Bachelor thesis
Computer Science

Radboud University

A high-performance threshold implementation of a BaseKing variant on an ARM architecture

Author:
Tim van Dijk
tim.vandijk96@gmail.com

First supervisor/assessor:
prof. dr. Joan Daemen
joan@cs.ru.nl

Second supervisor:
drs. Kostas Papagiannopoulos
k.papagiannopoulos@cs.ru.nl

Second assessor:
prof. dr. Lejla Batina
lejla@cs.ru.nl

July 18, 2017

Abstract

In this thesis we present a straightforward implementation and a threshold implementation of the block cipher DoubleKing, a variant of BaseKing that uses 32-bit words instead of 16-bit words and has a block size of a 384-bit. The implementations are optimized with a focus on throughput and are made for the ARM Cortex-M4 microcontroller that uses the ARMv7 instruction set. The straightforward implementation achieves a throughput of 5.54 cycles per bit and the threshold implementation achieves one of 25.2 cycles per bit. We performed a test for resistance against first-order differential power analysis on the threshold implementation using close to a million traces. We did not find significant leakages. Therefore we conclude the threshold implementation indeed does protect against first-order differential power analysis.

Contents

Chapter 1

Introduction

It is not sufficient to only think of cryptographic systems in terms of abstract mathematical transformations, turning some input into some output. In practice, in order to use such a cryptographic system, it needs to be implemented first. The resulting program will run on a given processor, in a given environment, and will therefore present specific characteristics. [14] Even if a cryptographic system is considered mathematically secure, information could still leak through implementation-specific physically observable phenomena, such as power consumption, timing, electromagnetic radiation and even sound. The class of physical attacks where an adversary tries to retrieve secret data from a cryptographic system by exploiting these characteristics, is called side-channel attacks. Embedded devices are typical targets for side-channel attacks, so software for these devices should include adequate protection against such attacks.[12]

The block cipher BaseKing was designed in 1994 by Joan Daemen as part of his doctoral dissertation. [3]. A few years later, an optimized implementation of it was made for ARMv7 along with a reference implementation written in the C programming language. [4]

ARM Cortex M4 is one of the most popular modern microprocessors for constrained embedded devices.[12] For this microprocessor, we provide two implementations of DoubleKing, a modified version of BaseKing that uses 32-bit words instead of 16-bit words. The first implementation is a highly optimized unprotected implementation. The other is an optimized threshold implementation that aims to provide resistance against first-order power analysis attacks. These implementations are written in ARM assembly because an implementation written in a high-level programming language is unlikely to produce optimal performance and there would be numerous leaks as we cannot control register allocation nor the order of instructions. Both ARM assembly implementations, along with python implementations for both BaseKing and DoubleKing can be found at `https://github.com/TimVanDijk/Bachelor-Thesis-Public`.

In the remainder of this thesis, we will take a look at the details, optimizations and design choices of these implementations. We will also discuss the side-channel analysis with which we have verified the threshold implementation's resistance against first-order power analysis attacks.

Chapter 2

Preliminaries

2.1 BaseKing

In 1994, Joan Daemen designed the block cipher BaseKing as part of his doctoral dissertation. In this dissertation, he presents a new approach for the design of encryption schemes, ciphers and cryptographic hash functions, among which a design strategy for block ciphers. He then uses this strategy to design two ciphers, each specified with different choices for the variable parameters. One of the resulting ciphers is BaseKing.

BaseKing is a block cipher with a round function that consists of symmetric step functions. It operates on blocks consisting of twelve 16-bit words, which gives a block size of 192 bits. The key length is 192 bits as well. Both encryption and decryption are performed in eleven rounds and a final output transformation. In each of the eleven rounds, five transformations are applied to the intermediate result which we call the state. The state is denoted by a_0 to a_{11}. The five transformations are (in order):

- Key addition
- Mixing layer
- Early shift
- Nonlinear transform
- Late shift

The final output transformation consists of key addition followed by the mixing layer after which the order of the words is inverted.

In the next five subsections, we will take a closer look at each of the transformations to briefly explain what they do and what they accomplish.

2.1.1 Key addition

First the key is added to the state, then the round constant is added to some of the words. The words to which the round constant is added are a_i for $i \in \{2, 3, 8, 9\}$. The round constants are generated with a LFSR that can be described with the following pseudo-c program[4]:

```
q[0] = 0x000B;
if ((q[j+1] = q[j]<<1) & 0x0100) q[j+1] ^= 0x0111;
```

These constants are of course the same for each run of the program. This means that we can precompute the round constants generated by this LFSR. The computed round constants are: 0x0b, 0x16, 0x2c, 0x58, 0xb0, 0x71, 0xe2, 0xd5, 0xbb, 0x67, 0xce and 0x8d.

2.1.2 Mixing layer

BaseKing uses the wide trail strategy to gain resistance against linear and differential cryptanalysis. This is achieved by the iterated alternation of a nonlinear transformation and a transformation with high diffusion. In BaseKing, the mixing layer achieves diffusion of branch number 8 by making each word in this transformation's output depend on seven words in the input.[4] This is done by adding six words to each word in the state. These six words are found at offsets 2, 6, 7, 9, 10 and 11 from the word we are adding to. We can therefore describe the transformation as follows (with indices in modulo 12):

$$a_i \leftarrow a_i \oplus a_{i+2} \oplus a_{i+6} \oplus a_{i+7} \oplus a_{i+9} \oplus a_{i+10} \oplus a_{i+11}$$

2.1.3 Early shift

The early shift performs a bitwise circular shift on each of the words in the state. The amount of bits that each word is rotated depends on the rotation constants: r_0 to r_{11}. There are twelve constants, which means that each word has its own rotation constant. These constants are: 0, 8, 1, 15, 5, 10, 7, 6, 13, 14, 2 and 3. We can denote this transformation by: $a_i \leftarrow a_i \ll r_i$.

