Side-Channel Analysis of ALE

Jakob Girstmair
jakob.girstmair@gmail.com

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Prof. Karl-Christian Posch
Assessor: Thomas Korak

September, 2015

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Ich bestätige hiermit, diese Arbeit selbständig verfasst zu haben. Teile der Diplomarbeit, die auf Arbeiten anderer Autoren beruhen, sind durch Angabe der entsprechenden Referenz gekennzeichnet.

Jakob Girstmair

Acknowledgements

I would like to thank my parents for supporting me through these strange times that we live in, all my friends for motivating me to hang in there, Sandra Grinschgl for her unbreakable optimism and the staff at IAIK for their unmatched expertise, never-ending patience and unflinching support. I have to thank you all.

l

Abstract

In this master thesis we examined the ALE authenticated encryption algorithm towards its practical implementation and its resistance to side-channel attacks. ALE is a lightweight authenticated encryption scheme based on AES-128 whose use in the initialization stage of the algorithm can be used for first-order DPA attacks on ALE.

For this we provided our own optimized low-area implementation which showcases a smaller datapath of the AES-core than compared to what was previously achieved with AES implementations of a similar architecture. We have shown that this architecture of AES is even more vulnerable to DPA attacks than was originally assumed due to the effects high-fanout registers of the FPGA have on the switching noise. We have also examined the influence single-pole Butterworth filters have on the success rate of DPA attacks and examined our DPA results in regards to the characteristic frequency found in the correlation results and the recorded power traces.

Next we implemented two FPGA-specific generic countermeasures to improve on the side-channel resistance and proposed ways to improve them. The first countermeasure randomizes the clock signal by switching between multiple differently phase-shifted clock signals which yielded promising results but was weaker than originally described in its proposal. We improved on this countermeasure by lowering the frequency of half of the phase shifted signals by a factor of 16 which results in a mixed operating frequency on the design. The second countermeasure involves the randomized switching of short-circuits as means to generate noise on the power traces. This countermeasure not only proved hard to implement but was also very ineffective.

Lastly we utilized Welsh's t-test as a means to further examine the effectiveness of the implemented countermeasures.

Keywords: ALE, AES, side-channel, DPA, t-test, frequency-dependence, ASIC

Kurzfassung

Im Zuge dieser Masterarbeit wurde der ALE-Algorithmus hinsichtlich seiner praktischen Implementierung und seiner Resistenz gegenüber Seitenkanalattacken überprüft. ALE ist ein leichtgewichtiger und authentifizierender Verschlüsselungsalgorithmus, der auf AES-128 basiert. Die unveränderte Nutzung von AES-128 während der Initialisierungsphase von ALE wurde benutzt, um DPA-Attacken auf ALE durchzuführen.

Um diese Attacke durchführen zu können, wurde im Rahmen dieser Masterarbeit eine low-area-optimierte ASIC-Implementation von ALE angefertigt. Diese implementation erreicht einen kleineren Datenpfad beim AES-128-Kern als vorangehende Implementationen mit ähnlichen Architektur-Konzepten. Wir haben gezeigt, dass dieser Architekturansatz für AES-128 weitaus angreifbarer ist, als ursprünglich angenommen wurde, diese fehlende Resistenz kann zum Teil auf eine Aufsummierung des relevanten Stromverbrauchs beim Speichern der Information im Schieberegister zurückgefürt werden. Des Weiteren wurde der Einfluss von Tiefpassfiltern erster Ordnung auf die Erfolgsraten der DPA-Angriffe untersucht und die DPA-Ergebnisse in Hinblick auf das Phänomen der Eigenfrequenz der verwendeten FPGA untersucht. Diese Eigenfrequenz ließ sich sowohl in den Kurven der korrekten Schlüsselhypothese in den Ergebnissen der DPA als auch in den aufgezeichneten Stromverbrauchskurven beobachten.

Als Nächstes wurden zwei FPGA-spezifische Gegenmaßnahmen implementiert, um die Resistenz gegen Seitenkanalangriffe zu erhöhen. Die erste Gegenmaßnahme randomisiert das interne Clock-Signal der Implementation durch ein randomisiertes Umschalten zwischen verschieden phasenverschobenen Clock-Signalen. Diese Gegenmaßnahme lieferte vielversprechende Ergebnisse, die aber leider hinter den Ergebnissen des ursrprünglichen Vorschlags lagen. Des Weiteren wurde diese Gegenmaßnahme in dieser Masterarbeit verbessert, indem ein Teil der verschieden phasenverschobenen Clock-Signale auf ein Sechzehntel der ürsprünglichen Frequenz verlangsamt wurde. Die zweite Gegenmaßnahme nutzt das randomisierte An- und Abschlaten von Kurzschlüssen, um Rauschen auf den aufgezeichneten Stromverbrauchskurven zu erzeugen. Es zeigte sich, dass diese Gegenmaßnahme sowohl schwer zu implementieren als auch sehr ineffektiv ist.

Abschließend wurde Welshs T-Test alternativ zur DPA-Attacke benutzt, um die Effektivität der Gegenmaßnahmen besser beurteilen zu können.

Stichwörter: ALE, AES, side-channel, DPA, t-test, frequency-dependence, ASIC

Contents

Chapter 1

Introduction

Throughout the last two decades computers have become increasingly integrated into all aspects of human life. The first interactions of our children with a computer -maybe a tablet or a smartphone- will probably take place before they can even speak. Computers have become so ubiquitous, so diverse and in many aspects independent from human interaction that the mere task of listing their applications and purposes seems exhausting.

But as the years went by our trust in the security of information technology has always been the achilles heel of this integration. Passwords, communication, any recordings of our daily lives will always be of interest to other people we do not want to share this information with. As many of the devices we use process or generate data we deem as private the security with which these are handled has to be evaluated and reviewed with great care. Security issues like *Heartbleed* have been present in the media with increasing coverage[1] and it is safe to say that the continued engagement of nation states to circumvent and water down existing security measures will be a defining question of the 21st century.

The work of the scientific community with its analysis and proposals of security schemes and paradigms has been one of the pillars against this lack of trust. A defining achievement of this scientific approach is our ability to use many cryptographic tools like the *Advanced Encryption Standard (AES)* for diverse applications with certain amounts of trust and ease.

One of the most pressing and current issues regarding IT security is the standardization of *authenticated encryption (AE)*. *AE* includes all block cipher[2] modes which simultaneously provide *confidentiality*, *integrity*, and *authenticity* assurances and has been discussed by researchers in detail since 2000 [12]. *Authenticity* is necessary for every implementation of secure communication and has often been added through a combination of *confidentiality* and *authenticity* modes. The need for dedicated *AE* modes of operation emerged after it became clear that the combination of *confidentiality* and *authenticity* modes is prone to errors [17]. This realization was followed by attacks on existing implementations (e.g. OpenSSH [18], TLS-encrypted cookies [19]) while other implementations still fail to use *authentication* by default or allow to disable *authentication* for a gain in performance (e.g. IPSec [20, 21, 22], VMWare) and are still vulnerable.

Many *AE* modes of operation like *CWC* [23], *GCM* [24], *OCB* [25] and *EAX* [26] have been proposed and implemented over the last years but some of them have their own complications. *OCB* for example outperforms *GCM* but is hindered by its patented

[1] As an example see: http://bits.blogs.nytimes.com/2014/04/18/heartbleed-internet-security-flaw-used-in-attack/

[2] A block cipher denotes any encryption which operates on a fixed number of bits (ergo a *block*).

status held by Philip Rogaway. *EAX* on the other hand received standardization as a slightly modified version called *EAX prime* [27] in *ANSI C12.22* which sacrificed its proof of security for better performance and which was broken by Kazuhiko Minematsu et al. [28] in 2012. In order to standardize an authenticated encryption scheme that matches or surpasses *GCM's* performance the CAESAR challenge was started. Its submission Deadline was set to May 2014 and it will finish in late 2017. Submissions have predominantly been based on sponge constructions (e.g. ASCON [29]) and AES (e.g. AES-OTR [30]).

These proposals also reached the domain of *lightweight* security which has its own specific set of resource constraints. Especially aspects like low-power, low-energy and low-area are important design factors. Lightweight security aims at applications like security in RFID-tags which also raises the exposure of these devices to side-channel attacks. *Side-channel attacks* use information gained by the physical implementations of cryptographic systems. This information can include power consumption [51], emitted sound [55], timings of encryption steps [52] or electromagnetic emissions [51]. A popular example for such an attack is a *differential power analysis (DPA)* [51] attack which utilizes the statistical analysis on the power consumption of the cryptographic system under attack.

The first encryption schemes to be proposed with the *lightweight* aspect in mind were encrytpion modes without *authentication. Lightweight* security schemes were proposed for stream ciphers[3](e.g. Trivium [31], Mickey [32], Grain [33]), block ciphers (e.g. DESXL [34], PRESENT [46], HIGHT [35], mCrypton [36]) and dedicated hash functions (e.g. Spongent [37], Photon [38], Quark [39]) but omitted authenticated encryption until recently.

AE schemes with a focus on lightweight encryption have been relatively rare. Examples are *Grain-128a [40]*, *Hummingbird-2 [41]*, *Fides [10]*, *APE [42]* and *ALE [1]*. *ALE* will be the focus of this thesis. *ALE* has been proposed by Bogdanov et al. in 2013. It is based on *AES-128* and uses a *Pelican MAC* to facilitate *authenticity*. Two implementations were realized by its creators. One was a software implementation to showcase its performance and another was done in hardware (*ASIC*[4]) to project its uses as a ligthweight encryption scheme. The latter uses code protected by copyright law and omits many registers[5] needed for independent usage of the circuit.

This leads us to the contributions of this master thesis. We will propose a hardware (*ASIC/FPGA*[6]) implementation of ALE that is relatively lightweight and can be operated independently and without the need to load every message block twice which was part of the proposal for a ASIC implemenetation for ALE by [1] which had no additional registers besides registers with the purpose to hold key and state values. Also no code protected by copyright will be used. This implementation also uses a different mixcolumns module than used by most of the current lightweight implementation of AES [6]. This achieves a smaller state module when compared to [6]. Futhermore we take a closer look at the side-channel vulnerability of ALE. Due to the fact to its close relationship with *AES* this will also re-evalutate the sidechannel vulnerabilities of AES.

During the course of this thesis we also observed unusual correlation behavior while performing a DPA attack. Although our design and attack hypothesis of AES are similar to that proposed and used by [6] we did not register short bursts of succesful correlation

[3]Also called a *state cipher*, typically one bit is combined with the corresponding bit of its internal state

[4]Application-Specific Integrated Circuits are integrated circuits that are intended for specialized use and cannot be repurposed

[5]A *register* defines a hardware construction to hold and save data

[6]FPGA: *Field-Programmable Gate Array*, defines an integrated circuit that is configurable after manufacturing

peaks but instead calculated a gradually increasing DPA result whose distinguish-ability of the correct hypothesis for one key byte actually lingered well into the timings of the second AES-round[7] and beyond.

We also evaluate the merits of two FPGA-specific countermeasures proposed by [4] with the goal to mitigate these vulnerabilities. The first countermeasure is called *Clock Randomization* and uses the FPGA's own DCMs[8] and a random number generator (RNG) to randomize the internal clock[9] of the integrated circuit. This countermeasure is evaluated over variyng degrees of available clocks to be switched between and has proven to be effective with a rather small effort to implement.

The second countermeasure we implemented are controlled short-circuits in the FPGA which are also controlled by the same RNG and functions as low-cost noise generator. The merit of this countermeasure on its own is debatable but can still mitigate attack results at a rather low cost of FPGA resources especially if the RNG can be shared with other countermeasures.

The remainder of this thesis is organized as follows: **Chapter 2** explains *AES* and *ALE* in more detail and explains some of the discrepancies between the two antedecent implementations of *ALE*. In **Chapter 3** we will take a closer look at our hardware implementation of *ALE* and possible variations. This chapter also discusses how the countermeasures were implemented and examines how many resources were used and which constraints were imposed on the design. **Chapter 4** will give an introduction into the nature of DPA attacks and discuss how a fitting attack was devised for the implemented design. We will look at our measurement setup and the results of a DPA on *ALE* in **Chapter 5**. **Chapter 6** is reserved for the discussion of DPA results of our measurements with the countermeasures turned on. And finally in **Chapter 7** we will give a summary and a outlook on further research.

1.1 Related Work

The proposal of *ALE* [1] acts as one of the centerpieces to this work and is heavily referenced in Chapter 2. It was used as a blueprint (in combination with its software implementation of *ALE)* for our own implementations and is used for comparison in almost every step of the way. As goes with many other encryption schemes -especially ones for *AE*- it uses many other works which are of importance to this thesis.

The core building block of *ALE* is *AES* which was proposed by Joan Daemen and Vincent Rijmen in 2002 [3] which is also referenced heavily in Chapter 2. AES serves as the de facto standard block cipher in current crypto systems. Another work regarding *AES* is Moradi et al.'s currently smallest ASIC implementation [6] of *AES* which provided the AES core for the ASIC implementations of *ALE* in its original proposal [1]. Also in our work, the AES implementation is based on [6]. The AES implementation by Moradi et al [6] also uses Canright's substitution box (Sbox) [14], which is the smallest version for this module yet. Another implementation of *ALE* was proposed by Bogdanov et al. [9] who created a more performance oriented implementation of *ALE* to encrypt FPGA bitstreams.

[7] A more detailed look into how AES works is given in Chapter 2

[8] Digital Clock Managers are built-in into many FPGAs by multiple vendors and are used to manipulate clock signals

[9] the clock signal is a periodic signal used to control the registers of an integrated circuit

Further building blocks of *ALE* are the Pelican MAC [16] function and the LEX leak [15] stream to generate its authentication tag and ciphertext respectively. Both have been iterated in this work. *ALE* also incorporates an added AES keyschedule which has also been used by ASC-1 [11] and shares some of its ideas with Fides [10] which omits the AES core.

ALE has also been already attacked by Khovratovich et al. [7] and Wu et al. [8] although they used different implementations and used specialized attacks which have little in common with our DPA approach. While [7] used a software attack using forged states in a so called LOCAL attack, [8] attacked *ALE* through its cipher generation which reduced *ALE's* authentication security to 97-bit.

The ideas for FPGA-specific countermeasures including clock randomization and the integration of short circuits were provided by Tim Güneysu et al. [4] where they used them on a different implementaion of *AES* which was based on T-tables. The design for the MixColumns module was proposed by Hua Li et al. [13] in 2005 but did not turn out as area efficient as they claimed.

This thesis also relies heavily on Stefan Mangard et al.'s work [2] regarding the execution of DPAs and their optimization including the use of moving average filters. Other important aspects of this work are its guidelines on how to use Hamming weights (HWs) and Hamming distances (HDs) in order to establish hypotheses on power consumption levels.

Chapter 2

Introduction to ALE

In this chapter we take a closer look at the ALE authenticated-encryption algorithm. ALE stands for **A**ES-Based **L**ightweight Authenticated **E**encryption. ALE is heavily based on Rijandel's AES [3] encryption method but uses a LEX-leak [15] to generate a more stream-oriented cipher. It also produces an authentication tag based on the Pelican-MAC [16] function to enable detection of ciphertext manipulation.

The original authors of ALE [1] created two different reference implementations for ALE. One was written in C to take advantage of the NI-instruction set of modern Intel-CPU architectures. The other was created to showcase ALE's affinity to low-area ASIC implementations and was written in a hardware description language. The latter uses code protected by copyright and was unavailable at the time of writing. This thesis understands ALE as it was conceived in the most recent version of the reference C implementation for NI-enabled architectures which features some discrepancies, changes and additions to the algorithm described in its original paper. These differences are discussed on their own in Section 2.5 of this chapter.

This chapter will otherwise be more focused on the specification of ALE. But before we begin with a more detailed discussion of ALE and the creation of its authentication tag, we will give a short overview into its main building block, the advanced encryption standard (AES) in Section 2.1. Further on, we will conclude with a short summary. Before concluding this introductory section we discuss some basic information on ALE.

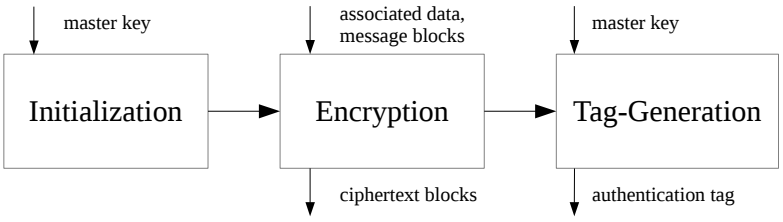

Figure 2.1: Stages of ALE

ALE is an authenticated encryption algorithm which means that it transforms plaintext into ciphertext while also providing an authentication tag which can be used to verify the integrity of the transformed data. This added data integrity is rendering alterations,

forgeries and other manipulations of the ciphertext highly unlikely. ALE's main building block is the advanced encryption standard (AES) which is used and has been used in many cryptographic applications and is well understood by the research community.

ALE is processed in three stages: *Initialization, Encryption* and *Tag-Generation* as can be seen in Figure 2.1. The master key is used as an AES-key during the first and final stages. The initialization stage is used to set up the internal states of ALE, the encryption stage can be used to process associated data and to transform plaintext into ciphertext. The tag-generation stage is used to create the authentication tag.

ALE uses two 128-bit wide internal states for its operations. The two internal states of ALE are called *data state* and *key state*. The *data state* is used as the state of all AES operations after initialization and is also influenced by additions with blocks of plaintext or associated data. The initialized *key state* is used as its AES-key counterpart and is only updated by the regular key-schedule of AES. The 128-bit size of these states also influences the block length for the master key, plaintex and ciphertext which are all defined to have a length of 128-bit (or 16 bytes) per block. An overview of ALE's input and output data is displayed in Table 2.1. Information on associated data in these table is kept in parentheses as associated data is an optional feature. As discussed earlier the next section focuses the Advanced Encryption Standard(AES) which serves as the main building block for ALE.

Stage	Inputs		Outputs	
	Name	*Bitlength*	*Name*	*Bitlength*
Initialization	Master Key IV	128 128	-	-
Encryption	(Associated Data Blocks) Message Blocks	$(l * 128)$ $n * 128$	 Cyphertext Blocks	 $n * 128$
Tag-Generation	Master Key	128	Tag	128

Table 2.1: IO-Data of ALE

2.1 The Ten-Round AES Encryption

The advanced encryption standard (AES) or originally called Rijandel [3], was chosen by the U.S. National Institute of Standards and Technology (NIST) to be the official successor to the data encryption standard[1] (DES) in 2001. There are three official versions of AES, all of them feature a block size of 128 bits for their data state while each having different key sizes of 128, 192 and 256 bits respectively. These versions are referred to as AES-128, AES-192 and AES-256. From here on, when talking about AES we mean AES-128.

One of the most important features of AES is its use of a matrix representation for all of its operations. AES uses two 128-bit wide blocks as inputs. These are transposed from one-dimensional *arrays* as the key and message blocks $k = [k_0, k_1, k_2, ..., k_{15}]$ and $m = [m_0, m_1, m_2, ..., m_{15}]$ into four by four matrices as can be seen below in Equation (2.1). Each element of the former array and the matrices below represent one byte. These matrices are also important to consider when we discuss the LEX-Leak of ALE in one of the following subsections.

[1] puplished in 1977 by the NSA

$$K = \begin{pmatrix} k_{0,0} & k_{0,4} & k_{0,8} & k_{0,12} \\ k_{0,1} & k_{0,5} & k_{0,9} & k_{0,13} \\ k_{0,2} & k_{0,6} & k_{0,10} & k_{0,14} \\ k_{0,3} & k_{0,7} & k_{0,11} & k_{0,15} \end{pmatrix}, M = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \tag{2.1}$$

A regular AES encryption consists of ten rounds, each one of these includes a key addition with the respective round key. These round keys are calculated consecutively through a process which is summarized under the term *key-schedule* (KS) which we will discuss first.

The last column of the k-matrix is rotated one element upward. Then the elements of this column are sent through the Rijandel Sbox. In the next step these Sbox values are xored with the first column of the k-matrix. The first byte is also xored with the current round constant. This forms the first column of the round key. The remaining columns are results of xor operations of the respective element with the same-row element of the previous column. This can be seen in Equation (2.2). The variable t determines the current round and the function $S()$ describes the Sbox-transforation function.

$$\begin{aligned} k_t &= \begin{pmatrix} k_{t,0} & k_{t,4} & k_{t,8} & k_{t,12} \\ k_{t,1} & k_{t,5} & k_{t,9} & k_{t,13} \\ k_{t,2} & k_{t,6} & k_{t,10} & k_{t,14} \\ k_{t,3} & k_{t,7} & k_{t,11} & k_{t,15} \end{pmatrix} \\ &= \begin{pmatrix} k_{t-1,0} \oplus S(k_{t,13}) \oplus RCON(t) & k_{t,0} \oplus k_{t-1,4} & k_{t,4} \oplus k_{t-1,8} & k_{t,8} \oplus k_{t-1,12} \\ k_{t-1,1} \oplus S(k_{t,14}) \oplus 0 & k_{t,1} \oplus k_{t-1,5} & k_{t,5} \oplus k_{t-1,9} & k_{t,9} \oplus k_{t-1,13} \\ k_{t-1,2} \oplus S(k_{t,15}) \oplus 0 & k_{t,2} \oplus k_{t-1,6} & k_{t,6} \oplus k_{t-1,10} & k_{t,10} \oplus k_{t-1,14} \\ k_{t-1,3} \oplus S(k_{t,12}) \oplus 0 & k_{t,3} \oplus k_{t-1,7} & k_{t,7} \oplus k_{t-1,11} & k_{t,11} \oplus k_{t-1,15} \end{pmatrix} \end{aligned} \tag{2.2}$$

$t = 1, 2, ...$ (round) ; $k_0 = K$ (the master key)

The Round Constant (RCON) is a value dependent on the current encryption round and can also be represented by a set of *constant* values. In order to save memory these constants can also be calculated ad hoc by using the following steps. The Round Constant is initialized with the value *1* for the first round. The following round constants are results of a times-2 operation within Rijandel's finite field ($GF(2^8)$)[3]. The result of this operation can be calculated with bit arithmetics quite easily via a one-bit shift-rotation to the left and three xor operations of the old values of bits *three*, *two* and *zero*, with the seventh bit. The seventh bit is the most significant and left-most bit.

$$\begin{aligned} RCON(i+1) &= b(i+1)_7.b(i+1)_6.b(i+1)_5.b(i+1)_4.b(i+1)_3.b(i+1)_2.b(i+1)_1.b(i+1)_0 \\ &= X2(RCON(i)) \\ &= b(i)_6.b(i)_5.b(i)_4.b(i)_3 \oplus b(i)_7.b(i)_2 \oplus b(i)_7.b(i)_1.b(i)_0 \oplus b(i)_7.b(i)_7 \\ RCON(1) &= 0.0.0.0.0.0.0.1 \\ i &= 1, ... \end{aligned} \tag{2.3}$$

This concludes the key-schedule. Next we will look into the state. Each of the ten rounds of AES uses the following round transformations to manipulated the state:

1: SubBytes

All byte values of M are replaced with the values of their respective Sbox transformations. Most commonly this is achieved with a lookup-table but can also be calculated within Rijandel's finite field ($GF(2^8)$).

2: ShiftRows

The rows of the matrix M are shifted to the left. The shifting distance is determined by the row number as can be seen in Figure 2.2. The first row remains unshifted, the second is shifted by one, the third by two and the fourth by three columns.

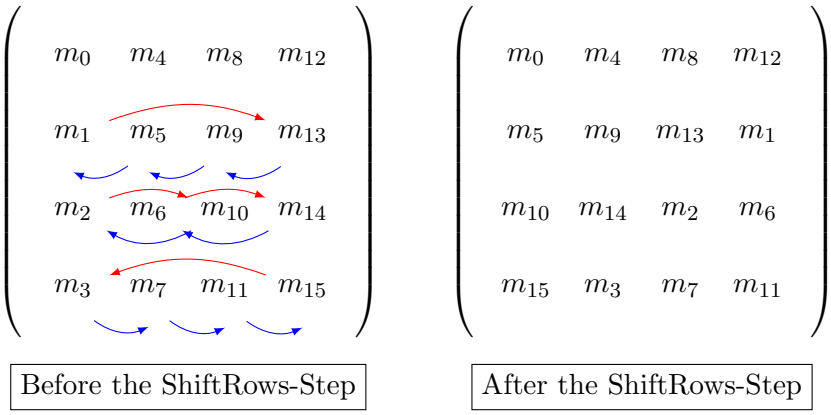

Figure 2.2: ShiftRows-Step on matrix M

3: MixColumns

Each column of the matrix M is substituted by the result of the MixColumns operation. The multiplications and additions of this operation are calculated in the finite field ($GF(2^8)$). The computation is conducted as shown in Equation (2.4).

$$\begin{aligned}\begin{pmatrix}\bar{m}_i\\ \bar{m}_{i+1}\\ \bar{m}_{i+2}\\ \bar{m}_{i+3}\end{pmatrix} &= \begin{pmatrix}2 & 3 & 1 & 1\\ 1 & 2 & 3 & 1\\ 1 & 1 & 2 & 3\\ 3 & 1 & 1 & 2\end{pmatrix}\begin{pmatrix}m_i\\ m_{i+1}\\ m_{i+2}\\ m_{i+3}\end{pmatrix} = \\ &= \begin{pmatrix}2m_i + 3m_{i+1} + m_{i+2} + m_{i+3}\\ m_i + 2m_{i+1} + 3m_{i+2} + m_{i+3}\\ m_i + m_{i+1} + 2m_{i+2} + 3m_{i+3}\\ 3m_i + m_{i+1} + m_{i+2} + 2m_{i+3}\end{pmatrix} \\ i &= 0, 4, 8, 12\end{aligned} \tag{2.4}$$

4: AddRoundKey

All elements of M are xored with the current elements of k_t, which were calculated with the aforementioned key-schedule.

Adding to this the *AddRoundKey*-step is also calculated as the very first step ahead of the first round and the *MixColumns*-step is omitted in the last (e.g. tenth) round. This concludes the Advanced Encryption Standard. Now we will look how AES was used by ALE to form an authenticated encryption algorithm.

2.2 Initialization

In order to set the two states of ALE up for encryption and the possible processing of associated data, three full AES encryptions have to be calculated as can be seen in Figure 2.3. First an initialization vector (IV) has to be AES-encrypted with the master key. This IV is 128-bit long and is suggested to be generated by a counter. Second, the master key is used to encrypt a 128-bit long vector consisting of 0s. In the third AES encryption the results of the first two encryptions are used as key and input. The result of AES encryption of the IV is used as the key while the result of the AES encryption of the zero vector is used as the state. The result of this third ten-round AES encryption is the initialized *data state*.

The result of the first AES encryption in which the IV was encrypted is then also submitted to one additional round of the AES keyschedule with the round constant (RCON) set to '0x6c' which is the round constant for the eleventh round. The result is the initialized *key state*. As this key-schedule is never directly mentioned in the paper but is still in the reference implementation albeit unexplained the author has started to refer to it for the lack of a better word as the *solo key-schedule*. The solo key-schedule can be observed in the upper right corner of Figure 2.3 in the function box denoted as *AES KS 1Round*. It will be further discussed in section 2.5. With the initialization of the internal states complete, associated data blocks and message blocks can now be processed.

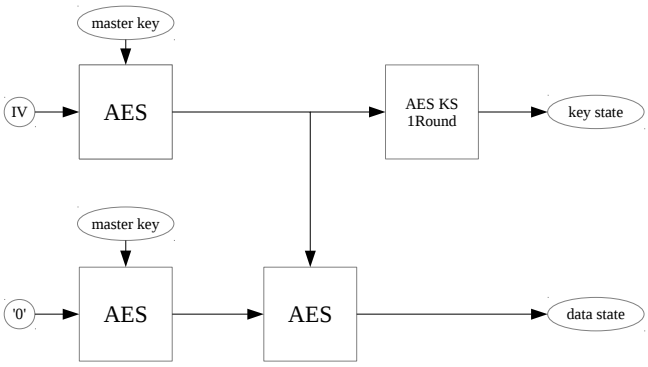

Figure 2.3: Initialization of ALE

2.3 Encryption

The encryption of one single message block or one block of associated data after the initialization is depicted in Figure 2.4. ALE encrypts 128-bit message blocks by performing the following steps:

AddRoundKey

Before the first round of AES is performed the key state is added into the data state, just like in a regular ten-round AES encryption.

4-Round AES
For each message block 4 rounds of AES are performed. After each round the LEX-Leak step is executed.

Word Reversal
Every message block is split up into four 32-bit long *words* which are used to generate the according *words* of the ciphertext. These words are used after each round of AES in reverse order. The order of the bytes inside those words remains intact.

LEX-Leak
To generate the ciphertext, the aforementioned words are added with selected parts of the *data state*, these parts are selected by the so called LEX-Leak, which consists of two masks. One is for odd rounds of the AES the other is for even rounds. These masks are represented by the following matrices. These four-by-four matrices are applied to the data state of ALE which also consists of a four-by-four matrix with one byte per element. A *1* represents a byte that will be xored with one byte of the message word while a *0* will be ignored. The order of the bytes extracted to the data state is defined as top-then-down before left-then-right.

$$odd = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, even = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{2.5}$$

Add
After four rounds of AES with LEX-Leak operations the whole message block is added into the *data state*. This is essential for the processing of associated data where the aforementioned LEX-Leak operations can be omitted.

After the last block message data has been encrypted, ALE should proceed to its *tag-generation* stage. The reference implementation and the original ALE paper left unclear whether the switch into the tag-generation stage should be done automatically or how this should be communicated to the algorithm.

2.4 Tag Generation

After the final message block has been processed the current *data state* serves as the input to a 10-round AES encryption with the original master-key as can be seen in Figure 2.5. The result of this encryption is called the authentication tag. After this all states can be reset.

2.5 Discrepancies between Paper and Reference Implementation

In the following paragraphs we are going to discuss some discrepancies between the original ALE paper [1] and the available reference implementation and some omissions by the former. Some of these are needed to completely specify ALE while others are unnecessary deviations. First of all there is no handling of the padding in the reference implementation which is promised by the paper as

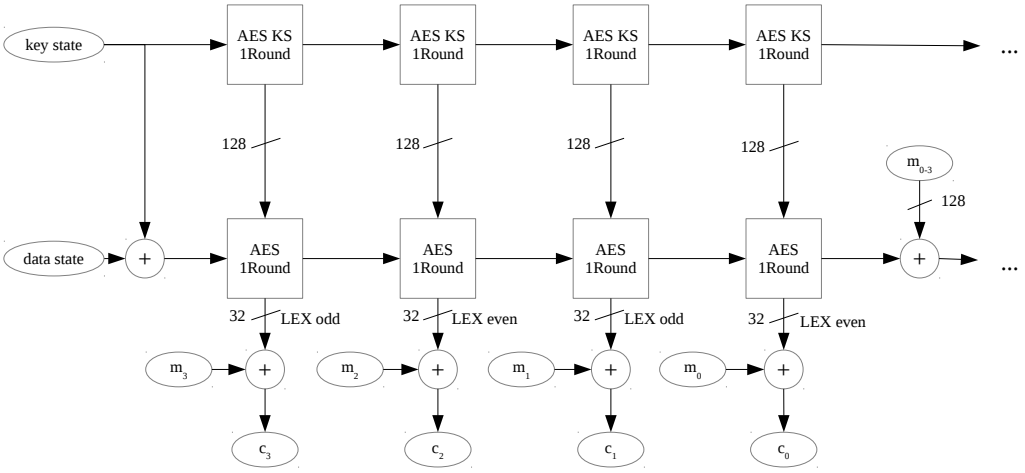

Figure 2.4: Encryption of one message block

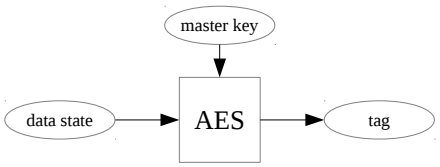

Figure 2.5: Tag-Generation of ALE

> »For the last block of M the *exact required number of most significant bits* are taken from the leak and xored to the last block (without padding) to produce the last bits of ciphertext, and m_t is xored to the data state.[1]«

The paper proposes a padding-mode similar to MD4. This includes the addition of a 1-bit followed by 0-bits and a 64-bit encoded message length encoded into the last message block. In reality the code has no other handling for the last message block so the padding is in fact xored into the *data state*. Second the paper proposes different masks for the LEX-Leak than are actually used in the implementation:

$$odd: \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, even: \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.6}$$

Another point is that the 1-round keyschedule in the Initialization stage is omitted by the paper, although it is mentioned in the beginning when comparing the algorithm to ASC-1.

> »the non-sequential order in which the AES-256 subkeys are used in ASC-1 (e.g. subkey 11 is needed already in the first round) [1]«

And at last the reverted order of the message words is omitted in the paper[1].

2.6 Summary

ALE is an authenticated encryption algorithm that is heavily based on AES. Some minor specifics of ALE seem to be not set in stone yet. While others have filled the fuzzy details on their own with mixed success [7], we filled this lack of specification with a strict adherence to the reference implementation that was provided to us. Many aspects of ALE are intriguing, especially when it comes to its low-area qualities which we will discuss further in Chapter 3.

Chapter 3

Lightweight Implementation of ALE

In order to perform a side-channel analysis of ALE, the algorithm was implemented targeting a Xilinx Virtex-II Pro FPGA platform. We describe this FPGA in more detail in Chapter 5. In this chapter we give a detailed overview of the design of this lightweight implementation of ALE and all of its components. We discuss each of the modules and the design in a top-down approach. In the first section, the global requirements are reviewed and the resulting design decisions are discussed, this is followed by the hardware architecture of ALE. There we discuss our top module which can be split into a control and a datapath module. This will give an overview of all the control signals which are wired between those two modules and their relations to the finite state machine (FSM) which we use as the control module and also discuss their purpose in the datapath.

We created an implementation of AES which is similar to Moradi et al.'s proposal in [6] but deviate from it in some aspects which we discuss in other sections of this chapter in more detail. The original proposers [1] of ALE claim that they also used an AES core based on [6]. This chapter also features more detailed discussions on other modules created for this implementation and the purposes of these modules in the following sections. These explanations also include a description of our solutions for a LEX-leak [15] and all other aspects needed to augment AES into a fully functioning version of *ALE*.

Further on we discuss the implementation of our FPGA-specific countermeasures as they were proposed by Tim Güneysu and Amir Moradi in [4]. One subsection is devoted to the random number generator (RNG) we implemented while each of the countermeasures -*clock randomization* and the randomized switching of *short circuits*- receive their own discussion regarding their implementation. Both of these countermeasures utilize the same RNG design which is discussed in Section 3.3.1.

At the end of this chapter we discuss the synthesis results and further optimizations. We also give a comparison of the results provided by [1], [6] and [13] with our own results. This chapter is then concluded by a short summary.

