2015-08-26

Lightweight Cryptography Meets Threshold Implementation: A Case Study for SIMON

Aria Shahverdi
Worcester Polytechnic Institute

Lightweight Cryptography Meets Threshold Implementation: A Case Study for Simon

by

Aria Shahverdi

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

by

August 2015

APPROVED:

Professor Thomas Eisenbarth Major Advisor	Professor Berk Sunar Thesis Committee
Professor Mostafa Taha Thesis Committee	Professor Yehia Massoud Department Head

Abstract

Securing data transmission has always been a challenge. While many cryptographic algorithms are available to solve the problem, many applications have tough area constraints while requiring high-level security. Lightweight cryptography aims at achieving high-level security with the benefit of being low cost.

Since the late nineties and with the discovery of side channel attacks the approach towards cryptography has changed quite significantly. An attacker who can get close to a device can extract sensitive data by monitoring side channels such as power consumption, sound, or electromagnetic emanation. This means that embedded implementations of cryptographic schemes require protection against such attacks to achieve the desired level of security.

In this work we combine a low-cost embedded cipher, Simon, with a state-of-the-art side channel countermeasure called Threshold Implementation (TI). We show that TI is a great match for lightweight cryptographic ciphers, especially for hardware implementation. Our implementation is the smallest TI of a block-cipher on an FPGA. This implementation utilizes 96 slices of a low-cost Spartan-3 FPGA and 55 slices a modern Kintex-7 FPGA. Moreover, we present a higher order TI which is resistant against second order attacks. This implementation utilizes 163 slices of a Spartan-3 FPGA and 95 slices of a Kintex-7 FPGA. We also present a state of the art leakage analysis and, by applying it to the designs, show that the implementations achieve the expected security. The implementations even feature a significant robustness to higher order attacks, where several million observations are needed to detect leakage.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Thomas Eisenbarth, for his guidance, patience and advice during my research. I also want to thank him for providing such an excellent atmosphere for doing research. His patience and support made it easy to overcome the problems I faced throughout my research. It was an honor to be able to work with him in the past two years.

I also want to thank Prof. Mostafa Taha for his help during this research. His expertise in this field was invaluable. I would like to thank my thesis committee, Prof. Berk Sunar, for his valuable suggestions and comments on my thesis.

I would like to thank Gorka Irazoqui Apecechea, Berk Gulmezoglu, Xin Ye, Wei Dai, Michael Moukarzel, Marc Green, Yarkin Doroz, Gizem Selcan Cetin, Mehmet Sinan Inci and Cong Chen for creating such a great atmosphere in Vernam Group.

Last but not least, I would like to thank my father Farhad, my mother Homa and my sister Bahar. They have provided me with support and encouragement during my graduate studies at Worcester Polytechnic Institute.

This material is in part based upon work supported by the National Science Foundation under Grant No. 1261399.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the past couple of years we have seen numerous small devices got connected to each other. Some of these devices are not considered to be critical in terms of security such as light bulb, smart TV and toaster while the others can be critical such as heart monitoring implants. Based on the application of these devices some minimum requirement is needed in term of transmitting data securely to servers or within devices.

The solution for transmitting data securely has been studied for a long time and cryptography provide us the secure channel. These algorithms work fine if we assume that the adversary has access only to the data channel. This is not the case anymore when we talk about small devices which can be found anywhere. The modern adversary can get close to the device, measure the electromagnetic emanation of the device. In some cases, an adversary has physical access to the device and can even connects a wire to that.

Once these assumptions are taken into account those secure algorithms are not secure anymore. An adversary with physical access to the device has the ability to do the most dangerous type of attacks. Considering these adversaries, having an implementation of the cryptographic algorithm is not enough and some type of mitigation against physical attack needs to be applied.

As soon as power analysis attacks were discovered by Kocher et al. in [KJJ99], effort has been made to propose ways in order to protect the implementation. One of the first contribution was done by Chari et al. in [CJRR99]. In this work the author first discusses the behavior of a device and how a device consumes power in general way. They looked at the CMOS devices and assume that the main source of power consumption is transition to other states and maintaining the current state does not need much power. Based on this assumption and also the need for making the intermediate value independent of key, they introduced splitting the state into several parts by using some random numbers. This division is done in a way that combining all the shares will recover the original value and combining all except one will not reveal any information.

One of the practical implementation using this scheme is done by Akkar et al. [AG01] for both AES and DES. Later, Oswald et al. [OMPR05] presented a way to mask the AES S-Box. As it was shown in [MPO05], these two implementations are vulnerable to more sophisticated attacks. Mangard et el. first show that by using Hamming weight as a power model they can not successfully recover the key from the protected design. However, they used the simulation to obtain a new power model which is basically toggle count for a specific output. They used the mean of the transition count as their power model and they showed that by doing so, the implementation is prone to practical attacks. This model works because the delay in the input of logical gates are not the same for each input. As a result of these differences in arrival time the output of a circuit will toggle couple of times before it reaches the final result. In a separate work, Mangard et al. [MPG05] show that the power consumption of a device is correlated to unmasked value in the presence of glitches.

One of the first efforts to counteract glitches is done by Fischer et al. in [FG05]. The first working solution, on the other hand, is proposed by Nikova et al. in [NRR06]. The idea is based on the secret sharing and it is called `Threshold Implementation`. One of the interesting feature of their scheme is the need of randomness only in the starting point of the algorithm and there is no need for fresh randomness after that. We are going to introduce this scheme in more details in Section 3.

There are several works published based on the idea of threshold implementation. Kutzner et al. in [KNP12] shows the implementation of 4-bit S-Boxes using 3 shares. Another work by Moradi et al. [MPL$^+$11] tries to implement the well-known cipher, namely AES in a small area. It was shown that the threshold version of AES can be implemented by using approximately 11000 GE. Bilgin et al. in [BGN$^+$14a] improve the result even more and implemented threshold implementation of AES using 9000 GE.

Recent works focus on the higher-order threshold implementation. For example, Bilgin et al. in [BGN$^+$14b] discussed the theory of higher-order threshold implementation as well as practical implementation. They also presented the resistance of their core by analyzing 300 million traces and showed that there is no leakage in those traces.

In this work, to analyze an implementation for leakage, a new methodology will be used which was proposed by Goodwill et al. in [GJJR11]. This leakage detection method can be used to observe whether the device leaks or not. An enhancement to this method is published by Becker et al. in [BCD$^+$13].

1.2 Our Contribution

In this work, we chose Simon as a cryptographic algorithm due to its small area overhead. We focused on one of the existing solution, i.e. threshold implementation, against an attacker with physical access to the device. We first investigate the vulnerability of unprotected Simon by presenting an actual attack as well as using leakage detection methods. As a method for securing Simon against side-channel attacks, a first order threshold implementation for Simon is proposed and its resistance is also shown by leakage detection method [STE15]. The equation for a core resistant against second order attacks is also proposed and its efficiency is also shown by leakage detection method based on actual power traces.

1.3 Outline of the work

In Chapter 2 we start by introducing the background on attacks and ways of protecting against them. In the same chapter we present a lightweight cipher, namely Simon in more detail. Then we introduce a mitigation method in Chapter 3. The protected version of Simon is introduced in Chapter 4. We present our analysis in Chapter 5 and conclude the work in Chapter 6.

Chapter 2

Background

Until around late nineties, the focus of research in cryptography was on proving that only by observing plaintexts and ciphertexts the key being used by the system will not be revealed. There has been some interests in breaking the cryptographic schemes by using some novel ideas such as inducing an error [BDL97] or measuring the computation time [Koc96]. The seminal work by Kocher et al. [KJJ99] was among these efforts which shows that by observing the amount of power the device uses during encryption, useful information can be extracted from the device, such as when a certain operation is being done. From now on these types of observation which leads to extraction of useful data are called leakage.

The most important information is the one that depends both on the plaintext and the key being processed. Using this information can result in obtaining the key. In order to protect the algorithm against these types of attacks numerous countermeasures have been proposed. In this section we take a look at how the attacks work and ways of protecting the device against them.

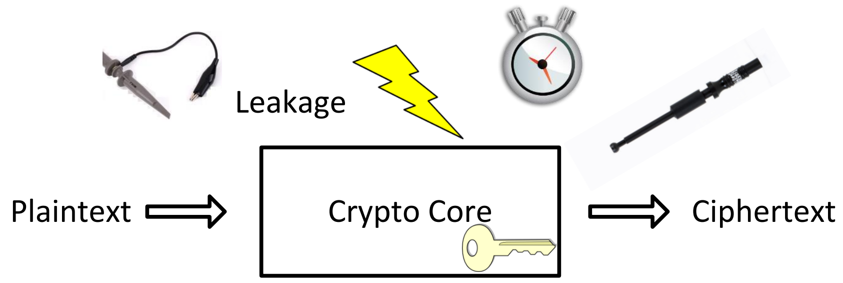


Figure 2.1: New Way of Looking at Cryptography

2.1 Side-Channel Attack

As it was stated, the leakage can help the attacker to extract useful information about the data being processed. As it can be seen in Figure 2.1, the attacker who has access to the device can simply send his desired plaintext to it.

Then the cryptographic algorithm, which is shown by Crypto Core in the Figure 2.1, returns the ciphertext by using the plaintext as an input and key as its secret internal value. The attacker has access to ciphertext and by having physical access to device he can also perform additional measurement in order to figure out how the device acts during the run time of the cryptographic algorithm. As it is also shown in Figure 2.1, there are different kinds of measurement that can be performed such as measuring the computation time and electromagnetic emanation. Power consumption is also one type of observation and throughout this thesis we are going to focus on it as a source of leakage.

As it can be seen in Figure 2.2, resistor R is placed in the route of VCC to GND. The amount of power that Crypto Device consumes will result in changes of current going through the device. By simply measuring the voltage V_o and dividing that value by R that current can be calculated. The power consumption of a device can be formulated as follows:

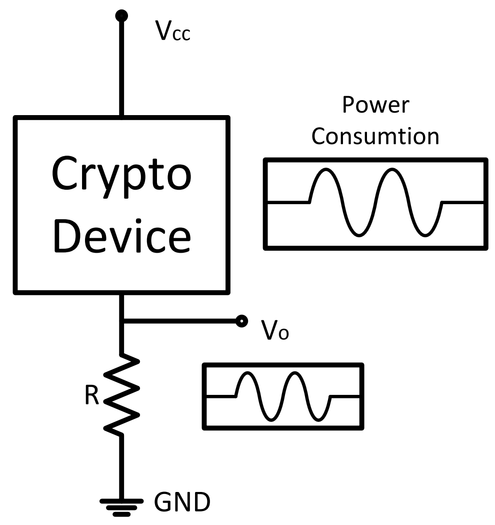


Figure 2.2: Power Measurement Setup

$$\text{Power[Crypto Device]} = Vcc \times \frac{V_o}{R} \tag{2.1}$$

The Equation 2.1 shows that the power consumption of a device depends directly on the V_o. The common setup for measuring power is as it is shown in Figure 2.2. The oscilloscope (which from now on we refer to it as a scope) records the value of V_o and we treat that value as a power consumption of a device.

There are two main types of attacks that can be done based on this power consumption. In the rest of this section we look at both of them.

2.1.1 Simple Power Analysis

In this type of attack it is assumed that the adversary has access to only one measurement or a few measurements. As it is crucial to know exactly what is happening in each time instance, in order for an attack to be successful, the attacker should know the details of the implementation. As and example for this type of attack we look

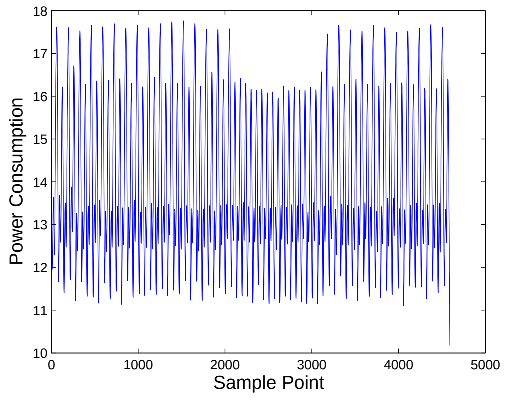


Figure 2.3: Power Consumption of an RSA algorithm

at an algorithm which performs RSA. Figure 2.3 represents the power consumption of a device which computes an RSA exponentiation using the square-and-multiply algorithm after filtering noises based on the work by Do et al. [DKH$^+$13]. The square-and-multiply algorithm performs squaring operation in each steps but multiplication is only performed when the bit in the exponent is equal to a one bit. As it can be seen in the figure, the peaks with smaller amplitude happen all the time. The larger peaks, on the other hand, only happens at some points. From this observation we can assign the power trace to the exponent. In the points where only one pick (smaller peaks) happens the bit in the exponent is 0. The other points which have both smaller and larger peaks can be corresponded to bit 1.

This attack works in this case because the implementation was completely known to us. This is not always the case.

The power consumption of the first round of AES can be seen in Figure 2.4. Although the AES algorithm is fully known to us, recovering the key from this figure, just by looking at it, is not a trivial task.

In the next subsection we introduce Differential Power Analysis (DPA) which can recover the key even in the scenarios where the details of implementation are not known to us.

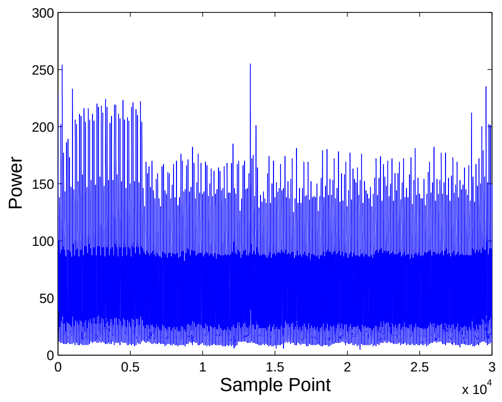


Figure 2.4: AES Power Consumption

2.1.2 Differential Power Analysis

In this type of attack the adversary does not use the details of the implementation. For attacking a device using DPA, large number of traces should be recorded. In contrast to SPA where we look at one trace over time, in a DPA attack statistical methods will be used to perform the attack. Figure 2.5 shows the steps to perform DPA attack, we introduce them in the following.

