

BACHELOR THESIS
COMPUTER SCIENCE



RADBOUD UNIVERSITY

**Implementing Xoodoo with
protection against single fault
attacks on Cortex-M4**

Author:

Raka Schipperheijn
S4582721

First supervisor/assessor:

Prof. J.J.C. Daemen
joan@cs.ru.nl

Second supervisor:

MSc. P.M.C. Massolino
P.Massolino@cs.ru.nl

Second assessor:

dr. P. Schwabe
peter@cryptojedi.org

July 7, 2019

Abstract

This thesis proposes a solution to single fault attacks in Xoodoo, a 48-byte cryptographic permutation that allows very efficient symmetric crypto on a wide range of platforms. Protection against fault attacks is important as fault attacks aim to provoke an error to force the algorithm into an unintended state and this allows for a Differential Fault attack. This can be prevented by applying parity bits to check the integrity during certain critical stages of the algorithm. The practical solution will be provided in a generic non platform related implementation and also a representation of an implementation designed for an ARMv7 micro controller. The last implementation shows a representation that contains the appropriate register management to run on a Cortex-M4. Unlike the code of the Xoodoo without redundancy, it is impossible to run an implementation with parity bits completely on the registers without any swapping to the RAM. Thus one of the goals is to reduce the performance cost as much as possible.

Contents

1	Introduction	3
2	Preliminaries	4
2.1	Xoodoo	4
2.2	Protection against fault attacks using Parity bits	7
2.2.1	Implementing parity bits against fault attacks on Xoodoo	7
2.3	Fault attacks	7
2.4	Cortex-M4	8
3	Research	9
3.1	Parity bits implementation	9
3.2	Formal Notation	10
3.2.1	Algorithm	10
3.2.2	High level implementation optimization of mixing layer θ	12
3.3	Informal proof of pseudo code	12
3.3.1	Mixing layer θ	12
3.3.2	Plane shifting ρ_{west}	14
3.3.3	Non-linear layer χ	15
3.3.4	Plane shifting ρ_{east}	17
3.4	Low level scheduling	19
3.4.1	Cortex-M4 specific modifications	19
3.4.2	Mixing layer θ	20
3.4.3	Plane shifting ρ_{west}	21
3.4.4	Addition round constant ι	22
3.4.5	Non-linear layer χ	22
3.4.6	Plane shifting ρ_{east}	23
3.5	Representation of an optimized Cortex-M4 implementation	24
3.6	Validity	25
3.6.1	Equivalence	25
3.6.2	Protection against single-fault attacks	25
3.7	Computing the performance loss	25
3.7.1	Method	25

4	Related Work	27
5	Conclusions	28
5.1	Future work	28
	Bibliography	31

Chapter 1

Introduction

Xoodoo [1] is a cryptographic permutation that is meant to work on a wide range of platforms whilst keeping a good trade off between security and performance. On a low-end target, the computation is done serially with a small footprint, and the high-end processor can fully exploit its capabilities with the evaluation of multiple instances of the building block in parallel [2]. However, Xoodoo currently lacks protection against fault attacks. Fault attacks allow a potential Differential Fault attack [3], where an attacker is able to recover the secret key by injecting a fault and comparing the differences in the ciphertext. This thesis focusses on single-fault attacks. These are limited to one modification per function per round, since executing multiple fault attacks is a lot more difficult and the solution is too complex for this thesis. The question of this thesis is if it is possible to provide a solution against single fault attacks whilst keeping the cost reasonable.

The importance of providing protection lies within the fact that the platform is unreliable. It should execute everything safely, but it does not, as an attacker can influence the platform. So we try to compensate this by adding countermeasures to at least detect any modifications. The solution provides protection with the use of parity bits to verify if any bits have been modified during execution. This is a very fast way to verify the integrity of the state of the algorithm. The solution is designed for ARMv7-M processors as these are on the low-end side of microprocessors, meaning that it cannot exploit the capabilities of Xoodoo with the evaluation of multiple instances of the building block in parallel. The solution will be tested on a Cortex-M4. One of the difficulties of providing an implementation on a Cortex-M4 is that there are only fourteen usable registers. The addition of parity bits whilst maintaining a good performance means it is not possible to execute Xoodoo purely on the fourteen provided registers. This means a part of the solution focuses on efficient scheduling and keeping swapping between the registers and RAM as low as possible.

Chapter 2

Preliminaries

To understand this thesis properly, some knowledge is required about the cryptographic algorithm Xoodoo and fault attacks. Furthermore it is also important to understand what the reasoning and implications are behind implementing the protected version of Xoodoo on a Cortex-M4 processor.

2.1 Xoodoo

Xoodoo takes as input a state A . The structure of state A is as follows: A consists of three horizontal planes of 4×32 bits. These planes lie on top of each other and are called a_0, a_1, a_2 . Each plane contains four lanes, for example: plane a_0 contains lanes $a_{00}, a_{01}, a_{02}, a_{03}$. Three lanes that lie on top of each other are called sheets, for example: a_{00}, a_{10}, a_{30} . All bits have coordinates (x, y, z) where x represents the lane, z represents the place on the lane and y represents the plane. When three bits lie on top of each other (so the same x, z but different y , then we call that a column. A representation is shown in Figure 2.1.

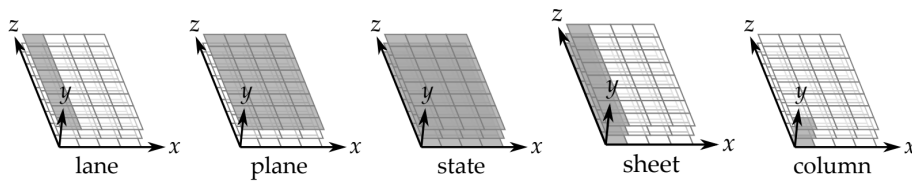


Figure 2.1: Structure of the state A [1]

Furthermore Xoodoo generates roundconstants rc_i for the desired amount of rounds. Then it applies a round function on the state n_r times. Each round consists of five functions:

Mixing layer θ

In this function, the sum of bits of the columns gets computed and this new plane gets shifted in the x, z coordinates. Finally each plane gets XORed with this new plane.