3.1 Global Requirements and Design Decisions

Our global requirements are first and foremost defined by a motivation towards area optimization with the goal to achieve a smaller AES core than what was achieved by [6]. Smaller area requirements lead to cheaper production costs if a design would eventually

get taped-out, a smaller area rquirement also comes with lower power-consumption requirements when compared with designs focused on the fastest possible computations. Since ALE was proposed as a tool for light-weight cryptography we also opted for an area optimization approach. The design should be able to function as an ASIC and on an FPGA and be cycle-efficient. It should be noted that the countermeasures implemented are FPGA-specific and are not suited for ASIC implementations.

To verify the correct functionality, the implementation also must be able to pass the same testvectors as the reference implementation of ALE [1] with the same ciphertext and tag output. This means that we chose to adapt the same LEX-mask as described in Chapter 2.3 and not the one seen in the figures of [1]. This necessitates an AES core capable of executing up to ten rounds of AES and the calculation of the eleventh round of the AES keyschedule.

We chose to communicate via a predefined communication standard (AMBA APB) as described in Subsection 3.1.1 and adopted a similar 8-bit architecture as [1] and [6]. We also chose to use the standard approach of seperating the design into a datapath and a controlpath.

One of our further-reaching design decisions was to actively use clock gating or enable signals for flipflops to control the flow of data. We also decided to generate Sbox and RCON results through logic gates and not through look-up tables (LUT).

3.1.1 Communication Interface

In order to set up a predefined way of communication for the implementation we decided to use an established communication standard for the input and ouput of data. We chose the *Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB)* [43] protocol by ARM due to the already existing serial-to-APB module which allows serial communication with computers and its simplicity. AMBA APB is used for low bandwith access to peripheral slaves and is part of other open-standard, on-chip interconnect specifications for system-on-a-chip (SoC) designs which were created by the ARM corporation.

We use AMBA APB with 32-bit wide bus signals for data transfers (IO). The signals we use can be seen in Figure 3.1. This figure shows the top module of our implementation with the interface signals on its outer left and outer right edges. The general purposes of these signals are as follows:

penable
: This signal indicates a data transfer via APB from the second sent or received block onward when set to high.

presetn
: This is the reset signal, generally used to reset the slave back to its starting point and resetting registers where necessary. This signal is *active LOW* which means a reset should take place if this signal equals zero.

pwrite
: *pwrite* is used to indicate the direction of a data transfer via APB from master to slave device. If set to HIGH (logical one) data is sent to the peripheral device, if LOW data can obtained *from* the device.

pwdata (32 bits wide)
This bus is used to send 32-bit wide data blocks to the peripheral device and should only be relevant if *pwrite* is HIGH.

prdata (32 bits wide)
This bus is used to read 32-bit wide data blocks from the peripheral device and should only contain relevant data if *pwrite* is LOW.

psel *(unused)*
This indicates that the peripheral device is selected. Due to the lack of other peripheral devices the need to communicate with this signal has been omitted by the control logic of our implementation.

paddr (8 bits wide) *(unused)*
This bus can be used to indicate to which address transferred data should be written. This bus was unnecessary for our implementation and was therefore left unused.

pclk
The clock signal whose rising edge times all transfers of information via the APB interface.

Taking a closer look at Figure 3.1, the control signals *penable*, *pwrite* and *presetn* are connected to the *control* module of our implementation. But also the data signal *pwdata[31]* is connected to this module. This is because we only use *pwdata[15]* - *pwdata[0]* for data transmission. We reduced our input bandwith due to the fact that it further improves our low area target and fits better into our architecture which heavily depends on 8-bit wide buses. The hence otherwise unused *pwdata[31]* signal is repurposed to indicate *if the last message block has been transmitted*. That means, the *pwdata[31]* signal serves as an additional interface *control* signal. This enables the implementation to know when to switch from ciphertext generation to tag generation. The decoding of information encoded in the message padding would be another albeit more expensive solution to detect this.

Also note that in order to be able to use this interface on an FPGA for a side-channel analysis, a pre-existing serial-to-APB interface in the FPGA bitstream was used as a intermediate module.

As was previously mentioned, the serial-to-APB module enables communication with an external controller (e.g. a PC running on the Windows operating system) which sends and receives its information one bit at a time. This reduces the input and output signals to one signal for each purpose (rx for receiving, tx for transmitting). The serial-to-APB module itself is available in VHDL. For the communication to work properly, information about the operating frequency and the baudrate[1] need to be hardcoded into the HDL files.

3.2 Hardware Design of ALE

As is common in modern hardware designs we divided our design into a datapath and a controlpath. The *datapath* module stores, processes and outputs the supplied data and the *control* module controls the flow of the data in the datapath. Figure 3.1 shows the *top* module containing a *datapath* module and a *control* module. The *control* module uses

[1]The baudrate is defined as symbols per second

multiple control signals to direct the *datapath* module. According to the signals set by the *control* module, the data is modified in the datapath.

In the following, the purpose of each of the control signals depicted in Figure 3.1 is discussed:

state_select, key_select, temp_select and zero
: These signals determine which input will be shifted into the respective registers of the *state*, *key*, and *temp* modules. The *zero* signal controls a multiplexer regarding the input of the *state* register. This is needed in the initialization stage of ALE to fill the state register with zeroes.

add_select
: According to the value of this signal, the values of one of the output buses of either the *key* or the *temp* module is added to the output bus (state[127:120]) of the state register. The output of the multiplexer will be sent to an 8-bit XOR gate whose other input is an output bus of the *state* module.

state_en, key_en, tmp_en and lex_en
: These *enable* signals determine if the respective registers should update their values at the next positive edge of the clock (HIGH) or hold their current values disregarding the clock signal (LOW).

mx and shiftrows
: These signals are all used in the state module. The *mx* signal determines if the output of the mixcolumns module should be the input of the last column of the register while the *shiftrows* signal enables the ShiftRows transformation of AES as it was discussed in Section 2.1.

sel_and and sub_word
: These signals are used to control the keyschedule of AES.

loadpt
: This signal is used to store message blocks in the temp register.

even and shift
: In the LEX-leak module these signals control the flow of information. *even* denotes whether the current round of AES is odd or even while the signal named *shift* enables the registers of the *LEX-leak* module to perform a regular shift to the left. In an alternative implementation of the *LEX-leak* module the *shift* signal is replaced by a signal called *round1* which denotes whether the module can skip the preprocessing of the keyschedule (see Section 3.2.1 and Figure 3.9).

reset_rcon, step_rc, active_rc
: These signals are used to control the state and the output bus of the RCON module. A *step_rc* set to HIGH will set the internal value of the RCON module one step further. If the RCON module is set to inactive (*active_rc* set to LOW) it will set the value on its output bus to zero.

tag_out
: The *tag_out* signal sets the *prdata* bus to the result of the tag generation. Otherwise it outputs the results of the LEX-leak.

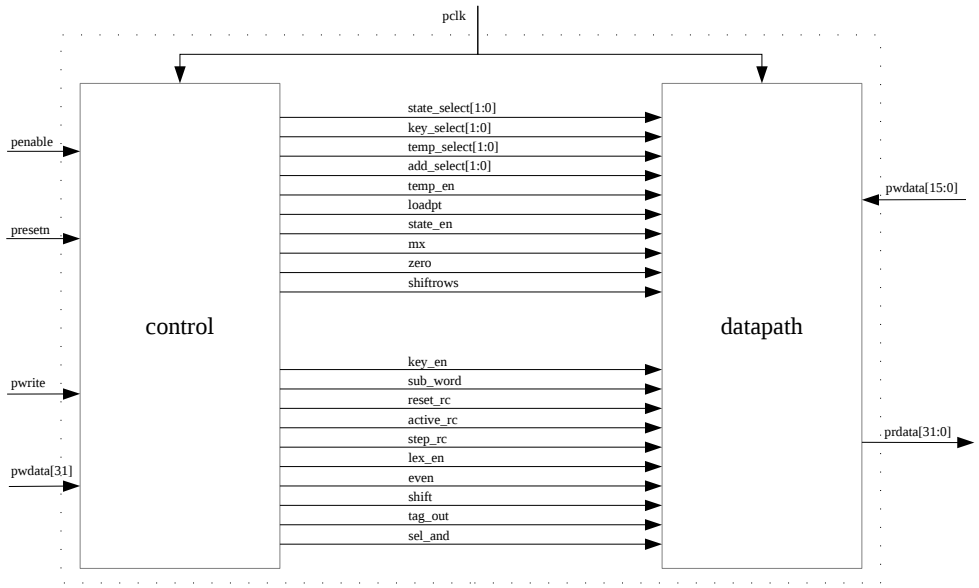

Figure 3.1: The *top* module with input, output and control signals

How the signals described above change the behavior of the datapath is discussed in more detail in Section 3.2.1. For further information on how they are set see Section 3.2.2.

Figure 3.1 also shows how the signals of the communication interface are routed and reveal some of the specifics of the communication between the interface master and our *ALE* implementation. These interface signals have been discussed in Section 3.1.1.

This section has featured information on the control signals used in the *top* module of our ALE implementation. The following subsections discuss the *datapath* module with its submodules and the *control* module, respectivily.

3.2.1 Datapath Module of ALE

The *datapath* module can be described as a modified version of the datapath proposed by [6] and can be seen in Figure 3.2. Besides the one module containing Canright's Sbox [14] it holds four specialized register modules: *state*, *key*, *temp* and *LEX-leak*. The state, key and temp modules include 16 registers each containing one byte (for an example see Figure 3.3) while the *LEX-leak* module holds four byte registers in order to be able to hold one word (32 bits) of the current ciphertext block (see Figures 3.9 and 3.10).

The *datapath* module uses 8-bit buses to transfer data almost exclusively with a few notable exceptions concerning the LEX-leak module and the output of data. We can also see this 8-bit structure at the core of the design where the *datapath* module has the ability to add two bytes of data and send the result through the Sbox in one clock cycle.

The interactions between the five modules contained in the *datapath* module is controlled by several multiplexers, which are discussed in the following paragraphs. All the multiplexers are controlled by signals originating from the *control* module. All of our explanations can be related to Figure 3.2.

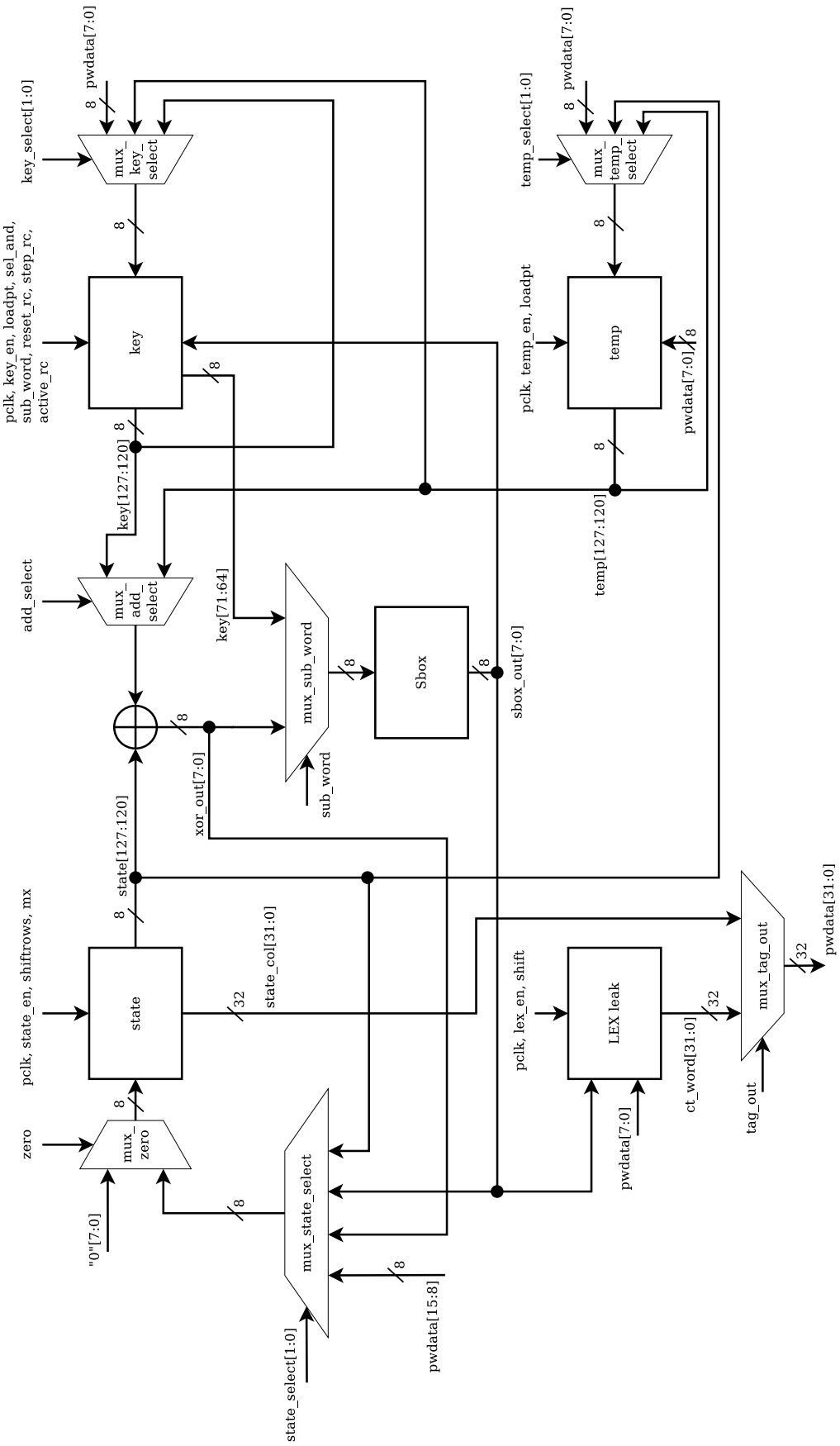

Figure 3.2: The *datapath* module

mux_state_select
The 8-bit multiplexer denoted in Figure 3.2 as *mux_state_select* allows to select between four inputs. The first (from the left) input is used to store data received from the AMBA APB interface. The second is reserved for the last operation of a single message block encryption in which the message block is added into the *state* register. The third input takes the output of the Sbox as is needed for the AES encryption. And finally the fourth input is used during the *initialization* stage (see Section 2.2 about this particular stage of *ALE*) where the result of the first AES encryption has to be stored temporarily in the *temp* register.

mux_key_select
The 8-bit multiplexer called *mux_key_select* has three inputs for the *key* module. The first (from the top) input is used to store data received from the AMBA APB interface. The second input takes the output of the *temp* module to load the master key or the initialized *key* from the *temp* module. The third input is used to rotate the key bytes during the key addition of the AES encryption.

mux_temp_select
The *mux_temp_select* 8-bit multiplexer provides three inputs for the *temp* module. The first input is used to store data received from the AMBA APB interface. The second input is used during the *initialization* stage of *ALE* where the result of the first AES encryption has to be stored temporarily in the *temp* register. The third input is used to rotate the bytes of the *temp* module in order to preserve the initialized *key* for the additional keyschedule (see Section 2.2).

mux_add_select
The *add_select* signal selects the byte that will be added to the current byte of the state. This enables the design to add bytes of the *temp* and *key* modules to bytes of the *state* module (whose signal is called *state[127:120]* the input signals of the multiplexer are called *key[127:120]* and *temp[127:120]* respectively). ALE necessitates this when a plaintext block which is stored in the *temp* register has to be added into the *state* module.

mux_sub_word
In order to enable an AES keyschedule it is required to substitute bytes of the round key using the *Sbox* module. This is provided by the multiplexer switched by *sub_word*. The *sub_word* signal is also used in the *key* module to change the inputs of its registers.

mux_zero
Another requirement of ALE is the loading of zero bytes into the *state* module during the *initialization* stage (see Section 2.2) for the second AES encrytption. This is achieved by applying an additional multiplexer.

mux_tag_out
The 32-bit multiplexer called *mux_tag_out* enables us to switch between the ciphertext words generated in the *LEX-leak* module and 32-bit words of the authentication tag which we process in the final AES encryption of ALE.

Some other points are of note in the *datapath* module. The *temp* and *LEX-leak* modules have another pwdata[7:0] input bus which is used to store or process plaintext inside of

them. Figure 3.2 omits input buses used by the less compact *LEX-leak* module which are selected byte registers of the *state* and *key* modules to comply with the given LEX-mask and a bus coming from the *temp* module which supplies a complete plaintext word.

In the following subsections the *state*, *key*, *temp*, *Sbox* and *LEX-leak* modules are discussed in a sequential order.

The State Module

The *state* module consists of sixteen registers (each 8-bit wide) which can be used as a simple shift register and is shown in Figure 3.3. The registers in Figure 3.3 are assembled to resemble the same four-by-four pattern the internal states of AES hold as can be seen for the matrices M and K in Equation (2.1). The same matrix pattern is used for the registers of the *key* and *temp* modules. This enables the module to load data from multiple sources as was dicussed above in the general *datapath* section.

In order for the registers to be clocked, they have to be enabled by the *state_en* signal first. This additional complexity in flipflop-design was introduced along with the use of scan flipflops to gain better control over the flow of data. The scan flipflops help to keep the area requirements low as they are significantly smaller than additional multiplexers.

The *shiftrows* signal is used to facilitate the ShiftRows step of AES (see Section 2.1) which selects the inputs of the registers as is indicated by the lower set of inputs of the multiplexers labeled with *'scan'* in Figure 3.3. The *mx* signal cuts the shift chain from the last byte to the byte into four self-contained shift loops along the rows of the registers to direct all four columns through the *mixcolumns* module.

The *state* module also contains a submodule called *mixcolumns* which transforms the first column of registers into the result of AES' MixColumns step of AES. The design of this module can be seen in Figure 3.5 and was taken from Hua Li and Zac Friggstad's [13] proposal. This is a derivation of the design proposed by [6] which uses a module which is designed for the first row of the MixColumns matrix multiplication as it was depicted in Equation (2.4). The module is instantiated four times for each of the four result bytes of MixColumns. The wiring of the input bytes is rotated according to multiplication matrix of Equation (2.4).

Our approach holds the advantage of smaller area requirements due to the reuse of the results of the *x2* modules which can also be observed in Figure 3.5. An *x2* module computes a simple duplication of a value inside of $GF(2^8)$ which corresponds to a shift operation to the left and a reduction modulo $x^8 + x^4 + x^3 + x + 1$ which translates to an xor operation with 100011011_2 if the most-significant bit (MSB) is set. This is the same function as was described in Equation (2.3). So these *x2* modules contain a simple bit-wise shift wiring to the left and three 2-input XORs of the MSB with the fifth, sixth and eighth significant bits. The MSB is also rotated to the place of the least significant bit (LSB) as can be seen in Figure 3.4.

Another advantage regarding area requirements is the omission of a 64-to-32-bit multiplexer which we do not need due to the further utilization of our *state_en* signal. This multiplexer was used in the design of [6] to circumvent the MixColumns-step of AES for the tenth round of AES while our design just disables the clocking of the registers used in the *state* module.

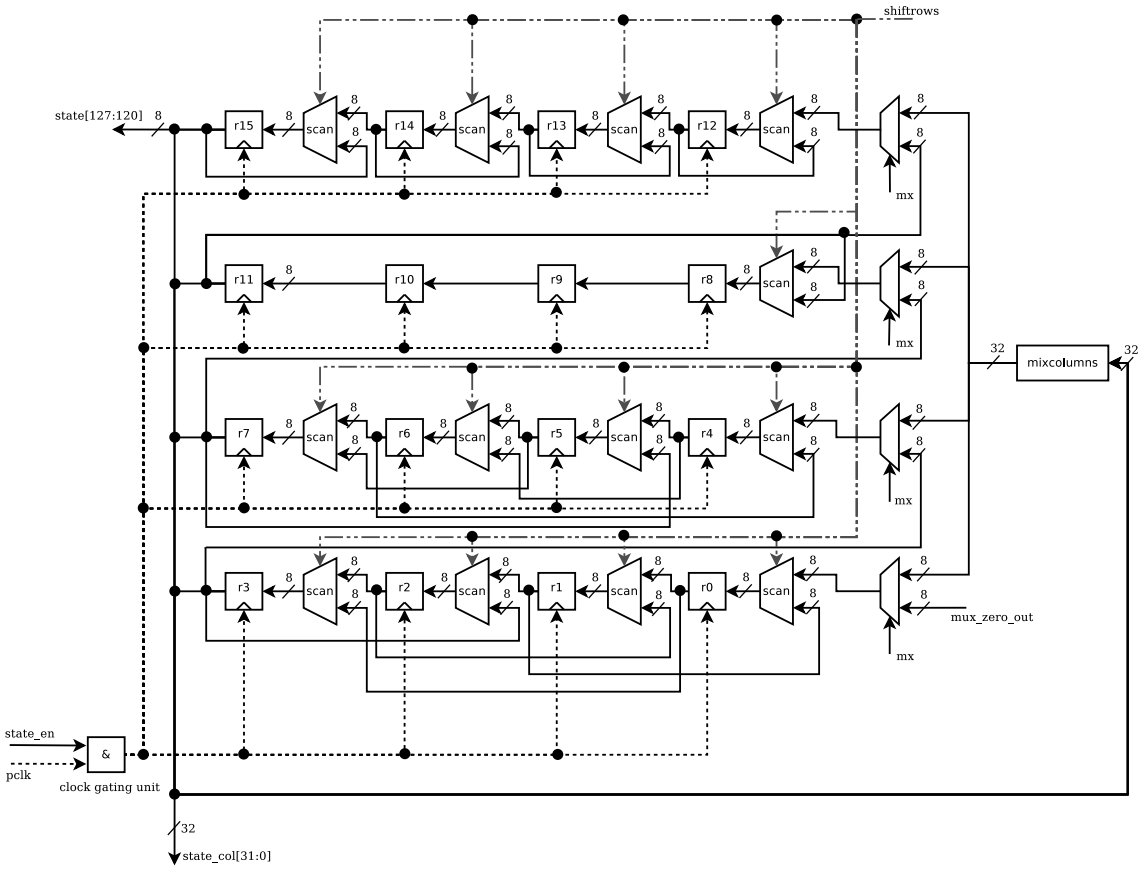

Figure 3.3: The *state* module

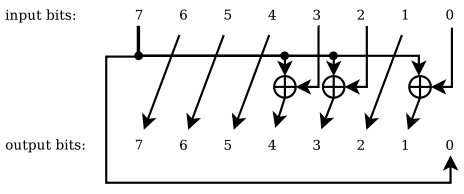

Figure 3.4: The *x2* module only consists of wires and three XOR gates

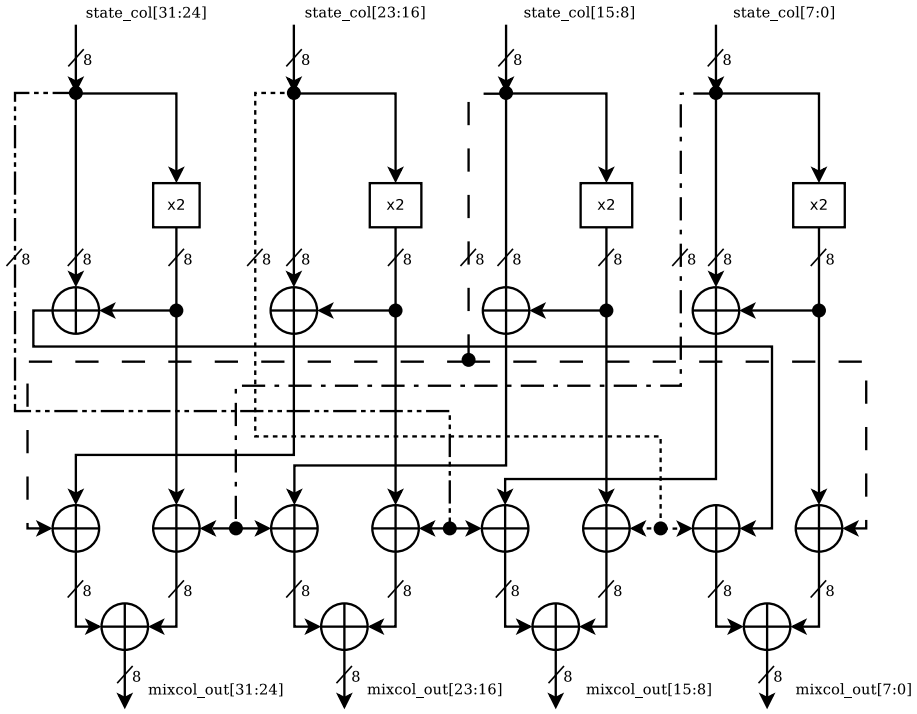

Figure 3.5: The *mixcolumns* module

The Key Module

The *key* module is designed similarly to what we established while discussing the *state* module. It is comprised of 16 byte registers which are clocked when the *key_en* signal is set to HIGH and utilizes scan flipflops to save some area.

A 2-to-1 multiplexer costs 2.33 gate equivalents (GE) and a positive edge triggered D-flipflop requires 5 GE to be able to store one bit. Scan flipflops on the other hand require 6 GE while combining the functionality of the multiplexer and the flipflop. This results in 1.33 GE saved per bit of storage that would otherwise require a multiplexer on its input. This approach to reduce area requirements has been used before in area efficient implementations of PRESENT [47], KATAN/KTANTAN [48] and AES [6].

The *key* module can be used as a simple rows-first shift register but also holds features designed to facilitate the processing of the AES *keyschedule*. As depicted in Figure 3.6 the registers also support a columns-wise shift operation when the *sub_word* signal is set to HIGH. This is used to shift the rightmost column of the registers through the Sbox module and XOR the results with the current RCON values and the values held in the leftmost register column of the module. The results of this operation are then shifted into the registers of first column. Note that the output to the Sbox is set to output the *second* byte of the last column first. This allows for an easy way to rotate this column as is needed to process the AES *keyschedule*. The *sub_word* signal is also used in the *datapath* module above to set the Sbox input to the *key[71:64]* signal of the *key* module.

The *RCON* submodule is used to output the correct *RCON* value and is controlled by three control signals. The *reset_rc* signal resets the internal 8-bit register of the module to its initial value of *00000001* if set to HIGH. The *active_rc* signal sets the output of the *RCON* module to the current value if set to HIGH and otherwise to *00000000* as only the

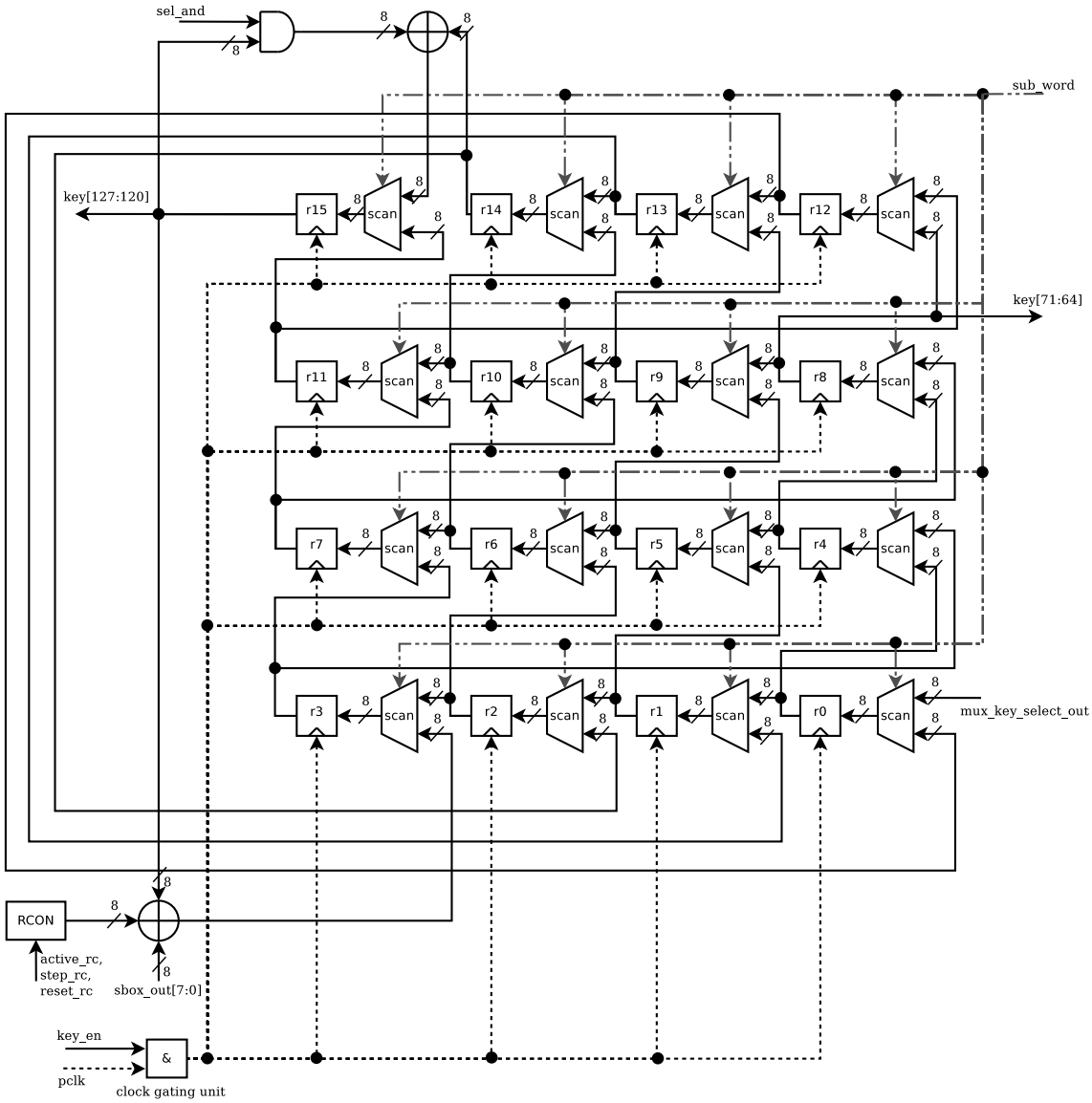

Figure 3.6: The *key* module

first of the four bytes of the column must be added with the RCON. The *step_rc* sets the internal register of the *RCON* module to the next value at the next positive clock edge. The next value is calculated inside of an *x2* module which we discussed while taking a closer look at our *mixcolumns* module and can be seen in Figure 3.4.

The remaining three columns are applied to the keyschedule while the module is set to function as a shift register and the *sub_word* signal is LOW by utilizing the *sel_and* signal. If the *sel_and* signal is set to HIGH the values of the current first and second register in the first row will be XORed and stored in the first register. This signal should only be set to HIGH when storing *keystate* values from the second to fourth columns in the first register of the first row to ensure a correct *keyschedule*.

The Temp Module

The *temp* module is specific to our implementation of *ALE* and was not proposed by [6] or [1]. It is used as a general storing tool enabling us to reduce the amount of communication overhead needed for the ALE encryption. It is used to store three types of data.

First, it is used to store the *masterkey* during the initialization. The goal is no to have to transmit it twice in that stage.

Second, it is used to store the result of the first AES encryption during the initialization stage. Otherwise this result must be transmitted outside of the implementation and reloaded twice afterwards as was done in the implementation by [1]. The first reloading would take place for the third AES encrytption of the initialization stage while the second reloading would be necessary to reset the registers of the *key* module and process the single *keyschedule* of the initialization stage.

Third, it is utilized to store the plaintext blocks used for the ALE encryption. Without this use of the *temp* module it would be necessary to reload all of the plaintext blocks through the global interface to add their values into the registers of the *state*.

As we have discussed with the *state* and *key* modules the *temp* module holds 16 byte registers which can be enabled to store values at their inputs when the *temp_en* signal is set to HIGH and utilizes scan flipflops to avoid the use of multiplexers. As is the case with the previously discussed main modules it supports a rows-first shift mode of operation while the *load_pt* signal is LOW. The overall design scheme of this module can be seen in Figure 3.7. This functionality of a shift register is utilized when storing the masterkey, storing intermediate results of the initialization stage from or to the module and when the current plaintext block needs to be added to the values of the *state* module.

If the *load_pt* signal is set to HIGH the module is set to a columns-first shift mode. During this mode of operation the data is retrieved from another input which is connected directly to the communication interface (*pwdata[7:0]*). This allows to load plaintext blocks without the need to transpose them for the otherwise rows-first design.

Another solution to load the message would be to transpose the message blocks before sending them. This would have the effect of added clock cycles for the encryption since the ciphertext words could no longer be created simultaneously to loading the respective words and would lead to other additions in the design like additional FSM states, another shifting mode for the *temp* module, etc.

This also enables us to transmit the words of a plaintext block in reverse order as is necessitated by the LEX-leak while filling the registers of the *temp* module in the correct order so they can later be added to the values stored inside of the *state* module.

The Sbox Module

In search of the currently smallest design of the AES Sbox we have chosen the proposal of Canright who investigated the possibilities of hardware requirements for the AES Sbox in [14]. As many similar proposals suggest he chose to split up the $GF(2^8)$ into nested fields $GF(2^4)$ and $GF(2^2)$ in order minimize the area requirements of the Sbox. The mathematical specifics can be found in [14] while a graphical representation of the Sbox was created for [6]. Our implementation is directly based on the optimized verilog source-code provided by [14] with a few alterations regarding the compatibility of Canright's code.