Choosing A Point to Attack DPA attacks can recover the key. The intermediate variable that is chosen should depend on a known value (X) and an unknown key (K). We focus on the AES algorithm in this example. As it can be seen in Figure 2.6, the chosen point is the output of S-Box layer.

$$Z = \mathsf{S} - \mathsf{Box}(Y) = \mathsf{S} - \mathsf{Box}(X \oplus K)$$

Measurement The next step is to run the cryptographic algorithm for a large number of known plaintexts. In our case we call them $d = (d_1, d_2, \ldots, d_D)$. The Crypto Device will perform encryption on this plaintexts and using the scope we record the power consumption during this process. The number of samples in each trace is denoted by T. The samples measured for encryption

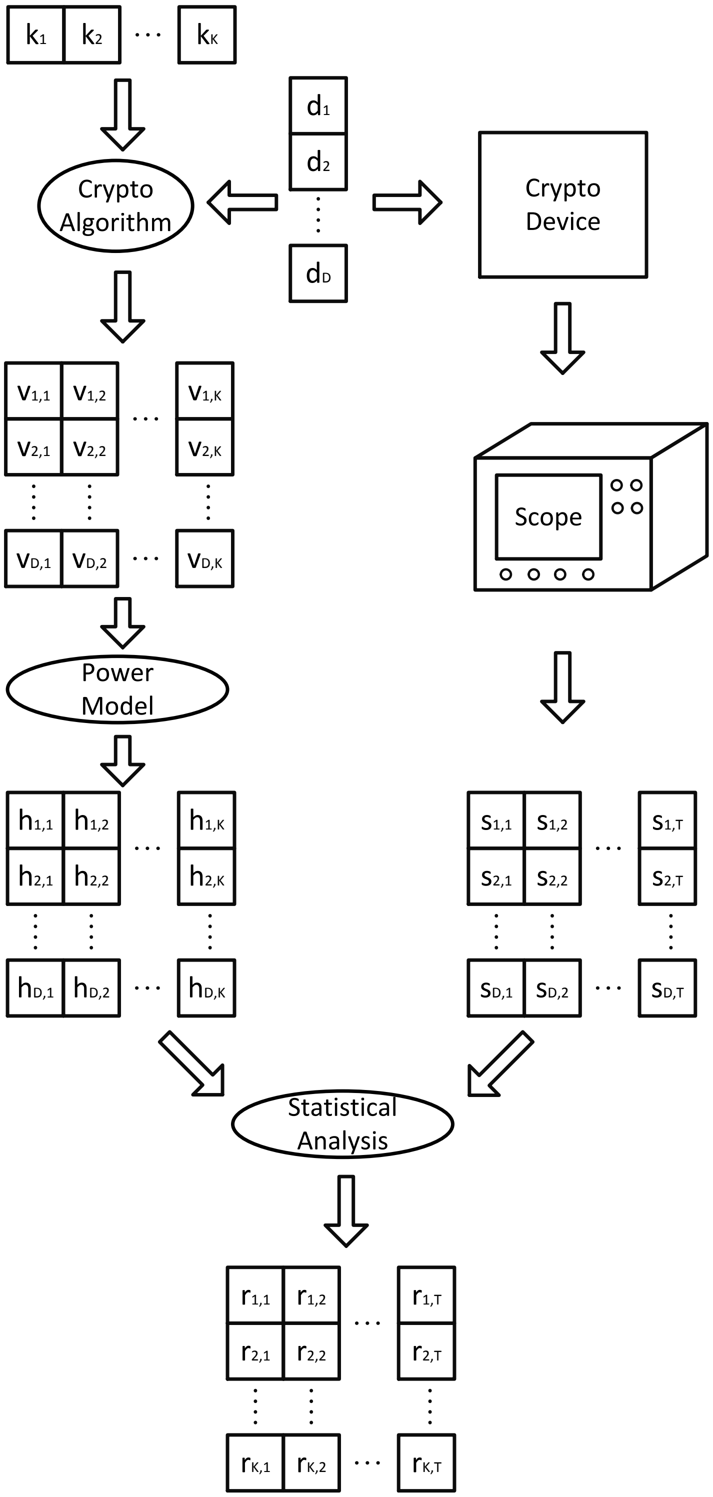

k_1 k_2 … k_K
d_1 d_2 … d_D
Crypto Algorithm
Crypto Device
$v_{1,1}$ $v_{1,2}$ … $v_{1,K}$
$v_{2,1}$ $v_{2,2}$ … $v_{2,K}$
$v_{D,1}$ $v_{D,2}$ … $v_{D,K}$
Scope
Power Model
$h_{1,1}$ $h_{1,2}$ … $h_{1,K}$
$h_{2,1}$ $h_{2,2}$ … $h_{2,K}$
$h_{D,1}$ $h_{D,2}$ … $h_{D,K}$
$s_{1,1}$ $s_{1,2}$ … $s_{1,T}$
$s_{2,1}$ $s_{2,2}$ … $s_{2,T}$
$s_{D,1}$ $s_{D,2}$ … $s_{D,T}$
Statistical Analysis
$r_{1,1}$ $r_{1,2}$ … $r_{1,T}$
$r_{2,1}$ $r_{2,2}$ … $r_{2,T}$
$r_{K,1}$ $r_{K,2}$ … $r_{K,T}$

Figure 2.5: DPA model

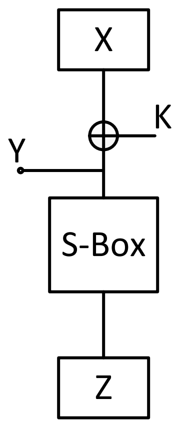


Figure 2.6: DPA analysis of AES

d_i is denoted by s_i The measured power consumption for input d_i is then $s_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,T})$.

Simulation In this step we build a matrix based on known input d_i and key hypothesis k_j. Each element of this matrix which is called V is denoted by $v_{i,j}$ and the equation for obtaining each element is

$$v_{i,j} = \mathsf{S} - \mathsf{Box}(d_i \oplus k_j)$$

In the real measurement since the key value is fixed, only one column of matrix V will be recorded. As a result of DPA attack the correct key will be found.

Modeling the Power Consumption Everything up to now was performed either using simulation or by measuring the actual power consumption, in this step we try to establish a link between them. The elements of matrix V represents the value of the intermediate step of the algorithm. In this step, information about the device is needed to estimate the power consumption of the device based on these intermediate values. Among the accepted models `Hamming distance` and `Hamming weight`, `Least Significant Bit` and `Most Significant Bit` can be used. The `Hamming distance` model is based on how many bits transition occur from one state to another, while the `Hamming weight` model just look at the result and does not care about the transition. In the LSB and MSB model only the right most and left most bit will take into account, respectively. The elements of this hypothesis matrix H is denoted by $h_{i,j}$ and they can be derived from $v_{i,j}$ as following

$$h_{i,j} = \mathsf{Model}(v_{i,j})$$

As it was mentioned the most common `Model` functions are `Hamming distance`, `Hamming weight` and `LSB` or `MSB` of a register. Based on how accurate this `Model` function represents the true behavior of the device, the quality of DPA attack will differ.

Comparing the Hypothesis Matrix with Actual Traces In this step a statistical tool such as correlation is needed. The goal of this step is twofold. The first result of doing the comparison will give some information on when the chosen point is being processed. The second outcome is giving some information on the actual key that was used in the Crypto Device.

Here we see an example of a DPA attack on the AES algorithm. The chosen point is the output of the first S-Box in the first level of AES. The number of traces that have been recorded is equal to 500 ($D = 500$), and each trace contains 30,000 points ($T = 30,000$). The matrices V and H are computed by computing the intermediate result and modeling that intermediate result to hypothesis matrix by using `Hamming weight` as power model. The result of correlation-based DPA attack can be seen in Figure 2.7.

The black trace shows the correct key hypothesis and the rest of them are the wrong hypothesis. As it can be seen for the correct key, the peaks show the time

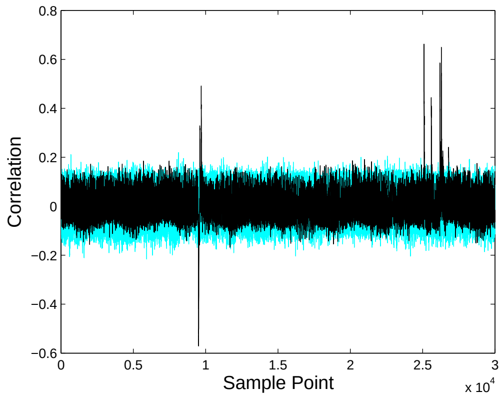
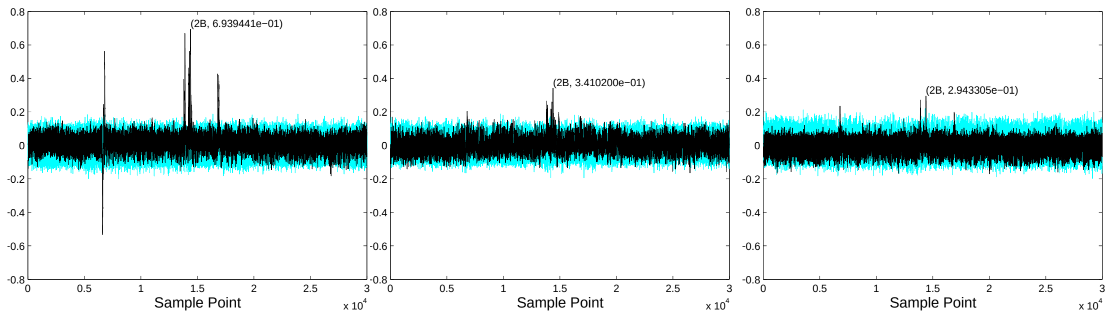


Figure 2.7: Result of correlation-based DPA attack on first round of AES



(a) HW as power model (b) MSB as power model (c) LSB as power model

Figure 2.8: Effect of different power model on correlation value

where the chosen point (in this case, the output of S-Box) is being processed. The correct key in the figure can be distinguished from the wrong ones, which means that the chosen power model was successful to describe the real power consumption of the device.

Figure 2.8 represents the result of DPA attack on the first S-Box by using different power model. As it can be seen the maximum value of correlation occurred in Figure 2.8a. This means the Crypto Device that has been attacked is probably leaking in a way that is close to `Hamming weight` model.

In the following we are going to look at some countermeasures that, to some extent, can prevent these type of attacks.

2.2 Side-Channel Countermeasures

As it was shown in the previous section, the attacks that are based on power analysis can extract the key from the side-channel leakages. The main reason that those attacks were successful was because of the fact that the power consumption depends one the intermediate value being processed. Countermeasures break this link. Since the introduction of such attacks, countermeasures to prevent them have started to developed [CJRR99, AG01]. In this section two of those countermeasures have been discussed.

2.2.1 Hiding

The goal of hiding is to make the power consumption of a device independent of the intermediate value. Hiding is implemented either through time or the amplitude domain.

The algorithm can randomly change the time allotted to complete the operation. This can be done by adding some random delay between two consecutive operations or change the order of the operations. By adding the random delay to the algorithm, the time instance where the leakage occurs will change. The attacker can try all different possible time instances and perform the attack for all of them. The order of some independent operations can be changed, e.g. S-Box look-up. By changing the order (also known as shuffling) the attack will become harder but not impossible. There are other ways to hide the power consumption which modify the amplitude.

In hardware, a natural way to achieve this is to perform several operations in parallel. Another way is to add a separate unit to the circuit to generate additional noise.

Another way to perform hiding is to design a circuit which consumes same amount of power all the time. The idea is to add a parallel logic to the circuit which processes the complement of the original data. If all the operands can be realized using this encoding the circuit will consume constant power all the time. One of the recent work is done by Cong et al. [CESY14]. In their paper they proposed to use a special encoding for data that keeps the complement value of the data inside the encoded value. They showed that by using this scheme they can reduce the correlation coefficient of the DPA attack.

All the countermeasures proposed in this section will make the attack harder but an adversary can break the algorithm with sufficiently many traces.

2.2.2 Masking

Another class of countermeasures is masking, which processes completely random values, created from the original values by adding random masks, during the algorithm and at the end combines them in a way that the correct result can be recovered [CJRR99].

Lets assume the secret value is x and some random number m is also available. From now on m will be called mask value. Masking can be applied both at the gate level and at the algorithm level. Gate level masking is more generic and can be applied to every new algorithm. On the other hand, the algorithmic level masking needs to be redesigned for new ciphers. In this work we focus on masking schemes that will be applied at the algorithm level.

There are two types of masking schemes, arithmetic masking and Boolean masking. In arithmetic masking the operation to create the random value is performed by doing either modular addition or modular multiplication. In this work, the independent values are obtained by `XOR`ing the secret with random mask which is a Boolean masking. The circuit will process $x \oplus m$ and m in parallel and independently. At the end the result of both circuit will be combined. The masking scheme can combine the original data and the mask in different ways such as addition, multiplication and modular addition (such as XOR). In the following we discuss how the masking based on addition modulo two (XOR) works.

Lets assume that the data to be processed is x and y and the masks generated for them are m_x and m_y, respectively. Secret value x and y will be divided into two shares as follows

$$\begin{aligned} x \text{ will become } \{x_m, m_x\} &\qquad x_m = x \oplus m_x \\ y \text{ will become } \{y_m, m_y\} &\qquad y_m = y \oplus m_y \end{aligned}$$

The different types of operation can be realized as following.

Linear Operation For performing such an operation, it is enough to do the same computation on each share separately. Lets assume the linear operation is L

$$L(x) = L(x_m \oplus m_x) = L(x_m) \oplus L(m_x)$$
$$L(x, y) = L(x_m \oplus m_x, y_m \oplus m_y) = L(x_m, y_m) \oplus L(m_x, m_y)$$

From the above equations it is shown that the operation can be done on each share separately and the final result of the operation is XOR of the result of each share.

Non-linear Operation The only non-linear operation in modulo 2 is AND and the steps to perform the masked version of AND is as follows which is based on the work by Trichina et al. [TKL05].

$$\begin{aligned} NL(x,y) = x \otimes y &= (x_m \oplus m_x) \otimes (y_m \oplus m_y) \\ &= (x_m \otimes y_m) \oplus (x_m \otimes m_y) \oplus (m_x \otimes y_m) \oplus (m_x \otimes m_y) \end{aligned}$$

In order to use the result of the multiplication in the next stage of the circuit a mask should be added to the above equation. We denote this mask with m_z and then the output masking of the multiplication becomes

$$z \text{ will become } \{z_m, m_z\} \qquad z_m = (x \otimes y) \oplus m_z$$

Figure 2.9 shows the implementation of such a circuit.

The masking algorithm explained above works under one condition and that is the necessity that all the input to the circuit arrives at exactly the same time. In hardware ensuring such a condition is not possible. Every input to the masked `AND` comes from different part of the circuit and the delay of each path is different. Even if we assume that all inputs are coming at the same time, there might be some delay added by the circuits inside the gates.

The different arrival time for the circuit causes the output to toggle a couple of times before reaching the final result. This effect is called glitches. Mangard et al. in [MPG05] showed that a masked implementation of `AND` gate is not secure against DPA attacks. In another work, Mangard et al. [MPO05] successfully attacked the masked version of AES. They built a power model based on the simulation of back-annotated netlist of their design. The power model was obtained by counting the number of transition in the simulation also known as toggle count model. The number of transition depends on the input of the circuit, even if the input is masked.