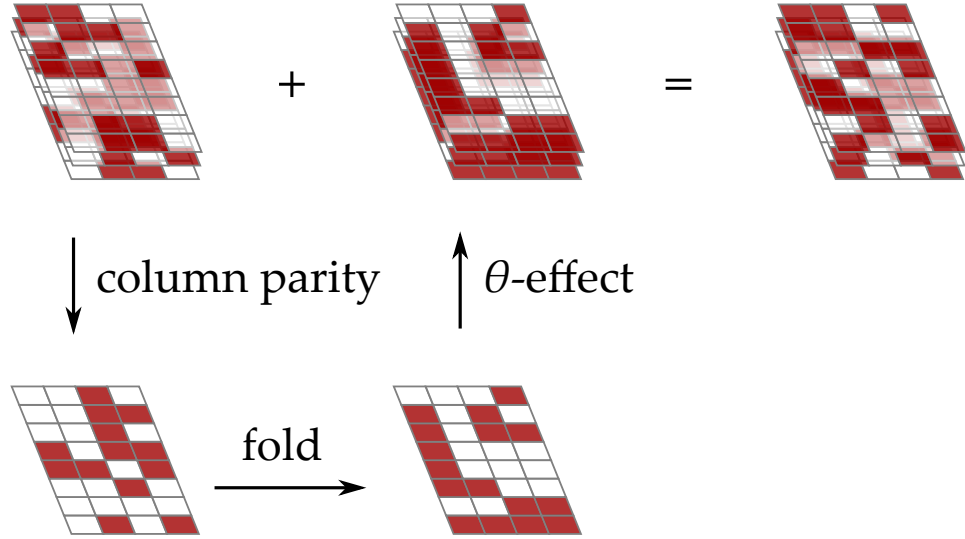


Figure 2.2: Effect of θ on a plane [1]

Addition of the round constant ι

In this function, the first plane gets XORed with the round constant.

Plane shifting ρ_{west} & ρ_{east}

In these functions, the second and third plane gets shifted in the x, y coordinates.

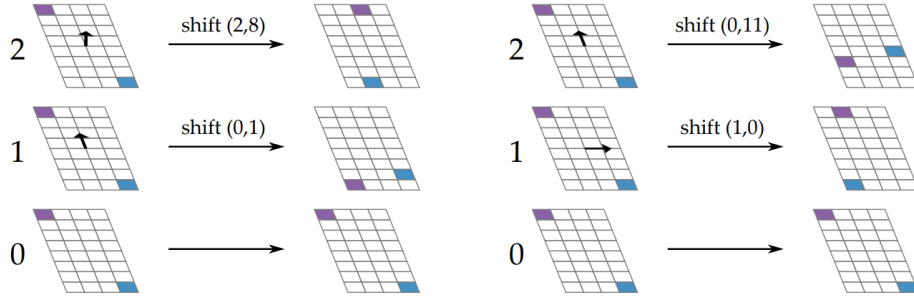


Figure 2.3: Effect of ρ_{west} (left) and ρ_{east} (right) on a state [1]

Non-linear layer χ

In this function each plane gets modified by using an XOR on the complement of another plane and by applying an AND with the remaining plane. This can be done sequentially thus allowing it to use less memory.

This structure allows Xoodoo to be able to only require twelve 32-bit registers to store all values and to do the computations with just two additional 32-bit registers.

However, the straightforward implementation of Xoodoo is weak against fault attacks as there is no way to detect a modified bit. This means that if a bit gets modified during a certain round the integrity of the ciphertext is void.

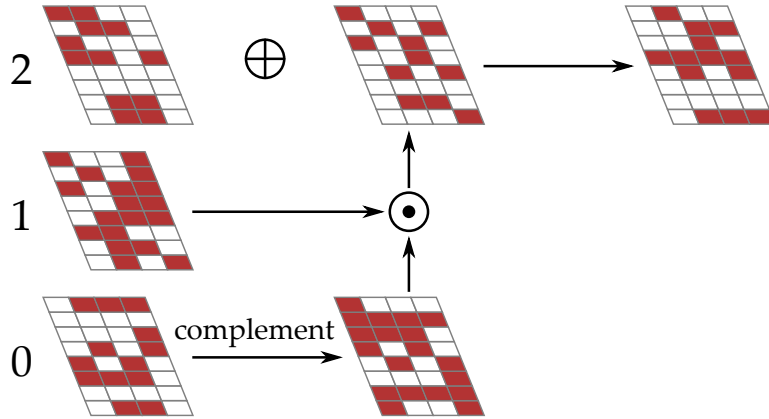


Figure 2.4: Effect of χ on a plane [1]

2.2 Protection against fault attacks using Parity bits

Using parity bits is a method of detecting errors. This is accomplished by taking the sum of the bits of the entity that has to be protected and append a parity bit. There are two methods of parity:

- Even parity: sum of bits + parity bit has to be even. So if the sum of the bits is odd, then the parity bit is 1 and if it is even then the parity bit is 0.
- Odd parity: sum of bits + parity bit has to be odd. So if the sum of the bits is odd, then the parity bit is 0 and if it is even then the parity bit is 1.

If a bit gets modified the parity check will fail as the sum of the bits + parity bit is, depending on the type of parity, either odd or even. However, this only works when an odd number of bits are modified as for an even number of modifications the parity check will still be satisfied. An example would be the bit string 10110 with an even parity type. The sum of the bits is odd, thus the parity bit is 1 and we get 101101. Now two bits gets flipped and the new value is 001001, which still satisfies the parity check as the sum of these bits is still even.

2.2.1 Implementing parity bits against fault attacks on Xoodoo

Because we only protect against single fault attacks, the problem with parity bits not working when there is an even amount of errors stated above is not relevant.

A new parity plane gets created by using even parity. After each function we can check if parity still holds by computing the XOR of the columns and comparing this to the parity plane. During the execution, of a function every modification to the original bits also has to be done on the parity bit. Therefore, if there are no improvements or shortcuts used, the computation takes at least twice as long. Improvements are thus required.

2.3 Fault attacks

Fault attacks[4] are active attacks against cryptographic implementations on a hardware level. In an active fault attack, the attacker has to be able to interact with the target device by interfering the calculation. There are different ways to interfere with the calculation [5] and a few examples are:

- Clock glitches, such as temporary overclocking of the CPU
- Sudden quick voltage spikes

- Overheating [6]
- Electromagnetic interference

These methods can bring the algorithm in an unintended intermediate state where certain values are modified. What this means is that the output of the execution is not equal to the output of a regular execution with no fault attacks.

2.4 Cortex-M4

The Cortex-M4 [7] is a low cost RISC ARM processor designed for micro controllers. The reason for using this particular processor is that it has enough registers to be able to run the original Xoodoo version fully on the registers. It has sixteen registers [8]:

- $r_0 - r_7$ are general-purpose registers accessible for both 16-bit and 32-bit instructions.
- $r_8 - r_{12}$ are general-purpose registers accessible for only 32-bit instructions.
- r_{13} stores the stack pointer and is thus reserved at all times.
- r_{14} is the Link Register and is able to be used as a general-purpose register as long as the Branch with Link instruction is not used.
- r_{15} stores the program counter and is always reserved.