An overview of the model can be seen in Figure 3.8. All submodules use combinational logic to calculate the result. After the input byte has been transformed from $GF(2^8)$ to

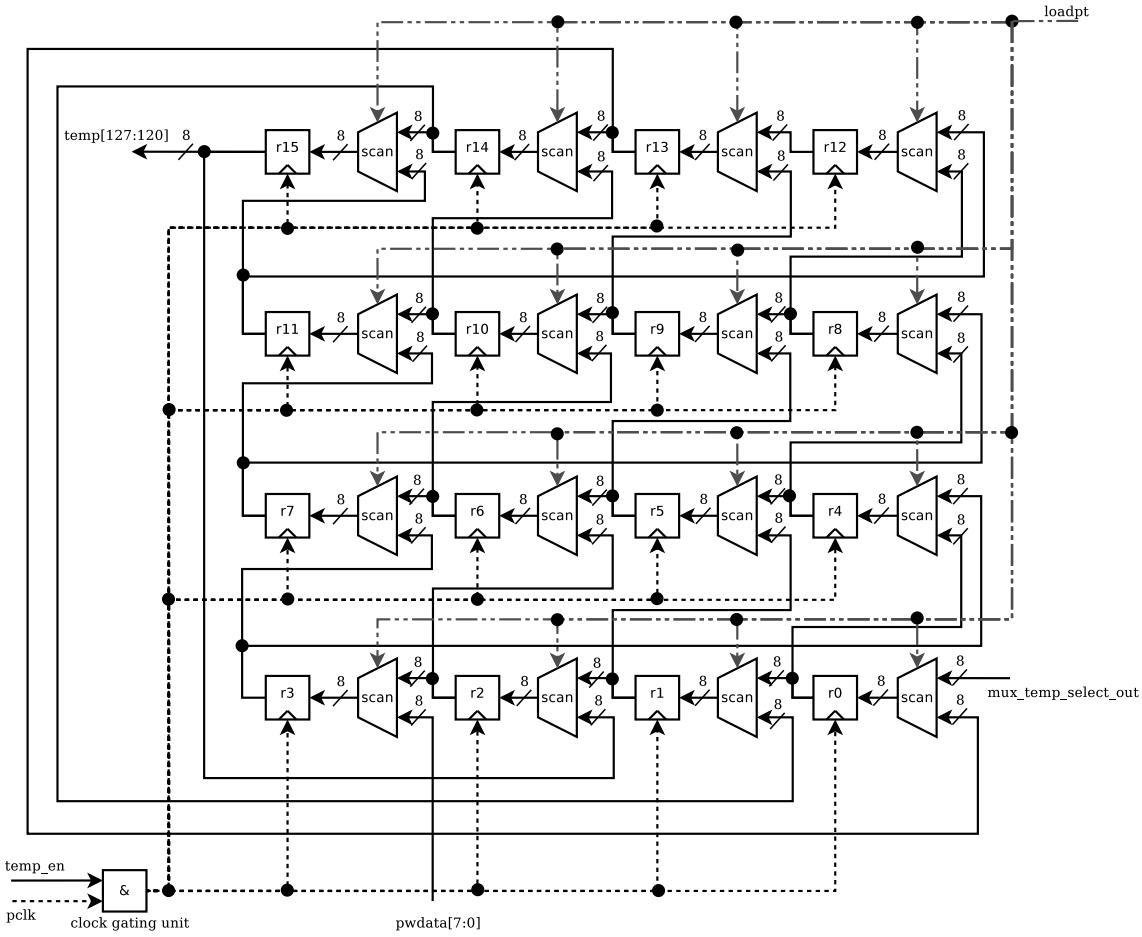

Figure 3.7: The *temp* module

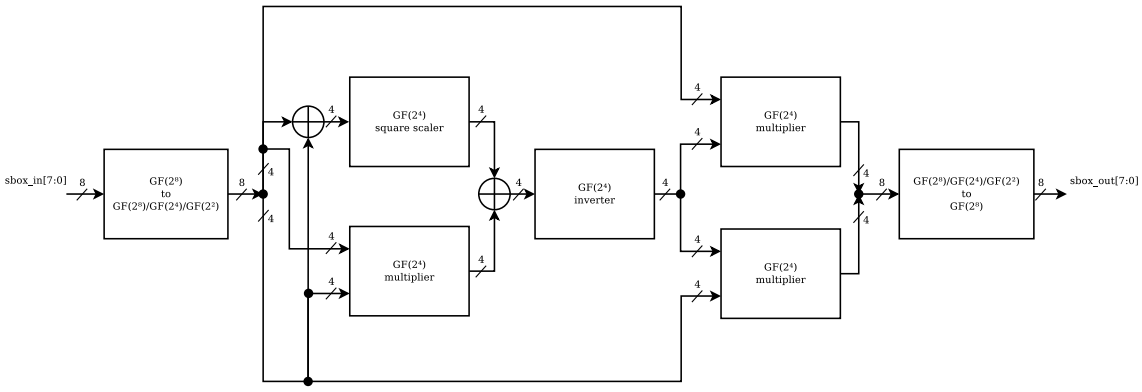

Figure 3.8: Overview of the Sbox module

$GF(2^8)/GF(2^4)/GF(2^2)$ which includes about 14 xor gates and additional logic inverers, the byte is split up to perform multiplications in $GF(2^4)$ which include calculations using submodules in $GF(2^2)$, these submodules are not displayed in the figure for the sake of space. After these multiplications the byte is reassembled (right-most side of Figure 3.8). and converted back to $GF(2^8)$.

The LEX-Leak Module

Two different approaches for implementing a module for the LEX-leak function are presented in the following. While the first design is a straight-forward solution and serves as a reference, the second design includes several improvements to decrease the area requirement of the module. The second design also reduces the amount of FSM states needed and also the number of clock cycles it takes to create the ciphertext is smaller.

Here we also discuss the challenges the LEX-leak imposes on us. Assuming we leave the AES core unaltered, it seems quite expensive to select the correct 4 bytes out of 16 during every round. One can either route the results of the SubBytes step of AES directly from the *state* module after their calculation is complete or pick them out of the shifting cycle when the respective bytes needed for the LEX mask are created.

Additional problems arise from the LEX mask itself. Because of the alternative version used by the reference implementation not only have the selected bytes to be changed for odd and even AES rounds but the correct transposition from the rows-first architecture into a columns-first ciphertext must also be altered accordingly. And another fact to consider is that the key schedule of the first round is omitted, resulting in altered requirements regarding the keyschedule. It should be noted that the way in which we have to generate our ciphertext requires a reverse order of 32-bit plaintext words of one 128-bit plaintext block. Our first decision was to keep the modifications of the communication through the interface as small as possible. We transmit the plaintext words in a reversed order but apart from this no alterations regarding the order of the bytes inside the words have to be made.

1. The first implementation of the LEX-leak module:

   The decision mentioned in the last paragraph gave way to our first design seen in Figure 3.9. This design circumvents the overall 8-bit wide dataflow for a one-cycle generation of a complete ciphertext word. To achieve this the plaintext word is loaded into the *temp* module in parallel to the steps of the mixcolumns and subword steps taking place inside of the *state* and *key* modules respectively. After this the ciphertext word is generated. All the bytes of the LEX mask and the ciphertext word are directly wired into the *LEX-leak* module and can be switched in-between in relation to odd and even rounds. This is rather straightforward for the bytes wired in from the *state* and *temp* modules as can be seen in the bottom half of Figure 3.9. What makes this approach so unattractive is how the bytes from the *key* module have to preprocess the last steps of the keyschedule depending on how far to the right (columns-wise) they are according to the LEX-mask as can be seen in the top area of the figure. This preprocessing is also unnecessary in the first AES round due to the omission of the keyschedule as we discussed above leading to an additional layer of multiplexers and a control signal called *round1*.

   As announced earlier, this approach is very raw and leaves a lot of room to area optimization, one could for example perform a 16-cycle loop of the *key* module and

perform the keyschedule then and omit the need for the XOR gates which can be seen in the top half of Figure 3.9 at the cost of more cycles and a little bigger *control* module.

2. The second implementation of the LEX-leak module:

   In our second design however we chose to stay as close to the given AES core and its 8-bit wide bus structure as possible. The goal of this design was to pick the results of the AddKey step (the *sbox_out[7:0] signal in the datapath*) according to the LEX-mask and storing them in the *LEX-leak* module. Then we can add the bytes of the plaintext word (from the pwdata[7:0] signal) as we load them while using the *LEX-leak* module as a shift register with the *shift* control signal set to HIGH. This design can be seen in Figure 3.10. A few things are of note here. First, the *lex_en* signal has to be HIGH on very specific cycles according to the LEX-mask. Second, we have to take care of the order in which we want the bytes to be stored inside of the *LEX-leak* module. This is where the remaining multiplexers come into play creating two additional shift patterns for the module. During odd rounds the bytes originating from *sbox_out[7:0]* are shifted through the fourth, third, first and then second register, while they are shifted through the second, fourth, third and then first register during even rounds. This results in a correct byte order when adding the plaintext words. Otherwise we would either have to send specially arranged plaintext words and rearrange them in a correct order inside of the *temp* module or we would have to create our own state inside of the FSM of the *control* module with still almost the same area requirements in the *LEX-leak* module.

   Compared to our first design this module requires 35 percent of the area while being one cycle faster (the first design requires its own encryption cycle). Other optimizations would eventually require additional preconditioning of the plaintext blocks.

3.2.2 The Control Module of ALE

In this section we take a look at the FSM of our implementation. FSMs can be used to design and describe sequential logic circuits and have a number of states. When used as control logic, a FSM has the purpose to set and unset its output signals to control the datapath.

The FSM featured in our implementation can be seen in Figure 3.11 where its states are grouped into the three stages of ALE (initialization, encrytption, tag generation). Besides its main state, the FSM uses the following internal attributes to determine its next state:

AES_ESCAPE

This type determines the next state after a ten round AES encryption has finished. The possible values are: IV, ZERO, FINAL, TAG.

If set to IV, the value indicates that the AES encryption of the initialization vector has been processed and that the relocation of the encryption result into the *temp* module should be done next.

If set to ZERO, the value indicates that the AES encryption of the zero vector during the initialization stage has been processed and that the values held by the *temp* module should be copied into the registers of the *key* module.

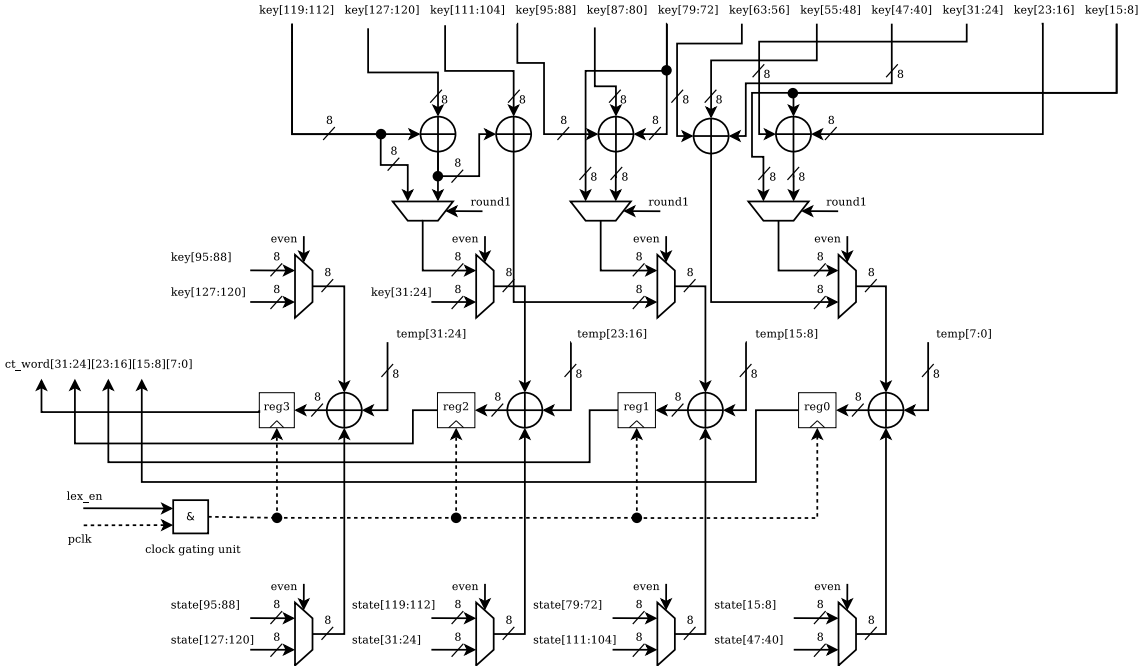

Figure 3.9: The first implementation of the *LEX-leak* module

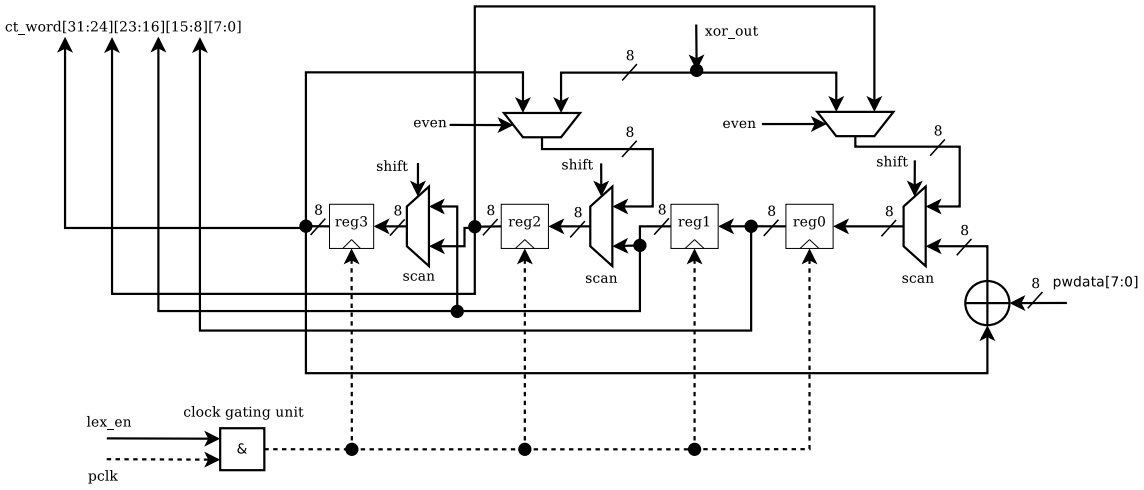

Figure 3.10: The second, more compact implementation of the *LEX-leak* module

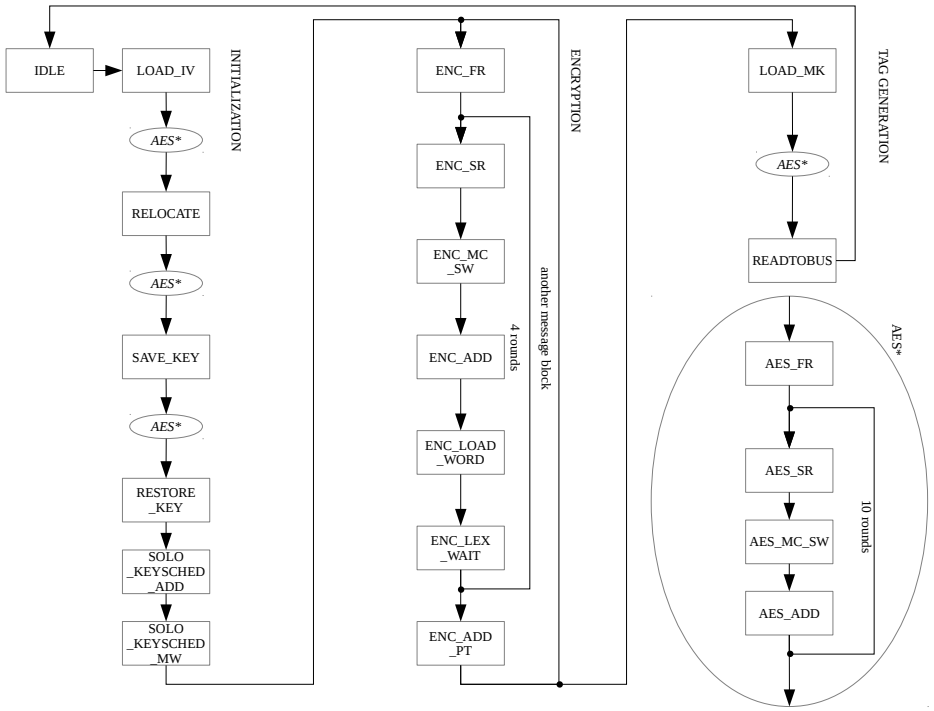

Figure 3.11: The states of the *FSM* which are setting the control signals

If set to FINAL, the value indicates that the last AES encryption of the initialization stage has been processed and that the values held by the *temp* module should be copied again into the registers of the *key* module.

If set to TAG, the value indicates that the AES encryption which creates the authentication tag has been processed and that its result should now be read out to the AMBA APB interface.

aes_counter
This counter is used to determine the progress of the current AES phase during AES_FR, AES_ADD, ENC_FR add ENC_ADD where it counts up to 15. During AES_MC_SW and ENC_MC_SW it counts up to 3.

lex_counter
This counter is used to determine the current AES round. This counter counts up to ten during the initialization and tag-generation stages and to four while encrypting a single plaintext block.

last_block
This internal register is used to decide if the encryption stage is complete.

Figure 3.11 shows all the states of the FSM. The states required for a complete AES encryption are used four times for every complete ALE encryption. The states can be described as follows:

IDLE
This state is meant as a starting point, the FSM will set itself to this state if reset.

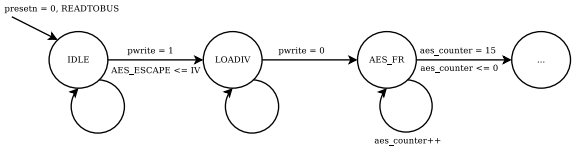

Figure 3.12: State transitions at the beginning of the FSM.

In this state both counters are set to 0, the RCON is set to 1 and the AES_ESCAPE is set to IV. The FSM will leave this state if *pwrite* is set to 1.

LOAD_IV

In this state the IV and key bytes are retrieved via the interface. The key bytes are stored twice: one copy is stored in the *key* module while the other is stored in the *temp* module If *penable* is 1 the clocking of the *state*, *key* and *temp* modules are enabled, otherwise they are not. This state will be left and AES_FR will be entered if *pwrite* returns to 0.

AES_FR

During this state the first round of AES begins. The key and state bytes are added and submitted to the Sbox from which the resulting bytes are stored in the *state* module. The input of the *state* module is set to the output of the Sbox while the *key* module is set to a loop. Also the value inside of the *RCON* module is always reset to *1* in this state with the *reset_rc* signal set to HIGH. After 16 clock cycles counted by the *aes_counter* signal the state is set to AES_SR.

AES_SR

SR is an acronym for *shiftrows*. The registers of the *key* module are disabled. The lex_counter is increased by one and the value inside the RCON module is set to the next one if it is no longer the first round. The next state (AES_MC_SW) is reached immediatly.

AES_MC_SW

This state is set to last four clock cycles in which the *mixcolumns* operations take place. Meanwhile the first column of the new round key is calculated. Both key and state registers are enabled. The *mx* and *sub_word* signals are set to HIGH. The

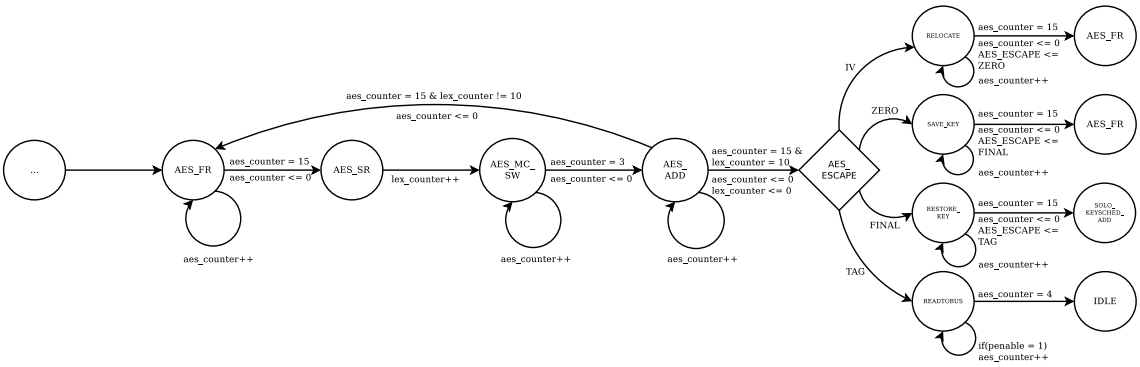

Figure 3.13: State transitions of the AES states.

active_rc signal is set to HIGH only during the first clock cycle. The next state (AES_ADD) is reached after the fourth clock cycle.

AES_ADD

The AES_ADD state is very similar to the AES_FR state. Until the *lex_counter* signal reaches a value of 10 indicating that the final round of AES was reached the state behaves as follows. The key and state bytes are added and submitted to the Sbox from which the resulting bytes are stored in the *state* module. The input of the *state* module is set to the output of the Sbox while the *key* module is set to a loop. After 16 clock cycles counted by the *aes_counter* signal the state is set to AES_SR. Additionally it sets the *key_and* signal to HIGH if the *aes_counter* signal currently is *not* set to the value of 3, 7, 11 or 15.

When the *lex_counter* signal has reached the value of 10 the input of the *state* module is set to the output of the the 8-bit XOR in the *datapath* module (Figure 3.2. After 16 clock cycles counted by the *aes_counter* signal the state is set to RELOCATE, SAVE_KEY, RESTORE_KEY or READTOBUS depending on the current value of the *AES_ESCAPE* variable.

RELOCATE

This state takes 16 clock cycles of time. It is used to load zero values into the *state* module. Also the result of the previously calculated AES encryption is relocated from the *state* module to the *temp* module as is the masterkey which is currently held in the *temp* module to the *key* module. All these operations take place concurrently with the *zero* signal set to HIGH. The respective *select* signals of the *key* and *temp* modules are set to the corresponding values.

SAVE_KEY

The SAVE_KEY state is very similar to the RELOCATE state. It takes 16 clock-cycles to complete between the second and third AES encryptions as is also shown in Figure 3.11. This state allows the design to *copy* the values in the *temp* module to the *key* module while the values held in the *state* module must remain unaltered. All of these three modules are enabled in this state, with the inputs of the *state* and *temp* modules set to loop their contents while the input of the *key* module is set to receive data from the *temp* module.

RESTORE_KEY

This state is used to transfer 16 bytes from the *temp* module to the *key* module. The only significant difference to the SAVE_KEY state is that it's next state is called SOLO_KEYSCHED_SW.

Also in the last cycle the *step_rc* signal is set to high to set the value inside of the *RCON* module to the value corresponding to the eleventh round of AES.

SOLO_KEYSCHED_SW

This state facilitates the same steps of the AES keyschedule as AES_MC_SW but disables the *state* module during its duration which is four clock cycles. The *sub_word* signal is set to HIGH. The *active_rc* signal is set to HIGH only during the first clock cycle. Note that the value inside of the *RCON* module is currently set to the value of the RCON in eleventh round of AES. The next state (AES_ADD) is reached after the fourth clock cycle.

SOLO_KEYSCHED_ADD

This state facilitates the same steps of the AES keyschedule as AES_ADD but disables the *state* module during its duration which is four clock cycles. It sets the *key_and* signal to HIGH if the *aes_counter* signal currently is *not* set to the value of 3, 7, 11 or 15.

ENC_FR

During this state the first round of AES begins. The key and state bytes are added and submitted to the Sbox from which the resulting bytes are stored in the *state* module. The input of the *state* module is set to the output of the Sbox while the *key* module is set to a loop. Also the value inside of the *RCON* module is always reset to *1* in this state with the *reset_rc* signal set to HIGH. After 16 clock cycles counted by the *aes_counter* signal the state is set to ENC_SR.

ENC_SR

SR is an acronym for *shiftrows*. The registers of the *key* module are disabled. The lex_counter is increased by one and the value inside the RCON module is set to the next one if it is no longer the *second* round of AES. The next state (ENC_MC_SW_LW) is reached immediatly.

ENC_MC_SW

When compared to other states this state is very similar to the AES_MC_SW state that it also sets the *even* control signal according to whether the current round of AES is odd or even. The *state* module is enabled. The *key* module is only enabled if the *lex_counter* has the values 2, 3 or 4 since the keyschedule is omitted in the first round. This applies also to the *active_rc* signal which is otherwise set to HIGH when the first byte is loaded. The control signals *mx* and *sub_word* are set to high to facilitate the *mixcolumns* step and steps of the keyschedule when the *state* and *key* modules are clocked.

This state will be left and ENC_ADD will be entered after four cycles.

ENC_ADD

During this state the remaining *addkey* and Sbox permutation steps of AES are processed. The key and state bytes are added and submitted to the Sbox from which the resulting bytes are stored in the *state* module. The input of the *state* module is set to the output of the Sbox while the *key* module is set to a loop. Additionally it sets the *key_and* signal to HIGH if the *aes_counter* signal currently is *not* set to the value of 3, 7, 11 or 15 and the *lex_counter* variable holds values above *1*. It sets the *even* control signal according to whether the current round of AES is odd or even. Also, while in an odd round it sets the *lex_en* signal to HIGH if the *aes_counter* holds the values 1,4,6 or 14. If the round is even the *lex_en* signal is set to HIGH if the *aes_counter* holds the values 0,2,10 or 12

After 16 clock cycles counted by the *aes_counter* signal the state is set to ENC_LOAD_WORD.

ENC_LOAD_WORD

This state is very similar to the LOAD_IV state. The *load_pt* and *shift* signals are set to HIGH, and both the *LEX-leak* and *temp* modules are enabled if both *penable* and *pwrite* are set to HIGH. Both *state* and *key* modules stay disabled during this state.

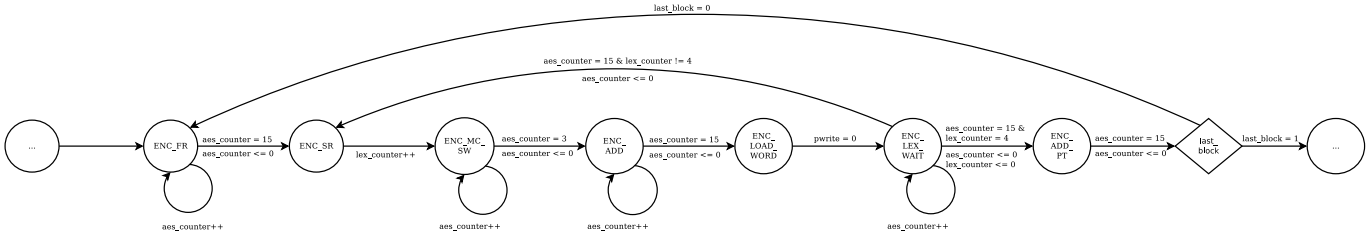

Figure 3.14: State transitions of the encryption states.

After 4 clock cycles counted by the *aes_counter* signal the state is set to ENC_LEX_WAIT.

ENC_LEX_WAIT

In this state our design waits for the AMBA master to retrieve the calculated ciphertext word. The clocking of the *state*, *key*, *temp* and *LEX-leak* modules is disabled during the duration of this state. If the *penable* signal is HIGH the state ENC_ADD_PT or the state ENC_SR is entered depending on whether the *lex_counter* has a value equal to 4 which suggests that a whole plaintext block has been encrypted.

ENC_ADD_PT

This state is used to add the values of the *temp* module which holds the values of the current plaintext block into the values of the *state* module. The clocking of the *state* and *temp* modules are enabled in this state while the clocking of the *key* module is turned off. The input of the *state* module is set to take the output of the 8-bit XOR in the *datapath* module, while *temp* module is set to loop on itself. Additionally the *add_select* signal is set to let the signals from the *temp* module through the multiplexer.

After 16 clock cycles the state determines whether to switch to ENC_FR or LOAD_MK if the last_block register is set.

LOAD_MK

This state is similar to LOAD_IV but only key bytes are retrieved via the interface. The key bytes are stored in the *key* module. If *penable* is 1 the clocking of the *key* module is enabled, otherwise it is not. This state will be left and AES_FR will be entered if *pwrite* returns to 0.

READTOBUS

After the last ten-round AES encryption to generate the authentication tag. The tag is read out from the *state* module to the prdata bus. This is facilitated by setting the *tag_out* and *mx* signals to HIGH. Depending on the *penable* signal which also controls the clocking of the *state* module, the state is left after each of the four words from the *state* module has been read out to the AMBA master and is set back to the IDLE state.

3.3 Countermeasures

After our first successful implementation attempt of *ALE* we went on to implement two FPGA-specific countermeasures to reduce the feasability of DPA attacks on the design. We

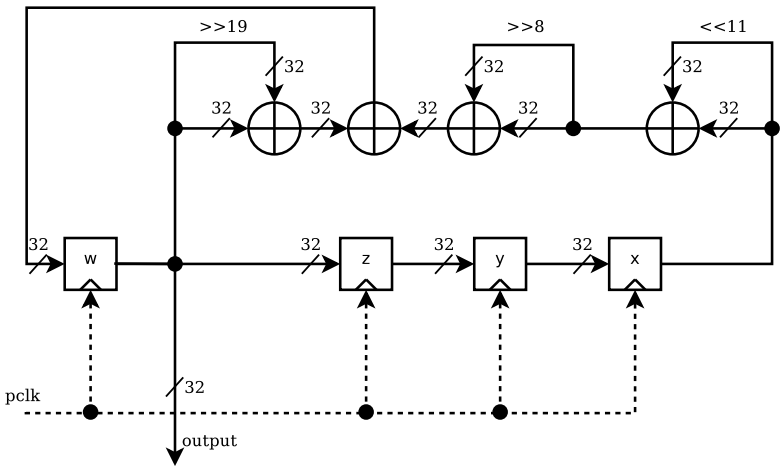

Figure 3.15: Hardware structure of *Xorshift*

also analyzed them and compared how the needed effort for a successful DPA attack differs to the unprotected implementation. Other compared parameters are the total correlation value, usage of FPGA resources and ease of the implementation. The countermeasures we chose were proposed by [4] as they seemed interesting regarding their focus on FPGAs.

The first countermeasure we implemented is called *clock randomization* i.e. randomizing the timing behavior during critical calculations which appear to be suitable for hardening the design against DPA attacks. The second countermeasure utilizes random *short circuits* to randomize power consumption during critical calculations.

Both of these countermeasures use a pseudo-random number generator, which we chose to be a *Xorshift Random Number Generator (RNG)* due to its relative ease of implementation. Further details on the RNG are discussed in the following subsection.

3.3.1 The Xorshift Random Number Generator

The Xorshift RNG was first proposed by George Marsaglia in [44] and is a pseudo random number generator that requires a rather small amount of resources. Although a simple implementation of Xorshift which lacks an additional non-linear step (which would result in a stronger Xorshift* [44] or Xorshift+ [45]) is weak to some statistical attacks we chose the basic version as our generated sequence is never fully exposed to attacks. This is owed to the fact that the value of the 32 bits while they are controlling short circuits is masked in the limited variance of the power consumption and only 3 bits of the RNG are utilized while using it as a source for clock randomization. It should also be noted that due to the *always on* nature of this RNG (which also disregards resets) the chance of reoccurance of the same sequence point during critical calculations is highly unlikely. This version of Xorshift yields a sequence with a length of $2^{128} - 1$ of 32-bit wide values.

Figure 3.15 shows a hardware implementation of Xorshift. Take note that the shift operations of Xorshift are logical and not circular. So if this RNG were to be implemented in an area optimized setting this fact could be used to omit 38 gates. The registers *w, x, y, z* are all 32-bit wide and initialized with hardcoded, non-zero values. The register *w* also serves as the output. All registers are clocked with the standard clock of the circuit.

This also applies during active clock randomization phases where the RNG module remains the only one still clocked by the unaltered clock signal. For further details on

clock randomization see the following subsection.

3.3.2 Clock Randomization

Clock Randomization is defined as a method to randomize the clock cycles during critical calculations in order to impede DPA attacks by varying the occurences of positive clock edges for every trace recorded. In order to make the clock randomization on an FPGA feasible we use the clock buffers (CBs) and digital clock managers (DCMs) which are provided by the FPGA. Clock Buffers here refer to multiplexers which can switch between different clock signals and keep them stable while switching. DCMs offer many other functionalities but we are only interested in one of them, their ability to output multiple phase-shifted clocks simultaneously. Although the FPGA we used[2] features eight DCMs we were only able to use a maximum of two DCMs for clock randomization due to routing limitations of the designated clock network which inhibited access to additional clock buffers. These limitations made the inclusion of additional DCMs in the randomized clock-tree impossible.

Figure 3.16 shows the implementation structure of our clock randomization with eight clocks derived from two DCMs. The clock signal is selected by three different bits *r0, r1* and *r2* of the current value held by the RNG. We also use the additional control signal *clkr_on* to turn clock randomization on or off. This is necessitated by the need of a constant clock during communication states. Originally we wanted to use another CB to switch between randomized and unrandomized clock but had difficulties generating a valid bitstream file. So we chose to implement the three AND gates at the left-hand side of Figure 3.16 as a way to activate clock randomization.

We also implemented smaller pools of clock signals to randomize between the phase distance of these clocks set to a maximum[3]. The scaling in behavior and effectiveness against DPAs with varying degrees of clock randomization will be discussed in a later chapter.

3.3.3 Short Circuits

Short circuits (SCs) can be used on an FPGA as a noise generator. The basic idea is to turn SCs on and off randomly during vulnerable operations which changes the power values of each recorded trace-sample. As a result the SNR and the exploitable leakage used by a DPA attack should decrease.

We opted to have the switching of SCs always turned on as their occurrences do not impede the operation of our design. This choice also translates to no additional control signals in our design. Implementing SCs themselves proofed to be a bit problematic. [4] proposed the following process to implement SCs on Xilinx Virtex-II Pro FPGAs:

1. Create one SC manually with the FPGA Editor by Xilinx
2. Transform the SC design into an .xdl[4]-file and manipulate the configuration into an actual SC there. Then transform the result back into a standard design file (.ncd).

[2] Xilinx Virtex-II Pro 2vp30

[3] two clocks: 0 and 180 degrees, four clocks: 0, 90, 180 and 270 degrees

[4] The Xilinx Design Language (XDL) can be used to directly change device configurations through a text editor

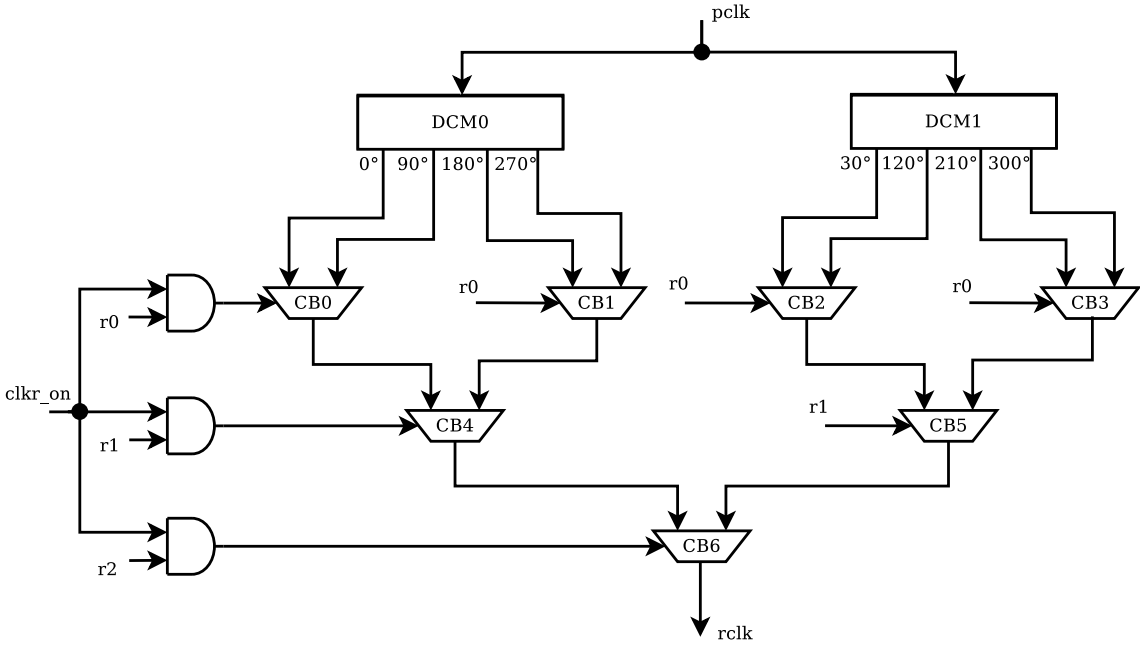

Figure 3.16: Clock randomization with 8 different clock signals

3. Save the design as a hard macro (.nmc) and instantiate it multiple times in your main implementation. Make sure that the outputs of your RNG are connected to the inputs of the instantiated SC modules.