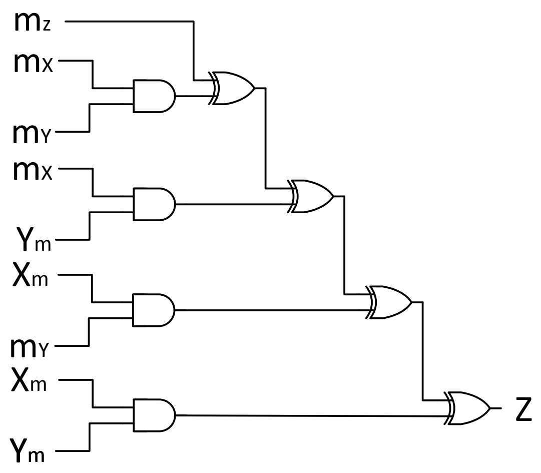


Figure 2.9: Masked AND gate

Finally, they showed the possibility of attacking the masked version of AES in the presence of glitches. This type of attacks motivates us to look at the other masking schemes which can withstand glitches. This type of countermeasures is introduced in Section 3.

2.3 Simon

Classical ciphers were designed with having the confidentiality of the plaintext given only the ciphertext in mind. Recently some small ciphers, also known as lightweight cryptographic ciphers, have been proposed for embedded systems. These lightweight solutions were designed for environments where the area is a limitation. SIMON and SPECK are two ciphers that have been recently proposed by NSA [BSS$^+$13, BSS$^+$15] and it is shown that their implementation is low-cost. These two lightweight ciphers accepts different types of plaintext and key as an input. The block size can be 32, 48, 64, 96 and 128 bits. For each input size, they have a set of allowable key sizes ranging from 64 bits to 256 bits. In this chapter we look at SIMON in more detail.

SIMON is designed to be efficient in hardware. SIMON will be denoted as SIMON$2n/mn$ in which $2n$ shows the size of input block and mn is the size of key. For performing key schedule the input key will be divided into m blocks of size n bits each. For example SIMON96/144 denotes a cipher with 96 bits plaintext which accepts keys of size 144 bits and for performing key schedule the key will be divided into 3 blocks with size of 48 bits. SIMON has Feistel network structure which can be implemented efficiently in hardware. Each round of SIMON has simple operations, namely, bitwise `XOR` and bitwise `AND` and also circular shifts which is simply done by proper wiring. Based on the requirements imposed on the designer, different configuration for SIMON can be selected. Table 2.1 represents those configurations. In the following we discuss in details how each part of the cipher works.

Plaintext Size (bits)	**Key Size** (bits)	Key Words (m)	Rounds Constant	Rounds (T)
32	**64**	4	z_0	32
48	**72**	3	z_0	36
	96	4	z_1	36
64	**96**	3	z_2	42
	128	4	z_3	44
96	**96**	2	z_2	52
	144	3	z_3	54
128	**128**	2	z_2	68
	192	3	z_3	69
	256	4	z_4	72

Table 2.1: Parameters for SIMON

2.3.1 Round Function

As it can be seen in Figure 2.10, the plaintext will be divided into two parts each consists of n bits. The structure is basically a Feistel network. The round function G which maps the input state to output state can be written as follows.

$$G(L, R) = (F(L, k_i) \oplus R, L) \tag{2.2}$$

Function F consists of shifting the input to the left by 1,2 and 8 positions which are shown as $S^1(.)$, $S^2(.)$ and $S^8(.)$, respectively. It also has `AND` and also `XOR` with the round key. Function F can be represented as follows.

$$F(L, k_i) = \Big(S(L) \otimes S^8(L)\Big) \oplus S^2(L) \oplus k_i \tag{2.3}$$

All rounds of Simon are the same with only difference that round keys will be different in each stage. It is worth noticing that Simon uses basic logic elements and shifting can also be handled by wiring. The two mentioned properties made Simon to be highly efficient in hardware. Low area design for Simon can be achieved easily because first of all logic gates in the design are simple and second because round functions can be used for all the stages. In the following we will look at the key schedule of Simon.

2.3.2 Key Schedule

Based on Table 2.1 the proper setting for key schedule can be extracted. Once the number of key blocks is known, the key schedule will be done based on either one of the Figure 2.11a, 2.11b or 2.12 if the number of key blocks is 2,3 or 4, respectively. The key schedule consists of shifting, `XOR` with key and also with constant. The

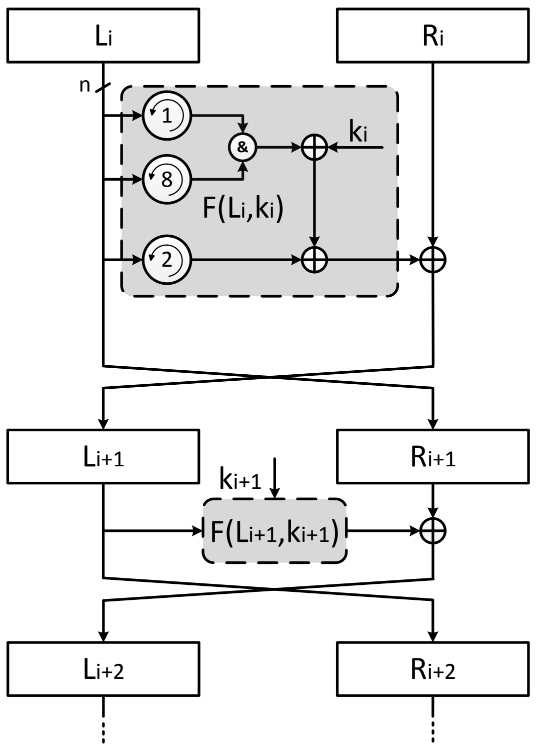
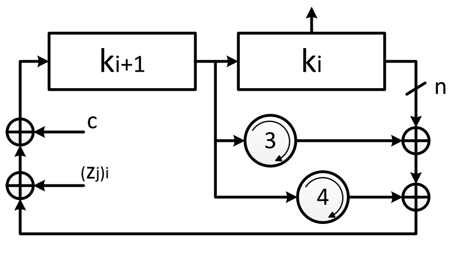
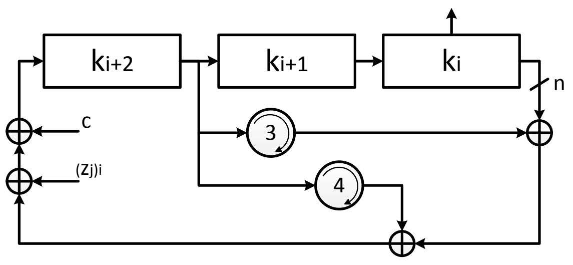


Figure 2.10: SIMON round function



(a) SIMON key schedule (2 blocks)



(b) SIMON key schedule (3 blocks)

Figure 2.11: SIMON key schedule (2 and 3 blocks)

circular shift is to the right, in contrast to round functions in which the rotations were to the left. In each round the result will be `XOR`ed with a both constant value. Constant c is equal to $2^n - 4 =$ `0xff...fc`, and it is the same for all rounds. The other constant value is z_j which is chosen based on Table 2.1.

Constant Sequence In order to obtain constant sequence z_js for $j = \{0, 1, 2, 3, 4\}$ we do the followings. The matrices U, V and W are defined as below

$$U = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}, V = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}, W = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Three sequences u, v and w are defined as below, where $(u)_i$ denotes the bit at position i^{th} of the sequence u. All of these sequences has period of 31, so it is enough to compute the first 31 bits. In the following the first 62 bits of those sequences are shown in hexadecimal notation.

$$(u)_i = (0,0,0,0,1)U^i(0,0,0,0,1)^T \quad , \quad u_0u_1u_2\ldots u_{61} = 3E8958737D12B0E6$$
$$(v)_i = (0,0,0,0,1)V^i(0,0,0,0,1)^T \quad , \quad v_0v_1v_2\ldots v_{61} = 23BE4C2D477C985A$$
$$(w)_i = (0,0,0,0,1)W^i(0,0,0,0,1)^T \quad , \quad w_0w_1w_2\ldots w_{61} = 212CF8DD4259F1BA$$

Let t denotes the sequence of 0s and 1s with period of 2, i.e. $t = t_0t_1t_2\ldots = 010101\ldots$. The first 62 bits of each constant sequence z_j is as follows, z_0 and

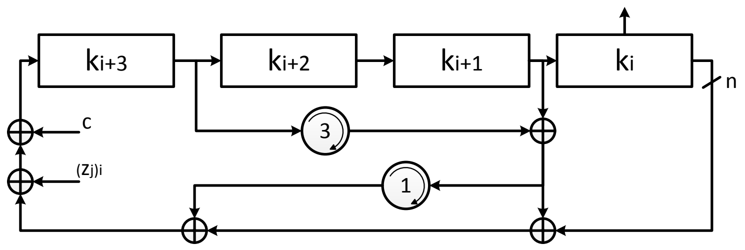


Figure 2.12: Simon key schedule (4 blocks)

z_1 have the period of 31 the rest has the period of 62.

$$(z_0)_i = (u)_i \quad , \qquad z_0 = 3E8958737D12B0E6\ldots$$

$$(z_1)_i = (v)_i \quad , \qquad z_1 = 212CF8DD4259F1BA\ldots$$

$$(z_2)_i = (t)_i \oplus (u)_i \quad , \quad z_2 = 2BDC0D262847E5B3\ldots$$

$$(z_3)_i = (t)_i \oplus (v)_i \quad , \quad z_3 = 36EB19781229CD0F\ldots$$

$$(z_4)_i = (t)_i \oplus (w)_i \quad , \quad z_4 = 3479AD88170CA4EF\ldots$$

Then the equations for computing the round keys are as follows for each case

$$k_{i+m} = \begin{cases} c \oplus (z_j)_i \oplus k_i \oplus S^{-3}(k_{i+1}) \oplus S^{-4}(k_{i+1}) & \text{if } m = 2 \\ c \oplus (z_j)_i \oplus k_i \oplus S^{-3}(k_{i+2}) \oplus S^{-4}(k_{i+2}) & \text{if } m = 3 \\ c \oplus (z_j)_i \oplus k_i \oplus S^{-3}(k_{i+3}) \oplus S^{-4}(k_{i+3}) \oplus S^{-1}(k_{i+1}) \oplus k_{i+1} & \text{if } m = 4 \end{cases}$$

The above algorithm will be done for $0 \leq i < T - m$. The result of the key schedule will be used in the round function. The first m levels in round function is done by using the input key and the rest of the Simon will use the keys computed by key schedule unit.

Chapter 3

Glitch-Free Implementations

The masking scheme which was introduced in the previous chapter can not resist DPA attacks, because the glitches have not been taken into account. In this chapter we introduce threshold implementation which is type of masking provably secure against first order DPA attacks, even in the presence of glitches. Threshold Implementation countermeasure was proposed by Nikova et al. in [NRR06].

3.1 Threshold Implementation

Threshold implementation is an (n, d) secret sharing in which d is equal to n, or all the shares are required to construct the secret value. The secret value is denoted by X and its shares are represented by $X_1, \ldots, X_n$ which is represented by $\hat{X}$. The set of $n-1$ shares which is missing X_i is denoted by $\hat{X}_i$. Share generation function is a simple `XOR`, and it can be realized as follows:

Definition 1 *(share generation) For dividing secret X into n shares, $n-1$ random*

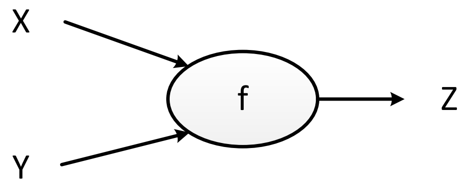


Figure 3.1: Simple function

value M_i will be generated and the shares are

$$\begin{cases} X_i = M_i & \textit{for } 1 \leq i \leq n-1 \\ X_n = \left(\sum_{i=1}^{n-1} M_i\right) \oplus X & \end{cases}$$

In this section we only focus on the functions with only one output. Furthermore, assume function f consists of two input value X and Y and produces Z as an output, e.g., $Z = f(X, Y)$ and it can be seen in Figure 3.1.

The shared version of output Z is denoted as $\hat{Z} = (Z_1, \ldots, Z_n)$. As it can be seen in Figure 3.2 each output share Z_i is produced by new function called f_i. The input to each function f_i comes from some input shares X_i and Y_i. The selection of the input shares is discussed in the rest of this section especially in 3.1.2 and 3.1.3.

The next part of this section is dedicated to necessary properties for constructing a threshold implementation of a function. These properties are called *Correctness*, *Non-completeness* and *Uniformity*.

3.1.1 Correctness

We have seen that output value is divided into n shares. Correctness means that by combining those output shares the original output can be retrieved in a correct way. In other word as it can be seen in Figure 3.2

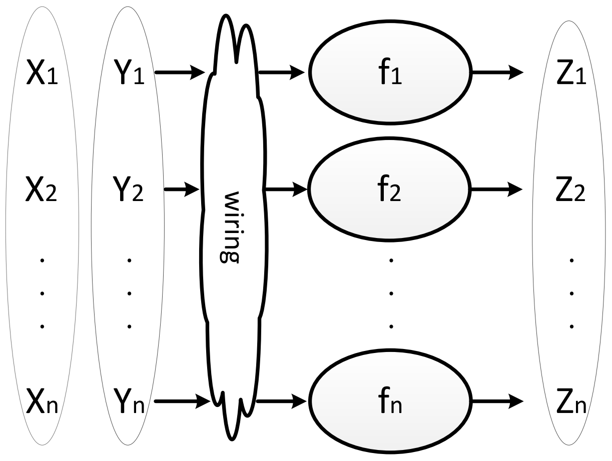


Figure 3.2: Simple function broken into shares

$$X = \bigoplus_{i=1}^{n} X_i = X_1 \oplus X_2 \oplus \dots X_n$$
$$Y = \bigoplus_{i=1}^{n} Y_i = Y_1 \oplus Y_2 \oplus \dots Y_n$$
$$Z = \bigoplus_{i=1}^{n} Z_i = Z_1 \oplus Z_2 \oplus \dots Z_n$$

3.1.2 Non-completeness

Non-completeness means that the equation used to evaluate any output share should be missing at least one input share. This requirement enforces that the information required to compute the secret value (all the shares) is not present in the system at any time instance. Hence, any vulnerability in the implementation (e.g. glitches) cannot leak the secret key. This property can be simply achieved if function f is linear. We assume that the wiring in the Figure 3.2 is in a way that $Z_i = f_i(X_i, Y_i)$, it can be shown that this wiring has non-completeness (each function depends on only one input share) and correctness.

$$Z = \bigoplus_{i=1}^{n} Z_i = \bigoplus_{i=1}^{n} f_i(X_i, Y_i) = f(\bigoplus_{i=1}^{n} X_i, \bigoplus_{i=1}^{n} Y_i) = f(X, Y)$$

In case function f is non-linear the wiring should be done in a different way. Lets assume that function f is a simple multiplication and each input and output shares are divided into three shares. One possible way of doing the wiring is to rewrite the equations in a way that function f_i depends on $\hat{X}_i$ and $\hat{Y}_i$

$$\begin{aligned} f(X,Y) = XY &= (X_1 \oplus X_2 \oplus X_3)(Y_1 \oplus Y_2 \oplus Y_3) \\ &= X_1Y_1 \oplus X_1Y_2 \oplus X_1Y_3 \\ &\quad + X_2Y_1 \oplus X_2Y_2 \oplus X_2Y_3 \\ &\quad + X_3Y_1 \oplus X_3Y_2 \oplus X_3Y_3 \end{aligned}$$

Then the output shares are written as

$$\begin{aligned} Z_1 &= X_2Y_3 \oplus X_3Y_2 \oplus X_2Y_2 \\ Z_2 &= X_3Y_1 \oplus X_1Y_3 \oplus X_3Y_3 \\ Z_3 &= X_1Y_2 \oplus X_2Y_1 \oplus X_1Y_1 \end{aligned}$$

It is obvious to see that each Z_i is missing X_i and Y_i component and also combining all Z_i will results in the correct output, i.e., XY.