If the Branch with Link instruction is not used, then the available amount of registers is 14 which is the required amount to execute a Xoodoo round in the registers exclusively. Because the Cortex-M4 is a lower end device the impact of adding more instructions to support the use of parity bits is more noticeable, and thus easier to measure.

Chapter 3

Research

The implementation was constructed by the following process:

- 1) Designing the implementation of the parity bits
- 2) Creating a formal notation based on 1)
- 3) Writing an informal proof to show that the parity bits provide protection
- 4) Creating a non optimized and non platform based pseudo code implementation that computes one round
- 5) Creating a non optimized and non platform C++ code implementation
- 6) Creating a C++ representation of an optimized Cortex-M4 processor implementation
- 7) Testing the representation of the Cortex-M4 implementation

The actual C++ code of 5) and 6) can be found in the corresponding git repository of this thesis [9], as well as some tests.

3.1 Parity bits implementation

The implementation has to have the following security claim: **For each function in each round, any modifications on a single plane will be detected.** It is assumed that the input state of the first round and any operation outside the round are valid, thus this thesis only focuses on attacks within the round function. Every function in the round has to be modified to support the addition of the parity plane and every function will have to check if the parity is still satisfied.

To implement the use of parity bits, ideally four additional registers are required compared to the original implementation of Xoodoo to store the

parity bits themselves. However, when using a Cortex-M4, swapping is required as the fourteen usable registers are insufficient. However, this step has to be minimized, because it is very slow.

3.2 Formal Notation

In this section the formal notation of the implementation of Xoodoo with protection against single fault attacks will be shown. Note that the parity bits are denoted by S and to increase the readability of the formal notation, an apostrophe gets added every time a variable is modified. Furthermore because Xoodoo uses planes, two dimensional rotations are possible. These are denoted by $\lll (x, z)$.

3.2.1 Algorithm

Algorithm 1 Definition of Xoodoo[r] with r the number of rounds

Require: Number of rounds $r > 0$

$S = a_0 + a_1 + a_2$
 $rc[r] = generateRoundConstants(r)$
for Round index i from $1 - r$ to 0 **do**
 $(a'_0, a'_1, a'_2, S') = R_i(a_0, a_1, a_2, S, rc[i])$

Here R_i is specified by the following sequence of steps:

function ROUND FUNCTION $R_i(a_0, a_1, a_2, S)$:

Mixing layer $\theta(a_0, a_1, a_2, S)$
 Plane shifting $\rho_{west}(a_1, a_2, S)$
 Addition round constant $\iota(a_0, rc[i], S)$
 Non-linear layer $\chi(a_0, a_1, a_2, S)$
 Plane shifting $\rho_{east}(a_1, a_2, S)$

Algorithm 2 Definition of the round function R_i

function MIXING LAYER $\theta(a_0, a_1, a_2, S)$:

for $j \in 0, 1, 2$ **do**

$E_a \leftarrow S \lll (1, 5) + S \lll (1, 14)$

$a'_j \leftarrow a_j + E_a$

$E_s \leftarrow S \lll (1, 5) + S \lll (1, 14)$

$S' \leftarrow S + E_s$

 verifyEquals(S' , a_0 , a'_1 , a'_2)

function PLANE SHIFTING $\rho_{west}(a_1, a_2, S)$:

$S' \leftarrow S + a_1 \lll (1, 0) + a_2 \lll (0, 11) + a_1 + a_2$

$a'_1 \leftarrow a_1 \lll (1, 0)$

$a'_2 \leftarrow a_2 \lll (0, 11)$

 verifyEquals(S' , a_0 , a'_1 , a'_2)

function ADDITION ROUND CONSTANT $\iota(a_0, rc, S)$:

$S' \leftarrow S + rc$

$a'_0 \leftarrow a_0 + rc$

 verifyEquals(S' , a'_0 , a_1 , a_2)

function NON-LINEAR LAYER $\chi(a_0, a_1, a_2, S)$:

$a'_0 \leftarrow a_0 + \overline{a_1} \cdot a_2$

$S' \leftarrow S + \overline{a_1} \cdot a_2$

$a'_1 \leftarrow a_1 + \overline{a_2} \cdot a'_0$

$S'' \leftarrow S' + \overline{a_2} \cdot a'_0$

$a'_2 \leftarrow a_2 + \overline{a'_0} \cdot a'_1$

$S''' \leftarrow S'' + \overline{a'_0} \cdot a'_1$

 VerifyEquals(S''' , a'_0 , a'_1 , a'_2)

function PLANE SHIFTING $\rho_{east}(a_1, a_2, S)$:

$S' \leftarrow S + a_1 \lll (0, 1) + a_2 \lll (2, 8) + a_1 + a_2$

$a'_1 \leftarrow a_1 \lll (0, 1)$

$a'_2 \leftarrow a_2 \lll (2, 8)$

 VerifyEquals(S' , a_0 , a'_1 , a'_2)

function VerifyEquals(S, a_0, a_1, a_2) = $\begin{cases} \text{True} & \text{for } S = a_0 + a_1 + a_2 \\ \text{False} & \text{for } S \neq a_0 + a_1 + a_2 \end{cases}$

3.2.2 High level implementation optimization of mixing layer θ

During mixing layer θ , the sum of the sheets is computed. This is equal to the new parity plane and thus it can be used for this computation. However, this can lead to a potential fault attack where the rotation of the parity plane gets modified. This will result in both the parity plane and the state being modified. To prevent this, the computation of the rotation gets redone for every lane of the state and the parity plane.

3.3 Informal proof of pseudo code

In this section, the algorithm will be verified with the use of an informal proof. For every line of code it will be checked that the algorithm will detect whether we execute a fault attack on that specific line. The invalid variable will be highlighted with an *.

