightmost bit, and the input bit is a function of the output bit and the taps. LFSRs are used amongst other things to generate pseudo-random numbers. Figure 2.1 is an example of an LFSR with taps on cells 29, 20, 16, 11, 8, 4, 3 and 2.

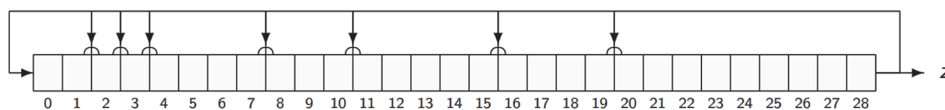

Figure 2.1: LFSR

2.4 RISC-V

RISC-V is an open-source instruction set architecture, originally developed as a university project in 2010 by Kryste Asanović at the University of California, Berkeley. The idea was to provide a free instruction set architecture

(ISA) for academic as well as industrial use. In 2015, the RISC-V Foundation was founded, and as of now comprises of more than 325 members, working on the project [2]. Already many implementations of RISC-V exist, with many more in development. In this thesis, we will be using the HiFive 1 development board designed by SiFive, using an E31 CPU with RV32IMAC support. RV32I means that the processor uses 32-bit instructions and registers, and MAC means that the processor supports integer multiplication and division (M), atomic mode for high performance portable software (A) and compressed mode which used compressed instructions for better code density (C).

Chapter 3

Elephant

3.1 Elephant construction

Elephant is an authenticated encryption scheme, created as an encryption algorithm for lightweight environments. It has three instances: Dumbo, Jumbo and Delirium. Each of these instances are parallelisable but also have a small state size. The Elephant mode is a nonce-based encrypt-then-MAC construction as depicted in Figure 3.1. It is permutation-based, and because only the forward direction of the permutation is used, no inverse of the permutation is needed. Each of the instances use an LFSR to generate the masks for the scheme. Dumbo and Jumbo both use Spongent as permutation, which performs well on hardware, while Delirium uses Keccak-f[200] as permutation and is more developed towards software use.

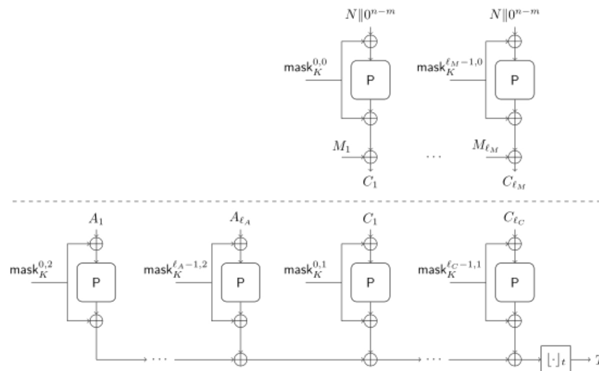

Figure 3.1: Elephant depiction

We now describe the generic authenticated encryption and decryption mode of Elephant as it is described in the original Elephant specification paper [3].

Let k, m, n, $t \in \mathbb{N}$ with k, m, $t \le n$. Let $P : \{0,1\}^n \to \{0,1\}^n$ be an n-bit permutation, and $\varphi_1 : \{0,1\}^n \to \{0,1\}^n$ be an LFSR. Define $\varphi_2 = \varphi_1 \oplus \mathrm{id}$. Define the function $\mathrm{mask} : \{0,1\}^k \times \mathbb{N}^2 \to \{0,1\}^n$ as follows:

$$\mathrm{mask}_K^{a,b} = \mathrm{mask}(K, a, b) = \varphi_2^b \circ \varphi_1^a \circ \mathrm{P}(K \parallel 0^{n-k}).$$

Encryption

The encryption algorithm, as described in Algorithm 1, gets as input key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^m$, associated data $A \in \{0,1\}^*$ and message $M \in \{0,1\}^*$, and outputs ciphertext $C \in \{0,1\}^{|M|}$ and tag $T \in \{0,1\}^t$.

Algorithm 1: Elephant encryption algorithm

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

Decryption

The decryption algorithm, as described in Algorithm 2, gets as input key $K \in \{0,1\}^k$, nonce $N \in \{0,1\}^m$, associated data $A \in \{0,1\}^*$ and ciphertext $C \in \{0,1\}^{|M|}$, and outputs message $M \in \{0,1\}^{|M|}$ if the tag is correct, or a $\perp$ sign otherwise.