Obviously, because of non-completeness property the attacker who has access to the output of one of the f_is can not infer anything about the input shares.

Nikova et al. [NRR06] proved that if the Equation 3.1 holds and two mentioned properties are satisfied, all the intermediate results of the circuit will be independent of inputs (X,Y) and output (Z). This equation ensures that for any input (x, y) all the valid sharing $(\hat{X}, \hat{Y})$ will happen with equal probability.

$$Pr(\hat{X} = \hat{x}, \hat{Y} = \hat{y}) = \alpha Pr(X = x, Y = y) \qquad \alpha \text{ is a constant value} \tag{3.1}$$

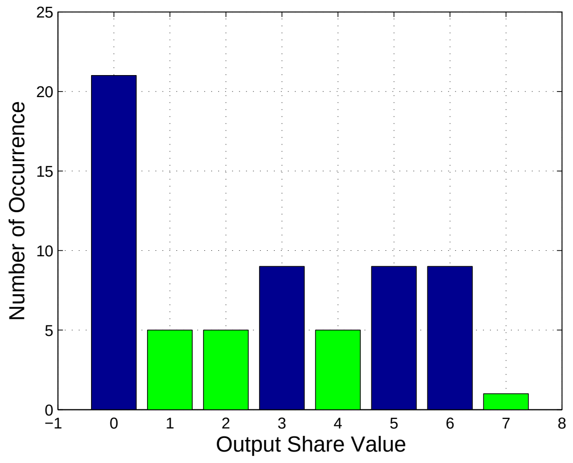


Figure 3.3: Distribution of output shares when multiplication function is implemented using three shares

These two properties are basic requirements for having threshold implementation. In practice the functions we are interested in are complex functions with so several levels. As the depth of the function grows there is a need for more shares. It is going to be impractical to implement those functions just by having a combinational logic. It was mentioned before that glitches occur due to difference in arrival time of each input of a logic gate and also random delay inside a logic gate. If we assume that an element can isolate its input timing and output timing we can break complex designs into pieces. This element which can isolate its input and output timing, as a result block the propagation of glitches, is called register. Our design can be simply turned into pipeline. In each stage of pipeline a simple functionality will be performed. In order for this design to be threshold implementation the input of each pipeline stage must satisfy the Equation 3.1. The input of each pipeline stage is the output of the previous stage. The next property is defined so that the output shares satisfy Equation 3.1.

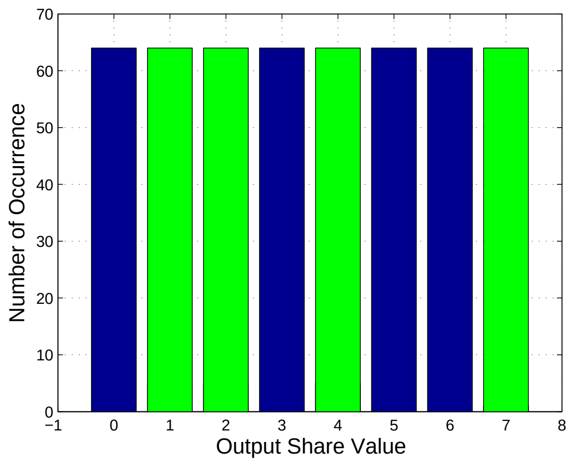


Figure 3.4: Distribution of output shares when multiplication function is implemented using three shares with extra randomness added to the equations

3.1.3 Uniformity

If the input shares are uniformly distributed, the output shares must also be uniformly distributed.

$$Pr(\hat{Z} = \hat{z} | Z = z) = \beta \qquad \beta \text{ is a constant value} \tag{3.2}$$

It can be shown that the multiplication introduced in Section 3.1.2 with uniformly distributed inputs does not satisfy the uniformity properties for the output. We did the analysis and as it can be seen in the Figure 3.3 the distribution of the output shares is not uniform. The x-axis in the figure denotes the number $Z_1Z_2Z_3$ in decimal notation.

In order to achieve the uniform distribution two approaches can be pursued. The first one is to add a new randomness to the set of previous shares in order to make them look random [MPL+11, BGN+14a]. The second approach is to increase the number of share and try to find a solution that satisfy all the mentioned properties [NRR06].

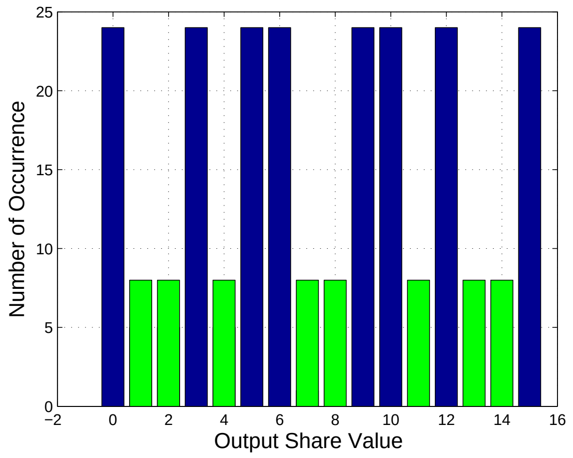


Figure 3.5: Distribution of output shares when multiplication function is implemented using four shares

Adding New Randomness In order to add new randomness we can add three new random values to each equation, namely R_1, R_2 and R_3. The R_is are independently and uniformly distributed random variable. In order to satisfy the correctness property the final result of $Z_1 \oplus Z_2 \oplus Z_3$ should kept unchanged. As a result adding these random numbers should not have effect on the result or in other word $R_1 \oplus R_2 \oplus R_3 = 0$. The set of new equations are shown below.

$$Z_1 = X_2Y_3 \oplus X_3Y_2 \oplus X_2Y_2 \oplus R_1$$
$$Z_2 = X_3Y_1 \oplus X_1Y_3 \oplus X_3Y_3 \oplus R_2$$
$$Z_3 = X_1Y_2 \oplus X_2Y_1 \oplus X_1Y_1 \oplus R_3$$

and the distribution of output shares can be seen in Figure 3.4.

Increase The Number of Shares Nikova et al. in [NRR06] chose the second approach and proposed a new set of equations for constructing the output shares.

$$
\begin{aligned}
Z_1 &= (X_3 \oplus X_4)(Y_2 \oplus Y_3) \oplus Y_2 \oplus Y_3 \oplus Y_4 \oplus X_2 \oplus X_3 \oplus X_4 \\
Z_2 &= (X_1 \oplus X_3)(Y_1 \oplus Y_4) \oplus Y_1 \oplus Y_3 \oplus Y_4 \oplus X_1 \oplus X_3 \oplus X_4 \\
Z_3 &= (X_2 \oplus X_4)(Y_1 \oplus Y_4) \oplus Y_2 \oplus X_2 \\
Z_4 &= (X_1 \oplus X_2)(Y_2 \oplus Y_3) \oplus Y_1 \oplus X_1
\end{aligned}
$$

The analysis for this set of equations can be seen in Figure 3.5. Comparing Figure 3.3 and Figure 3.5 proves that using the new of equations will result in the *Uniform* property to hold true and the construction based on this scheme is threshold implementation and glitch free.

Chapter 4

Design Methodology

Before starting the design, the designer should clearly specify the goals of the design. Being the fastest design or being the smallest design are examples of those goals. In this chapter we introduce the smallest implementation of Simon which only processes one bit at each cycle and apply the threshold implementation idea to that.

4.1 Bit-Serial Architecture of SIMON

As we have already seen in Section 2.3, Simon is a block cipher based on the Feistel structure. Simon accepts plaintexts of size 32, 48, 64, 96 and 128 bits. For each input size, Simon has a set of allowable key sizes ranging from 64 bits to 256 bits. The input is evenly split into two words, following the principles of Feistel structure. The key is also split into two to four words. The input key words which are used in the first rounds of Simon. The key scheduling algorithm is used to generate the following round keys. The number of rounds in Simon ranges from 32 rounds to 72 rounds. We focus on the Simon128/128 which which has 68 rounds and the input key will be divided into 2 blocks each one contains 64 bits.

The Equations 2.2 and 2.3 shows the round function. Assuming that the input words of round i are l_i and r_i, the output words are:

$$l_{i+1} = r_i \oplus l_i^2 \oplus (l_i^1 \wedge l_i^8) \oplus k_i \qquad r_{i+1} = l_i$$

The upper index X^s indicates left circular shift by s bits. This can be expressed in GF(2), where the `XOR` operation becomes addition and the `AND` operation becomes multiplication, as:

$$l_{i+1} = r_i + l_i^2 + (l_i^1 \times l_i^8) + k_i \qquad r_{i+1} = l_i \tag{4.1}$$

Also, assuming that the input words of the key, which are also the first round keys, are k_0 and k_1 (and possibly k_2 and k_3, depending on the key size), the next round key is computed as:

$$\begin{aligned} k_{i+2} &= k_i + k_{i+1}^{-3} + k_{i+1}^{-4} + \alpha_i && \text{Two and Three Words} \\ k_{i+4} &= k_i + k_{i+1} + k_{i+1}^{-1} + k_{i+3}^{-3} + k_{i+3}^{-4} + \alpha_i && \text{Four Words} \end{aligned} \tag{4.2}$$

where α_i is a the bitwise `XOR` of constant c and constant sequence $(z_j)_i$ as it was introduced in Section 2.3.2.

Aysu et al. in [AGS14] proposed a bit-serialized implementation of Simon where only one bit of the internal state is processed in each clock cycle. Hence, a single round of Simon completes after n cycles, where $2n$ is the size of input plaintext.

Moreover, two shift registers were used to store the internal states to simplify the control of sequentially processing and storing individual bits. In fact, the left share of the internal state is passed over as-is to the right share, hence only one shift register of the same size as the input block is actually needed.

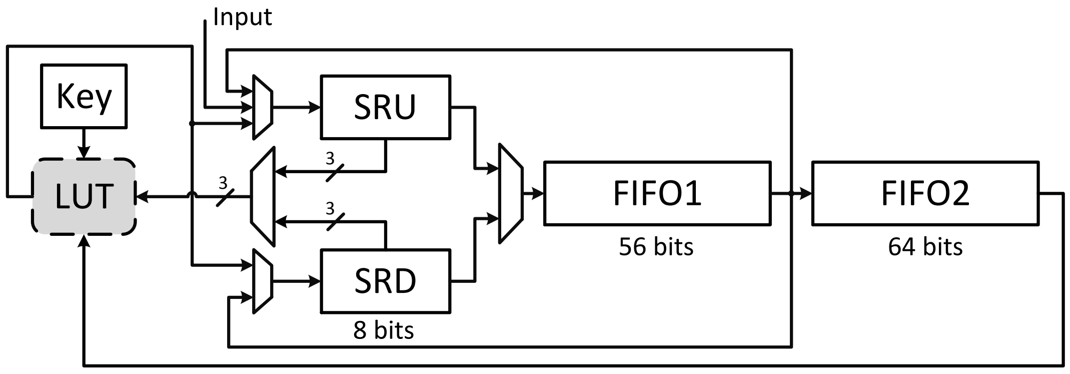


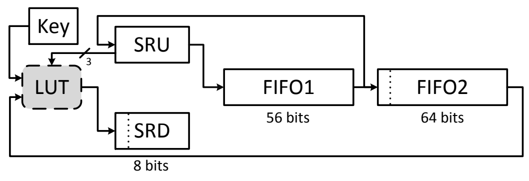
Figure 4.1: Data-path of the Simon cipher

Here, Simon is implemented as a special class of non-linear feedback shift registers, where the output of the feedback function changes the state only after completing the round function. Since the feedback function requires only four bits of the state, namely r_i, l_i^1, l_i^2 and l_i^8, only those bits need to be stored. This storage is realized by an extra 8-bit shift register. An overview of this implementation is shown in Figure 4.1.

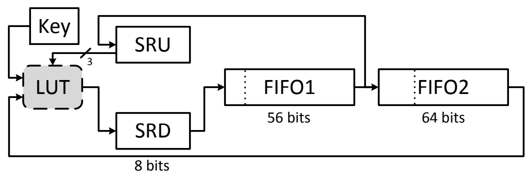
One of the main reason for using such a scheme for round function is the efficiency in area usage. As it can be seen in Figure 4.1, the basic elements used in the structure are shift registers which can simply goes into one slice of FPGA. The computation unit, i.e., LUT can also be mapped to one LUT of each slices. Shift Register Up (SRU) and Shift Register Down (SRD) can also be mapped into several slices. Although they are also simple shift registers, the fact that some internal registers needed for LUT make those logic spread to several slices.

4.1.1 Round Function

At first, the input is loaded into the Shift Register Up (SRU), FIFO1 and FIFO2. As it can be seen in Figure 4.2a, during the first 8 cycles, the look-up table (LUT)



(a) First 8 clock cycles



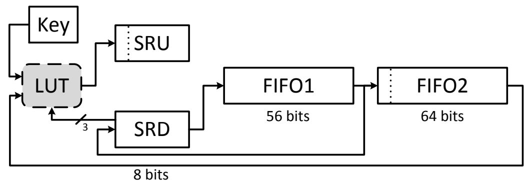
(b) Next 56 clock cycles

Figure 4.2: Data flow in even rounds of SIMON where SRD is used for saving newly computed data

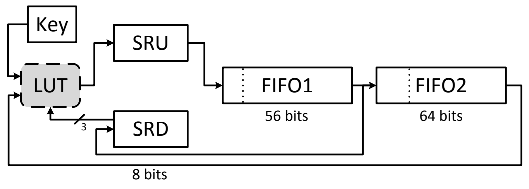
processes three bits from the SRU, a key bit and the output of FIFO2. It basically computes the result of Equation 4.1. The result is stored in the Shift Register Down (SRD). During this phase, SRD stores the new values, while SRU stores the old ones for further processing.

Once the SRD is full and before overflowing occurs, instead of SRU, SRD will be connected to FIFO1, where the new values will be stored. This change can be seen in Figure 4.2b. SRU will still work as the old register for storing old bit values from FIFO1 output. This phase continues for 56 cycles until the round is completed. As it can be seen, at the end of this round the state to be processed in the next round is stored in SRD, FIFO1 and FIFO2. Since the input for the LUT unit should come from SRD instead of SRU, there is a need for change in the data flow.