3.3.1 Mixing layer θ

```

1: function MIXING LAYER  $\theta(a_0, a_1, a_2, S)$ :
2:   for  $j \in 0, 1, 2$  do
3:      $E_a \leftarrow S \lll (1, 5) + S \lll (1, 14)$ 
4:      $a'_j \leftarrow a_j + E_a$ 
5:    $E_s \leftarrow S \lll (1, 5) + S \lll (1, 14)$ 
6:    $S' \leftarrow S + E_s$ 
7:   verifyEquals( $S', a_0, a'_1, a'_2$ )

```

line 2: Modified variable E_s :

$$\begin{aligned}
& a_0 + a_1 + a_2 \rightarrow S \\
& S \lll (1, 5) + S \lll (1, 14) \rightarrow E_s^* \\
& S \lll (1, 5) + S \lll (1, 14) \rightarrow E_a \\
& a_0 + E_a \rightarrow a'_0 \\
& S \lll (1, 5) + S \lll (1, 14) \rightarrow E_a \\
& a_1 + E_a \rightarrow a'_1 \\
& S \lll (1, 5) + S \lll (1, 14) \rightarrow E_a \\
& a_2 + E_a \rightarrow a'_2 \\
& E_s^* + S \rightarrow S'^*
\end{aligned}$$

The verification in line 7 will fail because:

$$a'_0 + a'_1 + a'_2 \neq S'^*$$

line 4: Modified variable: E_a is very similar to E_s

line 5: Modified variable: a'_0 :

$$\begin{aligned} a_0 + a_1 + a_2 &\rightarrow S \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_s \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_0 + E_a &\rightarrow a_0'^* \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_1 + E_a &\rightarrow a_1' \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_2 + E_a &\rightarrow a_2' \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ E_s + S &\rightarrow S' \end{aligned}$$

The verification in line 7 will fail because:

$$\begin{aligned} a_0'^* + a_1' + a_2' &\neq a'_0 + a'_1 + a'_2 = S' \\ a_0'^* &\neq a'_0 \end{aligned}$$

Modifying a'_1 and a'_2 have similar proofs

line 6: Modified variable: S' :

$$\begin{aligned} a_0 + a_1 + a_2 &\rightarrow S \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_s \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_0 + E_a &\rightarrow a_0' \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_1 + E_a &\rightarrow a_1' \\ S \lll (1, 5) + S \lll (1, 14) &\rightarrow E_a \\ a_2 + E_a &\rightarrow a_2' \\ E_s + S &\rightarrow S'^* \end{aligned}$$

The verification in line 7 will fail because:

$$a'_0 + a'_1 + a'_2 = S' \neq S'^*$$

line 7: If one of the inputs of VerifyEquals gets changed whilst the original input was equal to each other, this will result in a termination of the program.

3.3.2 Plane shifting ρ_{west}

1:	function PLANE SHIFTING $\rho_{west}(a_1, a_2, S)$:
2:	$S' \leftarrow S + a_1 \lll (1, 0) + a_2 \lll (0, 11) + a_1 + a_2$
3:	$a'_1 \leftarrow a_1 \lll (1, 0)$
4:	$a'_2 \leftarrow a_2 \lll (0, 11)$
5:	verifyEquals(S' , a_0 , a'_1 , a'_2)

line 2: Modified variable S' :

$$\begin{aligned}
 S + a_1 + a_2 + a_1 \lll (1, 0) + a_2 \lll (0, 11) &= S'^* \\
 a_1 \lll (1, 0) &= a'_1 \\
 a_2 \lll (0, 11) &= a'_2
 \end{aligned}$$

The verification in line 5 will fail as:

$$a_0 + a'_1 + a'_2 = a_0 + a_1 \lll (1, 0) + a_2 \lll (0, 11) = S' \neq S'^*$$

line 3: Modified variable a'_1 :

$$\begin{aligned}
 S + a_1 + a_2 + a_1 \lll (1, 0) + a_2 \lll (0, 11) &= S' \\
 a_1 \lll (1, 0) &= a'^*_1 \\
 a_2 \lll (0, 11) &= a'_2
 \end{aligned}$$

The verification in line 5 will fail as:

$$\begin{aligned}
 a_0 + a'^*_1 + a'_2 &\stackrel{?}{=} a_0 + a_1 \lll (1, 0) + a_2 \lll (0, 11) = S' \\
 a'^*_1 &\neq a'_1 = a_1 \lll (1, 0)
 \end{aligned}$$

line 4: Modified variable a'_2 is very similar to a'_1

line 5: If one of the inputs of VerifyEquals gets changed whilst the original input was equal to each other, this will result in a termination of the program.

3.3.3 Non-linear layer χ

```

1: function NON-LINEAR LAYER  $\chi(a_0, a_1, a_2, S)$ :
2:    $a'_0 \leftarrow a_0 + \overline{a_1} \cdot a_2$ 
3:    $S' \leftarrow S + \overline{a_1} \cdot a_2$ 
4:    $a'_1 \leftarrow a_1 + \overline{a_2} \cdot a'_0$ 
5:    $S'' \leftarrow S' + \overline{a_2} \cdot a'_0$ 
6:    $a'_2 \leftarrow a_2 + \overline{a'_0} \cdot a'_1$ 
7:    $S''' \leftarrow S'' + \overline{a'_0} \cdot a'_1$ 
8:   VerifyEquals( $S'''$ ,  $a'_0$ ,  $a'_1$ ,  $a'_2$ )

```

line 2: Modified variable: a_0 :

$$\begin{aligned}
a_0^* + \overline{a_1} \cdot a_2 &\rightarrow a'^*_0 \\
S + \overline{a_1} \cdot a_2 &\rightarrow S' \\
a_1 + \overline{a_2} \cdot a'^*_0 &\rightarrow a'^*_1 \\
S' + \overline{a_2} \cdot a'^*_0 &\rightarrow S''^* \\
a_2 + \overline{a'^*_0} \cdot a'^*_1 &\rightarrow a'^*_2 \\
S'' + \overline{a'^*_0} \cdot a'^*_1 &\rightarrow S'''^*
\end{aligned}$$

The verification in line 8 will fail because:

$$\begin{aligned}
\mathbf{a}'^*_0 + \mathbf{a}'^*_1 + \mathbf{a}'^*_2 &= a_0^* + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'^*_0 + a_2 + \overline{a'^*_0} \cdot a'^*_1 \\
\mathbf{S}''' &= a_0 + a_1 + a_2 + \overline{a_1} \cdot a_2 + \overline{a_2} \cdot a_0'^* + \overline{a'^*_0} \cdot a'^*_1 \\
a_0^* + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'^*_0 &\stackrel{?}{=} a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a_0'^* + a_2 + \overline{a'^*_0} \cdot a'^*_1 \\
& \quad a_0^* \neq a_0
\end{aligned}$$

Modifying a_1 and a_2 have similar proofs.

line 3: Modified variable: S :

$$\begin{aligned}
a_0 + \overline{a_1} \cdot a_2 &\rightarrow a'_0 \\
S^* + \overline{a_1} \cdot a_2 &\rightarrow S'^* \\
a_1 + \overline{a_2} \cdot a'_0 &\rightarrow a'_1 \\
S'^* + \overline{a_2} \cdot a'_0 &\rightarrow S''^* \\
a_2 + \overline{a'_0} \cdot a_1 &\rightarrow a'_2 \\
S''^* + \overline{a'_0} \cdot a'_1 &\rightarrow S'''^*
\end{aligned}$$