4. Compile your main implementation into a standard design file (.ncd) and then relocate the hard macros to spread-out places to avoid damaging the FPGA. This relocating is again done with the FPGA Editor

5. Generate a .bit[5]-file with the DRC (design rule checking) turned off.

As we tried to follow these steps we used the same toolchain as [4] claimed to have used but could not complete them. At step 4 of their proposal we could not proceed further because the Xilinx toolchain could not place hard macros which contain activated SC designs.

Our first idea to circumvent this roadblock was to compile our design with instantiated hard macros that contained only unactivated SCs. Then we could have activated them via the .xdl-file we would create from our compiled implementation. This idea unfortunately did not work. When trying to convert the compiled design (.ncd) to XDL the official Xilinx conversion tools gave segmentation faults. These segmentation faults only occur if the source file was generated with instantiated hard macros.

Another way to implement SCs on FPGA was proposed by [5]. They proposed to activate SCs on the .bit-file using a Hex Editor. In order to be able to tell which bits should be flipped, [5] developed a tool to analyze .bit-files. From here on out one could activate SCs which were instantiated though hard macros or by hand if one had sufficient knowledge about the .bit-files. Unfortunately we were unable to use the tool[6] developed

[5]Bitstream files are used to configure Xilinx FPGAs

[6]The tool is called ReCoBus-Builder and is developed and supported at the university of Erlangen. https://www12.informatik.uni-erlangen.de/research/recobus/

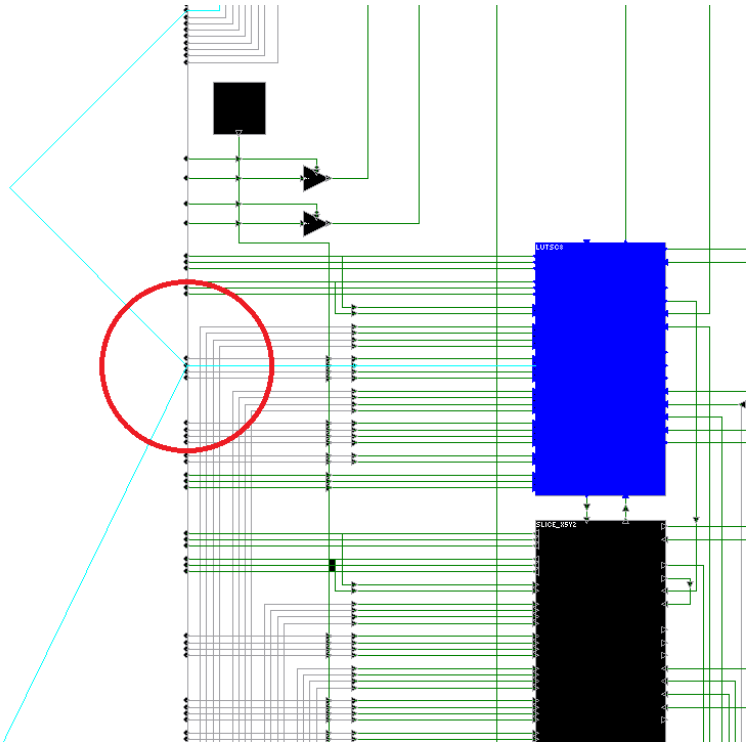

Figure 3.17: An activated short circuit as seen in the FPGA Editor. The circle indicates the position of the short circuit

by [5]. This is why we proposed our own way to successfully implement SCs on a Xilinx Virtex-II Pro FPGA:

1. Compile your implementation into a .ncd file. Make sure your RNG is not optimized away if it is not used for other purposes.
2. Draw the SCs you want to implement and connect them to the RNG ouputs with the FPGA Editor.
3. Redraw the connections you want to activate your SCs with and save them in another .ncd file.
4. Convert both .ncd files from steps 2 and 3 into XDL. Replace the connections of the former with the connections of the latter to activate the SCs.
5. Convert the implementation back to .ncd and generate the .bit file using the *bitgen* tool without DRC checks.

In order to get a better idea of our capabilities regarding SCs we conceived three different SC designs whose different impacts on the noise of our measurements will be discussed in Chapter 6. The three designs can be seen in Figure 3.18. Design 1 is closest to what was proposed by [4]. It features two flipflops (FFs) fed by different outputs of the RNG. The outputs of the flipflops create a short circuit on the same input of another slice.

Design 2 omits the need for a clock connection by replacing the flipflops with look-up tables (LUTs). Simplifying the SC construction even more is Design 3 which omits those

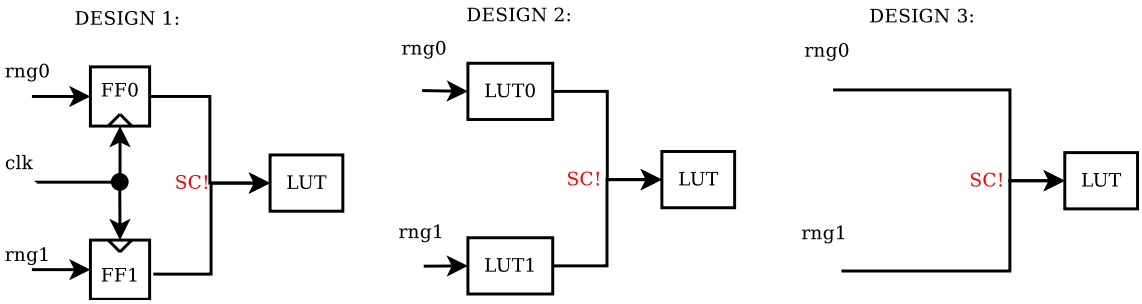

Figure 3.18: Three different approaches to SC designs.

intermediate components (FFs or LUTs) completely and just short-circuits the outputs of the RNG directly which results in the longest connections that are re-purposed as a SC.

Design	1	2	3
Amount of Slices per SC	3	3	1

Table 3.1: FPGA usage of each short circuit design

Another concern when implementing SCs is the maximization of the possible power consumption variance. Remember that one SC can occur if the input bits have different values.

$$SC_{r_i,r_j} \text{occurs if} (r_i \oplus r_j), i = 0, 1, ..., 31, j = 0, 1, ..., 31, i \neq j \tag{3.1}$$

Assuming we have a RNG which provides 32 random bits $r_0, r_1 ... r_{31}$ the amount of total possible connections is defined by the binomial coefficient $\binom{32}{2} = 496$. Due to the exclusive requirements of a SC occuring it is not possible to have 496 SCs activated at the same time. Take also note that the variance of actual SCs activated plays the most important role when using SCs as a countermeasure.

Take for example four random bits r_0, r_1, r_2, r_3 and connect them with two, three, four or six ($\binom{4}{2} = 6$) short circuits as seen in Figure 3.19.

If we assume that all four random bits have a probability of being 1 at 50% we can analyse at what combinations which SCs are active. The possible amounts of SCs active at the same time is shown in Table 3.3 while one example of how this listing came to be can be observed in Table 3.2. The best variance of values is achieved with three SCs while the implementation of four SCs will lead to the least amount of hiding DPA-relevant information by noise. It does not matter which connections are made for the SCs, but how many were implemented.

We connected the SCs after the pattern

$$SC_{r_i,r_{i+1}} \text{occurs if} (r_i \oplus r_{i+1}), i = 0, 2, 4, ..., 30 \tag{3.2}$$

for up to 16 SC instantiations. When implementing 32 SCs we connected the remaining SCs in this pattern

$$SC_{r_i,r_{i+3}} \text{occurs if} (r_i \oplus r_{i+3}), i = 0, 1, 2, ..., 15 \tag{3.3}$$

Besides implementations of varying amounts of short circuits (1,4,8,16,32; most of them with all three different SC designs) we created one implementation just containing

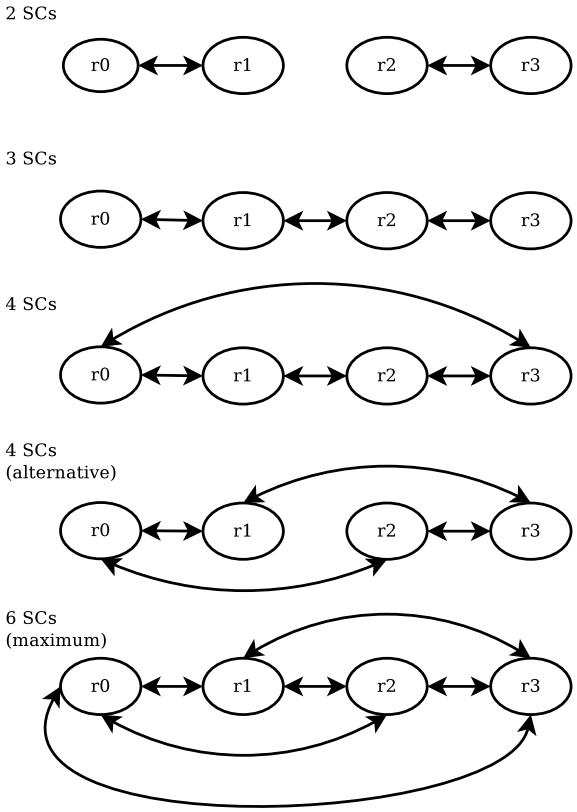

Figure 3.19: Multiple ways to use four random bits for short circuits. Arrows indicate a pairing for a short circuit.

bit values	active SCs	bit values	active SCs
0000	0	1000	1
0001	1	1001	2
0010	2	1010	3
0011	1	1011	2
0100	2	1100	1
0101	3	1101	2
0110	2	1110	1
0111	1	1111	0

Table 3.2: Possible active short circuits with 3 implemented short circuits on four random bits

Implemented SCs	Active SCs						
	0	1	2	3	4	5	6
2	4	8	4				
3	2	6	6	2			
4	2	0	14	0	2		
6	2	0	0	8	6	0	0

Table 3.3: Distribution of active short circuits with variyng amounts of implemented short circuits on four random bits

the RNG module for comparison purposes. It also should be noted that our proposed process of implementing SCs is rather unsuited for easy recreation, requiring a huge amount of manual work and time with the FPGA Editor. And it remains untested if similar procedures can succeed on more modern Xilinx FPGAs or on FPGAs by other vendors.

3.4 Synthesis Results

In this section we discuss the results of our implementation without FPGA-specific countermeasures and compare it to the results of the available related work. This is why we also synthesized our implementation for ASIC (using standard cell libraries) as it yields more comparable results. Table 3.4 showcases the differences between our results in gate equivalents (GE), the results of the original compact implementation of AES [6] and the original implementation of ALE [1]. One intersting observation is that the original implementation of ALE [1] claims to be smaller than the pure AES core proposed by [6]. [1] also claims that their implementation only needs to load the master key twice without using a third storage array like we do. Their claims are similar when it comes to plaintext blocks i.e. they achieve a one-time transmission of the plaintext blocks which are necessary to generate the ciphertext blocks and then need to be added into the state array up to four AES rounds later without any additional storage. We claim that the storage of the master key for the duration of a 10-round AES encrytpion without a third 128-bit register or the reversal of the 10 rounds of keyschedule seems highly unlikely. Also very unlikely is the successful storage of complete plaintext blocks for addition into the stream without any additional registers like we use in our *temp* module.

So in order to get a better comparison we took a closer look at the implementation of [6] which the original proposal of ALE [1] also claims to integrate. Comparing [6]

with our own implementation yields some insights. We managed to get all of our core components close or below the thresholds set by [6] except for our area requirements for the multiplexers and the control logic.

	ALE		AES [6]		ALE [1]
Technology	$UMC\,180\,nm$		$UMC\,180\,nm$		$STM\,65\,nm$
Sum	3 622 GE	100%	2 601 GE	100%	2 579 GE
	Single modules [GE] [%]				
State	699	19%	843	32%	
Key	747	21%	835	32%	
RCON	83	2%	89	3%	
Sbox	230	6%	233	9%	
Mixcolumns	349	10%	373	14%	
Key add	20	1%	21	1%	
Multiplexers	213	6%	128	5%	
Control logic	385	10%	72	3%	
Other	0	0%	7	1%	
Temp	678	19%			
LEX-leak	218	6%			

Table 3.4: Area results and comparisons

This holds especially true when comparing our values to the most similarily implemented modules like the *key, Sbox* or *RCON* modules. One notable derivation is the *state array* which is smaller in our implementation. This can be attributed to the the 32-bit wide multiplexer which is used by [6] to circumvent the mixcolumns module in the last round of AES. We do not need this multiplexer due to our additional use of clock-gating circuits controlled by the *state_en* signal to disable the registers during this state. Our need for more multiplexers in the datapath is explained by our additional requirements regarding ALE, for example its initialization necessitates additional multiplexers for relocation purposes. Our different approach towards the *mixcolumns* module has only manifested in neglible area savings which is surprising because the original proposal of this mixcolumns implementation [13] projected an area requirement of 140 "gates". It seems the authors confused gates with cells as 140 is the exact number of cells this module requires. Our biggest weakness is obviously the *control* module which takes more than five times the area required by Moradi et al's [6] highly optimized shift register they implemented instead of an FSM.

Compared to our first approach to the *LEX-leak* module we save a total of 504 GE with the module using only 35 percent of the area of our original approach. Although the module is now smaller by a relative amount our additional use of scan flipflops yielded about 37 additional GEs. Surprisingly enough we even saved 19 GE in our control module.

We used *Cadence Encounter(R) v08.10-s238_1* for our synthesis. This tool also projects a power usage of around 3.7 mW and a maximum frequency of 94.02 MHz. The critical path is also determined by this toolchain and starts inside FSM and ends with the storage into the *state* module utilizing the Sbox and requires a simulated time of 10 636 ps.

We also had to place-and-route our design to the target FPGA, which is a Xilinx Virtex-II Pro FPGA. The results of this can be observed in Table 3.5. The table also holds the place-and-route results for a design with an altered FSM which we did construct to improve our measurement setup (see Figure 5.1) in order to be able to verify the results

of the on-device AES-encryptions directly. The last segment of Table 3.5 is dedicated to the resources used when the 32-bit RNG of Section 3.3.1 is included into the implementation. All three sections of Table 3.5 implemented the serial-to-APB module to facilitate communication with a computer.

Complete ALE		Amount	Total	Percentage
slice utilization:	slice flipflops	551	27 392	2%
	4-input LUTs	1 061	27 392	3%
	occupied slices	550	13 696	4%
	used IOBs	13	416	3%
use of 4-input LUTs:	logic	1 033	1 061	97%
	route-through	12	1061	1%
	shift registers	16	1061	2%
FSM modified to AES-only		Amount	Total	Percentage
slice utilization:	slice flipflops	547	27 392	1%
	4-input LUTs	1 014	27 392	3%
	occupied slices	528	13 696	3%
	used IOBs	13	416	3%
use of 4-input LUTs:	logic	988	1 014	97%
	route-through	10	1 014	1%
	shift registers	16	1 014	2%
AES-only with added RNG module		Amount	Total	Percentage
slice utilization:	slice flipflops	611	27 392	2%
	4-input LUTs	1 091	27 392	3%
	occupied slices	567	13 696	3%
	used IOBs	13	416	3%
use of 4-input LUTs:	logic	1033	1 091	95%
	route-through	10	1 091	1%
	shift registers	48	1 091	4%

Table 3.5: Resources used on the Xilinx Virtex-II Pro (2vp30)

As can be deduced from the percentage numbers of Table 3.5, many of the FPGA's slices remain unused. This has an advantage for our SC countermeasure as many slices to construct SCs with them remain available. This can be beneficial to the countermeasure since the SCs can be spread out more evenly over the FPGA.

When reducing the FSM to function as an AES-only implemnation we save 22 slices through the optimization methods of the Xilinx tools in total. The 32-bit RNG module adds to this with 39 used slices.

The clock-randomization countermeasure uses seven CBs and two DCMs of the FPGA which can result in routing problems when additional independent clock routes should be added to the design. If other DCMs should be added (e.g. for dividing the operating frequency), the Xilinx place-and-route connects them to general routing and can not give the intermediate clock signals access to the dedicated clock nets.

3.5 Summary

In this chapter we have discussed our implementation of the ALE authenticated encryption method, which is based on a compact AES core and an LEX-leak extension of which we implemented two different versions. One of which has become our main implementation while we discussed the other implementation for a better understanding of the complexity a non-standard LEX-mask poses to a rows-first architecture.

We also discussed the implementation of the FPGA-specific countermeasures we have chosen to secure ALE against DPA attacks. There we described our source of pseudo randomization, and also the countermeasures known as *clock randomization* and *short circuits*.

With our implementation of ALE we have also shown that there is some room left for optimization regarding the area of a compact implementation of AES as proposed by [6], at least when it comes to the datapath. We have still some scepticism regarding the original ASIC implementation of ALE in [1], especially when it comes to the intermediate storage of data.

In the following chapter we will discuss differential power-analysis (DPA) in general and how we constructed a power model of our implementation to perform DPA attacks on a Xilinx Virtex-II Pro FPGA which processes AES encryptions based on our design.

Chapter 4

DPA Attack on ALE

The functional-verified implementation of ALE was analyzed according to possible side-channel leakages in a next step. By activating the implemented countermeasures one after the other, the side-channel leakage should be minimized. But before the discussion of these results is provided in the next chapters, the theory behind DPA attacks and how one specific DPA attack for this implementation was conceived are summarized in this chapter.

Differential power analysis (DPA) attacks are among the most popular forms of side-channel analysis according to [2]. Sections in this work attribute this to the fact that DPA attacks do not require detailed knowledge of the attacked device. When conducting a simple power analysis (SPA) in an encryption device it is necessary to attribute certain parts of a power trace to specific parts of the cryptographic algorithm (so called profiling). With DPA attacks this is not necessary since they treat all parts of a power trace equally so any point of the recorded time-frame could lead to a positive DPA result if that point features a strong correlation with the chosen power-model.

Another advantage of DPA attacks is their likely success despite extremely noisy power traces as the number of required traces for a successful DPA increases with increasing noise levels. This requires ownership of the device under attack for extended periods of time and also that the key must not change during the attack. Both of these concerns do not affect us in our research environment but should be kept in mind for the real-world feasibility of these attacks.

Differential power analysis works by evaluating fixed points of time regarding their dependence on data using significant amounts of power traces in the process. This gives way to our preconditions before attacking a device. We need to know which encryption method is used inside of the device in order to give the processed data and the power consumption a relationship for the evaluation. We need to have access to a communication interface through which we can provide the device with data and start the encryption or the decryption processes. Another requirement is the ability to measure one or more parameters that are related to the processed intermediate data (e.g. power consumption, electro-magnetic emissions, sound emissions).

In the remaining sections of this chapter we will take a closer look at how DPA attacks are realized in a general fashion. Then we will discuss how we used these generic steps to attack our implementation of ALE. Furthermore we will take a closer look at how other researchers created attacks on ALE. We will also discuss some other approaches (i.e. points of attack) we could have exploited.

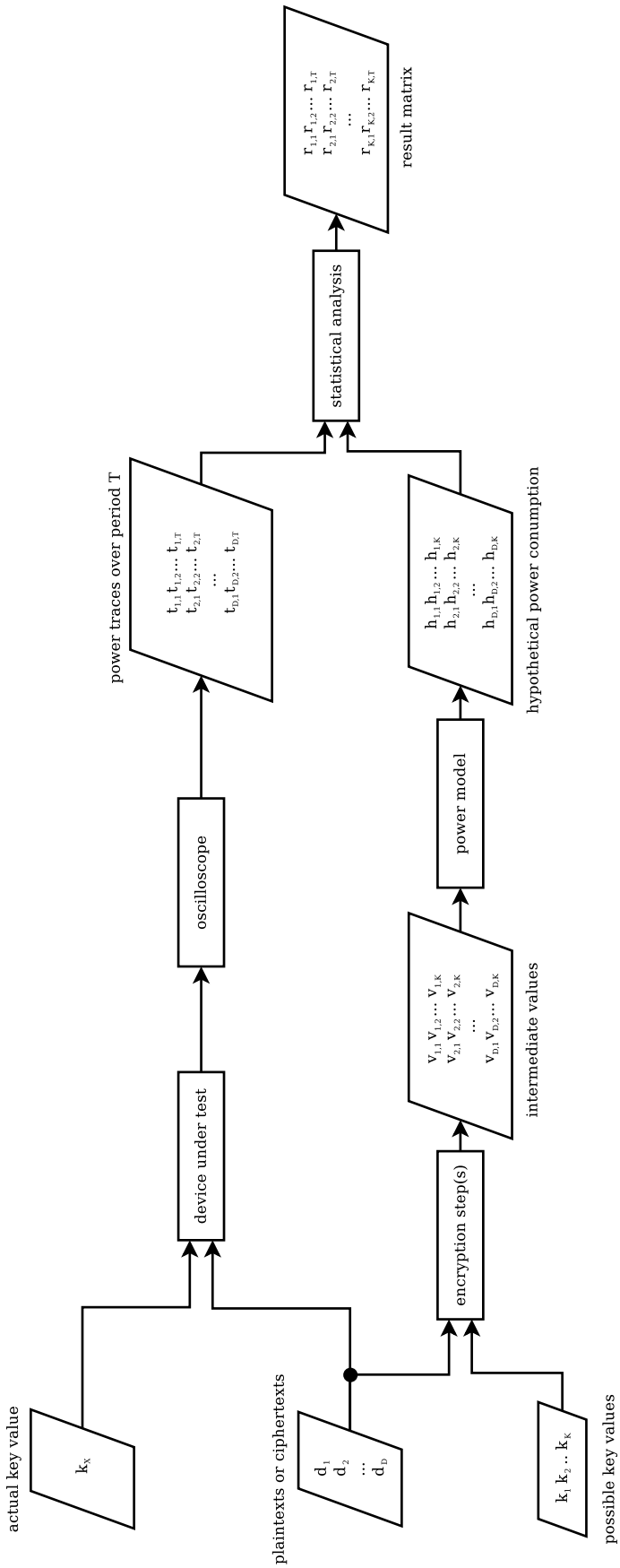

Figure 4.1: General overview of a DPA attack

4.1 The Basic Steps for Differential Power Analysis Attacks

For attacking a cryptologic implementation using DPA a few general steps were outlined by Stefan Mangard et al. in [2]. These steps are general in nature and their application on an actual cryptologic implementation can be followed in Section 4.2. To get an overview of how these steps are interconnected on can take a look at Figure 4.1 which shows a general overview of an DPA attack after a sensitive intermediate value has been determined. The general steps for a DPA can be summarized as follows:

1. Choosing a sensitive intermediate value

   The first step is to choose an intermediate value of the algorithm that is being processed in the device under test (DUT). This value must be the result of a function $f(d, k)$ where d denotes known data (a part of the plaintext or ciphertext) and k is a part of the key.

2. Measuring a parameter related to the sensitive intermediate value

   In the second step multiple samples of a parameter related to the sensitive intermediate value are recorded. These samples subsumed into so called traces with one trace per encryption. Not all recorded samples must be in direct relationship to the sensitive intermediate value but at least one part of these traces must contain the time interval of the calculation of the intermediate value chosen in Step 1. One also needs to keep track of the known data used during these trace recordings. These values are stored as a vector $d = (d_1, d_2, ..., d_D)$ -with D being the amount of recorded traces- as can be seen on the left-hand side of Figure 4.1. The values of one trace are stored as the vector $t_i = (t_{i,1}, t_{i,2}, ..., t_{i,T})$ with T being the amount of samples per trace.

   In order to record perfectly aligned power traces some sort of trigger signal is required. If such a signal is not available several techniques to align or preprocess these traces at a time post-recording exist. Alignments are usually done using pattern matching with the evaluation done with least squares or the help of the correlation coefficient (see Section 4.1.2). Preprocessing methods include integration (summing up the power consumption of multiple clock cycles), convolution of the power traces with a suitable window function or in some cases even the Fast-Fourier Transformation which depends on the spectral characteristics of noise and leakage.

3. Compute Intermediate Values

   Using the function chosen in Step 1, we calculate all possible intermediate values of this function using $d = (d_1, d_2, ..., d_D)$ and all possible key hypotheses $k = (k_1, k_2, ..., k_K)$. This calculation results in a matrix V with the dimensions $D \times K$.

$$V = \begin{pmatrix} v_{1,1} & v_{1,2} & ... & v_{1,K} \\ v_{2,1} & v_{2,2} & ... & v_{2,K} \\ ... & ... & ... & ... \\ v_{D,1} & v_{D,2} & ... & v_{D,K} \end{pmatrix} \tag{4.1}$$

Each of the columns of V belongs to one hypothetical value of the key. The further goal will be finding out which of the columns of V was calculated during the execution of the algorithm during the measurements of Step 2.

4. Choosing a Power Model

   As can be seen in Figure 4.1, the values that were calculated into matrix V are mapped to corresponding values of a power model into the matrix H. Power models are normally a very simple unit-less way of estimating power consumption. The two most popular models are the *Hamming weight (HW)* and the *Hamming distance (HD)*. Section 4.1.1 will discuss these power models with the addition of the zero-value power model.

   The choice of a power model strongly depends on the knowledge of the device under test (DUT). The better these simulated power values correspond with the actual power usage the better are the results of the DPA attack. A better DPA attack is defined by a smaller number traces needed for a successful deduction of the key value.

$$H = \begin{pmatrix} h_{1,1} & h_{1,2} & ... & h_{1,K} \\ h_{2,1} & h_{2,2} & ... & h_{2,K} \\ ... & ... & ... & ... \\ h_{D,1} & h_{D,2} & ... & h_{D,K} \end{pmatrix} \tag{4.2}$$

5. Statistical Analysis

   Now that the calculation results of V are mapped to the power values of H it is possible to compare each column of H to each point in time of the recorded traces. This results in a matrix called R with the dimensions $K \times T$.

   There are different algorithms to use for this comparison but when using the correlation coefficient higher values inside of R correspond to a better match of one key hypothesis with the samples of the power traces. We will use the *correlation coefficient* which we discuss in more detail in Section 4.1.2.

6. Evaluating the Result

   In this step the result matrix R is evaluated. If the correlation coefficient is used the highest values of R reveal at which point in time our chosen intermediate value was processed and with which key hypothesis this was done.

   If there are multiple conflicting maxima or none at all in R the DPA attack has failed. This can be due to too few recorded traces or wrong assumptions regarding the power model. It is also possible that the DPA attack was thwarted by countermeasures.

4.1.1 An Introduction to Various Power Models

In this section we will discuss the following three power models: the hamming weight (HW), the hamming distance (HD), and the zero value (ZV) model. A power model is used as a mean to *map* intermediate values to hypothetical power values. These power values do not have a designated unit and should be viewed as a way to differentiate the power consumption of one computation result to another.

The power models discussed here were also listed by Stefan Mangard et al. in [2] which serves as a point of reference for most research into power analysis attacks as it has compiled a huge amount of related research. Take note that it is possible to derive other power models from the power models described below for a more specific description of power usage if one has a better understanding of the algorithm and the attacked device.

- The Hamming Weight (HW) Power Model

  The Hamming Weight is the number of bits set to *1* of a given array of bits. For example $HW(1110_2) = 3$. The power model based on this function is rather simple. It assumes that the HW of a given processed value v_0 is proportional to the actual power consumption. This means that this power model is rather unsuited for CMOS technology whose actual power usage is proportional to which signals of its buses and registers switch during its clock cycle than the actual values that are held by them.

  If an attacker does not know the value which is replaced by v_0 he has not other choice but to use this model. Normally there is still some proportionality of the HW of v_0 and the actual power consumption especially if the value which is replaced by v_0 is a constant. Another aspect of why this model works is that transitions of values conducted in CMOS technology have slightly different power usages for zero-to-one and one-to-zero transitions. So if zero-to-one transitions have a higher power consumption, values with a larger HW will use more power than values with a lower HW.

  As a rule of thumb most attackers should only rely on the HW model if the use of the Hamming Distance model is unavailable. But as always there are exceptions, for example many processors set their buses to zero before loading new data onto them. In this case the HW model is obviously a better fit than the HD model. The choice of power model should always be given enough consideration.

- The Hamming Distance (HD) Power Model

  The Hamming Distance describes the number of bits that are flipped during a storing or bus transition. For example if v_0 is replaced by v_1 then the HD of this transition can be described as $HD(v_0 \rightarrow v_1) = HW(v_0 \oplus v_1)$. It follows that this model assumes that bit-transitions of zero-to-one and one-to-zero require the same amount of power and ignores static power consumption altogether.

  The HD model is very popular due to its relative simplicity and ease to compute while also delivering correct results for DPA attacks. Although not suited for describing the power consumption of combinational cells it is commonly a good fit when trying to model power consumption of buses and registers.

  In order to use the HD it is necessary to know two consecutive values inside of a register or a bus, this can be if a constant value (like for example: zero) is replaced or a other data known to the attacker is involved. So the HD power model usually requires some knowledge of the internal architecture of the DUT.

- The Zero-Value (ZV) Power Model

When trying to attack combinational logic, the Zero-Value model may apply. Instead of trying to model the power consumption of transitions or simple bit-weights of the processed data the ZV model attacks multiplications (including AND-gates).

The ZV model assumes that for some combinational architectures (like Sboxes or multipliers) a zero-bit results in much less power consumption than if the bit is set to one due to a multiplication by zero.

The ZV model is defined for each input bit of combinational logic as

$$ZV(v_{i,j}) = \begin{cases} 0 & \text{for } v_{i,j} = 0 \\ 1 & \text{for } v_{i,j} \neq 0 \end{cases}$$

4.1.2 The Correlation Coefficient

The correlation coefficient is one common method used to create a relation between hypothetical and measured power consumption. The correlation coefficient or more specifically the *Pearson product-moment correlation coefficient* generally is used to measure the linear correlation between two variables.

The coefficient is defined as the covariance between two variables divided by the product of their standard deviations. The equation for a statistical data sample is shown below in Equation (4.3). $\bar{x}$ and $\bar{y}$ are the mean values of the data vectors $\vec{x}$ and $\vec{y}$ while n describes the amount of data values available for each of the data vectors.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.3}$$

So if we want to apply Equation (4.3) on a DPA attack our two variables we want to correlate with each other are our modeled power values $h_{d,i}$ and the measured power values of $t_{d,j}$ with $i = 1, .., K$ denoting our possible key hypotheses and $j = 1, ..., T$ indexing each sample of a trace over a total of T. Lastly $d = 1...D$ indexes our processed plaintext values. This results in Equation (4.4) which assigns each key hypothesis i at a specific sample time j a correlation value $r_{i,j}$.

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)(t_{d,j} - \bar{t}_j)}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h}_i)^2 \sum_{d=1}^{D}(t_{d,j} - \bar{t}_j)^2}} \tag{4.4}$$

4.2 The Attack on ALE

We chose our sensitive intermediate value based on a standard approach to do an attack on AES which uses the relation of processed data and the key in the first round of AES when the plaintext is known. A similar approach to attack AES uses the relation of processed data and the key in the last round of AES when the computed ciphertext is available. This relation can be exploited bit-wise if the output of the key addition is the target of the attack. Similar techinques have been employed by Moradi et al. when they performed a side-channel analysis on their implementation of AES [6].

Another way to exploit this relation of input (or output) data with the key is byte-wise. This can be achieved when we want to attack the output of the SubBytes-step of AES. In this step the results of the AddKey operation are transformed through Rijandel's Sbox. When we consider that we know the plaintext that is being used as an IV for ALE

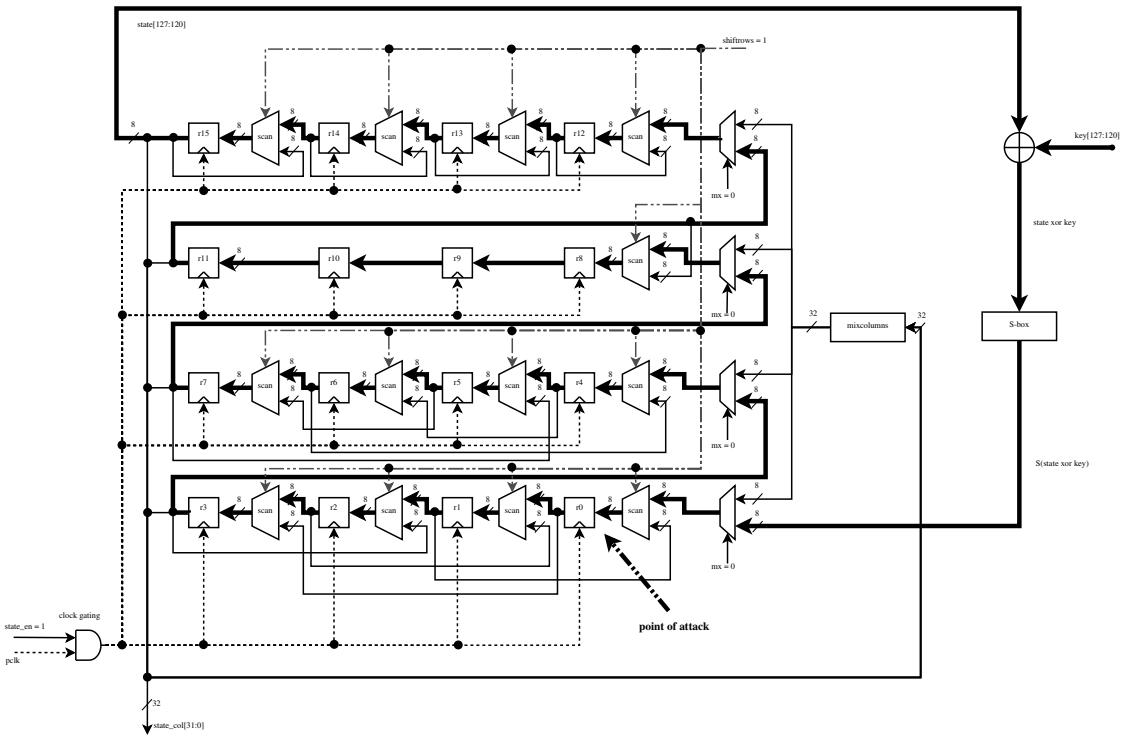

Figure 4.2: Point of attack in the implementation

which is a possibility since it will most commonly be generated by a counter then we can describe all possible outputs of one processed byte in the first round of AES through the Sbox as the vector $\bar{r}_i = S(p_i \oplus \bar{k}_i)$ where $S()$ is the Sbox transformation, the index i holds a value of 1 to 16, p_i is the respective plaintext byte while $\bar{k}_i$ represents all of the 256 valid values for the respective key byte. If we write k_i we assume to know the correct value of the respective byte.