As it was mentioned, in the next round, the functionality of SRU and SRD will be flipped. It can be seen in the Figure 4.3a that in the first 8 clock cycles, SRU will be used to store new values while SRD holds the necessary values needed for further computations. Once SRU is filled with new values its output will be connected to FIFO1. As it is represented in Figure 4.3b at this point the new values will be written into FIFO1 and SRD will keep holding the values for computation.



(a) First 8 clock cycles



(b) Next 56 clock cycles

Figure 4.3: Data flow in odd rounds of SIMON where SRU is used for saving newly computed data

4.1.2 Key Schedule

The structure of the key schedule is shown in Figure 4.4. The key will be loaded into Shifter1, FIFO and Shifter2. The output of Shifter2 is the key that will be used in each round of SIMON. The key that will be used in the first two rounds are the key loaded into in the first step and the key for the next rounds can be computed according to Equation 4.2.

During the first 4 clock cycles the output of LUT will be loaded into Shift Register (SR) and the data in FIFO will be moved to both Shifter2 and Shifter1. Once the SR is full it will save the data inside and the rest of the computed result of LUT will be moved to Shifter1 and FIFO will continue to move its data to Shifter2. This will keep going for the next 60 clock cycles and the first round of key schedule will be done.

From now on the input to LUT will be either from SR for the first clock cycle of each round and from FIFO for the next 63 clock cycles. During the first 4 clock cycles the output of SR will moved to both Shifter2 and Shifter1 and the result of LUT will moved to SR. Once SR is full it will stop shifting data and FIFO will fill Shifter2 and the result of LUT will move to Shifter1.

In each step constant is calculated by Equation 4.2 and the materials covered in Section 2.3.2.

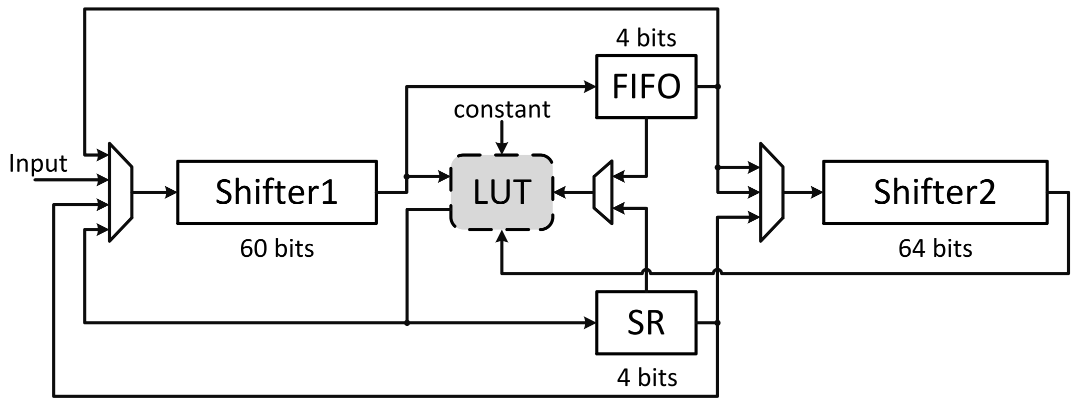


Figure 4.4: Key schedule of the Simon cipher

4.2 Loop Unrolling

The idea of loop unrolling first published in a work by Bhasin et al. [BGSD10]. They proposed a method to compute the result of DES algorithm in only one clock cycle. They showed that their implementation resist the correlation power analysis on Hamming distance and Hamming weight model if the datapath get cleared after each DES evaluation. Beaulieu et al. also proposed in [Smi15] to use the same method for protecting Simon against side-channel attacks. They implemented Simon in a way that computes four full rounds per clock cycle. Moradi et al. showed a correlation collision attacks on four unrolled encryption rounds of AES in [MMP11]. Since this method is not proven to be secure, in Section 5.1.1 we just present a practical attack on the four unrolled encryption rounds of Simon32/64.

4.3 Threshold Implementation of SIMON

Threshold Implementation of block ciphers have been published for AES [MPL$^+$11, BGN$^+$14a] and Present [KNPW13].

In this work, we propose the required equations to process Simon as a threshold implementation. Although a three shares implementation is required to overcome glitches in hardware modules, we start with a two shares implementation as a preliminary step.

4.3.1 Simon with Two Shares

In order to process SIMON in two shares, we use the following equations. We denote the random mask that affects the input plaintext as m_1 and m_2. The input words are given as:

$$\begin{aligned} (r_1)_0 &= m_1 \qquad (r_2)_0 = m_1 + r_0 \\ (l_1)_0 &= m_2 \qquad (l_2)_0 = m_2 + l_0 \end{aligned} \tag{4.3}$$

Then, the round functions can be expressed as:

$$\begin{aligned} (r_1)_{i+1} &= (l_1)_i \qquad (r_2)_{i+1} = (l_2)_i \\ (l_1)_{i+1} &= (r_1)_i + (l_1)_i^2 + (l_1)_i^1 \times (l_1)_i^8 + (l_1)_i^1 \times (l_2)_i^8 + (k_1)_i \\ (l_2)_{i+1} &= (r_2)_i + (l_2)_i^2 + (l_2)_i^1 \times (l_2)_i^8 + (l_2)_i^1 \times (l_1)_i^8 + (k_2)_i \end{aligned} \tag{4.4}$$

where k_1 and k_2 are the two shares of the round key. We use a different mask to process the key schedule. The size of the mask should be equal to the size of the key. Equations for splitting the key schedule into two shares are straightforward, being an entirely linear operation. It is just enough to split the key at the first step into to shares and run the key schedule on each of them separately.

This masking scheme is correct and uniform. However, it is not *non-complete* because the two input shares are required to process any output share. This masking scheme can work in software implementations if we enforce the order of processing the equation to be from left to right. Hence, we ensure that the compiler does not generate any intermediate variable that is free from the random mask. However, this masking scheme is not provably secure in hardware implementations where glitches can leak the relation between the two shares. In order for the secret-sharing scheme to provably work in hardware implementations, we need to enforce the requirement of *non-completeness*. Hence, we propose the three-sharing scheme in the next subsection.

4.3.2 Simon with Three Shares

The equations used to process SIMON in three shares follow the same reasoning of the two shares. Here, we use two random variables, each with the same size as the input plaintext. This generates three shares of each word, denoted by x_1, x_2 and x_3. The equations to process the r and l part are as follows:

$$(r_1)_{i+1} = (l_1)_i \qquad (r_2)_{i+1} = (l_2)_i \qquad (r_3)_{i+1} = (l_3)_i \tag{4.5}$$

$$\begin{aligned}
(l_1)_{i+1} &= (r_2 + l_2^2 + l_2^1 \times l_2^8 + l_2^1 \times l_3^8 + l_3^1 \times l_2^8 + k_2)_i \\
(l_2)_{i+1} &= (r_3 + l_3^2 + l_3^1 \times l_3^8 + l_3^1 \times l_1^8 + l_1^1 \times l_3^8 + k_3)_i \\
(l_3)_{i+1} &= (r_1 + l_1^2 + l_1^1 \times l_1^8 + l_1^1 \times l_2^8 + l_2^1 \times l_1^8 + k_1)_i
\end{aligned} \tag{4.6}$$

This masking scheme is correct, uniform and non-complete. It is non-complete because the equation used to process any output share (e.g. 1) does not include at least one input share (1). Although the system of equations in the data-path (every term in the equations aside from the key) is not invertible, i.e., its mapping is not guaranteed to be one-to-one, which suggests non-uniformity, uniformity is guaranteed by the randomness brought by the key shares (k_1, k_2 and k_3). The key shares are uniformly distributed as the system of equations to generate them is linear and invertible (assuming that the input random masks are uniform). Then, it is easy to prove that the result of addition in GF(2) between an arbitrary variable that is not necessarily uniform (the data-path) and a uniformly distributed random variable (the key shares), is uniformly distributed. This implies that the above system of equations is uniform. The distribution of the output shares of the above equations are demonstrated in Figure 3.4. Although the random variable used in one round depends on the random variables used in the previous rounds, this does not result in any vulnerability for univariate attacks that harvest information from a single point in the trace.

The number of randomness used in our design comes from the randomness in datapath and the randomness in key schedule. As we divided the plaintext into three shares we need two random mask each one being 128 bits. The same idea holds for key schedule where we need 256 bits random data in total. As a result, for threshold implementation of Simon there is a need for 512 bits randomness which is smaller than the previous works by Moradi et al. [MPL$^+$11] and Bilgin et al. [BGN$^+$14a] which uses 7680 and 7040 bits, respectively.

In order to design a threshold implementation for Simon there are two choices, parallel and serial. In both cases the state will be divided into three shares.

4.3.3 Parallel Simon

The parallel implementation uses three copies of the data-path and key schedule units, i.e. one for each share. Note that the three datapath units and key schedule units need only one instance of the control unit. Throughout this section we use $f(s,k)$ to denote the modular addition between key bit k and state bit s, i.e., $f(s,k) = s + k$. The state bit and key bit are as follows:

$$\begin{aligned} s &= r_\alpha + l^2_\alpha + l^1_\alpha \times l^8_\alpha + l^1_\alpha \times l^8_\beta + l^1_\beta \times l^8_\alpha \\ k &= k_\alpha \end{aligned} \tag{4.7}$$

where α and β denote different input shares.

The r part of each share can be easily obtained by shifting the l part of that share. For computing the l part the Equation 4.7 should be satisfied. As can be seen in Figure 4.5, the input to the function block comes from two shares (denoted by old) based on the above equation along with one bit from the key. The output is written into one share (denoted by new). The function block is implemented using LUTs. The old share is SRU (or SRD) and the new share is SRD (SRU), if the round is even (odd). The parameters α and β can be extracted from Equations 4.6. At each clock cycle the key schedule unit and data-path unit are enabled to ensure that new values are written for all three shares at each clock cycle.

In order to ensure that each output share is independent of at least one input share the "Keep Hierarchy" property of synthesize tool should be enabled. The keep hierarchy property ensures that parallel LUTs are synthesized so that they never share in one slice. The resistance analysis presented in the next section shows that this level of separation is sufficient for security.

Although no component of this core receives all three shares as an input, hence preventing glitches from leaking first-order information, the core as a whole still processes all three shares in the same clock cycle. Under rare circumstances, this might result in remaining first order leakage. For this reason, we propose the serialized version of the protected core where each share is strictly accessed in different clock cycles.

4.3.4 Serial Simon

The serial Simon processes only one share at each clock cycle as opposed to parallel implementation. More specifically, in each clock cycle, only one bit is computed and only one register is being shifted. So, updating the three shares takes three

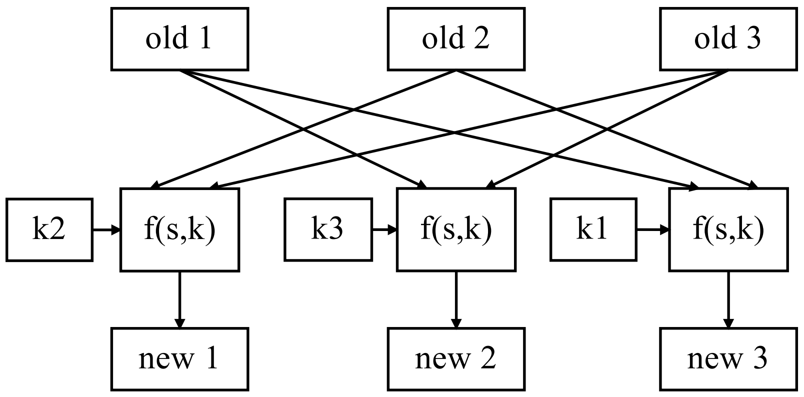


Figure 4.5: Architecture of parallel SIMON where all three shares are being processed at the same clock cycle

clock cycles. To ensure the correctness of the design, Read After Write (RAW) hazard should be prevented. This requires one extra register, added to one of the shares to save the previous value of that share. In order to reduce the overhead caused by the mentioned register, we modify the non-completeness of the equations in Section 4.3.2, such that shares 1, 2 and 3 are independent of shares 3, 1 and 2, respectively. The equation for this core is as follows:

$$(r_1)_{i+1} = (l_1)_i \qquad (r_2)_{i+1} = (l_2)_i \qquad (r_3)_{i+1} = (l_3)_i \tag{4.8}$$

$$\begin{aligned}
(l_1)_{i+1} &= (r_1 + l_1^2 + l_1^1 \times l_1^8 + l_1^1 \times l_2^8 + l_2^1 \times l_1^8 + k_1)_i \\
(l_2)_{i+1} &= (r_2 + l_2^2 + l_2^1 \times l_2^8 + l_2^1 \times l_3^8 + l_3^1 \times l_2^8 + k_2)_i \\
(l_3)_{i+1} &= (r_3 + l_3^2 + l_3^1 \times l_3^8 + l_3^1 \times l_1^8 + l_1^1 \times l_3^8 + k_3)_i
\end{aligned} \tag{4.9}$$

Based on the new set of equations, only share 1 will face the RAW hazard, so the extra register is added for share 1. Figure 4.6 illustrates the new architecture. Since the design is based on shift registers, adding an extra register is achieved by taking one register out of FIFO1 and adding it to SRU and SRD. The design ensures that at each cycle only one key bit along with proper states will go through the MUX. The computed result will then be routed in the DEMUX unit and written into the

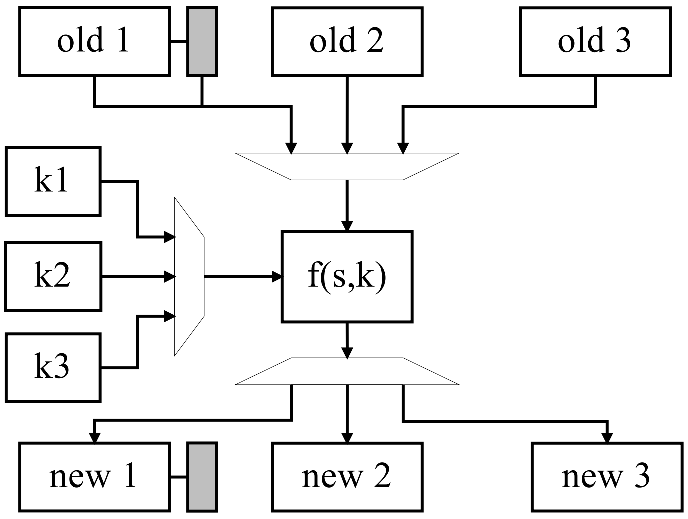


Figure 4.6: Architecture of Serial SIMON where only one share is being processed at each clock cycle

proper share.

4.4 Higher-Order Threshold Implementation of Simon

The implementations which were just introduced will resist against first order attacks but they are not resistant against higher order attacks. The set of equations for satisfying a design which withstand higher order attacks are more complex. The *non-completeness* property should be modified. Bilgin et al. [BGN+14b] proposed the following property as *non-completeness*. We should remember that in threshold implementation theory the original function f will be divided into n portions f_i and each one of them get some shares from the input.