The verification in line 8 will fail because:

$$\begin{aligned}
\mathbf{a}'_0 + \mathbf{a}'_1 + \mathbf{a}'_2 &= a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'_0 + a_2 + \overline{a'_0} \cdot a_1 \\
\mathbf{S}'''^* &= S^* + \overline{a_1} \cdot a_2 + \overline{a_2} \cdot a'_0 + \overline{a'_0} \cdot a_1 \\
a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'_0 + a_2 + \overline{a'_0} \cdot a_1 &\stackrel{?}{=} S^* + \overline{a_1} \cdot a_2 + \overline{a_2} \cdot a'_0 + \overline{a'_0} \cdot a_1 \\
a_0 + a_1 + a_2 &= S \neq S^*
\end{aligned}$$

line 4: Modified variable: a'_0 :

$$\begin{aligned}
a_0 + \overline{a_1} \cdot a_2 &\rightarrow a'^*_0 \\
S + \overline{a_1} \cdot a_2 &\rightarrow S' \\
a_1 + \overline{a_2} \cdot a'^*_0 &\rightarrow a'^*_1 \\
S' + \overline{a_2} \cdot a'^*_0 &\rightarrow S''^* \\
a_2 + \overline{a'^*_0} \cdot a'_1 &\rightarrow a'^*_2 \\
S'' + \overline{a'^*_0} \cdot a'_1 &\rightarrow S'''^*
\end{aligned}$$

The verification in line 8 will fail because:

$$\begin{aligned}
\mathbf{a}'^*_0 + \mathbf{a}'^*_1 + \mathbf{a}'^*_2 &= a'^*_0 + a_1 + \overline{a_2} \cdot a'^*_0 + a_2 + \overline{a'^*_0} \cdot a'_1 \\
\mathbf{S}''' &= a_0 + a_1 + a_2 + \overline{a_1} \cdot a_2 + \overline{a_2} \cdot a'^*_0 + \overline{a'^*_0} \cdot a'_1 \\
a'^*_0 + a_1 + \overline{a_2} \cdot a'^*_0 + a_2 + \overline{a'^*_0} \cdot a'_1 &\stackrel{?}{=} a_0 + a_1 + a_2 + \overline{a_1} \cdot a_2 + \overline{a_2} \cdot a'^*_0 + \overline{a'^*_0} \cdot a'_1 \\
a'^*_0 &\neq a'_0 = a_0 + \overline{a_1} \cdot a_2
\end{aligned}$$

line 5: Modified variable S' :

$$\begin{aligned}
a_0 + \overline{a_1} \cdot a_2 &\rightarrow a'_0 \\
S + \overline{a_1} \cdot a_2 &\rightarrow S'^* \\
a_1 + \overline{a_2} \cdot a'_0 &\rightarrow a'_1 \\
S'^* + \overline{a_2} \cdot a'_0 &\rightarrow S'''^* \\
a_2 + \overline{a'_0} \cdot a_1 &\rightarrow a'_2 \\
S'''^* + \overline{a'_0} \cdot a_1 &\rightarrow S'''^*
\end{aligned}$$

The verification in line 8 will fail because:

$$\begin{aligned}
\mathbf{a}'_0 + \mathbf{a}'_1 + \mathbf{a}'_2 &= a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'_0 + a_2 + \overline{a'_0} \cdot a_1 \\
\mathbf{S}'''^* &= S'^* + \overline{a_2} \cdot a'_0 + \overline{a'_0} \cdot a_1 \\
a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a'_0 + a_2 + \overline{a'_0} \cdot a_1 &\stackrel{?}{=} S'^* + \overline{a_2} \cdot a'_0 + \overline{a'_0} \cdot a_1 \\
a_0 + \overline{a_1} \cdot a_2 + a_1 + a_2 &= S + \overline{a_1} \cdot a_2 = S' \neq S'^*
\end{aligned}$$

line 6: Modified variable: a'_1 has a similar proof as a'_0 .

line 7: Modified variable: S'' has a similar proof as S' .

line 8: Modified variable: a'_2 has a similar proof as a'_0 .

line 9: Modified variable: S''' :

$$\begin{aligned}
a_0 + \overline{a_1} \cdot a_2 &\rightarrow a'_0 \\
S + \overline{a_1} \cdot a_2 &\rightarrow S' \\
a_1 + \overline{a_2} \cdot a'_0 &\rightarrow a'_1 \\
S' + \overline{a_2} \cdot a'_0 &\rightarrow S'' \\
a_2 + \overline{a'_0} \cdot a_1 &\rightarrow a'_2 \\
S'' + \overline{a'_0} \cdot a'_1 &\rightarrow S'''^*
\end{aligned}$$

The verification in line 8 will fail because:

$$\begin{aligned}
\mathbf{a}'_0 + \mathbf{a}'_1 + \mathbf{a}'_2 &= a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a_0 + a_2 + \overline{a'_0} \cdot a_1 \\
\mathbf{S}'''^* &\neq S''' \\
a_0 + \overline{a_1} \cdot a_2 + a_1 + \overline{a_2} \cdot a_0 + a_2 + \overline{a'_0} \cdot a_1 &= S''' \neq S'''^*
\end{aligned}$$

line 10: If one of the inputs of VerifyEquals gets changed whilst the original input was equal to each other, this will result in a termination of the program.

3.3.4 Plane shifting ρ_{east}

```

1: function PLANE SHIFTING  $\rho_{east}(a_1, a_2, S)$ :
2:    $S' \leftarrow S + a_1 \lll (0, 1) + a_2 \lll (2, 8) + a_1 + a_2$ 
3:    $a'_1 \leftarrow a_1 \lll (0, 1)$ 
4:    $a'_2 \leftarrow a_2 \lll (2, 8)$ 
5:   VerifyEquals( $S'$ ,  $a_0$ ,  $a'_1$ ,  $a'_2$ )

```

line 2: Modified variable S' :

$$\begin{aligned}
S + a_1 + a_2 + a_1 \lll (0, 1) + a_2 \lll (2, 8) &= S'^* \\
a_1 \lll (0, 1) &= a'_1 \\
a_2 \lll (2, 8) &= a'_2
\end{aligned}$$

The verification in line 5 will fail as:

$$a_0 + a'_1 + a'_2 = a_0 + a_1 \lll (0, 1) + a_2 \lll (2, 8) = S' \neq S'^*$$

line 3: Modified variable a'_1 :

$$\begin{aligned} S + a_1 + a_2 + a_1 &\lll (0, 1) + a_2 \lll (2, 8) = S' \\ a_1 &\lll (0, 1) = a_1'^* \\ a_2 &\lll (2, 8) = a_2' \end{aligned}$$

The verification in line 5 will fail as:

$$\begin{aligned} \cancel{a}_0 + a_1'^* + \cancel{a}_2 &\stackrel{?}{=} \cancel{a}_0 + a_1 \lll (0, 1) + \cancel{a}_2 \lll (2, 8) = S' \\ a_1'^* &\neq a_1' = a_1 \lll (0, 1) \end{aligned}$$

line 4: Modified variable a'_2 proof is similar to a'_1 :

line 5: If one of the inputs of VerifyEquals gets changed whilst the original input was equal to each other, this will result in a termination of the program.