Due to our knowledge of the implemented architecture, we can use this Sbox calculation for a power model based on the Hamming distance while an attack on the operation of the key addition seems less rewarding since its results are not intermediately saved into any kind of register but wired into the Sbox module itself. When we look at the moment when the first byte of the first round of AES is to be calculated we can write the result of the key addition which is then transformed by the Sbox as $r_1 = S(p_1 \oplus k_1)$. When this value is stored into the last register of the state module this operation will overwrite the value currently held at the last byte of this register which equals p_{16}. The Hamming distance of this can be viewed as

$$HD(p_{16} \leftarrow S(p_1 \oplus \bar{k}_1)) = HW(S(p_1 \oplus \bar{k}_1) \oplus p_{16}) \tag{4.5}$$

which we can use to calculate hypothetical power values in association with the first key byte. When attacking the remaining bytes of the key we can describe the HD as

$$HD(S(p_{i-1} \oplus k_{i-1}) \leftarrow S(p_i \oplus \bar{k}_i)) = HW(S(p_i \oplus \bar{k}_i) \oplus S(p_{i-1} \oplus k_{i-1})), i = 2...16 \tag{4.6}$$

due to the further workings of the shift register. The k_{i-1} of Equation (4.6) denotes the correct key byte which was found out a priori through another DPA calculation but can

HDs / *states*	AES_FR	AES_SR	AES_MC_SW	Total
$HD(p_{16} \leftarrow p_1, \bar{k}_1)$	16	0	0	16
$HD(S(p_1 \oplus k_1) \leftarrow S(p_2 \oplus \bar{k}_2))$	15	0	1	16
$HD(S(p_2 \oplus k_2) \leftarrow S(p_3 \oplus \bar{k}_3))$	14	0	2	16
$HD(S(p_3 \oplus k_3) \leftarrow S(p_4 \oplus \bar{k}_4))$	13	0	3	16
$HD(S(p_4 \oplus k_4) \leftarrow S(p_5 \oplus \bar{k}_5))$	12	0	0	12
$HD(S(p_5 \oplus k_5) \leftarrow S(p_6 \oplus \bar{k}_6))$	11	1	0	12
$HD(S(p_6 \oplus k_6) \leftarrow S(p_7 \oplus \bar{k}_7))$	10	1	1	12
$HD(S(p_7 \oplus k_7) \leftarrow S(p_8 \oplus \bar{k}_8))$	9	1	2	12
$HD(S(p_8 \oplus k_8) \leftarrow S(p_9 \oplus \bar{k}_9))$	8	0	0	8
$HD(S(p_9 \oplus k_9) \leftarrow S(p_{10} \oplus \bar{k}_{10}))$	7	0	3	10
$HD(S(p_{10} \oplus k_{10}) \leftarrow S(p_{11} \oplus \bar{k}_{11}))$	6	0	0	6
$HD(S(p_{11} \oplus k_{11}) \leftarrow S(p_{12} \oplus \bar{k}_{12}))$	5	0	1	6
$HD(S(p_{12} \oplus k_{12}) \leftarrow S(p_{13} \oplus \bar{k}_{13}))$	4	0	0	4
$HD(S(p_{13} \oplus k_{13}) \leftarrow S(p_{14} \oplus \bar{k}_{14}))$	3	1(→)	2	6
$HD(S(p_{14} \oplus k_{14}) \leftarrow S(p_{15} \oplus \bar{k}_{15}))$	2	1(→)	3	6
$HD(S(p_{15} \oplus k_{15}) \leftarrow S(p_{16} \oplus \bar{k}_{16}))$	1	1(→)	0	2

Table 4.1: Occurrences of the HDs during each FSM state of the first AES round

also be still an unknown key byte in which case the amount of possible key hypotheses is increased [1]. This was also the case for Moradi et al.'s [6] basic attack model. For $i = 2$ in Equation (4.6) the previosly found out key byte has resulted from the DPA which was computed with Equation (4.5);

Due to the nature of the state module these Hamming Distances should reoccur for multiple clock cycles. A simplified version of all components involved and according to our design can be seen in Figure 4.2. If we consider the HD as described in Equation (4.5) for example, the same HD occurs for 16 consecutive clock cycles until the value $r_1 = S(p_1 \oplus k_1)$ is written into the first byte of the register inside of the state module during the FSM state called AES_FR. The following HDs will each occur one cycle less. But we also have to consider these HDs during the next states of our FSM which can be seen in its AES optimized form in Figure 5.1. During the subsequent states of AES_SR and AES_MC_SW some of the HDs discussed above reoccur. During AES_SR regular shift operations to the left (second row from the top of Figure 4.2) and to the right (last row from the top of Figure 4.2) are carried out which reapplies to some of the HDs discussed above. The same is true for the shift operations facilitated during AES_MC_SW.

A complete summary of these occurrences can be seen in Table 4.1. These occurances translate into attack windows. The HDs with the most occurrences occur over 16 clock cycles while the last byte ($HD(S(p_{15} \oplus k_{15}) \leftarrow S(p_{16} \oplus \bar{k}_{16}))$) has the smallest time window. The attack window for these key bytes varies with the first *row* of the key bytes having the largest time window for a DPA attack.

Alternatives to this attack model could be an SPA on the second AES encryption during the initialization stage which always encrypts 16 bytes of zero values with the master key or a more specified attack during the shift operations of AES_MC_SW using a 32-bit wide key hypotheses.

Moradi et al [6] devised a similar attack on their design but left these multiple occur-

[1] two bytes give 65536 key hypotheses instead of 256

rences relatively untouched. We assumed that each of these occurrences would supply us with almost the same information leakage. This is why we assumed that small alterations of this attack would be of limited use. One of these alterations ensures a much better attack on the last key byte although its occurrence count is the same (2). It uses the HD between the 16th and the 13th bytes (first and last bytes of the last row) and occurs once during AES_SR and once AES_MC_SW. For a more extensive discussion why this occurs see Chapter 5.

$$HD(S(p_{13} \oplus k_{13}) \leftrightarrow S(p_{16} \oplus \bar{k}_{16})) = HW(S(p_{13} \oplus k_{13}) \oplus S(p_{16} \oplus \bar{k}_{16})) \tag{4.7}$$

4.3 Other Attacks on ALE

In this section we give a short summary of related attacks on ALE as they were conceived and realized by [7] and [8]. These attacks are not based on DPA but on algorithmic attacks which are based on differential cryptanalysis.

Both of these attacks used differential cryptanalysis on the LEX leak to produce a forgery attack. Please note that both of these attacks used the original LEX mask provided by [1] which was probably altered due to these attacks and make them more difficult since the new mask also varies in AES state rows and not only in columns.

First off was the so-called *LOCAL Attack* by Khovratovich and Rechberger[7] which used local collisions to produce the same authentication tags for different plaintexts and found out that the state information leaked through the ciphertext can lead to these collisions much faster. The probabilities they give are 2^{-102} when 2^{102} messages are available to 2^{-119} when only one message is available. After this they claim to be able to strengthen their attack to the point of full state recovery which allows universal forgery without knowledge of the master key.

The other attack [8] claims that the first one used an unfairly weakened version of ALE while using a half-sized implementation of ALE for their attack [2]. They used this implementation to show that the authentication security of ALE is only 97-bit and could be reduced to 93-bit if the whitening key layer is removed. On the other hand their attack seems to exploit the same leakage provided by the LEX Leak.

Both of these attacks show that they are highly dependent on the LEX mask which was altered in the reference implementation we received. This altered mask should at least provide a different challenge for such attacks.

4.4 Summary

In this chapter we have discussed how DPA attacks work in general and how we conceived our attack on our implementation of ALE. We also briefly discussed which other attacks have been put on the ALE algorithm up to now and how they focused more on differential cryptanalysis than differential power analysis. We are in the unique position where we can attack each byte of the key one after the other in a direct causality chain and each of the 16 key bytes will leak its information inside of a different time window.

In the next chapters we present the DPA attack results on ALE first without and then with countermeasures enabled.

[2] the block size was halved to 64 bits

Chapter 5

Side-Channel Analysis of ALE

In this chapter we will conclude the DPA attack on the unprotected FPGA-implementation of ALE. We will discuss why the attack on each byte of the key results in very different albeit successful correlation results.

First there is a short discussion of the used measurement setup on the SASEBO G-board. Then we discuss our results which were achieved while using the implementation with an operation frequency of 24 MHz. While doing this we made some observations on aspects of the correlation result which merited further investigation into their characteristics while changing the operation frequency. We conducted DPA attacks on operation frequencies ranging from 1.5 MHz up to 48 MHz and compared the characteristics of their correlation results. After discussing these results and how we achieved these alterations of operating frequency there will be a short subsection regarding our investigations into a phenomenon we call the *characteristic frequency of the FPGA*.

Lastly this chapter will conclude with a summarization of our findings and discuss potential points for future work. Some of these findings are expected results, while other results showcase some relatively unexplored power/correlation behavior.

5.1 The Measurement Setup

After a few test measurements it became clear that a reduction of ALE into a ten-round AES encryption shortens measurement time and accounts for a easier verifiability of the encryption. The basic AES result which is sent back to our workstation is more easily verifiable compared to the additional computational effort required for two additional AES encryptions, one single key schedule and at least one generation of a 128-bit ciphertext block which includes the even and odd round characteristic of the LEX mask.

Another simplification we introduced was the so called TRIGGER state which sets one of the available input/output pins to a logic HIGH and back to LOW four or eight clock cycles before AES_FR is entered[1]. The remodeled FSM for these two simplifications can be seen in Figure 5.1, no other module had to be changed. Although the complete ALE-encryption is no longer processed on the FPGA the attack remains the same and can still be used to evaluate the security of ALE.

[1]This gap of multiple clock cycles was introduced to minimize the possible overlap of power consumptions of AES_FR and the output signal whose influence on the power consumption can be measured for multiple cycles onward.

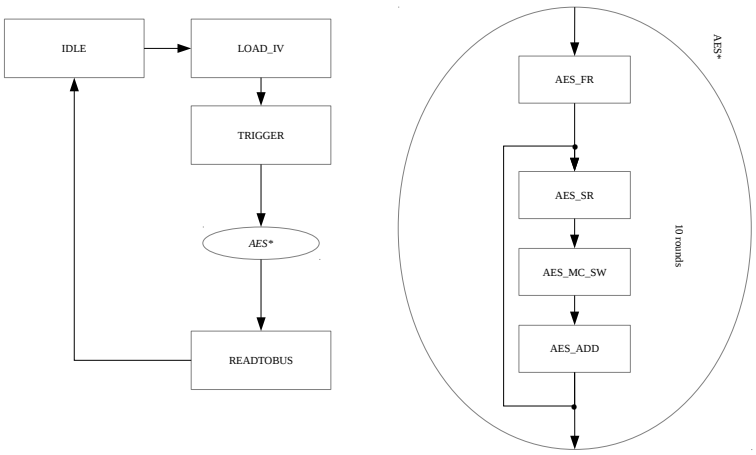

Figure 5.1: Modified FSM for faster measurements

The communication between workstation and DUT was achieved through an USB-to-serial interface and cable using the RS-232 serial-communication standard. The baudrate was fixed at 9600 for almost all measurements[2]. This serial communication was translated to AMBA APB by a serial-to-apb module included in our design through VHDL files and was already mentioned in Chapter 3. Keep in mind that this module necessitates hard-coded information about the baudrate and the operation frequency which results in separate FPGA configurations and bit-files[3] for altered operation frequencies even if no DCM-based reconfiguration would be necessary. This interface enabled communication between DUT and workstation to send input data and verify the correct calculation of the AES ciphertext.

The oscilloscope in use was a *Le Croy 1 GHz* oscilloscope which can take up to 2 Gigasamples per second (2 GS/s which translates to a interval of 0.5 nanoseconds between two samples) and is designated by the model number *LC584AM*. The aforementioned maximum sampling rate was used for almost all measurements. Exceptions were measurements which necessitated a very long recording window due to our clock-randomization countermeasure where it was reduced to 1 GS/s. The trigger delay was almost always set to zero with some exceptions where the trigger delay was set to wait for around three clock cycles which cut off most of the power consumption influenced by the trigger signal and started closer to the actual beginning of the AES_FR state.

A *Xilinx Virtex-II Pro (xc2vp10)* FPGA was used as the implementation platform. The FPGA is part of the SASEBO-G board [53]. No other IO pins than clock, rx, tx[4] and the trigger signal were used. The only exceptions were pins to LEDs for debugging purposes.

[2]The baudrate was halved to 4800 for measurements at an operation frequency of 1.5 MHz

[3]bit-files are used to configure Xilinx FPGAs

[4]rx and tx are standard denominations for serial communication pins. rx is for *r*eceiving and tx is usually designated for *t*ransmissions.

The serial communication is determined by the RS-232 standard which formally defines the signals used to communicate between a data terminal equipment (DTE) and a data circuit-terminating equipment (DCE) and was originally used for modems. Since the workstation used for communication measurements does not feature a RS-232 port a USB-to-RS-232 converter cable is used.

The SASEBO-G board which mounts our target platform has the following features:

- A printed circuit board (PCB) with 230 mm x 180 mm x 1.6 mm measurements, a FR-4 grade and eight layers.
- Two Xilinx Virtex-II Pro series FPGAs, one designated as the cryptographic FPGA (xc2vp7-fg456-5) the other designated as the control FPGA (xc2vp30-fg676-5). The FPGAs are connected through two 16-bit buses (data and address) and four control signals
- Two on-board oscillators with the frequency set to 24 MHz (for more details see further below)
- Power regulators which convert the 3.3 V supplied by an external power supply to 2.5 V, 1.8 V and 1.5 V for the FPGAs.
- Connectors for shunt resistors to apply power measurements on the FPGAs.
- Communication with external sources via an RS-232 connector to the control FPGA. Since the signals used by RS-232 have higher voltage levels (between -15 V and 15 V) than supported by the control FPGA a RS-232 level converter is also installed on the SASEBO board.

We used one differential probe to measure the voltage drop over the shunt resistor for statistical power analysis. The probe is a lab-fabricated differential probe with a bandwidth of 200 MHz and an amplification of 13. The differential probe measured the voltage drop on a 1.5 Ω resistor in the 1.5 VDD path to the FPGA.

The FPGA platform was programmed with a *Xilinx Platform Cable* and supplied by a *BST PSM 2/5A* with a supply voltage of 3.3 V. This power supply was also used to power the lab-fabricated differential probe.

A simplified version of the measurement setup can be seen in Figure 5.2. At last, the operation frequency was provided by an *Epson 24 MHz on-board* oscillator and by a *digimess FG 100* function generator which was used to generate operation frequencies close to 30 MHz. A short overview of all the hardware involved is given in Table 5.1.

5.2 Side-Channel Analysis at Varying Operating Frequencies

We conducted our first measurements on our implementation with an operating frequency set to 24 MHz and recorded ten thousand (10k) traces. We chose a recording window to observe 6 rounds of the AES encryption.

After the measurements we started applying the DPA attack (which was discussed in more detail in Section 4.2) on the recorded traces. Figure 5.3 shows the DPA result after we applied our HD power model for the first key byte according to Equation (4.5). The

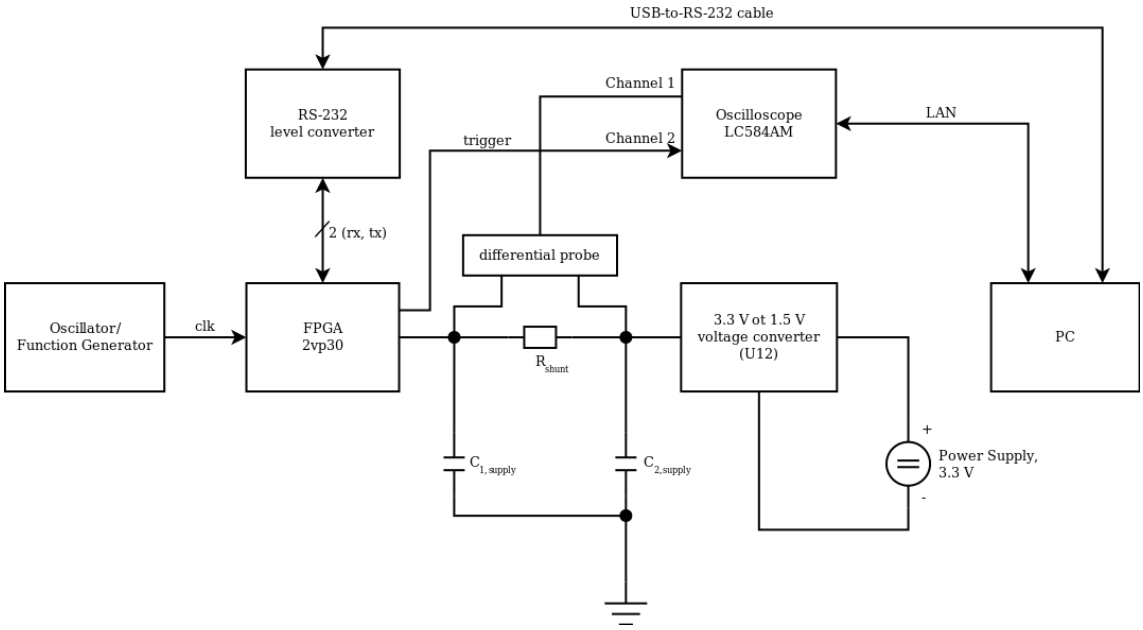

Figure 5.2: Overview of the measurement setup

Device	Name
FPGA	Xilinx Virtex-II Pro xc2vp10
FPGA Board	SASEBO-G
Serial Interface	USB-to-Serial, RS-232 standard
Workstation	Windows 7 and 8, MATLAB, Xilinx iMPACT 10.1
Oscilloscope	Le Croy LC584AM
Differential Probe	lab-fabricated 200 MHz probe with an amplification factor of 13
Power Supply	BST PSM 2/5A
Frequency Generators	Epson 24 MHz on-board, digimess FG 100
FPGA-Programming	Xilinx USB Platform Cable

Table 5.1: Table of used hardware for the DPA measurement

x-axis shows the simplified states of our FSM for the first three rounds of AES. The first observations which were made are the following:

1. The correct key hypothesis with value *237* (shown in black in Figure 5.3 is clearly distinguishable from the other wrong 255 key-hypotheses. This is an indicator that the HD power-model fits well.

2. The correlation value for the correct key hypothesis at the beginning of the ADD state is mall compared to the value at the beginning of the SR state.

3. The correct correlation curve stays distinguishable up until the beginning of the third ADD state.

4. The correlation values increase in four distinguishable steps although it was outlined that it should have 16 occurrences in Table 4.1.

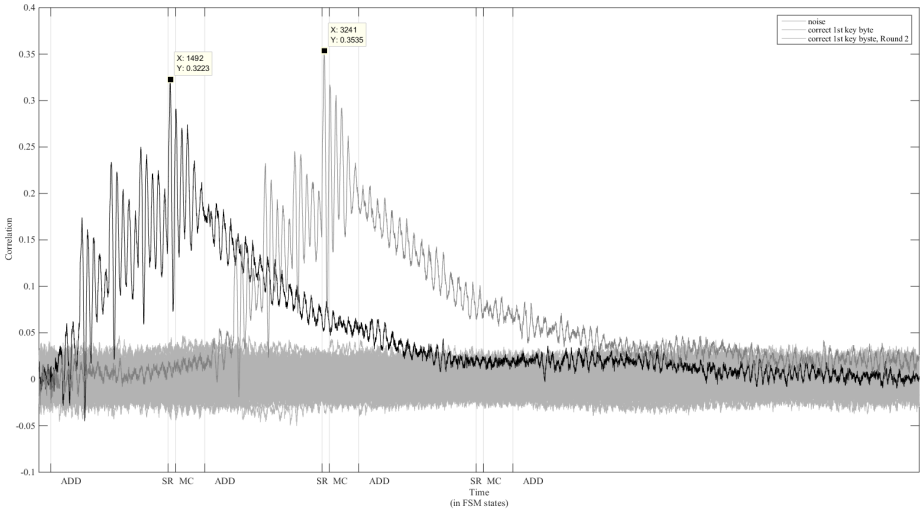

Figure 5.3: Correlation curve for the first key byte at 24 MHz (ten thousand traces)

5. Before the high positive peak, each of the four distinguishable peaks show a strong negative peak.

First, we tried to determine the reasons why the correlation after the first iterations of the AES states that remains distinguishable into the following AES states. Possible reasons are the result of the Xilinx optimization tools, other design altering effects or other physical effects (see below). So another DPA was conducted on the Sbox output of the next AES round. This is solely a profiling step and requires knowledge of all AES key-bytes to determine the possible values of the second ADD state. The result for the first key-byte of the second round can be seen in grey in Figure 5.3). As can be seen in this figure, the correlation curves of the first and second Sbox calculation rounds (seen marked as *ADD* in Figure 5.3) intersected. Another step to get clarity on this matter was a profiling step on the plaintext bytes. All of these (correlation) curves shared the same behavior regarding their atypical long distinguishability after the processed data the chosen power-model is based on has been overwritten by subsequent steps of the AES encryption.

An explanation for this was given by Stefan Mangard et al in [2] where they discussed the relations of switching noise, bandwidth and clock frequencies. Since integrated circuit designs commonly utilize supply capacitances to guarantee a stable power supply, these capacitances add parasitics to the power-supply chain. So when these capacitances are drained of their load the power consumption will rise in order to reapply the maximum load onto the capacitances. If the operating frequency of the clock is set faster than the capacitances can be refilled the measured power consumption of subsequent clock cycles overlaps and becomes continuous instead of only using power at clock based events. As an example see the rising power consumption curves during the ADD state in the plots of Figure 5.15.

This fast operating frequency causes the power consumptions of individual clock cycles to overlap, resulting in a higher amount of switching noise measured in each cycle but also leaks relevant information into subsequent cycles. Since the implemented design has

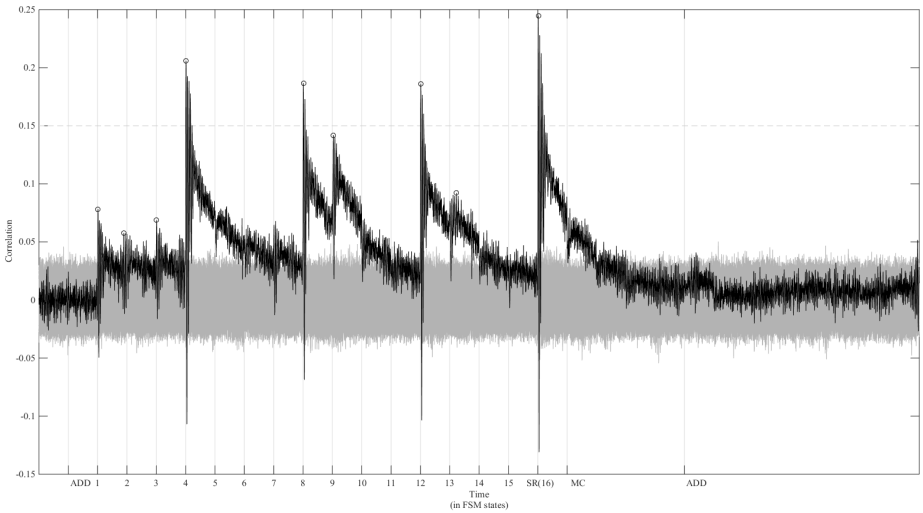

Figure 5.4: Correlation curve for the first key byte at 1.5 MHz (ten thousand traces)

multiple occurrences of the same characteristic HD, it can be argued that each occurrence adds to the characteristic power consumption of each recorded sample. This characteristic power information is "stored" in the power consumption until enough load is restored to the supply capacitance or enough unrelated power consumption has occurred which makes the characteristic power consumption unsignificant. For a deeper comparison on this matter see Section 5.2.3.

In order to get a better understanding of the occurrences related to the chosen power model, additional measurements on AES encryptions with the implemented design were taken with lowered operation frequencies. Figure 5.4 shows the correlation result for the first key byte at an operation frequency of 1.5 MHz. The correlation at an operating frequency of 1.5 MHz also needs a significant amount of time (see the end of the correlation curve in Figure 5.4) to become indistinguishable. What is interesting about this result are the qualitative differences each occurrence of the power model has on on the correlation value.

The correlation curve seen in Figure 5.4 should feature 16 individual peaks of equal value in the *ADD* state as was summarized in Table 4.1. What can be observed in Figure 5.4 on the other hand are about 9 peaks of varying correlation value in an interval of 16 clock cycles (see cycles 1, 2, 3, 4, 8, 9, 12, 13 and 16 in Figure 5.4). According to the design the HD should occur 16 times because of the use of the register in the state module as a shift register.

There are four peaks with correlation values above 0.15 (cycles 4, 8, 12, and 16), these peaks occur when the HD applies to a writing process on the first column of the registers inside of the state module. These registers are connected to the input signals of the *mixcolumns* module which increases the fanout of these modules considerably.

The same can be observed in Figure 5.5 which displays the correlation curve for the tenth key byte. Again, according to Table 4.1 the correlation curve should have 7 peaks in the ADD state and 3 peaks in the MC state. As was observed with the result for the first key byte the correlation has the highest values when the attacked HD passes through a register of the first column. One additional occurrence appears one cycle before the

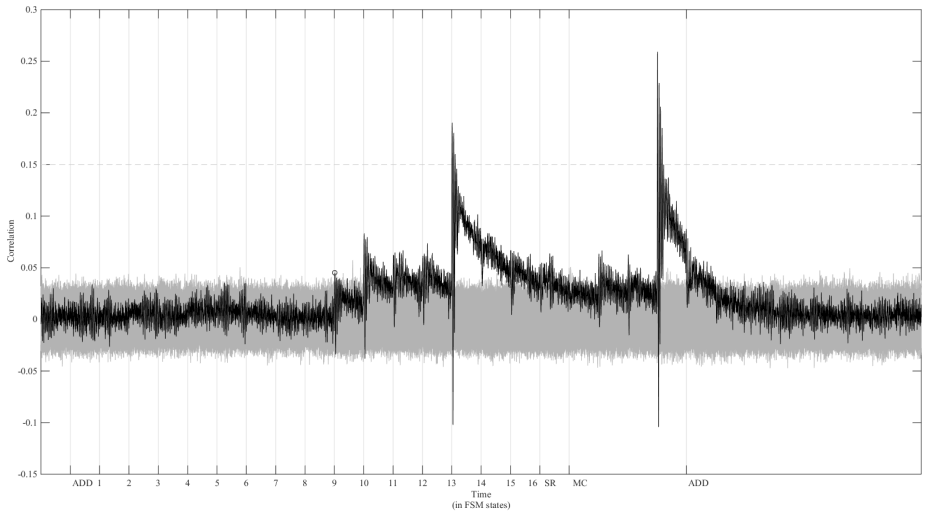

Figure 5.5: Correlation curve for the tenth key byte at 1.5 MHz (ten thousand traces)

expected occurrences and can be observed for every key byte beginning with the second. This can probably be attributed to the combinational switching of the Sbox module which produces power according to our HD one cycle before the corresponding value is stored inside of the state module.

Row\Column	1	2	3	4
1	17.12.12.10.9.8.9.9	0.0.0.0.0.0.0.0	0.0.0.0.0.0.0.0	1.1.1.1.1.1.1.1
2	15.9.9.8.8.9.8.9	0.0.0.0.0.0.0.0	0.0.0.0.0.0.0.0	1.1.1.1.1.1.1.1
3	18.12.12.9.8.9.8.8	2.2.2.2.2.3.2.3	2.2.2.2.2.2.2.2	2.2.2.2.2.2.2.2
4	16.10.10.9.8.8.8.10	2.2.2.2.2.2.2.2	3.3.3.3.5.5.3.3	2.2.2.2.2.2.2.2

Table 5.2: Fanout of the registers in the state module

The correlation behavior relates to the fanout of the place-and-route result of the design for the FPGA which can be observed bit-wise in Table 5.2. This information was obtained through the netlists provided by the Xilinx FPGA Editor. The first (from the left) column has significantly higher fanout due to its connection with the mixcolumns module. Byte [1, 1] is connected to the S-Box module which lengthened the connections originating from that byte and increased the fanout of this register, probably causing the biggest correlation-peak when the power-model applies to this register. A similar observation of increased correlation values in relation to the additional fanout due to the connection to the mixcolumns module was made by [56] where a similar architecture based on the proposal of [6] was attacked. The reduced fanout of zero for the register bytes at [2, 2], [2, 3], [3, 2] and [3, 3] occurs due to the optimization of the Xilinx tools where the slices associated with these registers and also the registers of the first column of the affected rows are contracted into 3-bit shift registers.

When we now consider all key bytes we can always observe high correlation peaks when the attacked byte passes through one of the high-fanout registers of the first column. On the other hand if we apply our power-model of Equation (4.6) to the last key-byte the

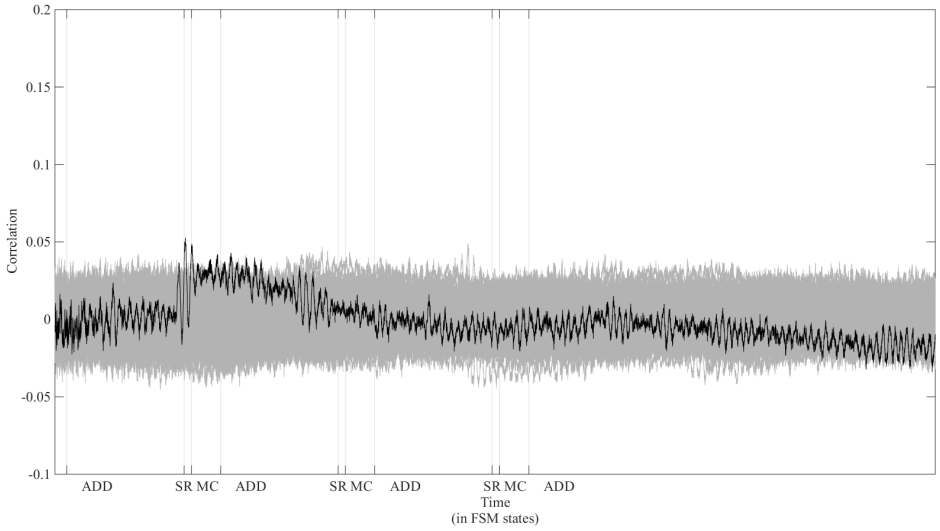

Figure 5.6: Correlation curve for the 16th key byte at 24 MHz

HD never applies to a register with high fanout. This can be seen in Figure 5.6 for our measurements with an operation frequency of 24 MHz where the correlation for the correct key value barely manages to reach the a value of 0.05. This low correlation also holds true for all other tested operation frequencies especially for 48 MHz where the correlation does not reach a distinguishable level with a measurement effort of 10k traces due to the stronger switching noise.

Since we have shown that HDs that apply to the first column of our register state deliver better correlation results one can change the attack on the last key byte to something more effective as was discussed earlier for Equation (4.7) where not the HD of subsequent bytes during the shifting of the ADD state is attacked but the register changes that occur during the SR and MC states. This enables the attack on the last key byte in relation to the first register of the last row and yields a better correlation result which can be observed in Figure 5.7 where the maximum correlation result is almost four times the value the attack yields with an unaltered power-model. This alteration of the attack was successful with all measured operation frequencies.

5.2.1 DPA with Added Low-Pass Filters

Since we could observe frequencies beyond our set operation frequency, measurements using the bandwidth limitation of the oscilloscope were conducted but did not change the correlation result in a significant way. After this low-pass filters with varying cut-off points were applied via MATLAB to our regular measurements. These low-pass filters are single-pole which behave similar to moving average filters discussed by Stefan Mangard et al. in [2]. The results of applying these filters were an increased smoothing effect on the correlation and overall lower correlation values. We also applied a bandwidth limitation of 25 MHz via the oscilloscope for one measurement of 10k traces which had a comparable effect as applying a low-pass filter after the measurements.

To evaluate the perceived amount of lower correlation for the correct key byte we analyzed the success rates for each low-pass filter. The results of these experiments can

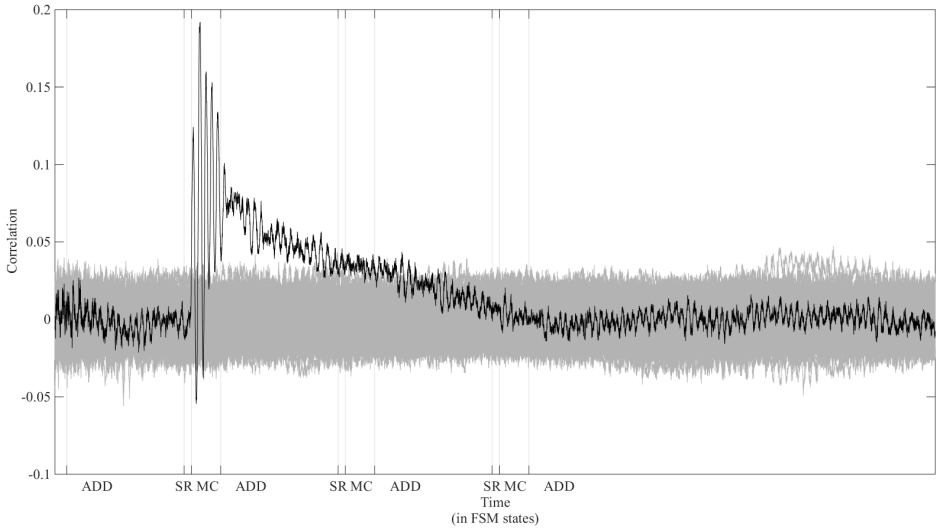

Figure 5.7: Correlation curve for the 16th key byte at 24 MHz with the improved power model

be seen in Figure 5.8. The success rates were calculated through the following means: For every amount of considered traces we selected ten sets of randomly chosen traces after the low-pass filter was applied to them and evaluated whether the correlation for the correct key byte surpassed the maximum noise-level of the given set. If this criterion was met the set increases the success rate of the considered amount of traces by 0.1 with a maximum of 1 if each set managed to fulfill the criterion.

With that in mind it can be determined that around 400 traces are enough to determine the first key byte when using our implementation with a success rate of 1. This is very low when considering that the almost identical implementation of AES by [6] necessitates 30k traces to yield a successful result. This result of [6] was achieved with no active countermeasures and an almost identical power model on the same SASEBO board (although on the smaller FPGA).

When considering the effects of added low-pass filters to the calculation, the results are clear. Applying these filters has no positive effect on the correlation result, to the contrary the absolute correlation values for the correct key hypothesis are reduced and the success rate of the DPA attacks are lowered to the point where additional traces are needed.