Algorithm 2: Elephant decryption algorithm

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 $\perp$

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

3.2 Delirium instance

Delirium or Elephant-Keccak-f[200], is the third variant of Elephant, and the one we will optimize in this thesis. It has a state size of 200 bits represented as a 5-by-5-by-8 array. Delirium has an expected security strength of 2^{217} (measured in the offline complexity) and supports an online complexity of 2^{74} bytes, provided that the Permutation Keccak is pseudorandom and an attacker cannot evaluate the encryption function twice with the same nonce. The key and tag lengths are 128 bits, and the nonce size is limited to 96 bits.

3.2.1 Permutation

Similar to the Elephant specification paper, we will now describe Keccak-f[200]. The permutation, Keccak-f[200], signifies the 18-round Keccak permutation of Bertoni et al [4]. The state $X \in \{0,1\}^{200}$ is represented as a 5-by-5-by-8 array $a \in \{0,1\}^{5\times5\times8}$, where for $(x, y, z) \in \mathbb{Z}_5 \times \mathbb{Z}_5 \times \mathbb{Z}_8$ the bit at position (x, y, z) is set as

$$a[x, y, z] = X[8(5y + x) + z].$$

Keccak-f[200] operates on a 200-bit input X as follows:

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

Where the functions $\theta, \rho, \chi,$ and ι are defined as follows:

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

For ρ the function $t[x,y]$ is defined as

t	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

and for ι, the round constants are given by

$$RC[i,x,y,z] = \begin{cases} rc[j+7i], & (x,y,z) = (0,0,2^j-1), \\ 0, & \text{otherwise,} \end{cases}$$

where rc is computed from a binary LFSR.

3.2.2 LFSR

The masks for the scheme are generated by a simple LFSR defined by the primitive polynomial:

$$p(x) = x^8 + x^6 + x^5 + x^4 + 1.$$

and in terms of bitwise operations as:

$$(x_0, ..., x_{24}) \mapsto (x_1, ..., x_{24}, x_0 \cap \quad 1 \oplus x_2 \cap \quad 1 \oplus x_{13} \ll 1)$$

where each x_i stands for an 8-bit word.

Chapter 4

Research

4.1 Research angle

As mentioned in the specification paper, there is a high degree of parallelism possible when implementing Delirium. If there are multiple cores available, multiple blocks can be processed in parallel. If the processor uses 16-or-more-bit registers, multiple calls to the Keccak function can be combined. Because the HiFive 1 board uses 32-bit registers, this call-combination technique should definitely be possible to implement. Also, since the reference implementation released with the Elephant specification paper was designed to work on 8-bit processors as well as on processors with bigger CPU-word sizes, it should definitely be possible to optimize a solely 32-bit version by using the aforementioned technique.

4.2 Optimization

4.2.1 Bit interleaving

In theory

Being able to process several blocks in parallel can greatly increase efficiency in Elephant, and since we have 32-bit registers, this is possible. Normally we process blocks of byte-sized elements, but by making full use of the 32-bit registers, we can combine four blocks of byte-sized elements into one block of 4-byte elements. This way, our permutation makes full use of the available 32-bit (4-byte) registers, meaning we can process four blocks at the same time. To do this we use the concept of bit interleaving. Bit interleaving is a technique described in *Keccak implementation overview* [5], which we can apply to our implementation. As demonstrated in Figure 4.1, when interleaving 4 bytes, the first bit is taken from each of the four bytes and then are concatenated. The same is done for all of the following bits of each of the four bytes, and these are concatenated to the result from the

earlier bits. This continues until all of the bits from the four bytes have been interleaved. De-interleaving works by using the position of each bit in the interleaved 32-bit word to determine to which of the four 8-bit words it belongs to.

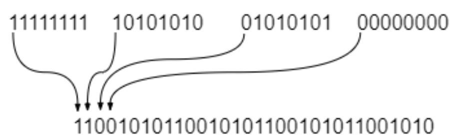

Figure 4.1: Four 8-bit words interleaved into one 32-bit word

Every bit-wise operation that is normally applied to each of the four bytes separately, can now be applied to the whole 32-bit word as long as that operation is adapted to work on 32-bit words. For example, a bit-shift of one now becomes a bit-shift of four. Any XOR or AND operation, that would normally be applied separately to each byte, now also needs to be interleaved in order to apply it to the whole 32-bit word at once.