2.1.4 Nonlinear transform

The transformation uses an S-box that takes 3 input bits and substitutes it with 3 output bits. There is a one-to-one relation between the input block and the output block. This makes it a permutation and ensures invertibility which is in some cases is useful for decryption. The criteria for an S-box are determined by the design strategy. BaseKing's wide trail strategy requires the S-box to be selected such that it results in the minimization of the worst-case differential probability and of the largest input-output correlation. Each

bit of each word is put in the S-box along with the bits at the same position in the words at offsets 4 and 8 from the current word. BaseKing uses the S-box shown in table 2.1.

in	000	001	010	011	100	101	110	111
out	111	010	100	101	001	110	011	000

Table 2.1: BaseKing's 3-bit S-box

This S-box can also be described in terms of bitwise boolean operations. Doing so allows us to compute many S-boxes at the same time. To be more precise, the number of S-boxes we can compute at the same time is equal to the processor's word length. The boolean operations that describe this S-box are: $a_i \leftarrow a_i \oplus (a_{i+4} \vee \overline{a_{i+8}})$.

2.1.5 Late shift

The late shift is similar to the early shift in the sense that it is a cyclic shift, however this time we rotate to the right. Rotating n bits to the right is of course the same as rotating $16 - n$ bits to the left. We also use the same rotation constants, although in reversed order. This leaves us with the following: $a_i \leftarrow a_i \gg r_{11-i}$.

2.2 DoubleKing

After discussing the idea of implementing BaseKing with its designer, Joan Daemen, we decided that in this day and age it would make more sense to implement a version with 32-bit words instead. We decided to call this 32-bit version DoubleKing. The specification of DoubleKing is exactly the same as BaseKing, except for the word size and the rotation constants. In DoubleKing, each of the twelve words is 32-bit which is double the word size of BaseKing, hence the name DoubleKing. BaseKing's rotation constants were made considering there are only 16 bits to rotate through. DoubleKing therefore requires different rotation constants. Joan Daemen provided a new set of rotation constants for DoubleKing. These constants are combinational numbers 2 out of i, with i starting from 1 and up to 12. This gives 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66. Doing this modulo 32 reduces the last few to 4, 13, 23 and 2. This is natural as rotating over a 32-bit word over 36 is the same as rotating over 4.

2.3 ARM Cortex-M4

In this section we will provide background information on the ARM Cortex-M4 which is the microcontroller that we chose to develop the implementation of DoubleKing for.

2.3.1 Registers

The ARM Cortex-M4 has 16 accessible registers: r0-r15. Of these, r0-r12 are general-purpose registers; r13 (sp) is the stack pointer; r14 (lr) is the link register and r15 (pc) is the program counter. All instructions can access r0-r14 and some can access r15 as well. In some cases, however, this leads to undefined behavior, for example, it is undefined what happens when r13 is used in logical operations such as eor (exclusive or).

2.3.2 Instruction set

The ARM Cortex-M4 is a 32-bit microcontroller that has an ARMv7E-M architecture and it supports the ARMv7-M instruction set as well as the Thumb instruction set.[1] The Thumb instruction set consists of instructions with a length of 16 bits and acts as a compact subset of the ARMv7-M instruction set whose instructions have a length of 32 bits. Not all of the ARMv7-M's instructions are available in the Thumb subset. For example, one cannot access some of the registers. Also, some ARM instructions can only be simulated with a sequence of Thumb instructions. In practise, when we compare programs that use the Thumb instruction set to their ARMv7-M counterpart, we notice that their code size is about 30% smaller and the increase in executed instructions is in the range of 9% to 41%.[9]

In applications where memory is limited, this trade-off is likely worth it. In our case memory is not an issue and we will focus solely on performance. Therefore, we will use the ARMv7-M instruction set.

2.3.3 Instructions

All instructions in the ARMv7-M instruction set are 32 bits long. Most, but not all instructions take one clock cycle to execute. Instructions such as ldr and str access SRAM and have a latency of two cycles, however consecutive executions of that instruction can be pipelined and will therefore only take one cycle.

To understand the implementation, we first need to understand the instructions that it uses. BaseKing is a bitslice cipher which means that it can be implemented using only bitwise logical operations and (cyclic) shifts.[4] Aside from those instructions we also make use of instructions to access memory such as str, stm, ldr and ldm. To control program flow we also use cmp (to test conditions) and branch instructions.

2.3.4 Barrel shifter

The ARM Cortex-M4 contains a barrel shifter, which is a piece of hardware that can shift or rotate a given value. Many instructions allow the use of the barrel shifter by sending the value of the second operand through the barrel shifter before it reaches the ALU. In most architectures this allows both the instruction as well as a shift or a rotate to be executed in the same cycle. For example, if we wanted to compute: r0 := r1 + 4×r2, we could achieve this with:

```
LSL r2, r2, #2    ; r2 := r2 << 2
ADD r0, r1, r2    ; r0 := r1 + r2
```

Both instructions require one cycle, so in total it takes two cycles. Because the add instruction allows the use of the barrel shifter, we can combine the two instructions as follows:

```
ADD r0, r1, r2, LSL #2    ; r0 := r1 + (r2 << 2)
```

This instruction only takes one cycle to execute.

2.4 Threshold implementations

There are many ways to protect against side-channel attacks. In this section we briefly discuss traditional approaches as well as the approach we applied to DoubleKing.

2.4.1 Traditional approaches

Traditional approaches to protect against side-channel attacks use random values to mask the data that is being processed. This is done such that operations on individual words of the intermediate result of the cryptographic algorithm are being executed independent of the secret key. A downside to these approaches is that one needs to be very careful when selecting the order of instructions, as executing instruction in the wrong order can leak information. Also, many approaches that were believed to be secure turned out leak information in the presence of glitches. Glitches naturally occur in hardware all the time, but can, for example, also be introduced in fault-injection attacks.[10]

2.4.2 A new approach

In 2006, Nikova, Rechberger & Rijmen presented threshold schemes: a new approach to protect against side-channel attacks. Threshold schemes are masking schemes that are provably secure against first-order differential power analysis attacks and are based on secret sharing, threshold cryptography and multiparty computation. Unlike traditional masking schemes, it protects against first-order differential power analysis, even in the presence of glitches. Also, the order in which instructions are executed does not matter, because each share misses information and therefore each share cannot leak information by itself. We will briefly discuss the properties of threshold implementations. For a complete explanation and the proof of these properties, we refer to the original paper.[10]

In threshold schemes sensitive variables are initially split over n shares such that:

1. the sum of the shares is equal to the sensitive variable;
2. knowledge of up to $n - 1$ shares does not provide information about the sensitive variable.

The first property is called correctness, and like all other properties, it holds not only at the start, but throughout the entire scheme. In other words, applying function f to the variable must have the same result as applying the shared function f' to each of the shares and then computing the sum of those shares. If this is the case, then f' is a correct sharing of f.