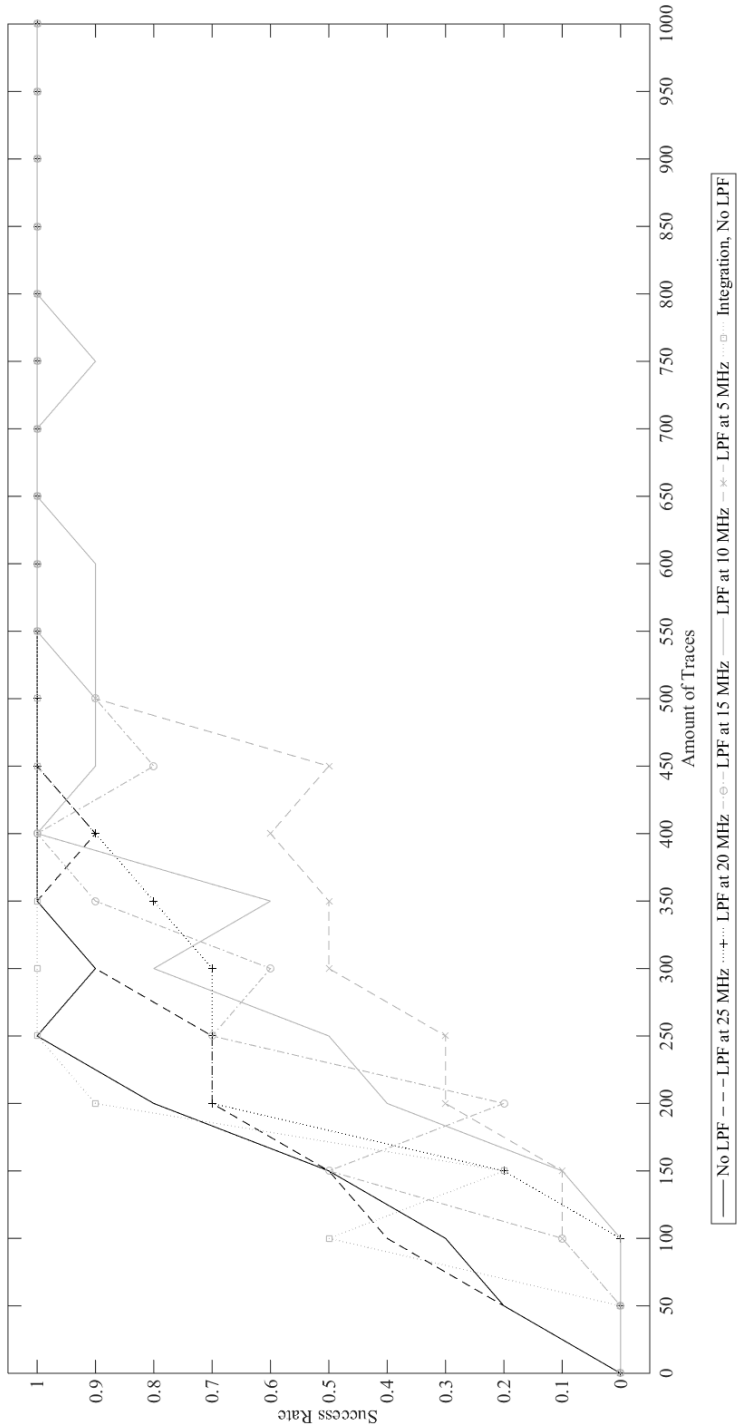

Figure 5.8: Success-rates of with random sets of traces with different low-pass filters

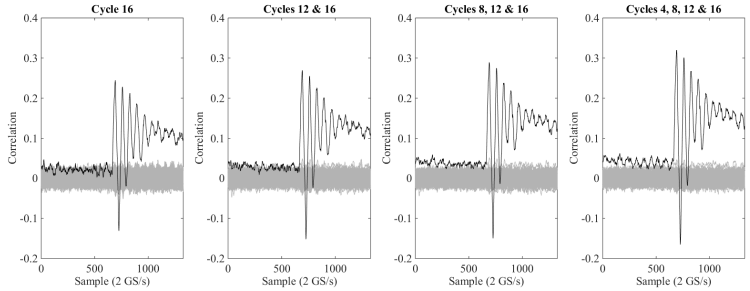

Figure 5.9: Correlation curve for the first key byte at 1.5 MHz using the integration method

1.5 MHz		32 MHz	
Cycles	Correlation	Cycles	Correlation
16	*0.2447*	**16**	***0.2779***
15, 16	0.1717	15, 16	0.2250
14, 16	0.1714	14, 16	0.2362
13, 16	0.2035	13, 16	0.2456
12, 16	*0.2696*	12, 16	0.2583
11, 12, 16	0.2170	11, 16	0.2167
10, 12, 16	0.2202	10, 16	0.2286
9, 12, 16	0.2501	9, 16	0.2403
8, 12, 16	*0.2893*	8, 16	0.2723
7, 8, 12, 16	0.2609	7, 16	0.2294
6, 8, 12, 16	0.2597	6, 16	0.2402
5, 8, 12, 16	0.2668	5, 16	0.2686
4, 8, 12, 16	***0.3197***	4, 16	0.2216
3, 4, 8, 12, 16	0.2946	3, 16	0.2272
2, 4, 8, 12, 16	0.2899	2, 16	0.2391
1, 4, 8, 12, 16	0.3073	1, 16	0.1901

Table 5.3: Integration attempts at 1.5 MHz and 32 MHz operating frequency (10k traces).

5.2.2 Integration of Multiple Clock Cycles

Stefan Mangard et al. proposed a way to improve DPA results in [2]. This method adds multiple clock cycles calculating the same intermediary result on top of each other and is called *Integration*. We applied this method to measurements taken the operating frequency set to 1.5 MHz and 32 MHz with 10k of traces each and applied the power model of the first key byte. The algorithm starts with the clock cycle containing the highest correlation peak (cycle 16) and integrates the clock cycles which improve the result until all relevant clock-cycles have been considered. The results can be seen in Table 5.3.

As can be seen from the table the results lead to a considerable increase of correlation values at an an operation frequency of 1.5 MHz, but only the clock cycles 4, 8 and 12 improve the result at 1.5 MHz and while the method has no positive effect on the correlation value at an operating frequency of 32 MHz. This is probably due to the differing power voltage levels which are integrated, which are more similar with an operation frequency of 1.5 MHz at the clock cycles where the result is improved. This integration method also increased the success rate as can be seen in Figure 5.8.

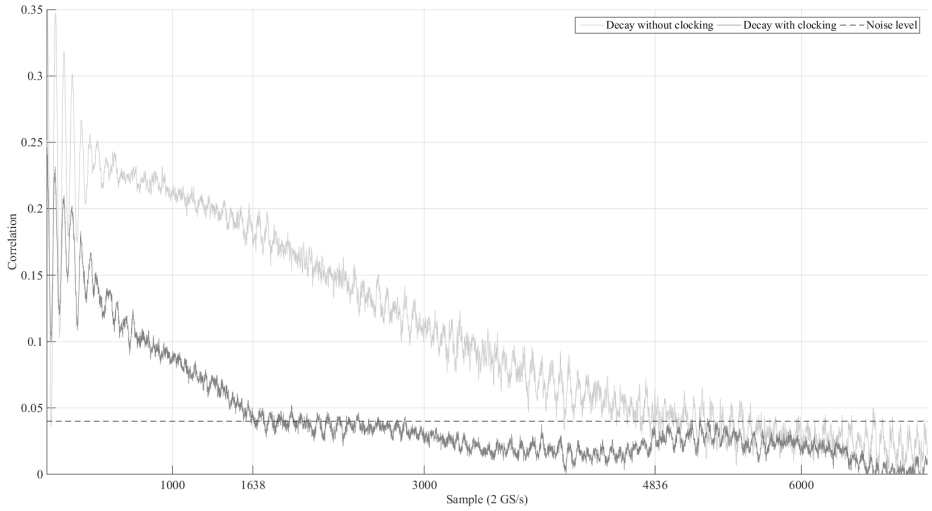

Figure 5.10: Comparison between untampered correlation decay and a decay influenced by switching registers at 48 MHz

5.2.3 The Correlation Decay

As was discussed earlier, the correlation result for the correct key-hypothesis remains distinguishable from the wrong key-hypotheses for multiple clock cycles or even multiple AES rounds for measurements with the operating frequencies higher than 12 MHz. We concluded that this aspect of the correlation was in part caused by the supply capacitances provided by the board and the FPGA itself as was also discussed by Stefan Mangard et al. in [2].

As a next step the influence of the switching noise on this phenomenon was examined. Due to the measurements taken with different operation frequencies we concluded that the added switching noise plays a factor in the correlation decay of the DPA. We also concluded that the more switching occurs during the distinguishable time-window of the correct key-byte the shorter the distinguishable time-window becomes.

Further another measurement of 10k traces was taken on a design which replaced the AES states after the first ADD state with a wait state halting the main registers of the implementation for multiple clock cycles. This measurement simulates an correlation decay without switching noise. The measurement was taken at an operation frequency of 48 MHz to compare it to the occurrence of the highest switching noise achievable with the measurement setup.

This comparison is displayed in Figure 5.10 over a time window of 7 000 samples (3 500 ns). The correlation of the halted implementation remains distinguishable significantly longer than it does with the standard implementation due to the lack of switching noise. The halted implementation features higher correlation values due to a smaller footprint on the FPGA after the Xilinx tools optimizing unused resources, decreasing the overall usage of the FPGA resources. This further decreased the DPA relevant noise.

The hence uninfluenced correlation decay is set at about *0.042* per 500 ns. When switching is active the correlation decays with *0.169* per 500 ns which is about four times the value-loss of the halted implementation. We can conclude that the switching of the

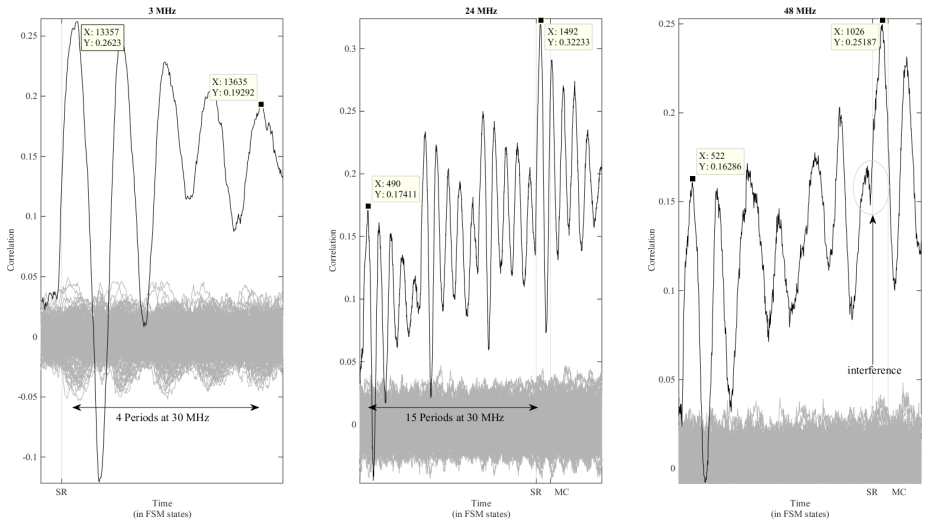

Figure 5.11: Correlation results for the first key byte at different operating frequencies. (10k traces)

main registers causes a faster correlation decay at an operation frequency of 48 MHz.

5.3 About the Characteristic Frequency of the Xilinx Virtex-II Pro FPGA

While conducting DPA attacks on the AES implementation similar to Moradi et al.'s proposal [6] on varying operating frequencies (1.5, 3, 6, 12, 24 and 48 MHz) we always observed a superimposed frequency of about 30 MHz on the correlation for the correct key hypothesis. Some examples of this phenomenon are displayed in Figure 5.11. At a sampling rate of 2 GS/s a period of 66.6 samples corresponds to a frequency of 30 MHz. The decaying oscillation shown in the left plot of Figure 5.11 has a frequency of about 29.6 MHz, the oscillation is held in most observed cases for 5 succeeding peaks which is then deteriorated by noise but still detectable. This deterioration does not occur if new correlation events (in our case the shifting of a register reiterating the same HD) occur. There is a chance that these events will break the phase of the correlation curve in the correlation plot for an operating frequency of 48 MHz as can be seen at certain sample points in the right correlation plot of Figure 5.11.

Figure 5.11 also shows part of correlation curve for the first key byte at an operation frequency of 24 MHz. Take note that the correlations still holds its frequency (the measurement window of 15 oscillation periods suggests a frequency of 29.9 MHz) and does not feature any breaks in its oscillation phase. We have determined earlier in this chapter that the strength of a correlation event is determined partly through the fanout of its affected output signals. In the case of this correlation curve the timing of the peaks where the value of the correlation is increased coincides with the correlation events connected to the biggest fanout values.

On the other hand the more minor correlation events between these peaks are not represented in the correlation result. Note that instead of a peak for each clock cycle

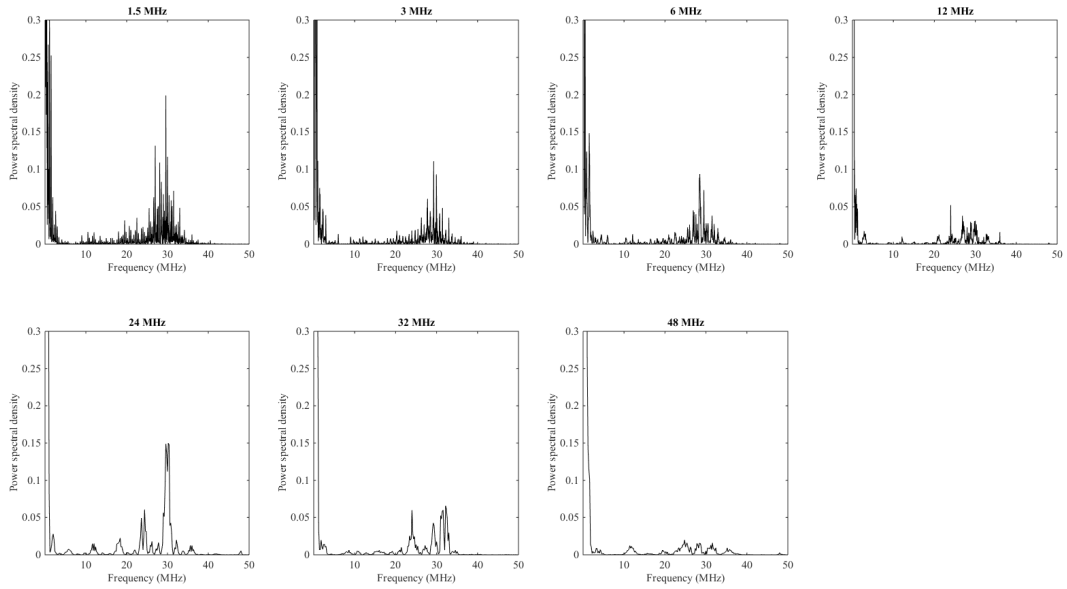

Figure 5.12: FFT result of the correct correlation curve of the first byte at multiple operating frequencies.

the correlation peaks with its characteristic frequency. Assuming that this frequency is 30 MHz the relationship between an operation frequency of 24 MHz and the characteristic frequency is 4/5 which results in an additional peak every four clock cycles (four clock cycles between each of the four major peaks).

When the operating frequency is doubled to 48 MHz (by employing a DCM) the relation between operating frequency and characteristic frequency is now 8/5. The right plot of Figure 5.11 shows the correlation result for the correct key hypothesis. We can see that the correlation still holds to its characteristic frequency albeit with major interferences regarding its phase. Note that there are also 12 clock cycles between the added data tips as there are for the plot at 24 MHz operating frequency.

We also conducted spectral analyses on the result vectors of the correct key hypothesis. Figure 5.12 holds multiple spectral analyses of DPA results of measurements taken with a determined set of operating frequencies. These spectral analyses show differing but noticeable amounts of spectral density around 30 MHz.

This characteristic frequency is probably caused by the resistances, capacitances and inductances of the circuit. So further measurements of the circuit were taken utilizing additional supply capacitors (10 μF, 220 μF or 1500 μF) and alternative shunt resistors (1.5 Ω, 1.8 Ω or 4.8 Ω). These alternative measurements setups were connected and examined for deviations in power consumption and altered correlation results.

The altered shunt resistors changed the measured voltage drop on the resistor but did not influence the correlation result. Adding additional capacitors to the FPGA also did not change the characteristic frequency but did result in altered voltage drops and correlation results as can be seen in Figure 5.13. This led us to believe that the characteristic frequency is solely caused by the inductances and capacitances of the FPGA and not related to the power-supply paths of the SASEBO board.

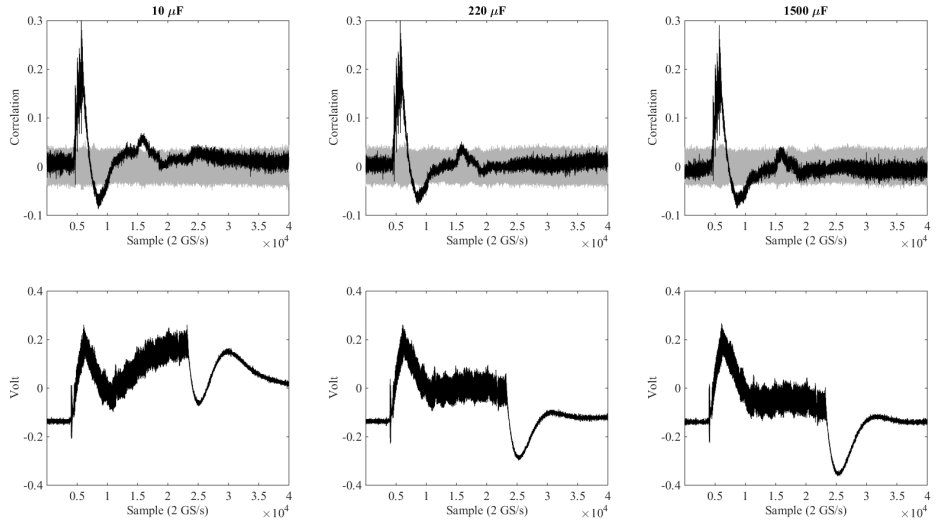

Figure 5.13: Correlation results and power-consumption plots with altered capacitances. (10k traces)

To further deepen our understanding of the characteristic frequency we approximated this frequency with the use of another DCM[5]and managed to lock an operating frequency of 32 MHz and compared the correlation results of multiple key bytes to one another.

To get a better understanding on this phenomenon we call the characteristic frequency one could take a closer look at the actual power traces. Figure 5.15 shows one thousand power traces for each operating frequency. The power traces of 48 MHz have no observable frequency. This indicates that the operating frequency is filtered/dampened by the capacitance of the FPGA acting as a low-pass filter. This capacitance probably also defines the characteristic frequency. The characteristic frequency of about 30 MHz is observable in all of the other power traces of 1.5 MHz to 32 MHz after a positive edge of the operating clock occurred.

We can also observe that the power consumption reaches a maximum at 30 MHz. In other measurements utilizing a function generator we determined that the actual maximum in power consumption is reached with the operating frequency set just below 30 MHz.

This is similar to reaching something akin to a resonance state which was hinted at by the characteristic frequency we observed at downward slopes of the correlation curves and their corresponding power traces. What is of interest in the context of this thesis are the effects this phenomenon has on the correlation curve due to our focus on security-aspects of the exposed side-channel.

We theorize that two possible effects could exist: First, an improvement of the signal-to-noise-ratio due to the higher exposure - read: higher min-to-max distance - of each clock cycle. Second, a faster correlation decay due to higher power consumption oscillation which could lead to a smaller “storage” window in the switching noise. We will align these possible effects with our observations and evaluate if these claims have any merit or further practical use.

The next logical step is headed at the evaluation of the correlation curves of correct key

[5]one DCM doubled the input frequency to 48 MHz, another divided the signal by 1.5 to 32 MHz.

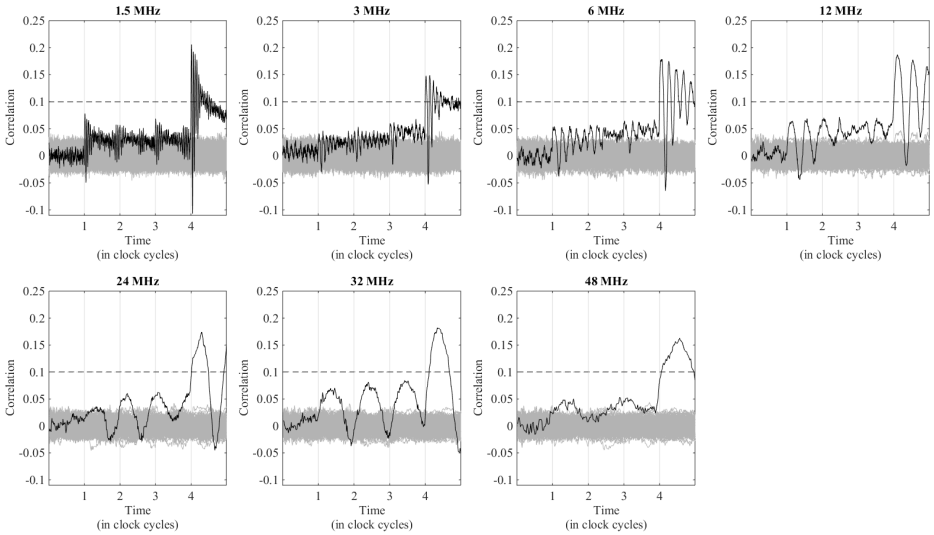

Figure 5.14: Correlation result for the first byte at multiple operating frequencies. (10k traces)

hypotheses for several key-bytes (the first, the second and the last). We should be experiencing alterations of the correlation behavior at different operating frequencies. First, the correlation should start to exhibit phase disturbances once an operating frequency above the characteristic frequency is reached. At this point, impulses sent by the operating clock can no longer be absorbed at a fast enough frequency. this leads to the reduction in power consumption at operating frequencies 31 MHz and above. Second, at operating frequencies close to the characteristic frequency the high-fanout registers should have a smaller impact on the correlation curve. This is due to the reduced amount of switching noise into subsequent registers. Low-fanout registers should have a higher impact on the correlation curve since their respective signal-to-noise-ratio has been increased when compared to the impact of high-fanout registers have at operating frequencies farther below and above the characteristic frequency.

The plots in Figure 5.14 show our correlation results at the first four clock cycles at which our power model applies for the first key byte at operating frequencies 1.5, 3, 6, 12, 24, 32 and 48 MHz with 10 000 recorded traces. Observations are: the phase of the correlation curves at 32 MHz and 48 MHz feature phase disturbances while correlation curves originating from measurements with lower operation frequencies do not. Also, measurements at 32 MHz result in the fastest correlation decay resulting in lower total correct correlation values when compared with 24 MHz and 48 MHz. This can be explained with the abridged influence high-fanout registers have on the switching noise.

When considering the results for the second byte (see Figure 5.16) these observations still hold true. Here the correct correlation at the first clock cycle does not originate from a register but from the combinational output of the Sbox module. The first clock cycles are visibly more masked than other clock cycles and when compared to the correlation curve at an operating frequency of 32 MHz.

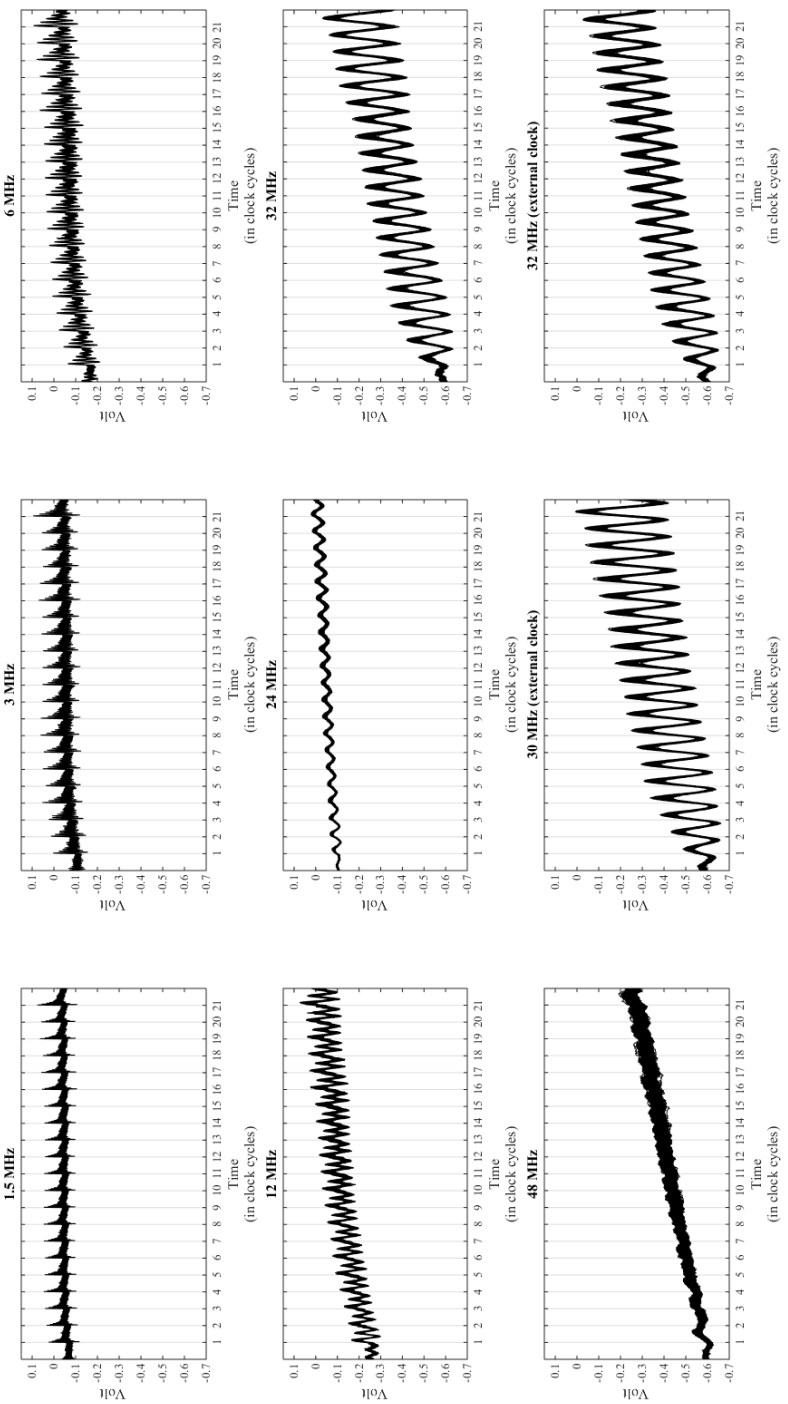

Figure 5.15: Power consumption for all cycles of the first AES round

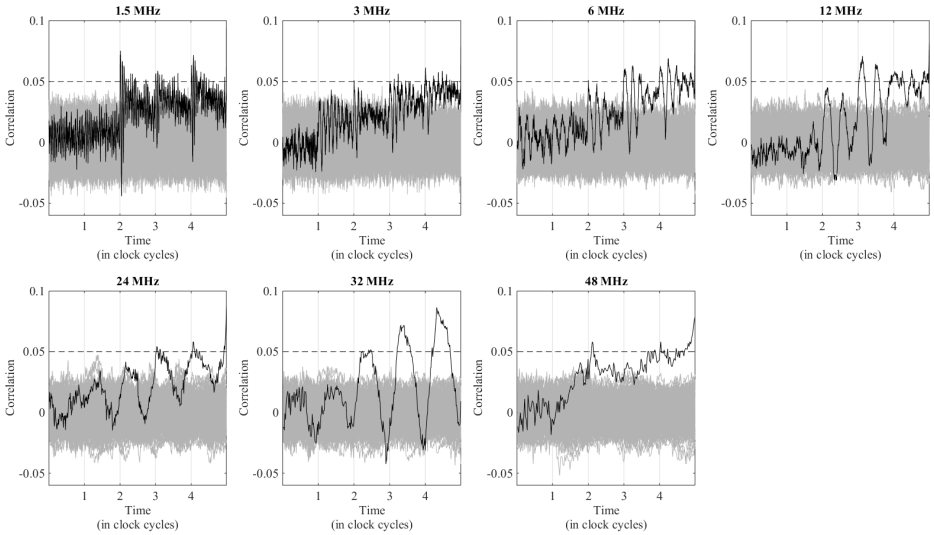

Figure 5.16: Correlation result for the second byte at multiple operating frequencies. (10k traces)

This effect is the most pronounced when considering our original power model for the last byte (Equation (4.6)) which does not feature a HD occurrence at a high-fanout register. Naturally this is the only key-byte where the correlation curve of 32 MHz surpasses other surrounding operating frequencies (namely 24 MHz and 48 MHz[6]) in total correlation value since the influence of a high-fanout register does not occur. The correlation results can be observed in Figure 5.17.

Also, when looking at a larger time-frame for this correlation curve (see Figure 5.18) we can see that the correlation decays faster than at other operating frequencies while oscillating with a significantly bigger amplitude. We have now argued for two effects occurring at the same time when the operating frequency approximates the characteristic frequency. The effect of a faster correlation decay is weakening DPA attacks that rely on multiple occurences of the same HD but is negligible for implementations that do not have multiple HD occurrences stacking on the switching noise. The effect of a seemingly better signal-to-noise-ratio can be beneficial to DPA attacks where the noise can not be reduced by lowering the operation frequency but approximated to the characteristic frequency.

But due to the singular experience with this FPGA it is hard to asses whether this effect is of use to other attacks on other FPGA models. We would like to suggest this open question as a possibility for future work of research.

5.4 Summary

In this chapter we have described our measurement setup for DPA attacks and discussed some evaluation methods regarding side-channel resistance. With these methods we have shown that the vulnerability of the AES implementation to DPA attacks is much higher compared to the findings of [6]. We require approximatly 400 traces for a success rate of 1 while in [6] 30k traces are required for the same result. This is a discrepancy of the factor

[6] 32 MHz also surpasses 30 MHz

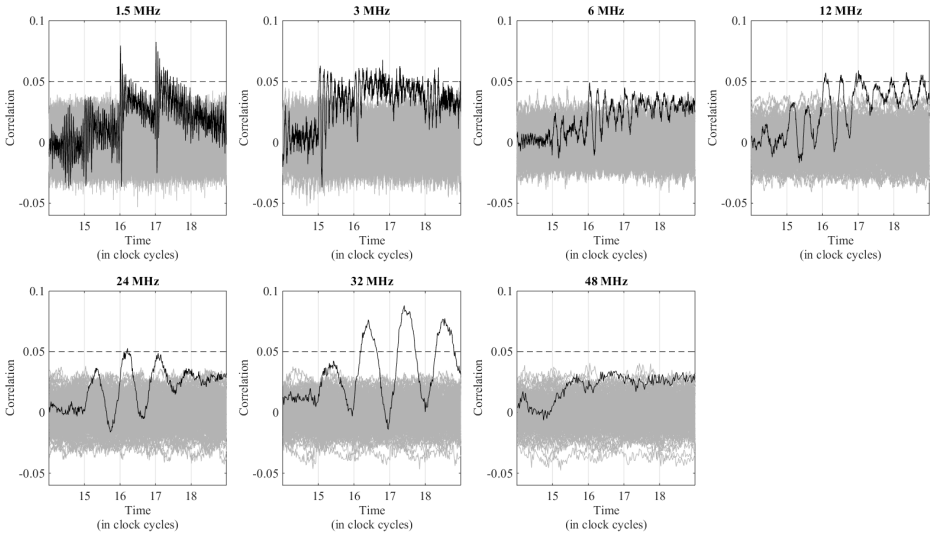

Figure 5.17: Correlation result for the last byte at multiple operating frequencies. (10k traces)

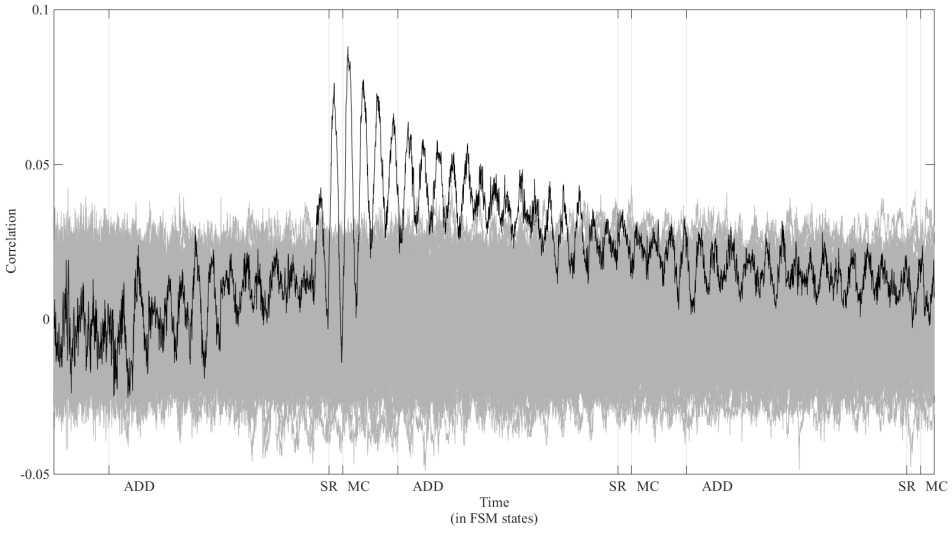

Figure 5.18: Correlation result for the last byte at an operating frequency of 32 MHz. (10k traces)

of almost 100. The attack on the AES implementation is essential for retrieving the key used in ALE encryptions.

We have explained different aspects of the behavior of the correct key-byte correlation in regards to the fanout of the implemented register, the noise originating from the trigger signal and the influence of switching noise. We could show a relation between the total maximum correlation value and the fanout of registers, how output switching influences the correlation result and how switching noise affects the decay of the correlation curve.

We have also shown the significance of a phenomenon we call the characteristic frequency and its sole dependence on the FPGA model in use. We have shown that this phenomenon can be exploited to achieve slightly better correlation results when no high-fanout registers or methods to slow the operation frequency (to avoid switching noise) are available. We have also shown how to improve a power-model when tailoring it to high-fanout registers (and probably nets).

Next we will look at the results yielded after implementing FPGA-specific countermeasures and the statistical evaluation technique known as t-test.

Chapter 6

Side-Channel Analysis with Countermeasures

In this chapter we take a look at the DPA results yielded after implementing the countermeasures described in Section 3.3. The two implemented countermeasures hindered the success of the DPA to varying degrees. This can be measured by an increased effort regarding the amount of traces required for a successful DPA attack.

The clock-randomization countermeasure proved to be suitable for a relatively easy implementation on varying operating frequencies while the implementation scale of the short-circuit countermeasure was hindered by the lengthy process required to alternate existing designs. Since the clock-randomization countermeasure was easy to implement and yielded promising results we decided to alter it by mixing two different operating frequencies together by employing an additional DCM. The results of this alteration are discussed in Section 6.1.4, where more details on the further success of this countermeasure are described.

The short-circuit countermeasure proved to be unsuitable for masking the relevant power information in our design at an operating frequency of 24 MHz since the power information is "stored" and added to for a relatively long time due to the effect of emptying the supply capacitances as was also described by Stefan Mangard et al. in [2]. This makes the countermeasure ineffective for later clock cycles (after the fourth clock cycle) since the noise it generates is insufficient. The short-circuit countermeasure also proved to be unsuitable for designs at lower operating frequencies as the additional measurement effort this countermeasure necessitates is much lower than was described by [4]. A deeper discussion of the results can be found in Section 6.2. But first, this chapter will discuss the results of the basic clock-randomization countermeasure, which was also evaluated with DPA attacks on different operating frequencies.

For further information on the measurement setup please refer to Section 5.1 since no major alterations were made for conducting measurements on the designs with countermeasures.