3.4 Low level scheduling

In this section the code will be explained together with the scheduling of the swaps between the RAM and registers. The variables notation is bound to the formal notation of the algorithm. There are fourteen usable registers which are denoted by: $r_0 \dots r_{11}, v_0, v_1$ and each register can contain one lane. To show that a variable has already been modified, we add an apostrophe to the variable. However, these will be dropped when the algorithm enters the next function. This means that the output of mixing layer θ is regarded as a'_{00} , but this variable is regarded as a_{00} in ρ_{west} .

3.4.1 Cortex-M4 specific modifications

This implementation is designed to be used on a Cortex-M4, thus meaning that the low level scheduling has to adhere to the specifications of the Cortex-M4 and to the ARMv7 assembly language. Thirteen registers can be used out of the box, with an optional fourteenth register. This register can store values during the execution of a single function as it normally contains the linkpointer. This also means it is required to store the linkpointer to the RAM to use that register.

The process of swapping in ARMv7 consists of two steps: loading the address of the variable into a register and read or write a value to/from the address to another register. For loading a value, it is possible to just use one register. An example would be to load value x into r_0 :

```
1  .global x                set global variable x
2  ldr r0, =x               load address of x to r0
3  ldr r0, [r0]             load the value of the address of x inside r0 into r0
```

However, to store a value into a variable in the RAM, we need two different registers. One to hold the address of the variable and one to hold the desired value of the variable. To write the value 7 into variable x :

```
1  .global x                set global variable x
2  ldr r0, =x               load address of x to r0
3  mov r1, #7               load the value 7 into r1
4  str r1, [r0]             store the value from r1 into the address of r0
```

So for storing a value to the RAM, register r_0 will be used, because a_{00} is the least modified lane as it does not get used in ρ_{west} or ρ_{east} . When a register contains the address of value x , the following notation will be used: $\&[x]$.

3.4.2 Mixing layer θ

At the start of the function, the state of the registers consists of:

$$\begin{aligned} r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_3, & v_1 &= \text{Linkpointer} \end{aligned}$$

First we have to store the value of the Linkpointer to the RAM as this value is used to for the return address of the round. Mixing layer θ starts off with calculating the first sheet, thus we first have to calculate E_a in v_1 with the use of S_3 . With E_a we can now calculate the new values of the first sheet, but we recompute E_a after each value to prevent the potential fault attack mentioned in 3.2.2. To calculate the new value of S_0 we first store the original value in v_1 as this is needed later on. Then S'_0 is calculated and stored in v_0 with the use of S_3 which is still in v_0 . After writing v_0 to S_0 we now have:

$$\begin{aligned} r_0 &= a'_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a'_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S'_0, & v_1 &= S_0 \end{aligned}$$

Before S'_0 can be stored in the RAM, a_{00} has to be stored first as it has been modified resulting in the following:

$$\begin{aligned} r_0 &= a'_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a'_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S'_0, & v_1 &= \&[a_{00}] \end{aligned}$$

Now S_0 is loaded back into v_1 and r_0 contains the address of S_0 :

$$\begin{aligned} r_0 &= \&[S_0], & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a'_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S'_0, & v_1 &= S_0 \end{aligned}$$

For the second sheet we do the same but now with S_0 to compute S_1 . Because S_0 is stored in v_1 instead of v_0 we switch the operations around, so this means all things that were stored v_0 in the first sheet is now stored in v_1 and vice versa. This results in the following state:

$$\begin{aligned} r_0 &= \&[S_1], & r_1 &= a'_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_1, & v_1 &= S'_1 \end{aligned}$$

Now we also do this for the third sheet but now with S_1 to compute S_2 . The use of v_0 and v_1 is equal to the computation of the first sheet. For the final sheet we use S_2 to compute S_3 and the use of v_0 and v_1 is equal to the second sheet:

$$\begin{aligned} r_0 &= \&[S_3], & r_1 = a'_{01}, & r_2 = a'_{02}, & r_3 = a'_{03} \\ r_4 &= a'_{10}, & r_5 = a'_{11}, & r_6 = a'_{12}, & r_7 = a'_{13} \\ r_8 &= a'_{20}, & r_9 = a'_{21}, & r_{10} = a'_{22}, & r_{11} = a'_{23} \\ v_0 &= S_3, & v_1 &= S'_3 \end{aligned}$$

Now to reduce the amount of swaps in the verification of the state, we first check if the third and fourth sheet of the state is equal to the third and fourth lane of the parity. Thus, we have to load S'_2 to v_0 . This gives us $v_0 = S'_2$, $v_1 = S'_3$. Now we do the verification and if this passes, we load a'_{00} , S'_0 and S'_1 to r_0 , v_0 and v_1 respectively and then verify the first and second lanes.

3.4.3 Plane shifting ρ_{west}

After mixing layer θ , the state of the registers is:

$$\begin{aligned} r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_0, & v_1 &= S_1 \end{aligned}$$

The problem in ρ_{west} is that the values required to calculate the new value of the parity bits are modified in ρ_{west} , thus we first have to calculate the S' before calculating a_0, a'_1, a'_2 .