Property d^{th}*-order non-completeness.* Any combination of up to d output of f_i must be independent of at least one input share.

Bilgin et al. showed that the d^{th} statistical moment of the power consumption of a device which satisfies the above property is independent of the unmasked input even in the occurrence of glitches.

For example, assume the function is $f(a,b,c) = a + b \times c$. The sharing of a, b and c will be denoted as a_i, b_i and c_i, respectively. One possible set of equations to satisfy the above property is as follows:

$$\begin{aligned}
y_1 &= a_2 + b_2c_2 + b_1c_2 + b_2c_1 & y_2 &= a_3 + b_3c_3 + b_1c_3 + b_3c_1 \\
y_3 &= a_4 + b_4c_4 + b_1c_4 + b_4c_1 & y_4 &= a_1 + b_1c_1 + b_1c_5 + b_5c_1 \\
y_5 &= a_5 + b_5c_5 + b_2c_5 + b_5c_2 & & \\
y_6 &= b_2c_4 + b_4c_2 & y_7 &= b_2c_3 + b_3c_2 \\
y_8 &= b_3c_4 + b_3c_5 + b_4c_5 & y_9 &= b_4c_3 + b_5c_3 + b_5c_4
\end{aligned} \tag{4.10}$$

In this equation the number of input shares is 5, while the number of output shares is 9. By keeping the sharing of y_i, the design will get bigger as more non-linear function is to be computed. Hence, there is a need for decreasing the number of shares. It was shown in [BGN$^+$14b] that the following construction which combines y_is is still secure against d^{th}-order DPA attack.

$$\begin{aligned}
z_1 &= y_1 + y_6 & z_2 &= y_2 + y_7 \\
z_3 &= y_3 + y_8 & z_4 &= y_4 + y_9 \\
z_5 &= y_5 & &
\end{aligned} \tag{4.11}$$

The logic where computes y_i should be separated from the unit which computes the z_i by registers.

Consider the case of SIMON. The equation for computing the left part is shown in Equation 4.1. The l part and r part of the equation are coming from the round functions and they should have the same number of shares. The key, on the other hand, can be divided into different number of shares. As long as the *correctness* property and d^{th}-*order non-completeness* holds it can be added to the same set of equations. The equations for processing the round function is as follows:

$$\begin{aligned}
y_1 &= (r_2 + l_2^1 \times l_2^8 + l_1^1 \times l_2^8 + l_2^1 \times l_1^8)_i \\
y_2 &= (r_3 + l_3^1 \times l_3^8 + l_1^1 \times l_3^8 + l_3^1 \times l_1^8)_i \\
y_3 &= (r_4 + l_4^1 \times l_4^8 + l_1^1 \times l_4^8 + l_4^1 \times l_1^8)_i \\
y_4 &= (r_1 + l_1^1 \times l_1^8 + l_1^1 \times l_5^8 + l_5^1 \times l_1^8)_i \\
y_5 &= (r_5 + l_5^1 \times l_5^8 + l_2^1 \times l_5^8 + l_5^1 \times l_2^8)_i \\
y_6 &= (l_2^1 \times l_4^8 + l_4^1 \times l_2^8 + k_1)_i \\
y_7 &= (l_2^1 \times l_3^8 + l_3^1 \times l_2^8 + k_2)_i \\
y_8 &= (l_3^1 \times l_4^8 + l_3^1 \times l_5^8 + l_4^1 \times l_5^8)_i \\
y_9 &= (l_4^1 \times l_3^8 + l_5^1 \times l_3^8 + l_5^1 \times l_4^8 + k_3)_i
\end{aligned} \tag{4.12}$$

After computing the y_is, the result will be stored in a register. In the next clock cycle, the stored values will be read from the registers and mixed together to reduce the sharing to 5 shares. The last part of the Equation 4.1, which is $(l^2)_i$ will be added too. Since there is one clock cycle difference the actual value for $(l^2)_i$ is shifted and now is present at $(l^3)_i$. The architecture of the design will slightly change to be able to read that value as well.

$$\begin{aligned}
z_1 &= y_1 + y_6 + (l_1^3)_i \qquad & z_2 &= y_2 + y_7 + (l_2^3)_i \\
z_3 &= y_3 + y_8 + (l_3^3)_i \qquad & z_4 &= y_4 + y_9 + (l_4^3)_i \\
z_5 &= y_5 + (l_5^3)_i
\end{aligned} \tag{4.13}$$

The amount of randomness in this core comes from the randomness needed in datapath as well as key schedule. Since we divided the plaintext into five shares we need four random mask each one being 128 bits. The key schedule, on the other hand, will be divided into three shares so 256 bits of randomness will be used in key schedule unit. In total, for higher order threshold implementation of SIMON there is a need for 768 bits randomness which is still smaller than the works by Moradi et al. [MPL^{+}11] and Bilgin et al. [BGN^{+}14a] which uses 7680 and 7040 bits, respectively.

4.5 Implementation Results

The proposed designs were implemented in Verilog HDL and synthesized using ISE 14.7. Table 4.1 represents the implementation result of the unprotected, threshold implementation and higher order threshold implementation when it is synthesized for Spartan-3 xc3s50. The first row for each mode of SIMON represent the unprotected core. The second row shows the result for the threshold implementation of SIMON and the third row shows the result for the higher order threshold implementation of SIMON.

Table 4.2 represents the implementation result of the unprotected, threshold implementation and higher order threshold implementation when it is synthesized for Kintex-7 xc7k70t. The first row for each mode of SIMON represent the unprotected core. The second row shows the result for the threshold implementation of SIMON and the third row shows the result for the higher order threshold implementation of SIMON.

Table 4.3 summarizes the results and provides a comparison to previous implementations on the same platform. Our proposed parallel implementation needs 96 slices when synthesized by setting the optimization goal to area. The occupied slices are less than three times of the unprotected design, since the control logic is not replicated for the parallel design. We also synthesized the parallel design by choosing speed as the main optimization goal, letting synthesize tool pick slices. The serial design is slightly larger than the parallel one, because of the overhead in control logic and some minor changes in the data-path, as discussed before. As highlighted in Table 4.3, our implementation is more compact than some unprotected ciphers, namely AES and PRESENT. In fact, the small AES implementation from [GB05] is also outperformed in all compared metrics, though that implementation is not protected against SCA. We implemented the higher-order SIMON only in parallel version. As it can be seen, the design is larger than the first-order resistant of SIMON. It can also be seen that because of the complex equations for higher order version the number of LUTs utilized in the design is significantly higher than the other two designs.

We synthesized the design for ASIC using Synopsys Design Compiler using the TSMC 90 nm cell library. The results are shown in Table 4.4. The results of the synthesize tool are divided by 5 (our estimation for the number of gates in each cell) to give the Gate Equivalents (GE) number. As it can be seen for the case of SIMON128/128 the threshold implementation core is roughly three times bigger than the unprotected version of the same core. The higher order implementation core is roughly four times bigger than the unprotected core.

We also compared the performance result with some known ciphers, namely, AES and PRESENT. The results are shown in Table 4.5 and it can be seen that even the higher order implementation of SIMON is smaller than the threshold implementation for AES. The other small cipher is `Katan` which accepts plaintext of size 32, 48 and 64 bits and the key size for all of them is 80 bits.

Design	**Area**			**Max. Freq.**	**Throughput**
	Slices	FFs	LUTs	(MHz)	(Mbps)
Simon32/64	29	29	53	125	6.25
	95	81	129	149	5.7
	150	109	216	143	4.9
Simon48/72	33	28	58	108	5.0
	92	74	135	146	5.5
	144	101	227	134	4.5
Simon48/96	39	32	67	99	4.5
	106	84	155	134	4.7
	161	111	247	121	3.9
Simon64/96	34	28	60	111	4.5
	89	74	136	158	5.3
	146	102	233	127	3.9
Simon64/128	36	31	64	111	4.2
	102	83	150	138	4.3
	159	111	248	130	3.7
Simon96/96	43	30	74	101	3.4
	107	74	156	138	4.1
	167	102	251	140	3.9
Simon96/144	44	30	77	101	3.3
	110	76	164	138	3.8
	172	104	263	134	3.5
Simon128/128	43	30	78	90	2.4
	96	68	150	145	3.5
	163	102	265	127	2.9
Simon128/192	48	30	87	88	2.3
	110	76	176	149	3.4
	169	104	277	138	3.0
Simon128/256	50	33	91	91	2.2
	121	85	194	148	3.2
	182	113	298	122	2.5

Table 4.1: Implementation result for SIMON on Spartan-3. The first row in each version represents the unprotected SIMON the second and third row represent the threshold implementation and higher order threshold implementation, respectively.

Design	**Area**			**Max. Freq.**	**Throughput**
	Slices	FFs	LUTs	(MHz)	(Mbps)
Simon32/64	51	80	101	364	14.0
	103	108	167	366	12.6
Simon48/72	44	74	97	401	15.1
	95	102	166	391	13.3
Simon48/96	49	83	104	274	9.7
	113	111	170	305	9.9
Simon64/96	52	72	99	277	9.3
	101	100	165	350	10.8
Simon64/128	57	81	104	366	11.4
	105	109	170	377	10.8
Simon96/96	56	72	109	275	8.3
	94	100	179	366	10.2
Simon96/144	51	74	113	318	8.9
	93	102	187	336	9.6
Simon128/128	55	71	107	315	7.6
	95	99	176	310	7.1
Simon128/192	53	73	116	345	8.0
	91	101	187	358	7.8
Simon128/256	60	82	124	346	7.5
	100	110	195	359	7.3

Table 4.2: Implementation result for SIMON on Kintex-7. The first and second row in each version represent the threshold implementation and higher order threshold implementation, respectively

Design	**Area**			**Max. Freq.**	**Throughput**
	Slices	FFs	LUTs	(MHz)	(Mbps)
AES [GB05]	264	N/A	N/A	67	2.2
PRESENT [YK09]	117	114	159	113	28.4
Unpro-SIMON [AGS14]	36	N/A	N/A	136	3.6
TI-Simon					
Parallel (*area*)	96	68	150	145	3.5
Parallel (*speed*)	108	178	172	191	4.6
Serial (*area*)	131	94	194	84	0.7
Serial (*speed*)	137	95	208	110	1
HO TI-Simon					
Parallel (*area*)	163	102	265	127	2.9
Parallel (*speed*)	167	106	270	149	3.4

Table 4.3: Implementation result on FPGA in comparison to other ciphers with similar key size. The numbers reported for AES and PRESENT are for unprotected version of them

Design	**Unprotected**	**TI**	**HO-TI**
SIMON32/64	454	1354	1741
SIMON48/72	548	1590	2087
SIMON48/96	642	1860	2362
SIMON64/96	689	2014	2635
SIMON64/128	805	2365	2992
SIMON96/96	813	2366	3217
SIMON96/144	982	2875	3719
SIMON128/128	1039	3044	4122
SIMON128/192	1265	3723	4795
SIMON128/256	1493	4415	5485

Table 4.4: Implementation result for different implementations of SIMON for ASIC. The reported numbers from the synthesize tool are divided by 5 to give an estimation for Gate Equivalents (GE) parameter

Design	**GE**
Unprotected	
KATAN-32 [BGN+14b]	1002
SIMON48/96 [BSS+13]	763
AES [MPL+11]	2400
PRESENT [BKL+07]	1569
SIMON128/128 [BSS+13]	1317
Threshold Implementation	
KATAN-32 [BGN+14b]	1720
SIMON48/96 [this work]	1860
AES [MPL+11]	11031
AES [BGN+14a]	8171
SIMON128/128 [this work]	3044
Higher Order Threshold Implementation	
KATAN-32 [BGN+14b]	2556
SIMON48/96 [this work]	2362
SIMON128/128 [this work]	4122

Table 4.5: Implementation result on ASIC for different ciphers. `Katan` has key size of 80 bits, `AES` and `Present` have key size of 128 bits

Chapter 5

Analysis

In this section, we propose a practical attack against the unprotected core of Simon128/128 as defined in [AGS14]. We highlight that, the previous SCA attacks proposed in [BGDN14] and [SSA14] were developed against the full-state implementation, and cannot be used against the bit-serialized version of our focus. Then, we show the results of this attack against the threshold implementation core along with a thorough leakage detection of the threshold implementation core and higher order threshold implementation core. We implemented this design in a way that the input to the core is already in masked form and the random masks are applied from an external source. Here, we use $x(a)_b$ to denote bit number $b \in [0:63]$ of the word $x : l \vee r$ in round number $a \in [1:68]$. x can also denote the key k. The practical test setup consists of a SASEBO-GII board to develop the hardware design, a Tektronix DPO-5104 oscilloscope to collect the power traces and a ZFL-1000LN amplifier to improve resolution of the collected traces.

5.1 Practical Attacks

The first step in DPA is to identify a sensitive intermediate variable, which depends on both the input data and the secret key in a non-linear equation with as low confusion as possible. Linear equations can also work (as used in [BGDN14]), but the attack in this case will need more traces to distinguish between the correct key and close-by ones. Low confusion means that the non-linear operation processes a small number of the key-bits. This is recommended to break the complexity of the secret key into smaller portions (divide-and-conquer). In this section first we look at one attack against the unprotected core of SIMON and present that performing the same attack on the threshold implementation of SIMON will not work anymore. Then we will look at one proposal by Beaulieu et al. [BSS+15] and present a valid way of breaking it.

5.1.1 Attack Against Loop Unrolling

Moradi et al. in [MMP11] presents the results of correlation collision attacks on different countermeasure including the loop unrolling model. They showed that by using this countermeasure the number of required traces to attack the algorithm will increase significantly. The number of traces to attack the unrolled version of AES will increase from 100,000 to 3,500,000 when the core processes four rounds per clock cycle instead of one round per clock cycle.

In this work, SIMON32/64 is selected as an example to simulate the behaviour of loop unrolling. Due to the structure of SIMON32/64 all the initial key will be used and there is no need for updating the key.

Let us assume that the unrolled version of SIMON maps the plaintext to the data presented at the fifth round of SIMON. Here, we use $x(a)_b$ to denote bit number

$b \in [0:16]$ of the word $x : l \vee r$ in round number $a \in [1:32]$. We select $r(5)_0$ as a point to attack. Based on the round function of SIMON we just have to write down the equation of $r(5)_0$ by only using the plaintext and initial key.