6.1 Results on the Clock-Randomization Countermeasure

This section will discuss our evaluations of various aspects of the clock-randomization countermeasure (CR[1]). First we discuss our choice of the clock-input of our PRNG module and its effects on the countermeasure in Section 6.1.1. Then a discussion to the effects of the clock-randomization countermeasure on DPA attacks is given in Section 6.1.2. Here we also discuss the results of several DPA attacks which were conducted on implementations of CR2, CR4 and CR8. Section 6.1.3 gives a short overview how alterations of the clock-randomization countermeasure were implemented while Section 6.1.4 analyzes the effects these alterations had on DPA attacks. A summary of these and other findings on this countermeasure is given in Section 6.1.5. Finally, we introduce another way to evaluate the leakage produced by various implementations of the clock-randomization countermeasure by applying the t-test on various measurement sets taken on all major implementations of this countermeasure in Section 6.1.6.

6.1.1 Choosing the Input of the PRNG

The clock-randomization countermeasure was at first implemented as was discussed in Section 3.3.2. At this section the clock input of the PRNG module was mentioned to be the unaltered clock provided by the on-board oscillator. But originally we considered two clock signals to drive the PRNG. First, we considered the aforementioned unaltered clock signal. This clock signal is supplied by the on-board oscillator and is used on the DCMs to produce the phase-sifted clock signals. The other option we considered was the result of the clock randomization itself. Which signal would suit our need for security against DPA attacks better did not seem clear and the decision made by [4] was not documented. So we chose to evaluate which signal would be a better solution. The results of this evaluation are summarized in this section.

The decision to use the input clock instead of the randomized clock was made after doing some preliminary measurements. These measurements were taken after altering the design to have the clock randomization always on and measuring the randomized clock which was routed on one of the output pins. We measured clock traces with the clock randomization set to use 2, 4 or 8 phase shifted clocks and altered the input of the RNG between the input clock to the randomized clock.

Each of these design variations was measured 1000 times with the measurement set to start after the first positive clock edge. These measurements were subsequently used to analyze the distribution behavior of the randomized clock periods. The plots inside of Figure 6.1 show histograms for the distribution of the randomized clock period for implementations of CR2, CR4 and CR8. It can be observed that no design variant causes the clock period to grow shorter than the unaltered clock of 24 MHz[2] while all variants cause the potential clock periods to increase. This lack of shorter clock periods is caused by the glitch-suppressing properties of the utilized clock multiplexers which are supplied by the FPGA.

[1] Implementations of the clock-randomization countermeasure which utilize two, four or eight phase-shifted clock signals are abbreviated with CR2, CR4 or CR8 respectively.

[2] 24 MHz equals 41.6 nanoseconds

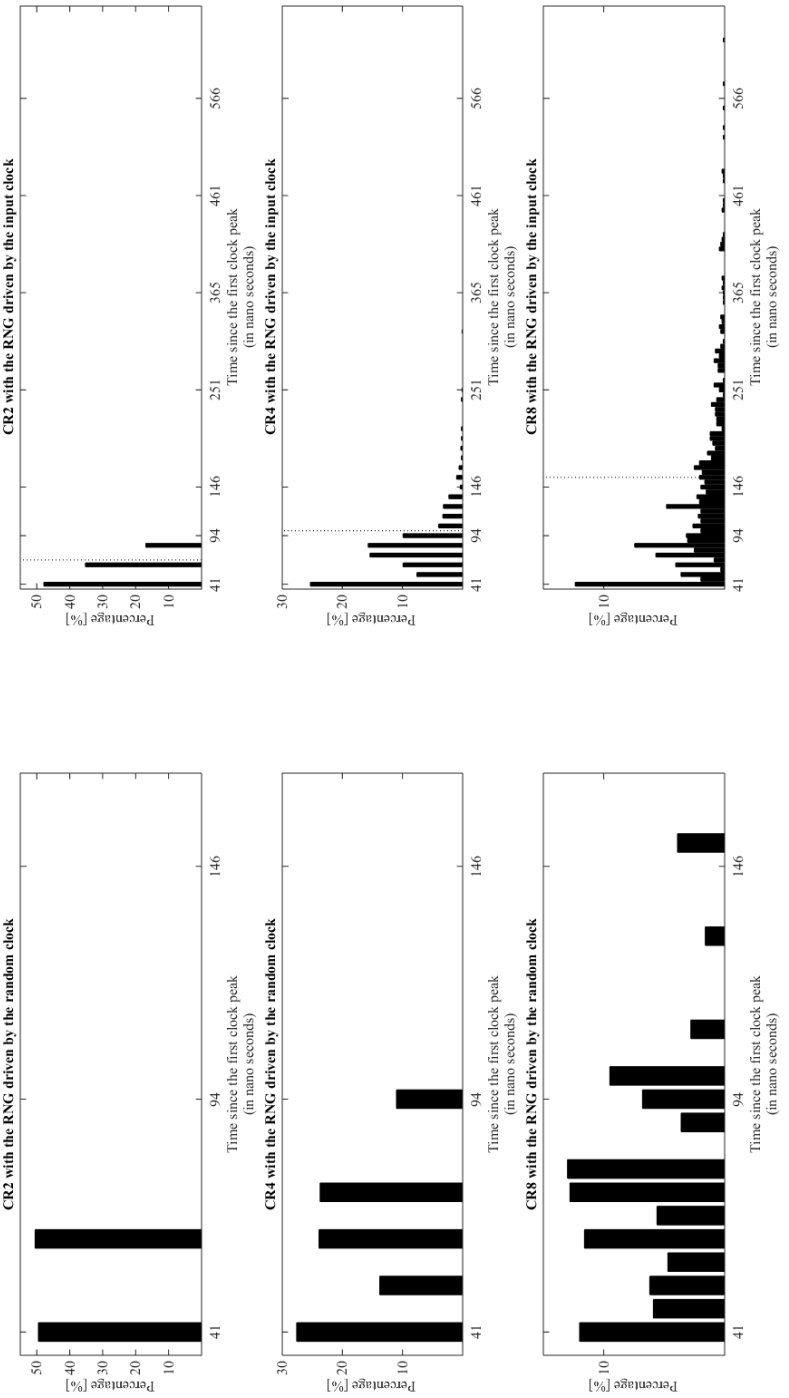

Figure 6.1: Histograms of clock periods on various clock-randomization implementations

Another observation is that the clock periods are farther distributed if the RNG is clocked by the input clock of the circuit and not by the randomized clock. When utilizing two or four phase-shifted clocks for the randomization about 17% of the clock periods will be longer than the longest clock periods measured when the input clock of the RNG is the randomized clock. This number almost doubles to 30% when eight phase-shifted clocks are utilized. These larger distribution windows also yielded better results when using the clock randomization against DPA attacks which is the reason why this way to clock the RNG became part our main implementation.

These larger distribution windows regarding the randomized clock periods can be explained by the combination of two effects. The first only occurs if the input clock of the RNG is the input clock of the device. When we compare the histograms at the top of Figure 6.1 which display the distribution of clock periods when two phase-shifted clocks[3] are utilized, we can see that there is a third bar of about 17% that equals to a phase shift of 360° which is twice the unaltered clock period (83.3 ns) for measurements taken with the RNG driven by the input clock of the device. To answer the question why this 360° shift only occurs in this case and not when the RNG is driven by the randomized clock we repeated the meaurements on the clock signal with two phase-shifted clocks but also captured the decision bit of the RNG. If the decision bit has the value 1 the clock signal which is shifted by 180° is activated by the clock randomization module. Otherwise the clock signal which is shifted by 0° is is activated by the clock randomization module. The three scenarios which we observed in the histogram were also present in this measurement and can be observed in Figure 6.2. If the clock signal remains unaltered the decision bit holds the value 0 or 1 during the critical decision time-frames. If the phase of the clock is altered by 180 degrees two different value inversions of the decision bit occur: The 0° clock signal is active and produces a positive clock edge while the decision bit changes from 0 to 1 or the 180° clock signal is active and produces a positive clock edge and the decision bit switches a half clock-period later (21 ns) from 1 to 0. Both of these cases result in the same clock alteration. The third case (i.e. the 360° shift) occurs when the 0° clock is selected and the decision bit switches to 1 at the positive clock edge and changes back to 0 before the 180° clock successfully has passed through the clock buffers.

The two latter cases show behavior that can only occur if the RNG is driven by the input clock of the device since changes of the decision bit can occur regardless of the current state of the randomized clock signal. This also explains the values of the first row of the histograms in Figure 6.1. If the RNG is driven by the randomized clock only two cases occur with a probability of about 50% each which only rely on the probability whether the RNG decision bit does not change (no alteration of the randomized clock and a clock period of 41.6 ns) or changes its value (a phase shift of 180° and a clock period of 62.5 ns). If the RNG is driven by the input clock of the device the last case is divided into two sub-cases. One subsumes that the decision bit switches from 1 to 0 or from 0 to 1 and keeps that value in the latter case for one clock cycle of the input clock. The other is less likely and requires that the decision bit switches from 0 to 1 and back to 0 in two subsequent clock cycles of the input clock.

The second effect is also present in clock-randomization variants where the RNG is driven by the randomized clock. The histogram for four phase-shifted clock signals where the RNG is driven by the randomized clock (second row, first column of Figure 6.1) has one additional non-zero value for clock periods of about 94n̈s which equals a phase shift

[3] 0° and 180°

RNG value	Phase		RNG value	Phase	Phase shift
00	0°	→	00	0°	0°
			01	90°	90°
			10	180°	180°
			11	270°	270°
01	90°	→	00	0°	270°
			01	90°	0°
			10	180°	*450°*
			11	270°	180°
10	180°	→	00	0°	180°
			01	90°	270°
			10	180°	0°
			11	270°	90°
11	270°	→	00	0°	*450°*
			01	90°	180°
			10	180°	270°
			11	270°	0°

Table 6.1: Phase-transition behavior of the clock-randomization countermeasure utilizing four phase-shifted clocks. The clock change is decided by the PRNG which is clocked by the randomized clock.

of 450°. To get a better understanding of how this clock period can occur we repeated the measurement while also recording the involved decision bits of the RNG. The five different cases for randomized clock periods can be seen in Figure 6.3. Decision bit 0 decides about the outcome of two clock buffers with the input pairs [0°,90°] and [180°, 270°] while bit 1 decides whether to let the clock signal pass from the outputs of the former or latter clock buffer. Again no clock alteration happens if the two decision bits keep their values. Phase shifts of 90 degrees occur if bit 0 changes its value from 0 to 1 while bit 1 remains unchanged. Phase shifts of 180° occur if bit 0 remains unchanged and bit 1 transitions from 0 to 1 or 1 to 0. Phase shifts of 270° can occur if bit 0 changes its value from 1 to 0 and bit 1 retains the same value which results in a negative phase jump of -90° which is equal to 270° or the same phase shift can occur if bit 0 changes from 0 to 1 and bit 1 inverts its value (which equals a switch from the 180° to 90° or from the 0° to 270° phase-shifted clock signals). All possible phase transitions of this measurement are contained in Table 6.1.

The phase shift of 450° occurs instead of phase shifts that should have a value of 90° where bit 0 changes from 1 to 0 and bit 1 inverts its value. We strongly believe that this effect is also caused by the utilized clock buffers which suppress the first positive clock edge after the change which delays the positive clock edge by another 360° (41.6 ns)[4]. This also explains why the corresponding histogram of Figure 6.1 has three possible clock periods with probabilities of about 25% each while the other two (the ones belonging to the 90° and 450° phase shifts) share the remaining 25%.

[4] 90° + 360° = 450°

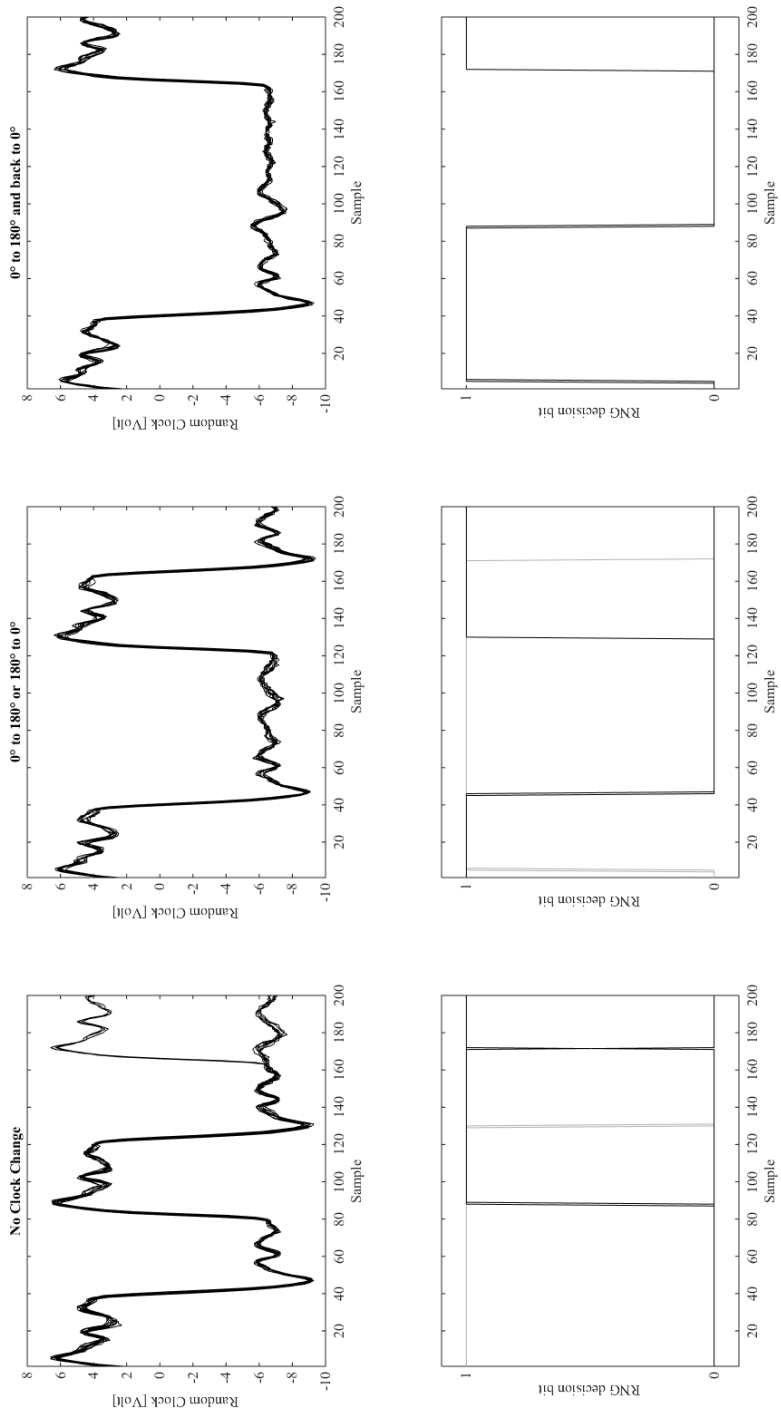

Figure 6.2: The three cases of clock alteration present with the clock randomization utilizing two phase-shifted clocks. The clock change is decided by the PRNG which is clocked by the input clock of the circuit.

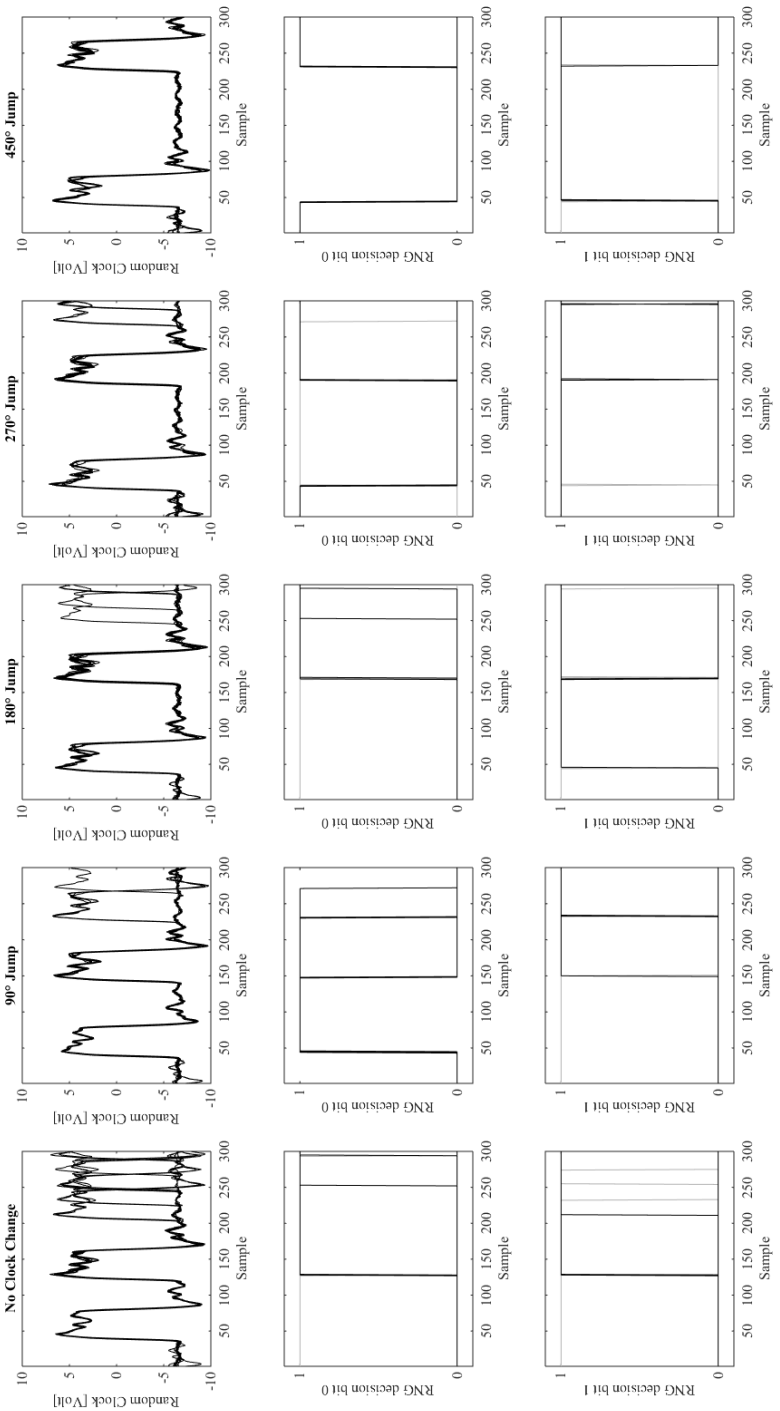

Figure 6.3: The five cases of clock alteration present with the clock randomization utilizing four phase-shifted clocks. The clock change is decided by the PRNG which is clocked by the randomized clock.

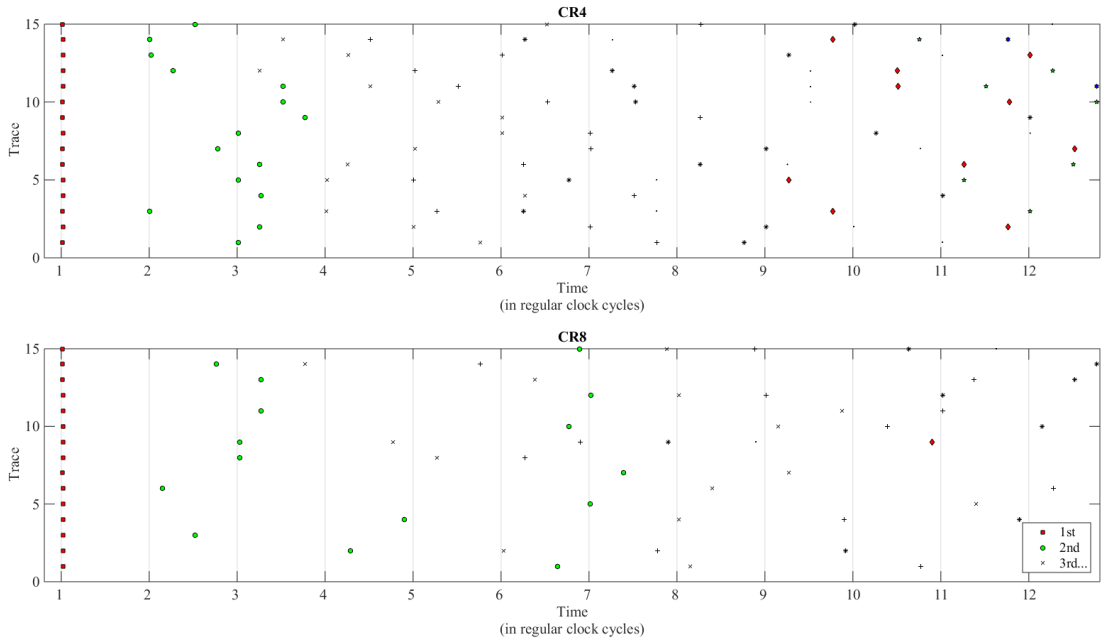

Figure 6.4: Clock randomization with four clocks and eight clocks. The PRNG is clocked by the input clock.

If these two effects are combined with each other (see the clock randomization using four phase-shifted clocks and the RNG driven by the input clock) and then also combined with the added complexity of another clock-buffer layer and four additional phase-shifted clocks of the clock randomization with eight phase-shifted clocks and the RNG driven by the input clock of the device multiple skips of positive clock-edges can occur. This leads to possible maximum clock-periods between positive clock-edges of almost 600 ns.

To get a better idea of the scope of the randomization effort and how it can variate overall calculation timings see Figure 6.4 which shows the placement of positive clock edges after a first positive clock edge. The top plot of Figure 6.4 shows the timing of the positive clock edges for a clock randomization with four phase-shifted clock signals for 15 recorded traces. As can be seen, the irregularity of the clock grows with each subsequent positive edge of the clock signal. This effect is exacerbated in the bottom plot of Figure 6.4 where the variance of one subsequent clock edge is increased as is expected with an implementation of the clock-randomization countermeasure that utilizes eight different phase-positions.

6.1.2 DPA Attacks on a CR-Secured Design

Next, we discuss the effects of the countermeasure on the DPA attack. As was discussed in Chapter 5, the correlation of most bytes stays distinguishable for many clock cycles when no countermeasures are applied, so it has to be determined how the interval of this correlation is affected by the randomization of the same time frame.

Since the interval is now calculated at different points in time and is of variable length, the total length of consideration for the DPA attack is increased. On the other hand the total correlation must be weaker since the calculation cycles are no longer aligned. As

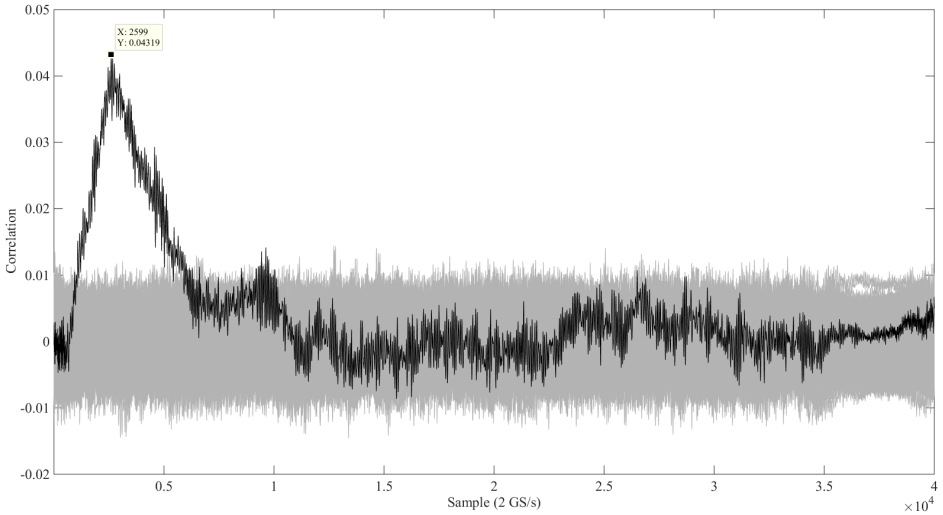

Figure 6.5: Correlation for the first key byte at an operating frequency of 24 MHz and an active CR countermeasure with four clocks, 100k traces

we discussed earlier in Chapter 4 one of the per-requisites for an optimal DPA attack is alignment of the traces. Re-alignment of the power traces is difficult due to the non-linear nature of the power traces recorded with this countermeasure and the jumps in phase of the clock signal.

Due to this, the total maximum correlation value of the correct key-hypothesis only reaches a fraction of the value of its counterpart of the implementation without active countermeasures compared to implementations of the clock-randomization countermeasure which utilize two or four phase-shifted clock signals. The result for the total maximum only reaches about 0.04 which it also does when only two clock signals are used albeit with a longer time period of distinguish-ability. The correlation result for the clock-randomization implementation with four phase-shifted clock signals can be seen in Figure 6.5.

Another aspect of this countermeasure is present when evaluating the subsequent key bytes whose attacked HD is exposed for fewer clock cycles. As the time window of exposure is reduced the worse the correlation result becomes. This is especially notable for key bytes corresponding with the last row of the register used in the state module where the correct key bytes can no longer be exposed by measurements with one-hundred thousand traces.

All the above aspects are featured even stronger when eight different clock signals are used. The maximum total value of the correlation curve for the correct first key-byte is now halved to 0.02 and the time window of distinguish-ability is further shortened. The result for the first key-byte is shown in Figure 6.6.

Also the DPA attack fails for the third row of registers in the state module when eight clock signals are used. This countermeasure is very successful considering the footprint of FPGA-resource usage necessary to implement it.

6.1.3 Clock Randomization with Mixed Operating Frequencies

In the search of improving this countermeasure we considered two approaches. First, one could add additional DCMs and phase-shifted clock signals into the implementation. Second, one set of DCM-clocks could be slowed down to randomly spread out the critical

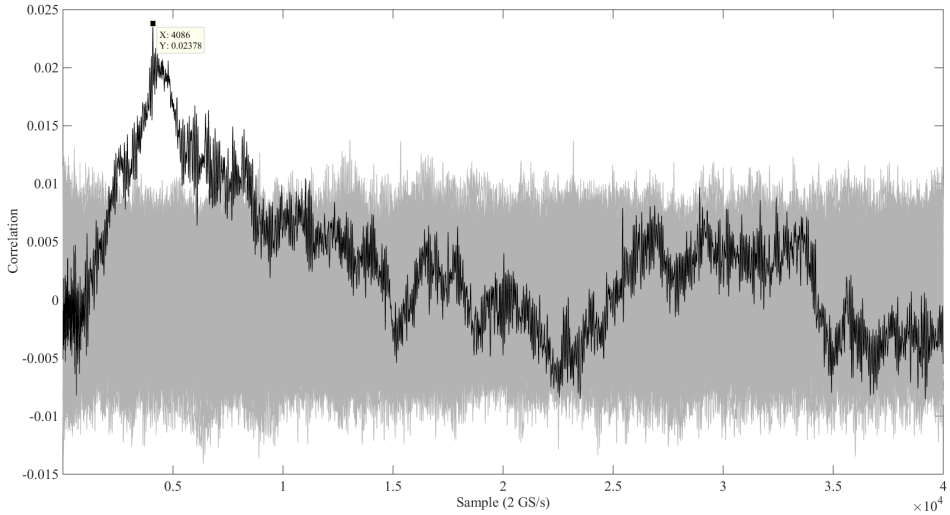

Figure 6.6: Correlation for the first key byte at an operating frequency of 24 MHz and an active CR countermeasure with eight clocks, 100k traces

cycles of the encryption even more which should further lower the maximum total correlation. Also this approach increases the measurement and DPA analysis effort because longer time-windows of measurement are necessitated.

Unfortunately the first approach can not be implemented without the chance of data corruption through clock glitches since not enough clock-buffer enabled multiplexers are available on the FPGA. We went on to implement the second approach by using an additional DCM to slow the clock down to 12 MHz and 1.5 MHz respectively. This also necessitated a reconsideration of how the RNG module should be clocked. Until now the RNG would be clocked by the main input clock, which guaranteed a chance of a clock-blocking event in one of the clock buffers, resulting in skips of clock-edge events. Since this clock must be the main 24 MHz clock of the design, the chance of considering positive clock events from the slower clock signals via the clock tree controlled by the RNG is low resulting in almost no further slow-down and interval widening of the AES encryption. Because of this, we reconnected the input clock of the RNG to the output of the RNG-controlled clock tree which guarantees whole clock periods after a clock switch has occurred which results in a longer encryption time-frame.

6.1.4 DPA Attacks on the Improved CR-Secured Design

When evaluating the countermeasure with four of the eight clock signals slowed to 12 MHz the impact the source of the input clock of the RNG module had on the correlation was insignificant, also the maximum total correlation of the first key byte remained almost the same with measurements done for both clock-source options. This is also the case when comparing the results with those obtained from the unaltered countermeasure design. The window of distinguish-ability is roghly the same as is the maximum total correlation for the correct first key byte at 0.02.

When the operating frequency of the second DCM is further slowed to 1.5 MHz the correlation result of the altered countermeasure shows a better masking of the correct key hypothesis. The correct key-byte hypothesis now has a maximum of 0.015 and is no

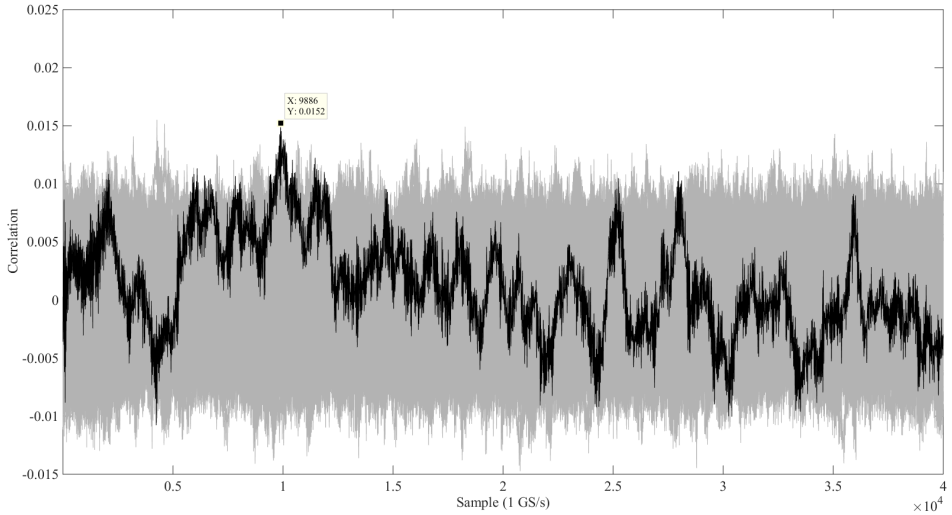

Figure 6.7: Correlation for the first key byte at an operating frequency of 24 MHz and 1.5 MHz and an active CR countermeasure with eight clocks, 100k traces

longer distinguishable from other maxima of wrong key hypotheses with a DPA calculated from 100k traces. This result can be seen in Figure 6.7. Since the exact timing of where the correlation maximum should occur is no longer known to the attacker all correlation maxima at around 0.15 must be considered hence the DPA has become more difficult for the first key-byte.

For this measurement the sampling rate had to be reduced to 1 GS/s in order to capture the relevant time window since the complete AES encryption now takes about 180 μs which is about 22-times of the 8 μs encryption time needed with no clock randomization. Also realignment of the clock cycles has been further hampered since the recombination of random 24 MHZ and 1.5 MHz clock-cycles has now to be applied to the measured power-traces.

From this point on we applied the second countermeasure which utilizes the randomized switching of short-circuits to the implementations of regular CR8, an 8-signal clock-randomization with the half of them slowed to 12 MHz and an 8-signal clock-randomization with the half of them slowed to 1.5 MHz. The first 2 implementations mentioned above showed no observable effect on the correlation curves. The implementation with half of the available clock signals slowed to 1.5 MHz slightly lowered the correct correlation below the noise-level. This however is still in the range of statistical deviation. The result for this implementation regarding the first key-byte can be observed in Figure 6.8. With this application of countermeasures the conventional DPA attack on 100k traces fails.

6.1.5 Summary of the Clock-Randomization Countermeasure

This section will discuss the results obtained when looking for the maximum correct correlation values of the first key byte in relationship to the amount of traces used in the DPA attack. It will also discuss the lower correlation achievements are linked to the additional time-requirement by slowed encryption process.

Figure 6.9 shows the success rates for the first key byte for multiple clock-randomization implementations. This plot was created by calculating the success rate as was discussed

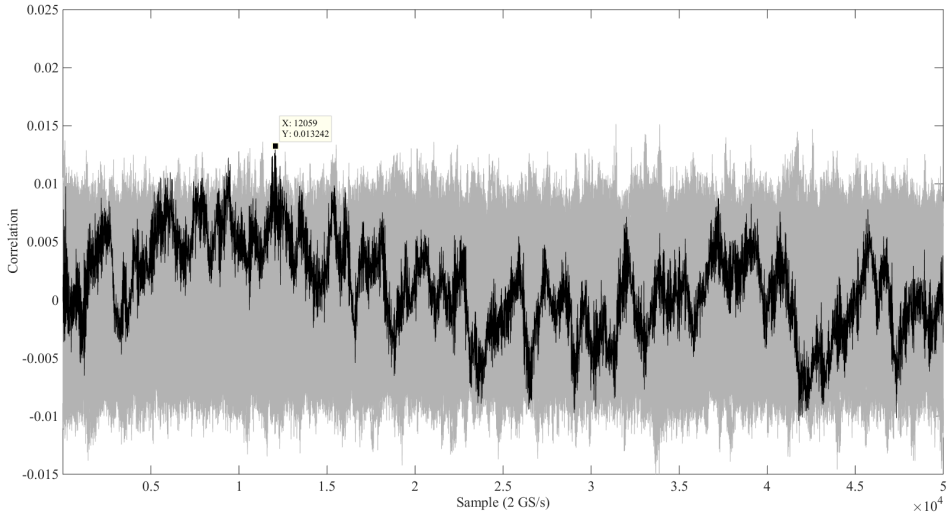

Figure 6.8: Correlation for the first key byte at an operating frequency of 24 MHz and 1.5 MHz and an active CR countermeasure with eight clocks and 16 short-circuits, 100k traces

	encryption time	percentage	maximum correlation
units	μs	%	[100 k traces]
NO CR	≈ 8	100	0.322
CR2	≈ 15	187	0.045
CR4	≈ 18	225	0.043
CR8	≈ 35	437	0.023
CR8 +12 MHz	≈ 30	375	0.020
CR8 +1.5 MHz	≈ 180	2 250	0.015
CR8 +1.5 MHz +16 SCs	≈ 180	2 250	0.013

Table 6.2: Results from the various clock randomization implementations.

in Section 5.2 and repeating the process 10 times and taking the mean of all 10 results. This was done to gather a better resolution[5]. It can be observed that the success rate is lowered by further deployments of clock signals and slowing part of these signals. Also the effects of 16 short-circuit instances appear to be negligible in the scope of this plot. It also seems that the implementation of clock randomization that utilized four phase-shifted clock signals at 12 MHz had minor adverse effects on the success rate.