The second property is called non-completeness. This property ensures that no correlation between a share and the sensitive variable exists. For a scheme to have this property, it is essential that there are no functions that operate on all shares at the same time. This can be challenging in nonlinear transformations.

Then there is a third property that we have not yet discussed: uniformity. Because in a threshold scheme transformations are often applied after one another, it is important that the output of each transformation is uniform as it will likely be used as input for a next transformation. It is an essential part in the proof of resistance against differential power analysis attacks, but in practise it is not straightforward to exploit a lack of uniformity. One way to show that a function is uniform is by showing that an inverse of that function exists.

Chapter 3

Related Work

In 2000, several techniques to protect bitslice block ciphers against power analysis attacks are presented by Daemen, Peeters & van Assche[4]. They extend the *full state splitting* method and show how it can protect against first-order differential power analysis (DPA) attacks. They call this the *bias vector method* and proceed by applying it, along with the method it is based on, to BaseKing, an example of a bitslice block cipher.

When using the full state splitting method, before computing the cipher, the state a is split in two shares: a' and a''. a' is generated randomly and $a'' = a \oplus a'$ such that $a = a' \oplus a''$. This way a' and a'' are independent of a. Only after the cipher computation is finished, the shares are recombined. As long as each computation does not involve both a' and a'', there is no correlation with a, and thus is the cipher protected against first-order DPA attacks.

The bias vector method is similar to full state splitting, but one of the two shares always consists of words that are either all-0 or all-1. Some of BaseKing's computations can be performed more efficiently on these bias states, thus improving performance. Although the bias vector method makes second order DPA attacks more difficult than full state splitting with two shares, decorrelation is only reached at bit-level and not at word-level.

In 2012, Bertoni, Daemen, Peeters, van Assche & van Keer[8] presented several implementations of the nonlinear transformation χ in the round function of Keccak-f, which is very similar to the nonlinear transformation in BaseKing. Among these implemenations are a two-share masking implementation and a hardware implementation using three-share masking. Like our threshold implementation for DoubleKing, their hardware implementation provides not only resistance against first order DPA, but also against glitches.

Similar to these papers, we provide a side-channel protected implementation for a cipher. To be more specific, we provide a threshold implementation of a BaseKing variant that is optimized for performance. This is the first

time a threshold implementation is made of BaseKing. Being a threshold implementation, it provides resistance against glitch attacks, which is something existing side-channel protected implementations of BaseKing do not.

Chapter 4

Straightforward Implementation

In this chapter, we explain the optimizations made in DoubleKing's implementation as well as its performance. Despite the main focus of this thesis being the threshold implementation, we figured an unprotected implementation could be very useful in situations where protection against first-order differential power analysis attacks is not necessary. Also, optimizations made in the unprotected implementation can likely be carried over to the threshold implementation.

The main obstacle we encountered when implementing DoubleKing is the lack of general purpose registers in the ARM Cortex M4. Remember there are only 16 registers: r0-r15. Twelve of these, r0-r11, are used to store the state. Because we cannot use r15, the program counter, this leaves us with three registers to work with: r12-r14. Although r13 and r14 are not intended to be used as general purpose registers, they can still be used for most purposes. Having three registers to work with was sufficient to keep the amount of memory accesses manageable. Figure 4.1 provides an overview of the transformations that are applied to the state in each round. The top part shows the first round. The bottom part shows the final transformation and the part in the middle shows what happens in each of the rounds in between.

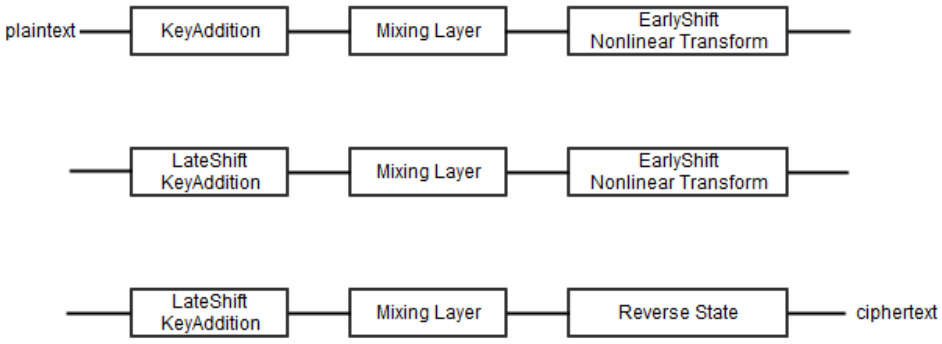


Figure 4.1: Overview of the unprotected implementation

4.1 Implementation details

Key Addition. Instead of computing the round constants on the spot, we hard-coded the round constants in memory. Although this solution requires memory access, it has our preference because it does not occupy one of our three precious open registers with the round constant. Using one of those three registers would cause a huge increase in memory accesses in the other transformations, therefore using memory in this transformation would have been inevitable anyway. When the key addition transformation is called, the round constant must be in r14. Because the xor operation is associative, we can add the constant before adding the key. This frees r14, which allows us to pipeline memory accesses required to load the key; instead of twelve accesses that each take 2 cycles for a total of 24 cycles, we now get six accesses that each take (2+1) cycles for a total of 18 cycles.

Mixing layer. A naive implementation of the mixing layer would require an overwhelming amount of memory accesses because all new words are to be computed in parallel and not sequential. For example, we compute a'_8 as follows: $a_8 := a_2 \oplus a_3 \oplus a_5 \oplus a_6 \oplus a_7 \oplus a_8 \oplus a_{10}$. If we compute the new a_i's in order, to compute a'_8 we need to retrieve a_2, a_3, a_5, a_6 and a_7 from memory because we have already modified them. This would be very costly in terms of clock cycles. We figured out a way to compute a'_0 to a'_5 using only the three spare registers. This was possible because fewer words were modified already. We also devised a method to compute a'_6 to a'_{11} using a'_0 to a'_5. This method uses that $a'_i = a_i \oplus a_{i-6} \oplus a'_{i-6} \oplus (a_0 \oplus a_1 \oplus \ldots \oplus a_{11})$. Let us take a moment to derive this.