First, the value of S'_0 is calculated and stored in v_0 followed by the computation of S'_1 which is stored in v_1 . Then these get stored in the RAM and S_2 and S_3 are stored in v_0 and v_1 respectively. Then we calculate S'_2 and S'_3 and store them to v_0 and v_1 . Now all the lanes of the parity have been computed and we have the following state:

$$\begin{aligned} r_0 &= \&[S_3], & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S'_2, & v_1 &= S'_3 \end{aligned}$$

We first compute a'_1 , but this requires the use of an empty register, thus we store v_1 to S_3 in the RAM and use v_1 for the computation of a'_1 . This is followed by computing a'_2 . ρ_{west} does not change a_0 so we now have the

state:

$$\begin{aligned}
r_0 &= \&[S_3], & r_1 = a_{01}, & r_2 = a_{02}, & r_3 = a_{03} \\
r_4 &= a'_{10}, & r_5 = a'_{11}, & r_6 = a'_{12}, & r_7 = a'_{13} \\
r_8 &= a'_{20}, & r_9 = a'_{21}, & r_{10} = a'_{22}, & r'_{11} = a'_{23} \\
v_0 &= S'_2, & v_1 &= a_{10}
\end{aligned}$$

We want to verify the state and as S'_2 is already in register v_0 , we load S'_3 into v_1 and then verify the third and fourth lane. If $S_2 = a_{02} + a_{12} + a_{22}$ and $S_3 = a_{03} + a_{13} + a_{23}$ we load S'_0 and S'_1 into v_0 and v_1 respectively and verify the first and second lane.

3.4.4 Addition round constant ι

As a_{00} is required for this function, it is loaded back into r_0 :

$$\begin{aligned}
r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\
r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\
v_0 &= S_0, & v_1 &= S_1
\end{aligned}$$

We load the round constant into v_1 and then compute S'_0 in v_0 and a'_{00} in r_0 . Because ι only affects the first lane of a_0 we only have to verify the first lane, which means we only have to check if $r_0 + r_4 + r_8 = v_0$.

3.4.5 Non-linear layer χ

After ι , the state of the registers is:

$$\begin{aligned}
r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\
r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\
v_0 &= S_0, & v_1 &= rc
\end{aligned}$$

The execution is split up into row halves, which means the first part is affecting the first and second lane of a_0 , the second part is affecting the first and second lane of a_1 and so on. We do this, because the order of execution is important as this affects the result. Because we do not have S_1 in the registers, we load S_1 into v_1 . Now, we compute the first lane of a_0 , then we store the value of a_{00} by using v_0 to store the address. Then we load S_0 back into v_0 and compute S'_0 and we do the same with the second lane, but we add this value to S_1 . This leaves us with:

$$\begin{aligned}
r_0 &= a'_{00}, & r_1 &= a'_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\
r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\
v_0 &= S'_0, & v_1 &= S'_1
\end{aligned}$$

Now we do the same with the first and second lane of a_1 and a_2 which results in:

$$\begin{aligned} r_0 &= a'_{00}, & r_1 &= a'_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_0''', & v_1 &= S_1''' \end{aligned}$$

Now that we have calculated the first and second sheets completely and the sheet do not affect each other, we can now verify the first and second lane of the state, which saves us some swaps compared to doing this at the end. After this, we store S_0 and S_1 to the RAM and load S_2 and S_3 into v_0 and v_1 . Now we do the exact same thing as before but now applying it to the third and fourth sheet, so computing the third and fourth lane of a_0 gives:

$$\begin{aligned} r_0 &= \&[S_1], & r_1 &= a'_{01}, & r_2 &= a'_{02}, & r_3 &= a'_{03} \\ r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_2', & v_1 &= S_3' \end{aligned}$$

Computing the third and fourth lane of a_1 and a_2 gives:

$$\begin{aligned} r_0 &= \&[S_1], & r_1 &= a'_{01}, & r_2 &= a'_{02}, & r_3 &= a'_{03} \\ r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a'_{12}, & r_7 &= a'_{13} \\ r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a'_{22}, & r_{11} &= a'_{23} \\ v_0 &= S_2''', & v_1 &= S_3''' \end{aligned}$$

We now load a_{00} back into the r_0 and verify the second half of the state.

3.4.6 Plane shifting ρ_{east}

After non-linear layer χ , the state of the registers is:

$$\begin{aligned} r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\ r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a_{12}, & r_7 &= a_{13} \\ r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\ v_0 &= S_2, & v_1 &= S_3 \end{aligned}$$

For each sheet the parity is based on the original values of the sheet, but the sheets do not interact with each other. The algorithm takes the same approach as with Chi where it is done by row halves. First S_2' and S_3' are computed as this value is still loaded in v_0 and v_1 respectively. This is then followed by the computation of the third and fourth lane of a_1' which gives

the following state:

$$\begin{aligned}
r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a'_{12}, & r_7 &= a'_{13} \\
r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a_{22}, & r_{11} &= a_{23} \\
v_0 &= S'_2, & v_1 &= S'_3
\end{aligned}$$

Then the third and fourth lane a'_2 are computed which gives:

$$\begin{aligned}
r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a_{10}, & r_5 &= a_{11}, & r_6 &= a'_{12}, & r_7 &= a'_{13} \\
r_8 &= a_{20}, & r_9 &= a_{21}, & r_{10} &= a'_{22}, & r_{11} &= a'_{23} \\
v_0 &= S'_2, & v_1 &= S'_3
\end{aligned}$$

Because ρ_{east} does not affect a_0 , we are finished with the second half of the state. This means we can directly verify the third and fourth lane of the state. If this passes S'_2 and S'_3 are stored in the RAM and S_0 and S_1 are loaded in v_0 and v_1 respectively. Now we do the same as with the third and fourth lane.

We first compute S'_0 and S'_1 followed by the computation of the first and second lane of a'_1 and a'_2 which gives the final state:

$$\begin{aligned}
r_0 &= \&[S_3], & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a'_{12}, & r_7 &= a'_{13} \\
r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a'_{22}, & r_{11} &= a'_{23} \\
v_0 &= S'_0, & v_1 &= S'_1
\end{aligned}$$

Now we verify the first and second lane of the state with the use of a_{00} , S'_0 and S'_1 inside r_0 , v_0 and v_1 respectively.

And finally the Linkpointer gets loaded back into v_0 giving:

$$\begin{aligned}
r_0 &= a_{00}, & r_1 &= a_{01}, & r_2 &= a_{02}, & r_3 &= a_{03} \\
r_4 &= a'_{10}, & r_5 &= a'_{11}, & r_6 &= a'_{12}, & r_7 &= a'_{13} \\
r_8 &= a'_{20}, & r_9 &= a'_{21}, & r_{10} &= a'_{22}, & r_{11} &= a'_{23} \\
v_0 &= S'_0, & v_1 &= \text{Linkpointer}
\end{aligned}$$

3.5 Representation of an optimized Cortex-M4 implementation

This version shows a representation of what the registry and swapping management of the code should be when running the algorithm on a Cortex-M4. It acts like it uses registers denoted by the variables $r_0 \dots r_{12}$ and v_0, v_1 and shows the swaps that are required. Because the register r_{14} is required, the Linkpointer has to be stored as this contains the return address of the current function. So to minimize the amount of swapping of the Linkpointer

itself, the whole round subsides into one big function. This representation is written in C and we can run tests on it to check its validity and its performance. It can be found in the repository of this project.