In practice

In practice, it is not a matter of "taking" bits from bytes and "putting" them somewhere else. This is because evaluating each byte bit-per-bit is too time consuming and impractical. Instead we realize bit interleaving by a series of bitshifts, bitwise XOR and bitwise AND operations. Since we need to go from 4×8 bit, to 1×32 bit, we have to space out each of the 8-bit words over a 32-bit word, such that bit interleaving is achieved. In order to do this, we interleave each of the 8-bit words with zeros, so that we get four 16-bit words with bit values alternating between the bit values from the 8-bit words and zeros. Next, each of the four 16-bit words gets interleaved with zeros again, so that we now have four 32-bit words, where each of the original bit values from each of the 8-bit words are now alternated with three zeros. We visualize this process in Figure 4.2.

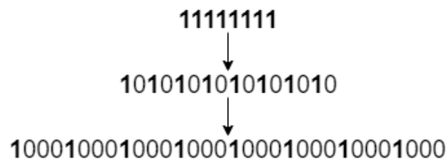

Figure 4.2: The 8-bit word "11111111" interleaved with zeros into a 32-bit word

Now, we perform a right bit-shift on 32-bit word two, three and four by one, two and three bits respectively. This puts all the values from the original 8-bit words in the correct position to finally combine the four 32-bit words into

one. Now that all values line up, we can perform a bitwise AND operation on each of the 32-bit words, creating a single 32-bit word consisting of the four original 8-bit words, now interleaved.

Theoretical improvement

The theoretical improvement of the Keccak-f[200] permutation by processing four blocks, where normally one would be processed, is 400%: no extra calculations have to be performed inside the optimized permutation. The only difference with the standard implementation is that instead of working with 8-bit words in 32-bit registers, Keccak-f[200] now works with 32-bit words in 32-bit registers. The LFSR can also be adapted to work on blocks of four, as well as the calculating of the masks for each block, although these components do not benefit as much from bit interleaving as the permutation. The process of bit interleaving itself also has a cost of course, but this process happens relatively few times; once before encryption or decryption, and once at the end, where we go back from an interleaved state to a normal state.

4.3 Implementation

In this section, we will describe how the implementation of bit interleaving works on the different components of Elephant. An implementation in C can be found at `https://github.com/Lemmenm/elephant-riscv`.

4.3.1 State representation

The standard version of Elephant has a state size of 200 bits (5-by-5-by-8), represented as an array of 25 (5-by-5) 8-bit words in the reference implementation. We only want to work with full 32-bit variables instead of 8-bit variables, so by replacing all byte variables with uint32_t variables, our state representation changes to an array of 25 32-bit words (5-by-5-by-32) with a total size of 800 bits. In this new representation, one block amounts to four blocks in the standard representation.

4.3.2 State transformation

Since we are using a new internal state representation, but also need to be able to use normal input data and return normal output data, two transformations of data need to happen: Once before encryption or decryption, where we receive a series of characters represented in bytes that need to be represented as a series of 32-bit words, and once after encryption or decryption where the output has to transform back to a byte representation.

Before encryption/decryption

There are two possible cases when transforming blocks to the new representation. The first and the easiest is when the amount of blocks that need to be transformed is a multiple of four. This means that all groups of four blocks consisting of 8-bit words can be interleaved to make one block of 32-bit words.
The second case is when the amount of blocks is not a multiple of four. Since the new representation needs four "old" blocks to transform into one new block, we have to use padding blocks filled with zero values to add to make the amount of blocks to a multiple of four.

After encryption/decryption

When transforming back to a byte representation of the data, we have to de-interleave each interleaved 32-bit block back to four blocks of bytes. Since it is possible that the amount of original blocks was not a multiple of four, we need to take make sure none of the data from the added padding blocks gets joined in the output data. This can be done by cutting off any output data which exceeds the message length variable.

4.3.3 Keccak-*f*[200]

The optimized implementation of Keccak-*f*[200] is not very different from the original one, except for the adaptation to support the new state representation. The Keccak-*f*[200] round constants need to be interleaved with themselves to be applicable to each part of the interleaved state, and the ρ-offsets and bit rotations need to be multiplied by four for the same purpose.

4.3.4 LFSR

The LFSR can also be changed to work on 32-bit words, but the original LFSR cannot be completely replaced. The masks of four blocks need to be interleaved, such that the new LFSR that uses 32-bit words can calculate the next block's mask for each of those interleaved masks at the same time. However, at the start of the algorithm, only the mask of the first block is known, as it is derived from the key. The second, third and fourth mask are derived sequentially from the first, which can only be done using the original LFSR setup. After calculating the masks for the first four blocks, and interleaving them, the new LFSR can continue calculating the masks for the next blocks using the 32-bit interleaved masks.