By definition, we know that

$$a'_i = a_i \oplus a_{i+2} \oplus a_{i+6} \oplus a_{i+7} \oplus a_{i+9} \oplus a_{i+10} \oplus a_{i+11}$$

By changing the order of the terms slightly, we get

$$a'_i = a_i \oplus a_{i+6} \oplus a_{i+2} \oplus a_{i+7} \oplus a_{i+9} \oplus a_{i+10} \oplus a_{i+11}$$

Then we consider all but the two leftmost terms. They might not seem to be very useful, but it is all about what is not in them. By adding all words in the state to that group, we get the words that are not in them. We also change the order of terms again.

$$\begin{aligned} a'_i &= a_i \oplus a_{i+6} \oplus (a_{i+1} \oplus a_{i+3} \oplus a_{i+4} \oplus a_{i+5} \oplus a_{i+6} \oplus a_{i+8} \oplus a_{i+0}) \\ &\quad \oplus (a_i \oplus a_{i+1} \oplus \ldots \oplus a_{i+11}) \\ a'_i &= a_i \oplus a_{i+6} \oplus (a_{i+6} \oplus a_{i+8} \oplus a_{i+0} \oplus a_{i+1} \oplus a_{i+3} \oplus a_{i+4} \oplus a_{i+5}) \\ &\quad \oplus (a_i \oplus a_{i+1} \oplus \ldots \oplus a_{i+11}) \end{aligned}$$

Now the second group is by definition equal to a'_{i+6}. Also, the indices are in modulo 12 and of course $a_i \oplus a_{i+1} \oplus \ldots \oplus a_{i+11}$ is simply the same as all words. To complete the proof, let us rewrite things a little.

$$\begin{aligned} a'_i &= a_i \oplus a_{i+6} \oplus a'_{i+6} \oplus (a_i \oplus a_{i+1} \oplus \ldots \oplus a_{i+11}) \\ a'_i &= a_i \oplus a_{i-6} \oplus a'_{i-6} \oplus (a_i \oplus a_{i+1} \oplus \ldots \oplus a_{i+11}) \\ a'_i &= a_i \oplus a_{i-6} \oplus a'_{i-6} \oplus (a_0 \oplus a_1 \oplus \ldots \oplus a_{11}) \quad \square \end{aligned}$$

Using this method we need to retrieve only one of the unmodified words from memory per computation of each of the words a_6 to a_{11}. We also managed to prevent a couple of memory accesses by postponing the finalization of some computations because they required a value that would need to be retrieved from memory at a later point in time anyway.

Early shift. We completely merged this transformation with the nonlinear transform. In the nonlinear transform, whenever a word is required in a computation, we shift it first in the same way the early shift would have done. Because of the barrel shifter this takes no additional cycles. A limitation of the barrel shifter is, however, that only one of the two operands can be shifted. We figured out a way to compute the S-box in four parts. The limitation of the barrel shifter becomes a problem at the start of each those four parts where we need to do the OR operation on two shifted values. To solve this problem we shift one of the operands first. The other operand we shift with the barrel shifter for free during the operation in which we require two shifted values. In total the early shift takes four times 1 cycle for a total of 4 cycles.

Nonlinear transform. As we already discussed in the preliminaries, the S-box that this transformation uses can be described in terms of boolean operations to compute entire words at once. Upon closer inspection of those boolean operations, it becomes apparent that we can compute the result of this transformation in four separate parts. These parts are $\{ (a_i, a_{i+4}, a_{i+8}) \mid 0 \leq i \leq 3 \}$. The new value of the words in each of these parts only depend on words in the same part. This way the three free registers are sufficient to perform the S-box transformation without accessing SRAM at all. All parts are computed in exactly the same way. By saving and reusing intermediate results that are saved in one of the free registers, we save a few cycles as well.

While developing the threshold implementation, we serialized the S-box. We decided not to revise the S-Box in the unprotected implementation because we expected the gain in performance to be minimal, and given the time constraints not worth the effort.

Late shift. Similarly to the early shift, we also merge this transformation with the one that comes after it. In this case, that transformation is the key addition of the next round. In the key addition we do not encounter the problem we had in the nonlinear transform and all shifts are done for free. Unfortunately we cannot use the merged version everywhere because in the first round there is no preceding late shift. In the first round we therefore use the unmerged version of key addition.

4.2 Results

We decided to count the cycles manually as making the processor on a board count them resulted in very unpredictable and unexpected values. When counting the cycles, we assumed that each instruction takes one cycle, except when it is a ldr or str instruction. In that case the first instruction takes two cycles and any subsequent instructions of that type take one. ldm and stm instructions were converted into equivalent ldr or str instructions.

In table 4.1 is a breakdown of the implementation, showing for each transformation what instructions were used and how many cycles it took.

Using table 4.1 we see that encryption of one block (384 bits) with DoubleKing takes 2127 cycles. This means we have a latency of 2127 cycles and a throughput of 5.54 cycles per bit. It has a code size of 1756 bytes and during execution uses 45 distinct words (= 180 bytes) in SRAM.

Transformation	Instructions	Cycles	Times used	Total cycles	% of total
Key Addition	eor x16, ldr x13	35	1	35	1.6%
Late Shift + Key Addition	eor x5, eor+ror x15, ldr x13	35	11	385	18.1%
Mixing Layer	eor x59, ldr x11, mov x3, str x8	89	12	1068	50.2%
Early Shift + Nonlinear Transform	eor x8, eor+ror x8, mov x1, mov+ror x3, mvn x8, orr x4, mvn+ror x4, orr+ror x8	44	11	484	22.8%
Reverse State	eor x18	18	1	18	0.8%
Control Logic	add x10, blt x10, cmp x10, ldr x63, mov x3, str x10	137	1	137	6.4%

Table 4.1: Breakdown of DoubleKing's unprotected implementation

There is no reference implementation of DoubleKing to compare it against, but there is a timing and SPA resistant implementation of BaseKing. That implementation requires 1949 cycles to encrypt 192 bits and has a code size of 1776 bytes. In about the same amount of cycles we encrypt twice as many bits, but the code size has increased by a factor of 2.3. The code size could be significantly reduced by using subroutines instead of macros, but it would come at the cost of a slight decrease in performance.

Chapter 5

Threshold Implementation

In addition to a straightforward implementation, we made a threshold implementation. Threshold implementations, when made correctly, provide provable theoretical security against first-order side-channel attacks. In this chapter we start off by providing an overview of the threshold implementation, afterwards we will take an in-depth look at each of the transformations to further discuss the details, optimizations and design choices.