The equation for $r(5)_0$ based on the data presented at round four is as follows.

$$r(5)_0 = l(4)_0$$

The bit $l(4)_0$ can be written using the data at round three as following.

$$l(4)_0 = r(3)_0 + (l(3)_{15} \times l(3)_8) + l(3)_{14} + k(3)_0$$

The data presented in the above equation can be presented by data at round two as following.

$$\begin{aligned}
&r(3)_0 = l(2)_0 \\
&l(3)_{15} = r(2)_{15} + (l(2)_{14} \times l(2)_7) + l(2)_{13} + k(2)_{15} \\
&l(3)_8 = r(2)_8 + (l(2)_7 \times l(2)_0) + l(2)_6 + k(2)_8 \\
&l(3)_{14} = r(2)_{14} + (l(2)_{13} \times l(2)_6) + l(2)_{12} + k(2)_{14}
\end{aligned}$$

Finally, we can show all the previous data by just using the plaintext at round one.

$$r(2)_{15} = l(1)_{15}$$

$$r(2)_{8} = l(1)_{8}$$

$$r(2)_{14} = l(1)_{14}$$

$$l(2)_{0} = r(1)_{0} + (l(1)_{15} \times l(1)_{8}) + l(1)_{14} + k(1)_{0}$$

$$l(2)_{14} = r(1)_{14} + (l(1)_{13} \times l(1)_{6}) + l(1)_{12} + k(1)_{14}$$

$$l(2)_{7} = r(1)_{7} + (l(1)_{6} \times l(1)_{15}) + l(1)_{5} + k(1)_{7}$$

$$l(2)_{13} = r(1)_{13} + (l(1)_{12} \times l(1)_{5}) + l(1)_{11} + k(1)_{13}$$

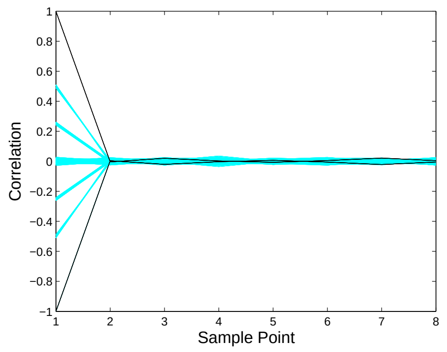
$$l(2)_{6} = r(1)_{6} + (l(1)_{5} \times l(1)_{14}) + l(1)_{4} + k(1)_{6}$$

$$l(2)_{12} = r(1)_{12} + (l(1)_{11} \times l(1)_{4}) + l(1)_{10} + k(1)_{12}$$

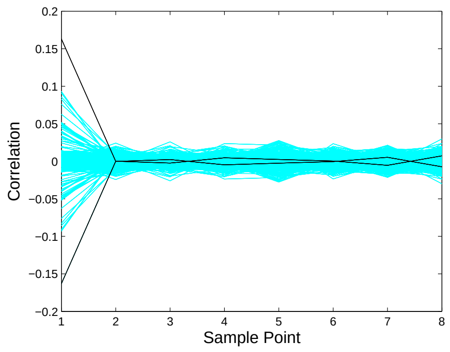
Now that we show bit $r(5)_0$ can be shown by only data presented at round one, we can do our attack. As it was mentioned before, the other property of SIMON32/64 is that all the key bits presented in the previous equations are also the initial key. This is not the case with the same unrolled version of SIMON128/128 where the key bits should also be extracted by key schedule function.

For the first simulation, only one bit, i.e., $r(5)_0$ will be saved as power consumption. The result of CPA by using Hamming distance model is shown in Figure 5.1a and as it can be expected the correlation coefficient is 1 for the correct keys. For the second simulation we assume more realistic power consumption and that is having all the 32 bits of data being present in the power traces. The result of the CPA attack by using Hamming distance model is also shown in Figure 5.1b. The correlation coefficient reduced significantly but the correct key can still be found.

There are more than one correct key in both of the scenarios and the reason is



(a) CPA results for the loop unrolled Simon when considering 1 bit



(b) CPA results for the loop unrolled Simon when considering all the state

Figure 5.1: Attacks against loop unrolling using simulated traces

that in the above equations some of the key bits will appear in a way that the `XOR` of two key bits will be in a function. In this case there are two correct key bits. By doing the CPA attack and iterate over all those 10 key bits presented in the above equation we could be able to recover 7 bits of the initial key. We can select different data at round five, i.e., $x(5)_i$ and try to extract different key bits each time.

5.1.2 Attack Against Unprotected Core

In order to satisfy the mentioned properties, we focus on attacking the output of the non-linear operation (the `AND` gate) in the second round of Simon, where the first key word $k(1)$ becomes part of $l(2)$ to compute $l(3)$. We do this analysis bit-by-bit following the bit-serialized implementation. The equation for the first bit of $l(3)$ is:

$$l(3)_0 = r(2)_0 + (l(2)_{63} \times l(2)_{56}) + l(2)_{62} + k(2)_0$$

where

$$\begin{aligned} r(2)_0 &= l(1)_0 \quad \text{, and} \\ l(2)_i &= r(1)_i + (l(1)_{i-1} \times l(1)_{i-8}) + l(1)_{i-2} + k(1)_i \end{aligned}$$

where $i \in \{62, 63, 56\}$ for this particular bit and the subtraction in indexes is done modulo 64. A similar equation can be written for all the bits of the internal state. In short, one bit of the left word in round three (e.g. $l(3)_0$) depends non-linearly on two key-bits ($k(1)_{63}$ and $k(1)_{56}$) and linearly on another two key bits ($k(2)_0$ and $k(1)_{62}$), along with some input data.

The second step of a successful DPA attack is to select an accurate power model, which is a function that converts the sensitive intermediate variable into relative power consumption. In this work, we use the Hamming Distance (HD) power model which is suitable for hardware modules. The HD represents the number of bit-flips between two clock cycles. For example, we focus on the activity of the first register of the left word, representing the operation of overwriting bit $l(3)_0$ by bit $l(3)_1$ between cycle 65 and 66. However, we first need to consider an equation for the system power consumption.

The system power equation of the unprotected structure (only one share) is:

$$P = P_{SRU} + P_{SRD} + P_{FIFO1} + P_{FIFO2} + N$$

where $P_{SRU}, P_{SRD}, P_{FIFO1}$ and P_{FIFO2} represent the power consumption of the SRU, the SRD and the FIFO registers, respectively. N is a noise component which represents the measurement noise along with all on-board activities that do not depend on the input data including the key-schedule circuit. We did not write a separate term for the LUT as its effect can be included in its output register, which is the first register of SRU or SRD depending on the clock cycle (SRU in our example). During the update of cycle 65/66 and following the HD model, the power consumption of each component is:

$$P_{SRU} = HW\Big(\big(l(3)_0||r(2)_{63:55}\big) \oplus \big(l(3)_1||l(3)_0||r(2)_{63:54}\big)\Big)$$

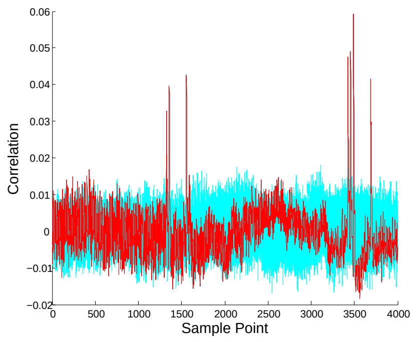
$$P_{SRD} + P_{FIFO1} = HW\big(l(2)^1 \oplus l(2)^2\big)$$

$$P_{FIFO2} = HW\Big(\,|(l(2)_0||r(2))|_{64} \oplus |(l(2)_1||l(2)_0||r(2))|_{64}\Big)$$

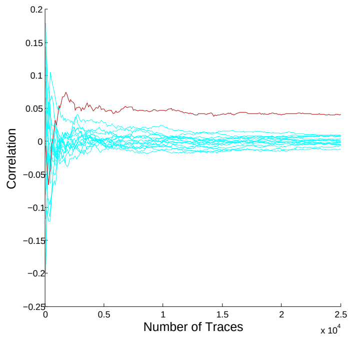
where HW is the Hamming weight function (the number of set-bits), X^s is a circular shift left by s bits and $|x|_{64}$ denotes trimming x to the first 64 bits. $P_{SRD} + P_{FIFO1}$ and P_{FIFO2} depend linearly on the plaintexts and the bits of $k(1)$. P_{SRU} is the only component in the system power consumption that depends non-linearly on key bits.

Figure 5.2a and 5.3a give the results of attacking the studied Simon cores with Correlation Power Analysis (CPA) [BCO04]. In this attack, we used a 4-bit key hypothesis to represent the non-linear key-bits involved in the computation of $l(3)_0$ and $l(3)_1$. Figures 5.2a and 5.2b show results for attacking the unprotected core. Figure 5.2a shows the correlation coefficient as a function of time. Figure 5.2b shows the correlation associated with the correct key against those of the incorrect keys as the number of analyzed traces increases. Although the results highlight the success rate of recovering only four bits of the secret key, the remaining key-bits could also be recovered by selecting another points in the algorithm using the same number of traces. These results shows that the unprotected core can be broken with less than 1200 traces.

Figures 5.3a and 5.3b show the results of the same attacks against the threshold implementation core. In this experiment, we collected 500,000 traces of the parallel version synthesized with speed optimization. If this core passes the attack and the leakage quantification tests, the serialized version will pass for being designed with

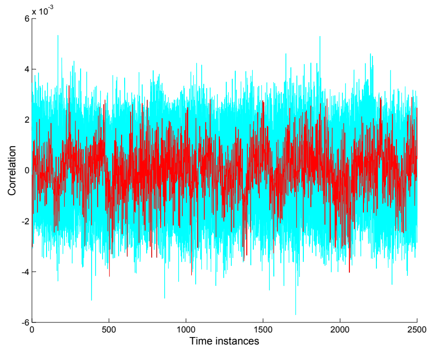


(a) CPA results for the core

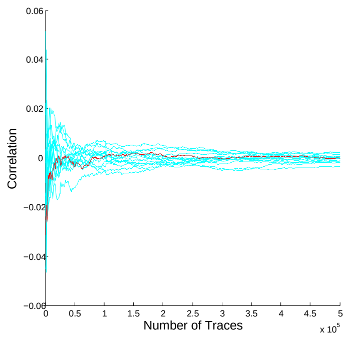


(b) Number of required traces to successfully break the core

Figure 5.2: Attack against the unprotected core, key can be extracted from the implementation



(a) CPA results for the core



(b) Number of required traces to successfully break the core

Figure 5.3: Attack against the threshold implementation core does not work with as many as 500,000 traces

more conservative assumptions. Although serial version is designed more conservatively one has to make sure that synthesize tool does not combine shares together which might be the case for MUXes in FPGAs. It is clear that the attack fails to recover any secret key, which supports our claim of secrecy.

5.2 Leakage Detection

Although the attack mentioned in Section 5.1.2 is necessary to prove the SCA-security of the proposed module, the attack examines the leakage of a single point in the trace which is not sufficient. The way to attack the threshold implementation core is either by proposing a more complex key extraction attack or using other generic methods to detect a leakage. The leakage detection technique examines the entire trace searching for any point where the leakage can be distinguished from random noise. Here, we do not use any key-recovery attack, but we use statistical tools to prove the indistinguishably of the collected traces. These tests are stronger than the previous DPA attack, as they search for the distinguishability in any trace point that may or may not lead to a full key recovery.

We use the test suite developed in [GJJR11]. This work was mainly proposed to satisfy two needs for such a detection methods. Firstly, there should be some clear parameter in order to pass or reject a device. Secondly, The should be done in an easy manner without the need for sophisticated attacks.This work gained a lot of attention recently and it was also used in [LMW14, BGN$^+$14b] to evaluate the effectiveness of their countermeasures.

The concept of the test is to gather some measurements and partitioned them into two group. Based on those measurements are obtained the partitioning method is going to be different. The measurements can be obtained from the set of randomly varying plaintext and the test based on that is called Random Versus Random (RVR). It can also be based on a fixed plaintext and randomly varying plaintext which is called Fixed Versus Random (FVR). The null hypothesis means that those two set have similar means and variance. The other hypothesize is that the mean of the two sets is different. The t-test performs the evaluation of the null hypothesize and determines with a confidence level whether two sets of measurement are from same distribution or not.

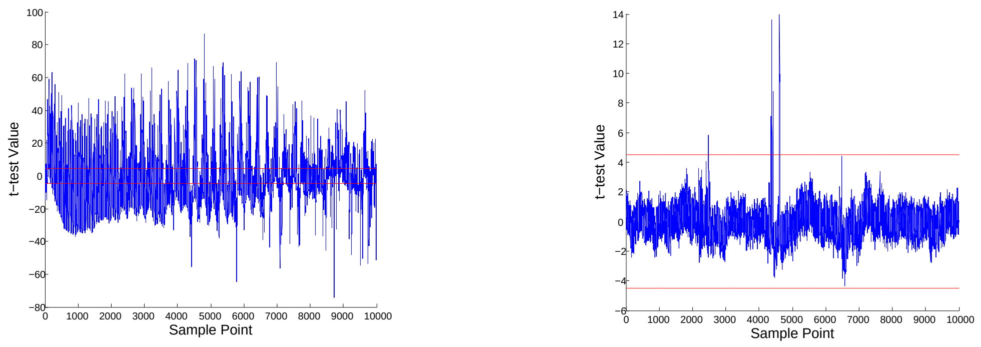
As it was mentioned the FVR test depends on collecting two sets of leakage traces, one with a fixed plaintext while the other with randomly varying plaintexts. The traces are collected in an interleaved way to minimize the effect of noise. We compute the sample mean (μ) and sample standard deviation (σ) of the traces in each set. Then, we compute the result of Welchs t-test:

$$t = \frac{\mu_a - \mu_b}{\sqrt{(\sigma_a^2/N_a) + (\sigma_b^2/N_b)}}$$

where a and b denote the two sets and N_i denote the number of traces in set $i : a \vee b$. The device fails the FVR test if the value of t exceeds a certain threshold. This threshold corresponds directly to the confidence level which was mentioned before. It is shown in [SM15] that if two sets of measurements have approximately equal number of traces and similar variance by choosing ± 4.5 as a threshold the confidence level is going to be %99.999. This threshold, i.e. ± 4.5, is also used in [GJJR11] and [LMW14].

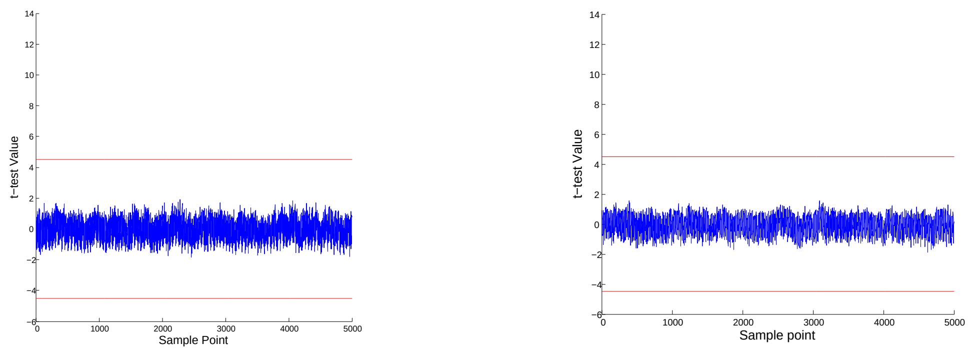
The RVR test applies the same analysis as above however, all the traces are collected with randomly varying plaintexts. In this case, the two groups of traces are separated based on an intermediate variable. We apply the RVR test to the HD between the first bits of the left and right words of the first two rounds.