4.4 Benchmarks and results

In order to compare the reference implementation from the Elephant specification paper and the optimized implementation, we count CPU cycles. To count those in a RISC-V processor, the following assembly code can be used:

```
.text

.globl getcycles
.align 2
getcycles:
    csrr a1, mcycleh
    csrr a0, mcycle
    csrr a2, mcycleh
    bne a1, a2, getcycles
    ret
```

By using the *getcycles* function before and after each encryption or decryption, we can calculate the difference between those two values and get the amount of cycles the algorithm took. This can also be used to calculate how long each permutation or any other operation took inside the algorithm. To generate test cases, we will use the test vector generation code provided by NIST for the Lightweight Cryptography Competition [6], with some changes of parameters to fit our needs.

4.4.1 Permutation

The performance of the permutation is not dependant on the length of the plaintext or additional data, since it always gets a fixed-size input block. As shown in Figure 4.3, we found that the original permutation costs 10598 cycles per call, so 42392 cycles for four blocks, while the optimized version costs 13760 cycles per four blocks. This means we have a speedup of 308% on the permutation.

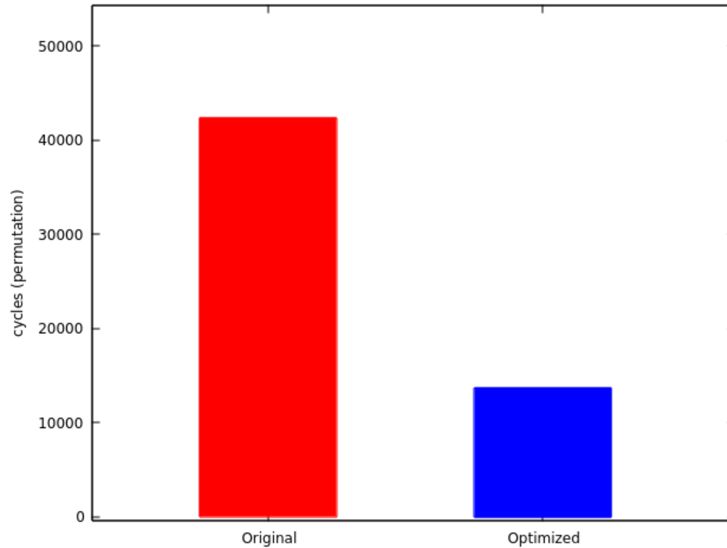

Figure 4.3: A comparison between the performance of the original Keccak-f[200] implementation and the optimized Keccak-f[200] implementation when processing four blocks

4.4.2 Varying message lengths

We will now look at a comparison between the benchmark and optimized implementation by increasing the plaintext size.

Encryption & decryption

As seen in Figure 4.4, there is not a great difference in encryption and decryption in terms of performance. We do note that messages with a low message length (under 200 characters) perform worse in the optimized implementation. This is because the effort of interleaving data to process four blocks simultaneously is wasted if there are very few blocks to process. For larger messages, the optimized implementation is a fair amount faster. When measuring the slope of the plots, we notice a slope of around 889.2 for the original implementation and a slope of 374.8 for the optimized implementation. To find the difference in growth between the original and optimized implementation, we divide the slopes of both plots and find $\frac{889.2}{374.8} = 2.37$, meaning the original implementation grows 2.37 times faster in CPU cycles than the optimized version, which means that the optimized implementation is 2.37 times faster.

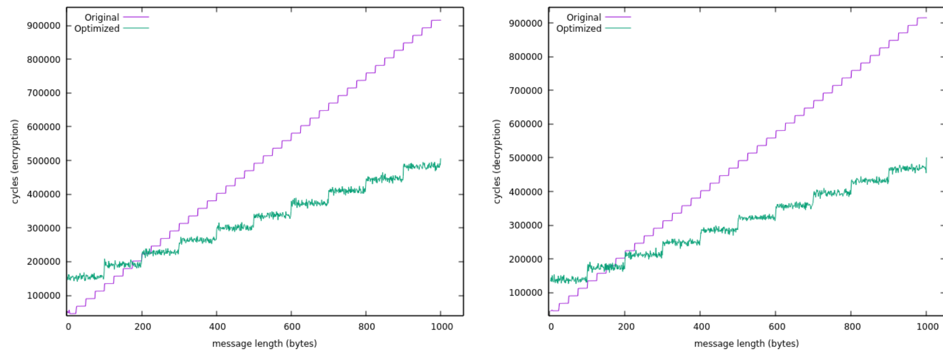

Figure 4.4: A comparison between the performance of the original implementation (purple) and the optimized implementation (green) in both encryption (left) and decryption (right).