5.1 Overview

Because of the linear step functions of BaseKing, we were able to reuse large parts of the straightforward implementation. One exception is the nonlinear transform. This is the only step function in which information from multiple shares is combined. We had to rewrite this transformation to ensure non-completeness while preserving correctness.

From the non-completeness and correctness property of threshold implementations follows that at least $d + 1$ shares are required to implement a function of degree d.[10] As we soon will see, the nonlinear transform, being a function of degree 2, has the highest degree of all functions used in DoubleKing. Therefore we need at least 3 shares.

During encryption, we start by using two random 384-bit strings to split the plaintext into three shares. Each of these shares consist out of twelve words. Because we only have 16 registers, just one of the three shares can be kept in memory at a time. Again the first round is different because there is no late shift that still needs to be done. To reduce the number of memory accesses we keep applying transformations to the share that is currently in the registers until we arrive at the nonlinear transform. At this point we can only continue once the other shares are ready to do the nonlinear transform as well. This is where we have to switch shares and access memory. This requires way fewer memory accesses compared to the naive solution where each transformation applied to each of the shares in succession. At the end

of the final output transformation the shares are combined and result in the ciphertext.

The code size of this implementation is drastically increased compared to the straightforward implementation. To counter this, we decided to sometimes use subroutines instead of macros. One problem is that we cannot use the link register to store the return address in, as we need that register for our computations. To solve this problem, at the start of each subroutine, we store the return address to SRAM. Once the subroutine is finished, we retrieve the address from SRAM and branch to it.

Figure 5.1 shows what transformations are applied to what shares in the threshold implementation. The top part shows the splitting of the plaintext into three shares and the first round. The middle part shows what happens in each of the other rounds and at the bottom we see the final output transformation along with the combining and reversing of the shares. If the names of two transformations are put inside the same block, it means that they have been merged.

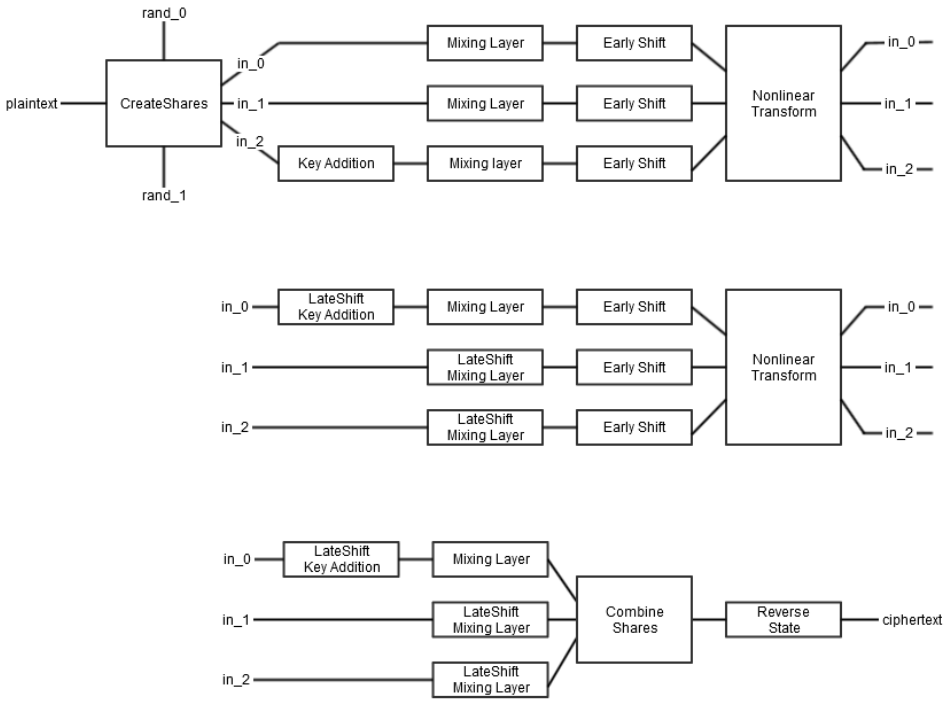


Figure 5.1: Overview of the threshold implementation

5.2 Implementation details

Create Shares. This transformation takes the plaintext and two random values that are the same length as the plaintext. It then computes:

$$in_0 = rand_0$$
$$in_1 = rand_1$$
$$in_2 = plaintext \oplus rand_0 \oplus rand_1$$

We perform the actual computation in two parts. We first compute the first half of each share and then we compute the second half. The main advantage of this approach is that it enables us to pipeline most memory accesses. A fortuitous coincidence is that it leaves us with half of one of the shares in the registers already. In our case, we end up with the second half of in_2 in r6-r11. To continue with the other transformations we only need to retrieve the first half instead of the whole share. Of course we could have ended up with an entire share in the registers, but that would be at the expense of our ability to pipeline the memory accesses and could cost us many cycles.

Key Addition. The key addition transformation from the straightforward implementation remains more or less untouched and can easily be reused in this implementation. Unlike the other linear transformations, we only apply it to one of the shares. This is natural because two key additions cancel each other out, so three key additions are equivalent to one key addition.

Mixing Layer. Like the key addition, we can easily reuse the mixing layer due to its linearity. A difference however is that we now need a version that is merged with the late shift. The merged version takes one cycle longer than the normal mixing layer. Compared to its unmerged counterpart, this is a reduction of 11 cycles. The merged version is used on the shares to which the key is not added except in the first round where no late shift is required.

Early Shift. This time the early shift is not merged with the nonlinear transform. As we soon will see, the threshold implementation version of the nonlinear transform has increased greatly in both complexity and code size compared to the one used in the straightforward implementation.

If an attempt would be made to merge the early shift with the nonlinear transform, the code size of the nonlinear transform would quadruple as each of the four slices requires different shifts. Also, we are uncertain if the shifts can efficiently be merged into the nonlinear transform in the first place because there seem to be many instructions that require not just one but two shifted operands. This would require us to spend one cycle to shift one of the operands first. Therefore, we expect the potential gain in performance to

be very small or even nonexistent. Because of time constraints, we decided not to attempt the merge.

Nonlinear Transform. Due to the nonlinearity of this transformation, we are required to completely rewrite the transformation such that it uses the shares correctly and the correctness property holds.

But before we deal with that, we first have another issue to deal with: the lack of general purpose registers. The S-box operates on three words at a time each split over three shares so we need nine registers to keep them in. Then we need nine more registers to store the resulting three words in as well as about 2 registers to store intermediate results in. So a naive implementation needs more registers than there actually are and would need to use SRAM to work around that.