Figure 5.4a and 5.4b report results of the FVR and the RVR tests for the unprotected core at 100,000 traces, respectively. Figures 5.5a and 5.5b report results for the threshold implementation core at 2,000,000 traces. We applied the aforementioned RVR tests and report results of only one intermediate variable (the HD in the first register during cycle 65/66). The unprotected core failed all the leakage



(a) Fixed vs. Random (b) Random vs. Random

Figure 5.4: Results of leakage detection for unprotected core using 500,000 traces



(a) Fixed vs. Random (b) Random vs. Random

Figure 5.5: Results of leakage detection for threshold implementation core using 2,000,000 traces

quantification tests (as expected), while the threshold implementation core did pass all the tests which again supports our claim of secrecy.

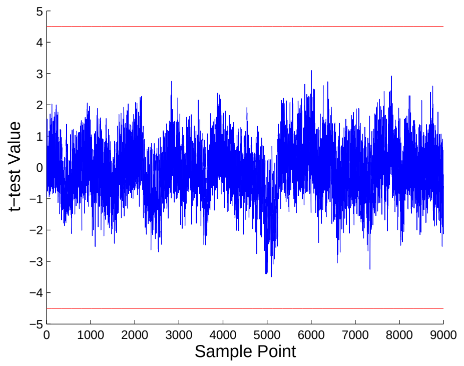
These tests can also be applied for higher order analysis but they require pre-processing before the statistical test. To perform higher order test the traces should be mean-free squared. Let us denote the random variable of the power traces with X and as it was stated we use μ and σ as sample mean and sample standard deviation, respectively.

In order to analyze second-order evaluations we use $(X-\mu)^2$ and for orders more than 2 we use $(\frac{X-\mu}{\sigma})^d$. The natural way of computing the higher-order analysis is by processing traces twice. First time to compute the mean and the second time to calculate the mean-free traces. This model of analysis can be quite time consuming since all the traces should be processed even if the device fails for rather small number of traces. Schneider and Moradi in [SM15] introduced a way to compute the mean-free traces in a one pass manner so that early exit from the analysis is possible if the leakage is found in the early stages. This method can be done while the traces are being collected so some overheads such as analysis time and some other delays regarding to saving traces in memory will be reduced.

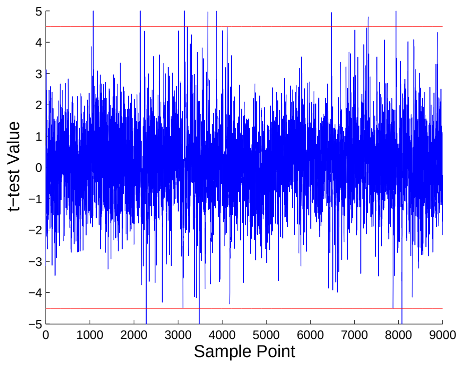
We gathered 20 million traces for the threshold implementation core as well as higher order threshold implementation core. The traces are for the first four round of SIMON. As it can be seen in Figure 5.6 the threshold implementation of SIMON leaks at second order analysis while being resistant against first order analysis. The number of traces are not sufficient to observe third and fourth order leakages.

As it was mentioned we gathered 20 million traces for the first four round of SIMON. It can be seen in Figure 5.7 that the higher order threshold implementation of SIMON does not leak at second order analysis as well as being resistant against first order analysis. The number of traces are not sufficient to observe third and fourth order leakages.

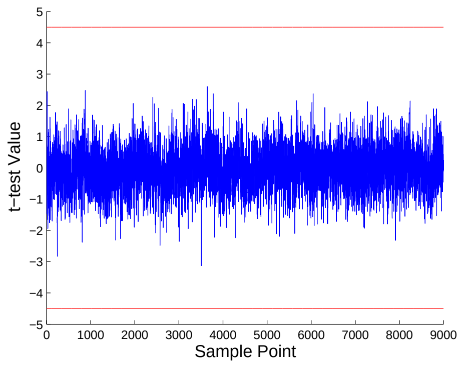
It is important to have an estimation on how many traces we need in order to detect higher order leakage. Figure 5.8 represents the progress of second order t-test value for one point throughout the measurements. As it can be seen, around 10 million traces the implementation starts to leak. Bilgin et al. [BGN$^+$14b] also presented the higher order analysis, in their paper they were able to detect fifth order leakages for 300 million traces while there was no evidence of third order leakage. The analysis of the higher order threshold implementation of SIMON in order to see third order leakage should be performed with more than just 20 million traces which is beyond the scope of this work. Another open problem is to estimate the number



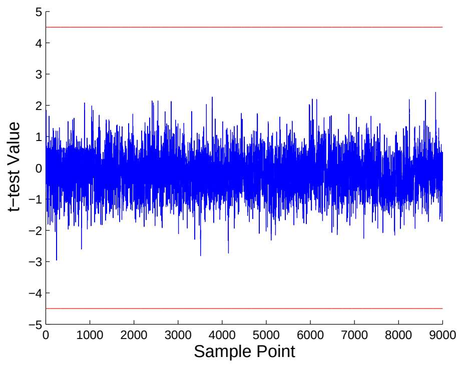
(a) First order FVR



(b) Second order FVR

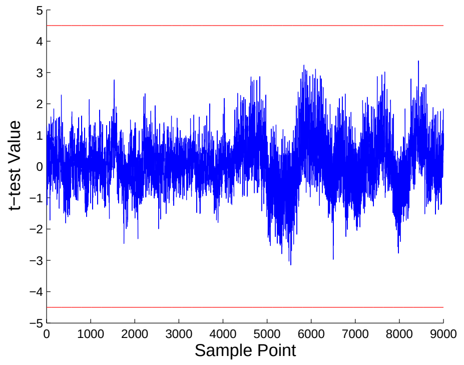


(c) Third order FVR

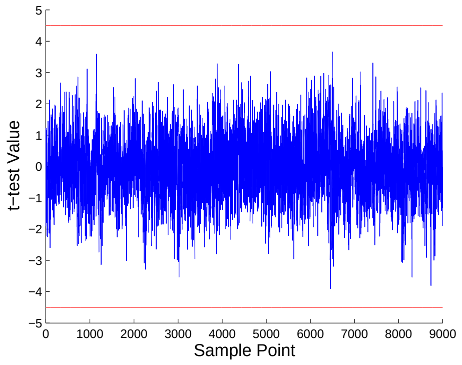


(d) Fourth order FVR

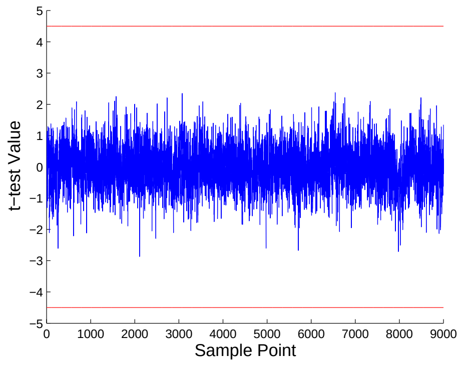
Figure 5.6: Leakage detection result for threshold implementation core for the first four round of Simon using 20,000,000 traces



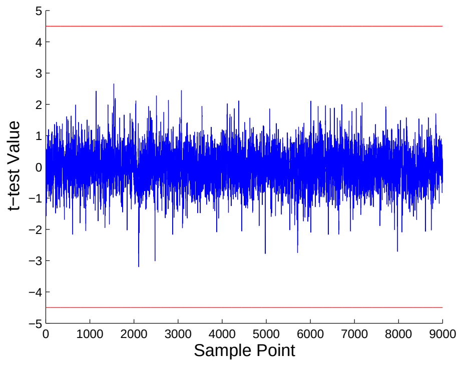
(a) First order FVR



(b) Second order FVR



(c) Third order FVR



(d) Fourth order FVR

Figure 5.7: Leakage detection result for higher order threshold implementation core for the first four round of SIMON using 20,000,000 traces

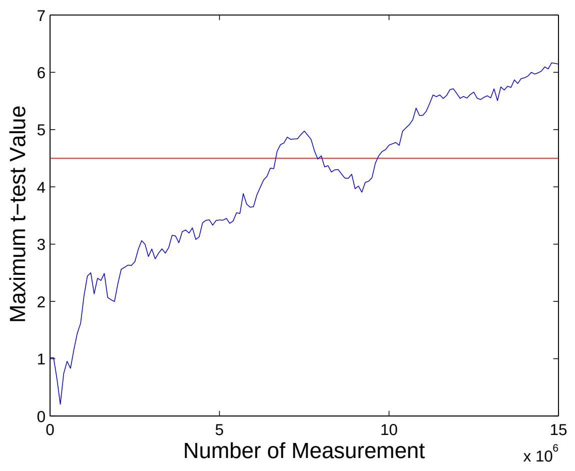


Figure 5.8: Progress of t-test value over 15 million traces for one point throughout the measurements

of required traces to be able to detect higher order leakages.

Chapter 6

Conclusion

In this work, we presented possible ways of protecting an implementation of a cryptographic cipher. Threshold implementation as a possible way of achieving this goal is introduced. We proposed a threshold implementation of SIMON block cipher that can be implemented in less than 100 slices of a low-cost FPGA platform. The thorough leakage detection for the threshold implementation of SIMON is also presented. We showed that the threshold implementation of SIMON is secure against first order attacks, but it is vulnerable against second order attacks. To fix the vulnerability against second order attacks, higher order threshold implementation of SIMON is introduced. We gathered 20 million measurement for this core and presented its resistance against first order and second order attacks. Our future work will be focused on the analysis of higher order threshold implementations with more number of measurements.

Bibliography

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An implementation of DES and AES, secure against some attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2001*, pages 309–318. Springer, 2001.

[AGS14] A. Aysu, E. Gulcan, and P. Schaumont. SIMON Says: Break Area Records of Block Ciphers on FPGAs. *Embedded Systems Letters, IEEE*, 6(2):37–40, June 2014.

[BCD$^+$13] G Becker, J Cooper, E DeMulder, G Goodwill, J Jaffe, G Kenworthy, T Kouzminov, A Leiserson, M Marson, P Rohatgi, and S Saab. Test vector leakage assessment (TVLA) methodology in practice, 2013.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Springer LNCS*, pages 135–152. 2004.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in CryptologyEUROCRYPT97*, pages 37–51. Springer, 1997.

[BGDN14] S. Bhasin, T. Graba, J.-L. Danger, and Z. Najm. A look into SIMON from a side-channel perspective. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2014*, pages 56–59, May 2014.

[BGN$^+$14a] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. A More Efficient AES Threshold Implementation. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology –AFRICACRYPT 2014*, volume 8469 of *Springer LNCS*, pages 267–284. 2014.

[BGN$^+$14b] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-Order Threshold Implementations. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology–ASIACRYPT 2014*, volume 8874 of *Springer LNCS*, pages 326–343. 2014.

[BGSD10] Shivam Bhasin, Sylvain Guilley, Laurent Sauvage, and Jean-Luc Danger. Unrolling cryptographic circuits: a simple countermeasure against side-channel attacks. In *Topics in Cryptology-CT-RSA 2010*, pages 195–207. Springer, 2010.

[BKL$^+$07] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. *PRESENT: An ultra-lightweight block cipher*. Springer, 2007.

[BSS$^+$13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.

[BSS$^+$15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, page 175. ACM, 2015.

[CESY14] Cong Chen, Thomas Eisenbarth, Aria Shahverdi, and Xin Ye. Balanced Encoding to Mitigate Power Analysis: A Case Study. In *Smart Card Research and Advanced Applications*, pages 49–63. Springer, 2014.

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology–CRYPTO'99*, pages 398–412. Springer, 1999.

[DKH$^+$13] Anh Do, Soe Thet Ko, Aung Thu Htet, Thomas Eisenbarth, and Berk Sunar. Electromagnetic Side-Channel Analysis on Intel Atom Processor. April 24, 2013.

[FG05] Wieland Fischer and Berndt M Gammel. Masking at gate level in the presence of glitches. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 187–200. Springer, 2005.

[GB05] Tim Good and Mohammed Benaissa. AES on FPGA from the fastest to the smallest. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 427–440. Springer, 2005.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation. Non-Invasive Attack Testing Workshop, 2011.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[KNP12] Sebastian Kutzner, Phuong Ha Nguyen, and Axel Poschmann. Enabling 3-share Threshold Implementations for any 4-bit S-box. *IACR Cryptology ePrint Archive*, 2012:510, 2012.

[KNPW13] Sebastian Kutzner, PhuongHa Nguyen, Axel Poschmann, and Huaxiong Wang. On 3-Share Threshold Implementations for 4-Bit S-boxes. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, volume 7864 of *Springer LNCS*, pages 99–113. 2013.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in CryptologyCRYPTO96*, pages 104–113. Springer, 1996.

[LMW14] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. Gate-Level Masking under a Path-Based Leakage Metric. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Springer LNCS*, pages 580–597. 2014.

[MMP11] Amir Moradi, Oliver Mischke, and Christof Paar. Practical evaluation of DPA countermeasures on reconfigurable hardware. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 154–160. IEEE, 2011.

[MPG05] Stefan Mangard, Thomas Popp, and BerndtM. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer Berlin Heidelberg, 2005.

[MPL$^+$11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology — EUROCRYPT 2011*, volume 6632 of *Springer LNCS*, pages 69–88. 2011.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 157–171. Springer, 2005.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307 of *Springer LNCS*, pages 529–545. 2006.

[OMPR05] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-box. In *Fast Software Encryption*, pages 413–423. Springer, 2005.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. Cryptology ePrint Archive, Report 2015/207, 2015. `http://eprint.iacr.org/`.

[Smi15] Ray Beaulieu Douglas Shors Jason Smith. Simon and Speck: Block Ciphers for the Internet of Things. 2015.

[SSA14] Dillibabu Shanmugam, Ravikumar Selvam, and Suganya Annadurai. Differential Power Analysis Attack on SIMON and LED Block Ciphers. In RajatSubhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering*, volume 8804 of *Springer LNCS*, pages 110–125. 2014.

[STE15] Aria Shahverdi, Mostafa Taha, and Thomas Eisenbarth. Silent Simon: A Threshold Implementation under 100 Slices. Cryptology ePrint Archive, Report 2015/172, 2015. `http://eprint.iacr.org/`.

[TKL05] Elena Trichina, Tymur Korkishko, and Kyung Hee Lee. Small size, low power, side channel-immune AES coprocessor: design and synthesis results. In *Advanced encryption standard–AES*, pages 113–127. Springer, 2005.

[YK09] P. Yalla and J. Kaps. Lightweight Cryptography for FPGAs. In *International Conference on Reconfigurable Computing and FPGAs, 2009. ReConFig '09.*, pages 225–230, Dec 2009.