Case distinction

Since our algorithm performs worse than the standard implementation on messages smaller than 200 bits, we implement a case distinction for messages below and above message length 200. For any messages with length smaller than 200, we simply use the standard implementation and for messages above that length, we use the implementation with bit-interleaving. As shown in figure 4.5, we now get an implementation that performs either equal or better to the original implementation.

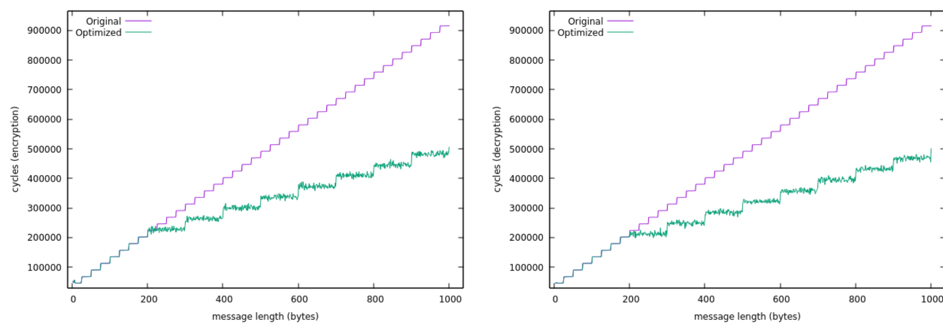

Figure 4.5: A comparison between the performance of the original implementation (purple) and the optimized implementation (green) in both encryption (left) and decryption (right), while using the new case distinction

4.5 Lower bound

When discussing the performance of an algorithm, having a lower bound of the amount of computational power it needs to run is a handy way to classify and judge the algorithm. Keccak-f[200] takes up most of the computational power of Delirium, and the parallelization of the calls to the Keccak-f[200] function have been the main optimization in this manuscript. So having an idea what the theoretical lower bound of the amount of cycles is and comparing this to the actual performance results, is a good way to get an idea of the performance of the algorithm.

4.5.1 Theoretical amount of cycles

Each round of Keccak-f[200] consists of five functions executed sequentially: θ, ρ, π, χ, and ι. Most of these functions are a combination of XOR's, AND's, NOT's, and the cost of each of these operations can be counted.

Cost of operations

On the RISC-V board, each of the XOR, AND, and NOT instructions cost 1 cycle. The custom ROL function consists of two shifts and a XOR, which amounts to 3 cycles. The lane size of the optimized Keccak-f[200] matches the 32-bit CPU word length, so each XOR, AND or NOT operation on a lane will cost only 1 cycle.

θ-function

The θ-function contains two loops of 25 XOR operations and 5 ROL operations followed by 5 more XOR operations, resulting in a total amount of:

$$\theta\text{-}total = 2 \times 25 \text{ cycles} + 5 \times 3 \text{ cycles} + 5 \text{ cycles} = 70 \text{ cycles}.$$

ρ-function

The ρ-function consists of 25 ROL operations, amounting to a total of:

$$\rho\text{-}total = 3 \times 25 \text{ cycles} = 75 \text{ cycles}.$$

π-function

The π-function does not contain any bitwise operations thus is not considered in the cycle count.

χ-function

The χ-function contains a loop of 25 XOR operations, 25 NOT operations and 25 AND operations amounting to a total of:

$$\chi\text{-}total = 25 \text{ cycles} + 25 \text{ cycles} + 25 \text{ cycles} = 75 \text{ cycles}.$$