To work around this issue we serialized the S-box. This means that we can compute the output words from the already computed output words and the untouched input words. At first we described the S-box as follows where x_0, x_1, x_2 are the input words and y_0, y_1, y_2 are the output words:

$$
\begin{aligned}
y_0 &= f(x_0, x_1, x_2) = x_0 \oplus (x_1 \vee \overline{x_2}) \\
y_1 &= g(x_0, x_1, x_2) = x_1 \oplus (x_2 \vee \overline{x_0}) \\
y_2 &= h(x_0, x_1, x_2) = x_2 \oplus (x_0 \vee \overline{x_1})
\end{aligned}
$$

As you can see, all output words are a function of the input words. Next is the serialized version in which output words are a function of output words and untouched input words:

$$
\begin{aligned}
y_0 &= f(x_0, x_1, x_2) = x_0 \oplus (x_1 \vee \overline{x_2}) \\
y_1 &= g(y_0, x_1, x_2) = x_1 \oplus (x_2 \vee y_0) \\
y_2 &= h(y_0, y_1, x_2) = x_2 \oplus (\overline{y_0} \vee y_1)
\end{aligned}
$$

Using these functions, we can overwrite input words with their respective output word, e.g. x_0 with y_0, without causing problems in the subsequent functions. This way, we no longer require SRAM to store intermediate results.

These functions were found using a trial-and-error strategy. It was not necessary to change function to compute y_0 because all input words were still available. To find $g(y_0, x_1, x_2)$ and $h(y_0, y_1, x_2)$, we repeatedly made a guess as to what it could be, then we made the corresponding truth table and compared it to the truth table of the original function. We stopped once we found a function with a matching truth table for each of the two functions, because if two functions have the same truth tables, they are equivalent.

Now that we have got the first problem out of our way, it is time to move on to the second task: the rewriting such that it handles the shares nicely. For each of the functions of the serialized S-box we use the same technique.

By applying it to the first function, $y_0 = f(x_0, x_1, x_2) = x_0 \oplus (x_1 \vee \overline{x_2})$, we will explain this technique. As the calculations are quite lengthy, some parts of the calculations are left out. The complete calculations for each of the functions can be found in the appendix.

We first rewrite the boolean expression into an algebraic expression in GF(2) one and simplify it. In the following functions, a + b means the sum of a and b in GF(2) and ab means the product of a and b in GF(2).

$$\begin{aligned} y_0 &= x_0 \oplus (x_1 \vee \overline{x_2}) \\ &= \dots \\ &= x_0 + (x_1 + 1)x_2 + 1 \end{aligned}$$

We must then realize that the words that it requires as input are actually split over three shares. For the sake of readability, in the calculations, we have named the shares differently: a, b and c instead of in_0, in_1 and in_2. Using this terminology, $x_i = a_i + b_i + c_i$. We use this knowledge to substitute the variables in the function with these expanded forms.

$$\begin{aligned} y_0 &= x_0 + (x_1 + 1)x_2 + 1 \\ &= (a_0 + b_0 + c_0) + ((a_1 + b_1 + c_1 + 1)(a_2 + b_2 + c_2) + 1) \end{aligned}$$

Afterwards, we simplify it. The result is a long list of terms.

$$\begin{aligned} y_0 &= (a_0 + b_0 + c_0) + ((a_1 + b_1 + c_1 + 1)(a_2 + b_2 + c_2) + 1) \\ &= a_0 + b_0 + c_0 + a_1a_2 + a_1b_2 + a_1c_2 + b_1a_2 + b_1b_2 + b_1c_2 \\ &+ c_1a_2 + c_1b_2 + c_1c_2 + a_2 + b_2 + c_2 + 1 \end{aligned}$$

Each term is composed of information from at most two shares. Next we split the terms in three groups where in each group information from one share is missing.

$$\begin{aligned} A_0 &= f_A(a, b) = b_0 + a_1b_2 + b_1a_2 + b_1b_2 + b_2 \\ B_0 &= f_B(b, c) = c_0 + b_1c_2 + c_1b_2 + c_1c_2 + c_2 + 1 \\ C_0 &= f_C(a, c) = a_0 + c_1a_2 + a_1c_1 + a_1a_2 + a_2 \end{aligned}$$

Each term is put in exactly one group. Therefore, when we add the groups we once again have y_0. This means the correctness property holds. The non-completeness property holds as well because we can compute y_0 in three parts, each in which information from one share is missing.

We still need to show that the uniformity property holds. We will prove it holds by showing that it is invertible. This means that a function $(A_0, \dots, C_2) \to (a_0, \dots, c_2)$ exists. Because the S-box is serialized, this is rather straightforward.

We start at the end i.e. functions h_A, h_B and h_C. At this point we have $(A_0, \ldots, C_2)$. Moving every value that we have knowledge of to the right-hand side and moving those we do not know to the left-hand side gives a function to compute a_2, b_2 and c_2 with, as is shown below:

$$b_2 = h_A(a, b)' = A_2 + A_0B_1 + B_0A_1 + B_0B_1 + B_0$$
$$c_2 = h_B(b, c)' = B_2 + B_0C_1 + C_0B_1 + C_0C_1 + C_0 + 1$$
$$a_2 = h_C(a, c)' = C_2 + A_0A_1 + A_0C_1 + C_0A_1 + A_0$$

Because we now know a_2, b_2 and c_2, we can repeat the same process on g_a, g_b and g_c to compute a_1, b_1 and c_1. Afterwards we repeat it on f_a, f_b and f_c to compute a_0, b_0 and c_0. We now have $a_0, \ldots, c_2$. This shows that the function is invertible.

Combine Shares. The combine shares transformation simply adds the three shares and is used at the end of the computations. Because of the correctness property, it results in the correct ciphertext. Instead of putting it at the very end, we put it before the reverse state transformation. This way, we combine the shares and then execute reverse state only once to reverse the combined shares. This is more efficient than the naive implementation, because we would then execute reverse state thrice before calling combine shares. This takes two additions executions of reverse state.