An attempt has been made to write the actual optimized ARM assembly implementation, but due to time constraints and the priority being on having an implementation that provides the protection, this is unfinished.

3.6 Validity

It has to be verified for the new implementation that the output is equivalent and that it actually provides protection against single-fault attacks. In this section, the methods to verify these statements are explained.

3.6.1 Equivalence

The first will be achieved by generating one state and perform a permutation from both versions of Xoodoo code on the same state. If the outputs are equal, then for that input the functions are equivalent. We attempt permutations of one, six and twelve rounds. It is assumed these functions are equivalent if there is a sufficient amount of inputs where the outputs are the same. The test ran for six hours without finding any non-equivalent outputs. Thus we assume that the two versions are equivalent. The implementation can be found in the repository [9].

3.6.2 Protection against single-fault attacks

The implementation follows the formal notation closely and thus it is assumed that by informally proving the formal notation, we can assume the implementation is protected against fault attacks.

3.7 Computing the performance loss

In this section the computation of the performance loss will be explained and the results will be revealed. Firstly keep in mind that there is only a representation of the swapping and register management and not an actual implementation. This means the test will only show the cost of computing the parity plane and does not account for swapping. This means the performance loss is probably higher.

3.7.1 Method

To be able to compare the performance, boundaries first have to be set:

- Only the performance of the functions inside the round are compared as those are the only functions that are modified. Functions like generating the round constant and any other preparation work that has to be done will not be accounted for. These should have the same performance between the two functions.
- Only the performance of a single round will be computed as the performance should be consistent between iterations and it can only just make the results more blurry as measuring longer functions has more risk of being inaccurate.

With these boundaries, the following method is proposed:

The idea is to first compute generate an initial state which is used for both versions. Then any preparation work before the round gets done. Now we use a high resolution clock to store the current time and we run a few rounds of the new implementation, as this means a longer run time and thus the impact of overhead on the actual time is lower. Then again use the high resolution clock to get the current time and subtract those times from each other. The same thing happens for the Xoodoo code without the protection and then the run times get compared. We compared the time difference ten thousand times and the result of it was that the new implementation runs around 2.8 times slower. These functions will create some overhead that will increase the value compared to the actual run time, but this should be consistent between the two versions and as the goal is to find the performance difference between the two versions, this should not give any problems. This test can also be found in the repository[9].

Chapter 4

Related Work

This paper proposes an implementation of fault protection to a specific algorithm. To my best knowledge there has not been any related work about providing this protection specifically for Xoodoo. There are studies that propose countermeasures against fault attacks in general. Simon et al.[10] proposes a novel approach for designing symmetric ciphers to resist fault injection. This approach consists of partitioning the state into equally sized limbs, adding an additional limb and computing an extended round function. When this method is run on a Cortex-M4 (which is also used in this research), it yields an overhead of 83% on the performance.

Other studies are by Schneider et al.[11] which introduces a countermeasure ParTI for cryptographic hardware implementations that combines the concept of a provably-secure masking scheme with an error detecting approach against fault injection. The work of Reparaz et al.[12] proposes a countermeasure CAPA that claims security against higher-order SCA, multiple-shot DFA and combined attacks.

Chapter 5

Conclusions

The goal of the research was to implement protection against single-fault attacks to Xoodoo with a reasonable performance loss. The total performance loss in run time is at least 2.8 times more than the original Xoodoo code. When measuring this on a Cortex-M4 with optimized code this will be higher. This is mainly because of the requirement to swap between the registers and memory which was not necessary in the original Xoodoo. As shown in the informal proof in section 3.4, the algorithm is protected against an attack where one of the planes is modified during a round.

5.1 Future work

There are some improvements that can be done on the suggested implementation:

- On a design level one can look at incorporating ρ_{west} into θ as this will remove the required swaps that are done in ρ_{west} . The same could be said about incorporating ρ_{east} into χ .
- On an implementation level, one can look at using different ARM instructions to reduce the amount of required instructions. For instance, ARM assembly has support for performing a rotation together with an AND, XOR or MOV in a single instruction.

Another interesting thing to look at, is to use a platform that supports enough registers to omit the requirement of swapping and then compare the performance between the proposed implementation and the original Xoodoo. This is the place where the proposed implementation loses the most performance as these instructions are slow and expensive.

One can also look at alternative implementations that are not based on the parity of the planes and find out if those provide a better performance with the same protection.

Finally one thing that this implementation does lack is a full implementation of Xoodoo. The implementation proposed in this paper only covers the functions inside a round. Parts like generating the round constant are still vulnerable to possible single fault attacks.

Bibliography

- [1] Joan Daemen, Seth Hoeffert, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. *IACR Cryptology ePrint Archive*, 2018:767, 2018.
- [2] Joan Daemen, Seth Hoeffert, Gilles Van Assche, and Ronny Van Keer. The design of xoodoo and xooff. *IACR Transactions on Symmetric Cryptology*, 2018(4):1–38, Dec. 2018.
- [3] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO ’97*, pages 513–525, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
- [4] Olivier Benot. *Fault Attack*, pages 452–453. Springer US, Boston, MA, 2011.
- [5] Martin Otto. *Fault attacks and countermeasures*. PhD thesis, Citeseer, 2005.
- [6] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 219–235. Springer, 2013.
- [7] Arm Ltd. *ARM Cortex-M4 Processor Technical Reference Manual*. ARM. <https://developer.arm.com/docs/100166/0001>.
- [8] Arm Ltd. *ARM Compiler toolchain*. ARM. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489f/DUI0489F_arm_assembler_reference.pdf.
- [9] Git repository of this thesis, June. <https://github.com/RakaPKS/Implementing-Xoodoo-with-protection-against-single-fault-attacks>.
- [10] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. Towards lightweight cryptographic primitives with built-in fault-detection. Cryptology ePrint Archive, Report 2018/729, 2018. <https://eprint.iacr.org/2018/729>.

- [11] Tobias Schneider, Amir Moradi, and Tim Güneysu. Parti-towards combined hardware countermeasures against side-channel and fault-injection attacks. In *Annual International Cryptology Conference*, pages 302–332. Springer, 2016.
- [12] Oscar Reparaz, Lauren De Meyer, Begül Bilgin, Victor Arribas, Svetla Nikova, Ventzislav Nikov, and Nigel Smart. Capa: The spirit of beaver against physical attacks. Cryptology ePrint Archive, Report 2017/1195, 2017. <https://eprint.iacr.org/2017/1195>.