ι-function

The ι-function contains 1 XOR operation, amounting to a total of:

$$\iota\text{-}total = 1 \text{ cycle}.$$

Total

Each round of Keccak-f[200] consists of each of the function parts executed sequentially, and one full execution of the whole of Keccak consists of 18 rounds. This amounts to a total of:

$$total = 18 \times (70 + 75 + 75 + 1) = 3.978 \text{ cycles}.$$

4.5.2 Discussion

We found a lower bound of 3978 cycles, which is certainly less than the measured 13760 cycles. The main reason why this is so different, is because while the ι-step consists of only one XOR operation, that operation is extremely costly (±500 cycles), since it has to look up the round constant in a list of 18 32-bit (4 interleaved 8-bit) round constants, which cannot all be kept in the registers, and are stored in memory instead. The round constants of the original Keccak-f[200] are only 8 bits in size, so they are easier to fit into the registers, which is why the original implementation suffers less from this problem. If we take away this giant cost, and treat it as an arbitrary XOR operation, we come to a total of 4778 measured cycles. The remainder of the difference between the lower bound and these newly measured cycles is due to loop control and arithmetic. An improvement of how the round constants are stored or used may be a good focal point in future work to further improve the algorithm.

Chapter 5

Related Work

There have been no earlier optimizations of Elephant on any platform, but with the progress of the NIST lightweight cryptography competition, an increasing amount of candidates are being implemented and optimized on a RISC-V environment. Research has been done on efficient cryptology on the RISC-V architecture [7], as well as multiple optimizations of crypto algorithms on RISC-V, such as SNEIK [8] and Ascon [9].

Chapter 6

Conclusions

We have optimized Elephant-Keccak-$f[200]$ for RISC-V by changing the state representation and functions to be able to process multiple blocks at the same time. This optimization performs especially well on larger message lengths because the cost of interleaving is more efficient with a greater amount of blocks. For message lengths under 200 bytes, we saw a decrease in performance, so we made use of the standard implementation of Elephant for those message lengths. This means that the optimized implementation performs either as good as, or better than the original.
Future work could include optimizing the usage of the Keccak round constants in the Keccak function or optimizing the two other instances of Elephant, Dumbo and Jumbo.

Bibliography

[1] Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process.

[2] "RISC-V Foundation". https://riscv.org/risc-v-foundation/.

[3] Beyne T., Chen Y.L., Dobraunig C., and Mennink B. "Elephant v1", Feb 2019.

[4] Bertoni G., Daemen J., Peeters J., and Van Assche G. "The keccak reference". Jan 2011.

[5] Bertoni G., Daemen J., Peeters J., Van Assche G., and Van Keer R. "Keccak Implementation Overview v3". May 2012.

[6] NIST Lightweight cryptography. https://csrc.nist.gov/projects/lightweight-cryptography.

[7] Stoffelen K. "Efficient Cryptography on the RISC-V Architecture". Cryptology ePrint Archive, Report 2019/794, 2019.

[8] Markku-Juhani O. Saarinen. "SNEIK on Microcontrollers: AVR, ARMv7-M, and RISC-V with Custom Instructions". Cryptology ePrint Archive, Report 2019/936, 2019.

[9] Jellema L. "optimizing Ascon on RISC-V". http://www.cs.ru.nl/bachelors-theses/2019/Lars_Jellema___4388747___Optimizing_Ascon_on_RISC-V.pdf.

[10] "RISC-V-getting-started". https://github.com/Ko-/RISC-V-getting-started/.

[11] "RISCV GNU Toolchain". https://github.com/riscv/riscv-gnu-toolchain.

[12] "RISC-V Openocd". https://github.com/riscv/riscv-openocd.

Appendix A

Appendix

A.1 How to set up the compiler

The compiler was installed on a Ubuntu 18.04 with Python 3 as default python, the used board was a HiFive 1 development board using an E32 CPU with RV32IMAC support. Following the RISC-V Getting started tutorial by Ko Stoffelen [10] we took the following steps:

1. Downloading and installing the RISC-V GNU toolchain
   (a) Clone the RISC-V GNU Toolchain[11] repository recursively
   (b) Install the listed prerequisites in the README
2. Apply the gdb.patch if Python 3 is your default Phython version
3. Configure and compile the GNU toolchain as described in the README of the repository [10]
4. Installing OpenOCD with RISC-V support
   (a) Clone the OpenOCD with RISC-V support repository[12]
   (b) Install the listed dependencies in the README
   (c) Follow the commands of the 'Building OpenOCD' section

A.2 How to download and compile and run optimized Elephant

The optimized elephant can be found and cloned at https://lemmenm.github.io/elephant-riscv/. The code can be compiled for RISC-V by running the 'make' command. To compile the code locally, run the 'make local' command. To flash the code to the Hifive 1 development board, execute './upload.sh main.elf', and to view the output from the board, execute './watch.sh'.