5.3 Results

In theory, this implementation protects against first-order side-channel attacks. It comes however at the expense of performance and code size. The decrease in performance can be explained by the fact that most transformations now need to be applied to all three shares instead of just one. Another important factor is frequent memory accesses. Although we tried to minimize the amount of accesses by switching shares as little as possible, it still has significant impact on the cycle count. In table 5.1 is a breakdown of the threshold implementation, showing for each transformation what instructions were used and how many cycles it took. The cycles were counted in the same way as they were counted for the unprotected implementation.

Transformation	Instructions	Cycles	Times used	Total cycles	% of total
Create Shares	add x5, eor x24, ldr x47, str x30	123	1	123	1.2%
Key Addition	eor x16, ldr x13	35	1	35	0.4%
Late Shift + Key Addition	eor x5, eor+ror x15, ldr x13	35	11	385	4.0%
Mixing Layer	eor x59, ldr x11, mov x3, str x8	89	14	1246	12.9%
Late Shift + Mixing Layer	eor x13, eor+ror x46, ldr x11, mov x1, mov+ror x3, str x8	90	22	1980	20.4%
Early Shift	mov+ror x11	11	33	363	3.7%
Nonlinear Transform	and x108, eor x204, mov x12, mvn x8	332	11	3652	37.9%
Combine Shares	add x3, eor x24, ldr x46, str x6	85	1	85	0.8%
Reverse State	eor x18	18	1	18	0.2%
Control Logic	bl x80, blt x10, cmp x10, ldr x788, mov x3, str x522	1803	1	1803	18.6%

Table 5.1: Breakdown of DoubleKing's threshold implementation

In total, encryption of 384 bits with this implementation takes 9690 cycles, so we have a latency of 9690 cycles and a throughput of 25.2 cycles per bit. The code size is 3592 bytes and during execution it uses 106 distinct words (= 424 bytes) in SRAM.

Chapter 6

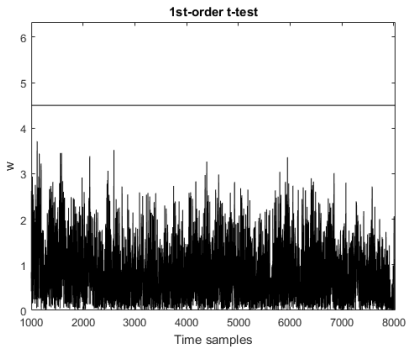
Side-channel Analysis

To test if the threshold implementation protects against first-order differential power attacks, we performed several t-tests. A t-test is a tool for distinguishing noisy signals and can be used as a preliminary test to see if there are leakages in an implementation of a cryptographic program.[13] In our case, the goal of the t-test was to see if we could distinguish traces of runs of the first round of DoubleKing with fixed plaintext from traces of runs with random plaintext.

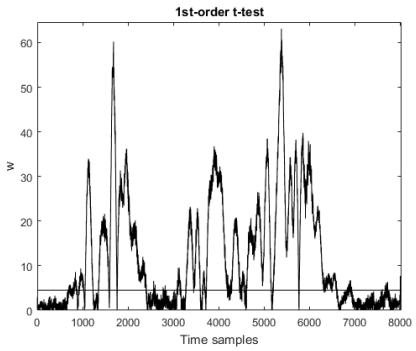
The theoretical security of a threshold implementation is that of a second-order attack. Unfortunately, distance-based leakages and other problems[2][5] cause deployed implementations to lose order of security. Solutions include increasing the order and/or hardening the implementation. The point of a threshold implementation is to make a first-order attack very inconvenient to the adversary such that he is forced to use a second-order attack instead. Next, we discuss how we examined if our implementation loses orders of security.

6.1 T-tests

We employed a random vs. fixed, first-order t-test. We first collected two distinct sets of traces. The first set, S_{fixed}, consists of traces of runs of the first round of DoubleKing where the plaintext was the same every time. The other set, S_{random}, consists of traces of runs where a uniformly random plaintext was provided. The types of runs were randomly interleaved and each run used the same encryption key.



(a) 450k fixed vs. 450k random t-test



(b) 5k fixed vs. 5k random t-test

A t-test tries to assess whether the two sets S_{fixed} and S_{random} stem from the same population. To do this it, uses a statistical test that takes into account the distance of their means, as well as their variability and cardinality.[7] For a more detailed description, we refer to [6]. If the result of the t-test says that the two sets do not stem from the same population, it implies leakage detection which could mean that the protective scheme is not working.[11]

6.2 Experiments

On an ARM Cortex-M4F core with a clock speed of 168MHz, we captured 300.000 power traces at a sampling rate of 500Msamples/second using the method described above. When we used these traces to do a first-order t-test with 130k fixed vs. 130k random traces, we did not detect leakages. We later repeated the experiment and captured close to a million traces. As we can see in figure 6.1a, even when we do a 450k fixed vs. 450k random traces t-test, we do not detect first-order leakage. Because of the protection that the threshold implementation offers, the boundary set at about $w = 4.5$ is not exceeded. Without going into too much detail, if that boundary is exceeded, it means that it has reached the level of confidence required to reject the null hypothesis (that says that the sets stem from the same population). Because we did not exceed that boundary, it means we cannot significantly distinguish the types of traces from one-another, implying no leakage was detected.

To verify we are looking at the right spot, we repeated the acquisition of traces, but this time the random number generator was disabled such that the implementation would use the same random values every time. This should render the threshold implementation useless. As expected, we were able to detect first-order leakage very quickly; using only 5000 traces of each type. As we can see in figure 6.1b, the boundary is exceeded at many points in time, implying leakage.

From these experiments we conclude that the threshold implementation on this particular device (the ARM Cortex M4) does not lose orders of security. This is despite the fact that at some points in the implementation, registers and memory buses are overwritten with information from the remaining missing shares. These problems have been shown to cause issues in AVR devices.[11]

Chapter 7

Results

We have made two optimized ARMv7 implementations of DoubleKing: a variant of BaseKing that uses 32-bit words. One implementation is unprotected and the other uses a threshold scheme which provides theoretical resistance against first-order DPA attacks. We confirmed the threshold implementation's resistance against first-order DPA attacks by conducting serveral t-test. Even with a t-test using close to a million traces we did not detect significant leakages. Therefore we conclude that the threshold implementation indeed does protect against first-order DPA attacks.

In table 7.1 several statistics on both implementations are shown. It takes the threshold implementation about 4.5 times as many cycles to encrypt as the unprotected implementation. This can easily be explained by the fact that most transformations now need be applied to three shares instead of just one. The nonlinear transform also is an important factor, because it is called eleven times and each time takes 332 cycles to complete. This is a significant increase compared to the unprotected implementation where each call only takes 44 cycles. Because there are only sixteen usable registers, shares need to be moved from and to memory regularly. This is takes about a thousand cycles. Due to limitations on code size we had to use subroutines instead of macros which cost us a few hundred cycles.

	Unprotected impl.	Threshold impl.
Number of cycles	2127	9690
Latency in cycles	2127	9690
Throughput in cycles/bit	5.54	25.2
Code size in bytes	1776	3592
SRAM size in bytes	180	424

Table 7.1: Comparison of the implementations

The difference in the amount of cycles between the unprotected implementation and the threshold implementation was to be expected. To a large extend, the remaining sources of overhead in the threshold implementation are inevitable. Despite the large increase, it could have been a lot worse if we had not used the optimizations discussed in chapter 4 and 5. That is why it seems reasonable to us to call it a high-performance implementation.

Bibliography

[1] ARM. *Cortex-M4 Devices - Generic User Guide*, 2010.

[2] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014.

[3] Joan Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis.* PhD thesis, 1995.

[4] Joan Daemen, Michael Peeters, and Gilles van Assche. Bitslice ciphers and power analysis attacks. In *International Workshop on Fast Software Encryption*, pages 134–149. Springer, 2000.

[5] Wouter de Groot, Kostas Papagiannopoulos, Antonio de La Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and arm: Friends or foes? In *International Workshop on Lightweight Cryptography for Security and Privacy*, pages 91–109. Springer, 2016.

[6] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 15–29. Springer, 2006.

[7] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation.

[8] B Guido, D Joan, P Michaël, VA Gilles, and VK Ronny. K implementation overview. 2012.

[9] Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of arm and thumb instructions. In *ACM SIGPLAN Notices*, volume 37, pages 56–64. ACM, 2002.

[10] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[11] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. *IACR Cryptology ePrint Archive*, 2017:345, 2017.

[12] Peter Schwabe and Ko Stoffelen. All the aes you need on cortex-m3 and m4. 2016.

[13] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations.

[14] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

Appendix A

Appendix

A.1 Equations for TI S-Box

Below are the complete calculations for each of the functions of the S-box in the threshold implementation.

$$
\begin{aligned}
y_0 &= x_0 \oplus (x_1 \vee \overline{x_2}) \\
&= x_0 + (x_1 \vee (x_2 + 1)) \\
&= x_0 + (x_1(x_2 + 1) + x_1 + (x_2 + 1)) \\
&= x_0 + x_1x_2 + x_1 + x_1 + x_2 + 1 \\
&= x_0 + x_1x_2 + x_2 + 1 \\
&= x_0 + (x_1 + 1)x_2 + 1 \\
&= (a_0 + b_0 + c_0) + ((a_1 + b_1 + c_1 + 1)(a_2 + b_2 + c_2) + 1) \\
&= a_0 + b_0 + c_0 + a_1a_2 + a_1b_2 + a_1c_2 + b_1a_2 + b_1b_2 + b_1c_2 \\
&+ c_1a_2 + c_1b_2 + c_1c_2 + a_2 + b_2 + c_2 + 1
\end{aligned}
$$

$$
\begin{aligned}
A_0 &= f_A(a, b) = b_0 + a_1b_2 + b_1a_2 + b_1b_2 + b_2 \\
B_0 &= f_B(b, c) = c_0 + b_1c_2 + c_1b_2 + c_1c_2 + c_2 + 1 \\
C_0 &= f_C(a, c) = a_0 + c_1a_2 + a_1c_1 + a_1a_2 + a_2 \\
y_0 &= A_0 \oplus B_0 \oplus C_0
\end{aligned}
$$

$$
\begin{aligned}
y_1 &= x_1 \oplus (x_2 \vee y_0) \\
&= x_1 + (x_2y_0 + x_2 + y_0) \\
&= y_0(x_2 + 1) + x_1 + x_2 \\
&= (A_0 + B_0 + C_0)((a_2 + b_2 + c_2) + 1) + (a_1 + b_1 + c_1) + (a_2 + b_2 + c_2) \\
&= A_0a_2 + A_0b_2 + A_0c_2 + A_0 + B_0a_2 + B_0b_2 + B_0c_2 + B_0 \\
&+ C_0a_2 + C_0b_2 + C_0c_2 + C_0 + a_1 + b_1 + c_1 + a_2 + b_2 + c_2
\end{aligned}
$$

$$
\begin{aligned}
A_1 &= g_A(a,b) = A_0b_2 + B_0a_2 + B_0b_2 + B_0 + b_1 + b_2 \\
B_1 &= g_B(b,c) = B_0c_2 + C_0b_2 + C_0c_2 + C_0 + c_1 + c_2 \\
C_1 &= g_C(a,c) = A_0a_2 + A_0c_2 + C_0a_2 + A_0 + a_1 + a_2 \\
y_1 &= A_1 \oplus B_1 \oplus C_1
\end{aligned}
$$

$$
\begin{aligned}
y_2 &= x_2 \oplus (\overline{y_0} \vee y_1) \\
&= x_2((y_0 + 1) \vee y_1) \\
&= x_2 + (y_1(y_0 + 1) + (y_0 + 1) + y_1) \\
&= x_2 + y_0y_1 + y_1 + y_0 + 1 + y_1 \\
&= x_2 + y_0(y_1 + 1) + 1 \\
&= (a_2 + b_2 + c_2) + (A_0 + B_0 + C_0)(A_1 + B_1 + C_1 + 1) + 1 \\
&= a_2 + b_2 + c_2 + A_0A_1 + A_0B_1 + A_0C_1 + A_0 + B_0A_1 \\
&+ B_0B_1 + B_0C_1 + B_0 + C_0A_1 + C_0B_1 + C_0C_1 + C_0 + 1
\end{aligned}
$$

$$
\begin{aligned}
A_2 &= h_A(a,b) = b_2 + A_0B_1 + B_0A_1 + B_0B_1 + B_0 \\
B_2 &= h_B(b,c) = c_2 + B_0C_1 + C_0B_1 + C_0C_1 + C_0 + 1 \\
C_2 &= h_C(a,c) = a_2 + A_0A_1 + A_0C_1 + C_0A_1 + A_0 \\
y_2 &= A_2 \oplus B_2 \oplus C_2
\end{aligned}
$$