When comparing our results with the ones given by [4] some discrepancies occur. Our clock randomization utilizing 8 phase-shifted clock requires on average 437% of the original time window while their implementation requires 377% which is a rather small difference. On average the implementation necessitates 100 times the amount of traces needed than without clock randomization while the implementation of [4] necessitates 1000 times the amount of traces needed than without the countermeasure.

When we compare our results of correct correlation maxima listed in the right column of Table 6.2 we can argue that the additional time needed by the encryption yields a relatively little benefit when regarding its lowered maximum correlation value.

[5] 1/100 instead of 1/10

6.1.6 Evaluation of the Leakage Produced by CR-secured Designs using t-tests

Further on we examined the effectiveness of the clock-randomization countermeasure with a statistical method called Welsh's t-test which can be used to determine wheter the main values of two different sets with different variances are identical or not. This tool has already been introduced as an evaluation method for applications regarding side-channel analysis by Gilbert Goodwill et al. in [49]. We applied this test for various datasets to further analyze the countermeasure. Equation (6.1) describes the statistic to be applied on a threshold evaluation:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}} \tag{6.1}$$

The variables X_i, s_i^2 and N_i are the mean, variance, and sample size of each data set.

First we applied the t-test on samples of fixed HDs. Our goal is to measure multiple occurrences of the same HD which is repeated in the 16 cycles of the ADD state and compare it to another specified HD. The t-test can evaluate how distinguishable both groups of measurements are and what influence the clock randomization countermeasure has on this distinguish-ability.

To achieve this, we set the state data to zero and the key data to Sbox inversions. Since we want to produce HDs by overwriting zero-values, we picked a value for each HD (zero to eight) to overwrite zero. We then calculated the Sbox inversions for each of these values which are supposed to be the results of the xor addition of the state and key. In this scenario all the state-bytes are zero and the key values hold the values of the Sbox inversion. To keep the occurrence of the same HD constant during the first AES-round the 16 key-bytes hold the same value. We recorded each HD one thousand times, the degree of freedom was constant at 2000 resulting in a threshold value of 3.1 which correlates with a distinguish-ability at probabilty of 99%. This threshold was derived from a table provided by NIST [50].

We measured these sets of one thousand traces each on our implementations outfitted with variations of the clock-randomization countermeasure. The statistic variable of the t-test can be observed in Figure 6.10 for the basic implementations (no countermeasure, CR2, CR4, CR8) of the clock-randomization countermeasure and in Figure 6.11 for altered versions of the clock-randomization countermeasure (CR4 24 MHz + CR4 12 MHz, CR4 24 MHz + CR4 1.5 MHz, CR4 24 MHz + CR4 1.5 MHz + 16 SCs) and also CR8 to ease the comparison between both figures. Both figures display t-test results of the comparisons between sets of HD0 with HD1, HD0 with HD4, HD4 with HD8 and HD0 with HD8. The t-test of HD0 with HD1 represents the smallest gap in HD while the t-test of with HD0 with HD8 represents the largest. We chooses these to see if both of these extremes pass the t-test threshold of 3.1 and whether they show different behavior. The t-tests of HD0 with HD4 and HD4 with HD8 represent the same gap in HD but were chosen to examplify the difference in behavior these t-test can nevertheless produce. This is mostly caused by the difference in fanout which was discussed back in Section 5.2 and also by the nonlinear effects the Sbox and other steps of the AES encryption which alter the behavior of these tests in later AES rounds considerably.

The t-test results clearly indicate that the mean values of each examined HD get closer the more clock signals are used for clock randomization and become almost identical when the clock randomization utilizes slower clock signals at 1.5 MHz while a combination with 12 MHz does not achieve this effect. This effect is caused by the distribution of calculation

timings due to the clock-randomization countermeasure. The effect increases with the strength of the countermeasure and also with the progress of time as can be observed in Figure 6.4.

To get a better understanding we repeated the t-tests with various HDs but altered how the HDs are created. In this case the HD is created from the overwriting process of the Sbox result to a non-zero value. In this case we chose a constant non-zero value which had also to be added to the result of the Sbox inversion values (see above) in order to get the needed values for the key byte. We did this for multiple reasons. First, we wanted to see the t-test performed over the complete AES encryption for multiple implementations of the clock-randomization countermeasure to be able to better compare the time scale of the leakage. We also wanted to eliminate our reliance on zero-value registers. And another goal was to conduct t-tests between fixed HDs (like HD0) and sets that contain measurements of random HDs. With this we hoped to get information how these fixed HDs are distinguishable from a "mean" HD we generated from these randomized sets.

The results for these t-tests can be observed in Figure 6.12 which captured the whole AES-encryption process of one 128-bit block for various implementations of the clock-randomization countermeasure. Here it can also be observed that the clock randomization weakens the distinguish-ability with more effective implementations of the countermeasure. This also entails delays of the most distinguishable calculations. When the enhanced countermeasure with four clock signals at 1.5 MHz is used the lower initial distribution of positive clock edges leads to a better distinguish-ability at the beginning of the AES-encryption process. This is because this implementation uses the randomized clock to drive the PRNG module instead of the standard input clock, a discussion on this topic was conducted in Section 6.1.1.

These t-test results also indicate that the mean values of each examined HD get closer the more clock signals are used for clock randomization and become almost identical when the clock randomization utilizes slower clock signals at 1.5 MHz. Note that the distinguish-ability observed after the first round is not of interest when trying to obtain key information from the first round as our DPA attack using known plaintext and the HD power-model are not applicable there. As we can now see from the leakage of the encryption that peaks that are distinguishable when no countermeasure is active get alaised by the clock edge distributions imposed by the clock-randomization countermeasure. T-tests which compare fixed HDs to randomized HD sets show a better distinguish-ability particularly at the beginning of the encryption process which hints on why our DPA attacks remained feasible. Also when we conducted t-tests between different sets of random HDs the results stayed close but sometimes reached above the threshold criterion indicating that the criterion was set too strict.

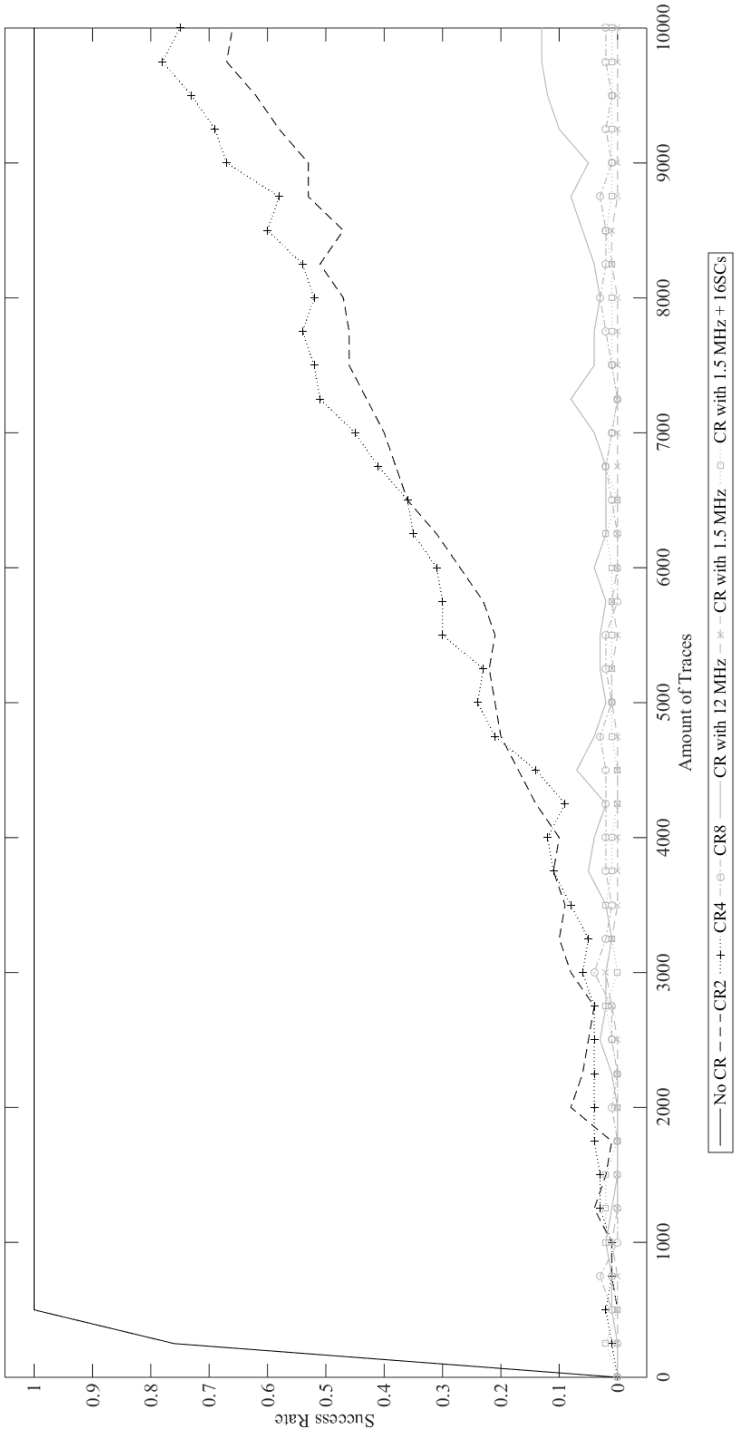

Figure 6.9: Success rates of various clock-randomization implementations

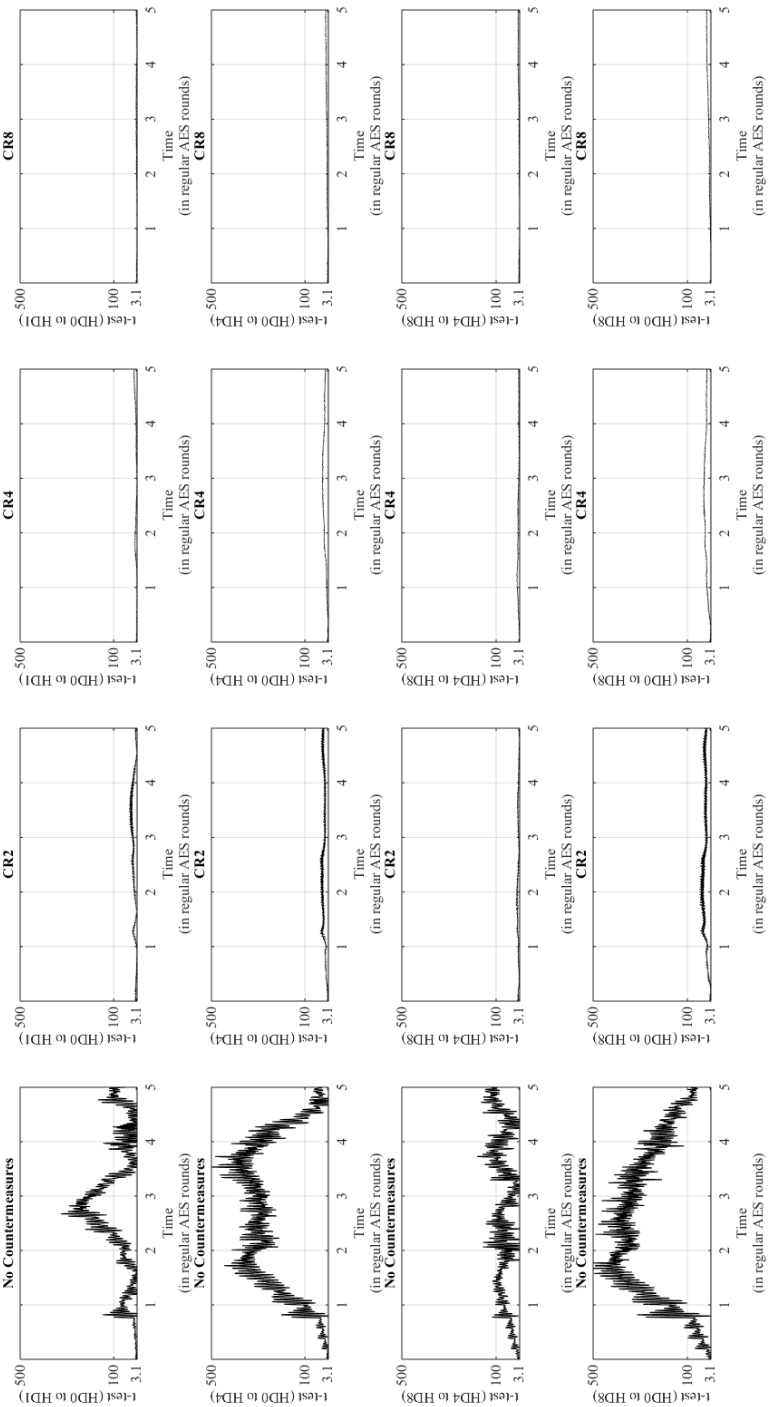

Figure 6.10: Results of Welsh's t-test on various clock-randomization implementations

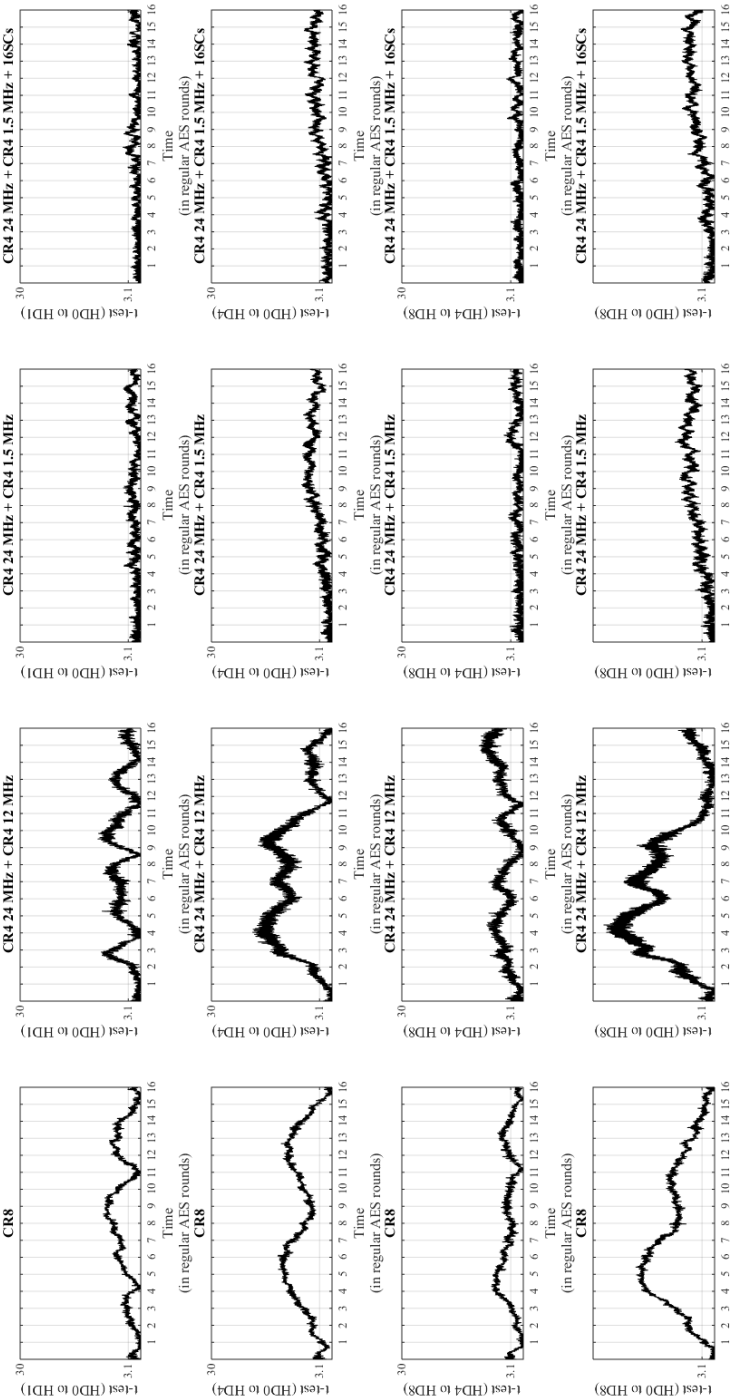

Figure 6.11: Results of Welsh's t-test on various clock-randomization implementations

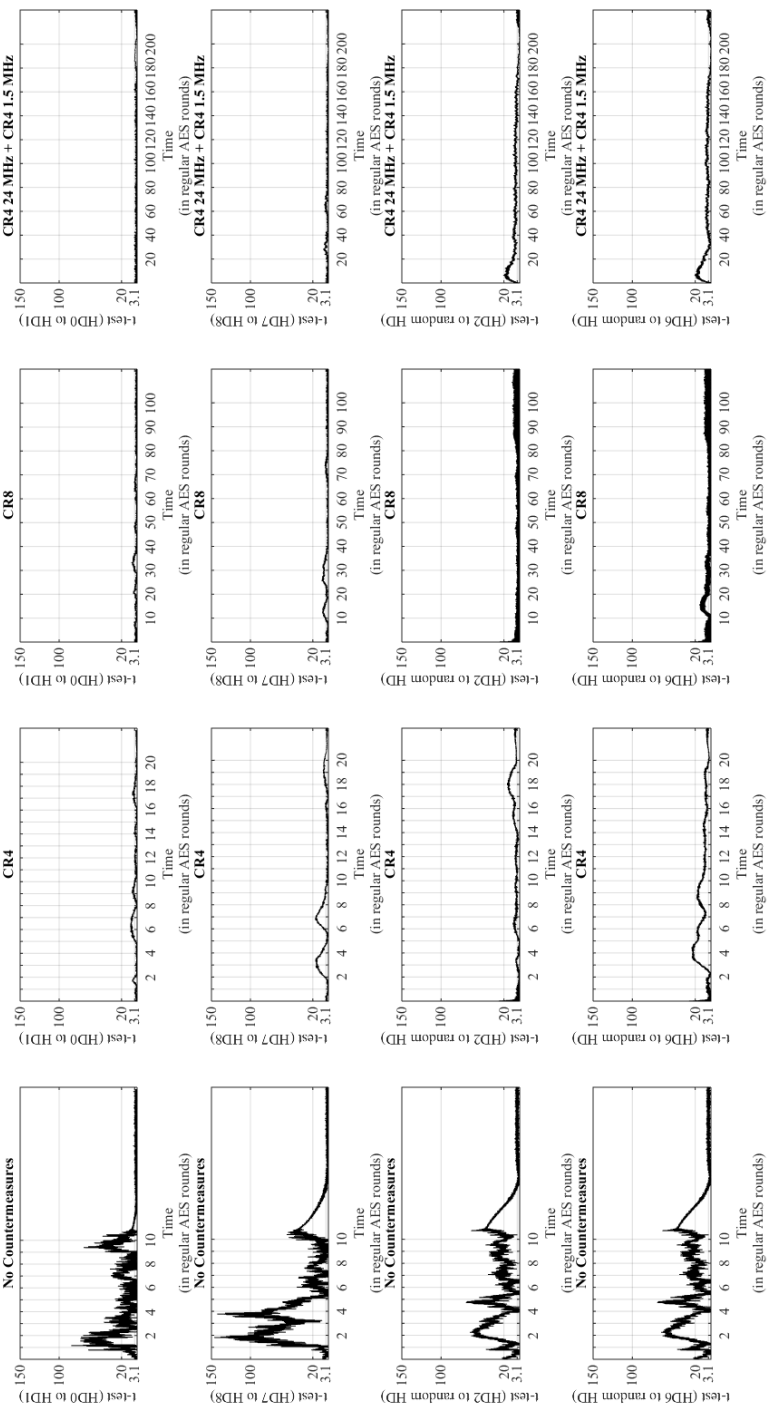

Figure 6.12: Results of Welsh's t-test on various clock-randomization implementations

6.2 Results on the Short-Circuit Countermeasure

We implemented all three SC designs as was discussed in Chapter 3 with one instance at first but could not detect any influence on the correlation results obtained from 10k measurements. Next we repeated the process with 16 SC instances which yielded a minor masking of the correlation curve for the correct key-byte for our design with three LUTs. Next we implemented 32 SC instances which was double the amount used by [4] but yielded weaker results than the implementations with just 16 SC instances for every design.

These correlation results seemed rather disappointing considering [4]'s claim of the need for a minimum of 8k traces per measurement with this countermeasure while their unprotected design required a minimum amount of 3k traces for a successful attack. So [4] conclude that this countermeasure more than doubled the minimum amount of traces. This is much better than our results which had showed no major impact on the success rates of the DPA attacks.

In order to better understand the reduced leakage of this countermeasure we additionally examined it with the same t-test scheme we applied on the clock-randomization countermeasure. The results of the t-tests can be seen for the period of the first 16 cycles of the encryption in Figures 6.13 for measurements taken at an operating frequency of 24 MHz and 6.14 for measurements conducted at an operating frequency of 1.5 MHz. It can be observed that the mean values of each set become more distinguishable the farther the shifting has proceeded. As expected this trend is accelerated when registers with high fanouts are passed. Ultimately the additional SC instances make the sets of recorded HDs less distinguishable. But it does this in very low increments.

Considering each SC instance needs 3 slices of the FPGA, 32 SC instances need about an additional 17% of the slices of the AES implementation equipped with an RNG module this is not very effective and even less effective than what was observed by [4]. Because of this and the complications involved with implementing the countermeasure we can not recommend this proposal for a countermeasure for an actual security-related implementation.

It is also unclear whether the toolchain-exploitations needed to achieve the SCs can be replicated on more modern Xilinx FPGAs.

6.3 Summary

In this chapter we have shown how both of the implemented countermeasures affected the correlation behavior of the correct key hypothesis. The clock-randomization countermeasure proved to be rather cost-effective regarding the use of FPGA resources but showed a significant increase in time necessary to calculate the encryption result.

We then proceeded to enhance this countermeasure by slowing half of the clock signals down to 12 MHz and 1.5 MHz which resulted in decreased correlation values with the slow to 1.5 MHz and reduced the correlation value even farther after we combined this implementation with 16 short-circuit instances. The major cost of this enhancement was the time used by the encryption since it now needs at average 22.5 times the time to calculate the encryption result.

We have shown that the short-circuit countermeasure is rather ineffective and expensive regarding the amount of FPGA resources needed. We also designed ways to evaluate these countermeasures with the help of Welsh's t-test which led to further insights into the effects of the countermeasures.

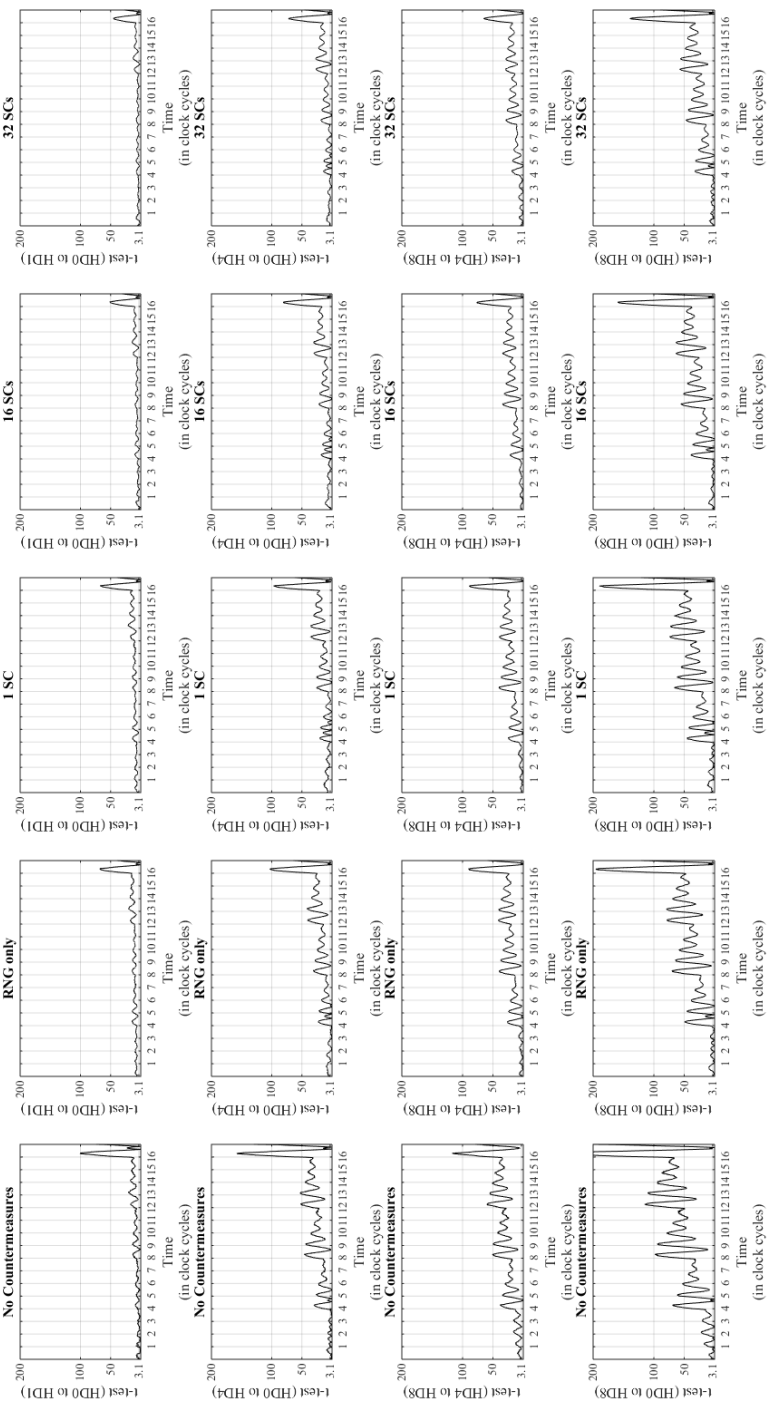

Figure 6.13: Results of Welsh's t-test on various SC implementations at 24 MHz

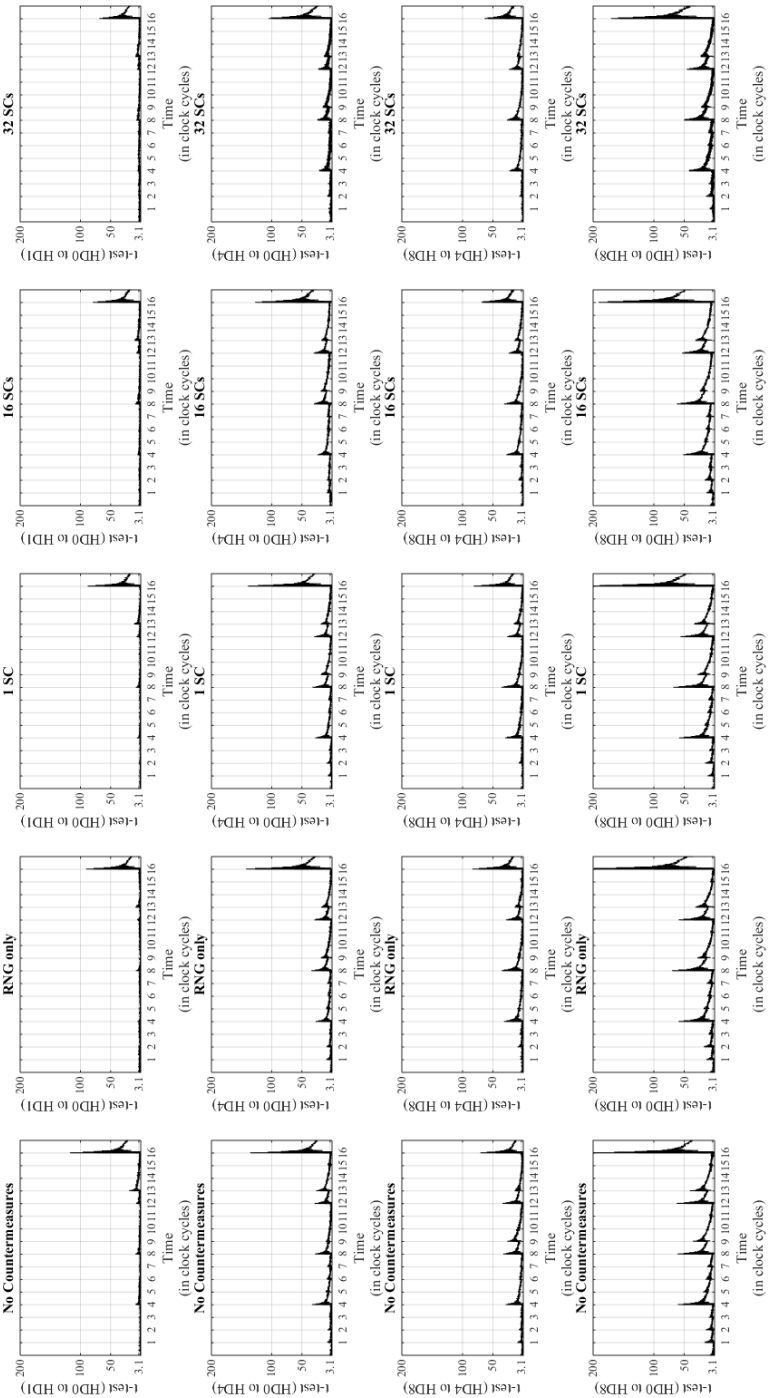

Figure 6.14: Results of Welsh's t-test on various SC implementations at 1.5 MHz

Chapter 7

Conclusions

Through the course of this thesis we have shown that the specification of ALE needs improvement to ensure standardized implementations which could be provided by the original authors of [1]. Also we have shown that the original ASIC implementation relied heavily on saving intermediate values online which we solved by adding a third 16-byte storage module called temp and still achieved comparable results regarding the area requirements of the implementation.

We have also discussed the influences supply capacitances and shunt-resistor values have on the correlation result. Since we experienced a correlation windup caused by switching information leaking into subsequent clock cycles, we have shown that this effect is minimized when setting the operating frequency close to the characteristic frequency which could be observed both in correlation results and the measured power traces. The characteristic frequency is set by the implementation of the FPGA itself and has no relation to power-supply setup provided by the evaluation board.

We have shown that there is a relationship between the fanout of a given net to its contribution to the correlation result. A higher fanout leads to increased correlation values when compared to the same information applied to nets with a lower fanout. This effect may also be mitigated by setting the operation frequency near to the characteristic frequency since low-fanout nets seem to contribute with a slightly better signal-to-noise-ratio in these measurements. It is left to further research whether these observations can be further utilized to hinder or enable DPA attacks.

We have also shown that low-pass filters set below the operating frequency measurably lower correlation values. Since the side-channel resistance of the AES implementation is much lower than what was observed by their original proposers [6] it begs to question what caused these discrepancies in results.

The FPGA-specific generic countermeasures performed worse than what was originally found by [4]. The improvement of the clock randomization countermeasure by significantly slowing the half of the used clock signals closed this gap but came at the cost of significantly slower encryption throughput. We did preliminary testing with 16 clock signals instead of 8 and experienced data corruption through clock glitches caused by a lack of additional clock buffers, but these findings were inconclusive. Maybe this further enhancement of the clock-randomization countermeasure can be researched on more modern FPGA technologies.

The countermeasure involving the randomized switching of short-circuits proved to be without any bigger merit besides further improving on the clock randomization countermeasure by causing a small decrease in correct correlation values. Since the implementation of short-circuits proved to involve a lot of time and resources with the dated

FPGA-Editor software and causes a significant increase of used FPGA resources we do not recommend any further research involving this countermeasure.

Overall we modified a standard approach to be used as the power model to attack AES-128 and were successful with it. At first we underestimated the effects multiple fanout values and the leakage of switching information had on our results caused by attacking a 16-byte shift register. Then we successfully explained most features the correct correlation curves exhibit.

This thesis re-examined many of the findings and proposals set by our peers and managed to add to their findings or highlight discrepancies in their approaches and findings. In the end we have primarily shown that many of the papers and proposals published in modern IT-security research could benefit from approaches to make it easier to reproduce their results.

Appendix A

Definitions

A.1 Abbreviations

AE	Authenticated Encryption
AES	Advanced Encryption Standard
ALE	AES-based Lightweight Encryption
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
CB	Clock Buffer
CR	Clock Randomization
DCM	Digital Clock Manager
DES	Data Encryption Standard
DPA	Differential Power Analysis
DRC	Design Rule Checking
DUT	Device Under Test
FF	FlipFlop
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GE	Gate Equivalent
GF	Galois Field
HD	Hamming Distance
HDL	Hardware Description Language
HW	Hamming Weight
IT	Information Technology
IV	Initialization Vector
KS	Key Schedule
LEX	Leak Extraction
LUT	Look-Up Table
MAC	Message Authentication Code
PC	Personal Computer
PRNG	Pseudo-Random Number Generator
RC / RCON	Round Constant
RFID	Radio-Frequency IDentification
RNG	Random Number Generator
SASEBO	Side-Channel Attack Standard Evaluation BOard

SC	Short Circuit
SCA	Side-Channel Analysis
SNR	Signal-to-Noise Ratio
VHDL	VHSIC Hardware Description Language
XDL	Xilinx Design Language
XOR	Exclusive Or
ZV	Zero Value

Bibliography

[1] Andrey Bogdanov, Florian Mendel , Francesco Regazzoni, Vincent Rijmen and Elmar Tischhauser, *ALE: AES-Based Lightweight Authenticated Encryption.*

[2] Stefan Mangard et al. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*Springer, 2007.

[3] Joan Daemen, Vincent Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard.* Springer, 2002.

[4] Tim Güneysu and Amir Moradi *Generic Side-Channel Countermeasures for Reconfigurable Devices* Springer, 2011.

[5] Beckhoff, Christian, Dirk Koch, and Jim Torresen *Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration* Field Programmable Logic and Applications (FPL), 2010 International Conference on. IEEE, 2010.

[6] Amir Moradi et al. *Pushing the Limits: A Very Compact and a Threshold Implementation of AES* Springer, 2011.

[7] Dmitry Khovratovich and Christian Rechberger *The LOCAL Attack: Cryptanalysis of the Authenticated Encryption Scheme ALE* Springer, 2014.

[8] Wu, Shengbao, et al. *Leaked-state-forgery attack against the authenticated encryption algorithm ale.* Advances in Cryptology-ASIACRYPT 2013. Springer Berlin Heidelberg, 2013. 377-404.

[9] A. Bogdanov et al. *Efficient and Side-Channel Resistant Authenticated Encryption of FPGA Bitstreams* ReConFig, 2012.

[10] B. Bilgin et al. *Fides: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware* Springer 2013.

[11] Jakimoski, Goce, and Samant Khajuria. *ASC-1: an authenticated encryption stream cipher.* Selected Areas in Cryptography. Springer Berlin Heidelberg, 2012.

[12] Mihir Bellare and Chanathip Namprempre *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm* Springer, 2000.

[13] Hua Li and Zac Friggstad *An Efficient Architecture for the AES Mix Columns Operation* ISCAS, 2005.

[14] D. Canright *An Efficient Architecture for the AES Mix Columns Operation* Naval Postgraduate School Monterey CA, 2005

[15] Alex Biryukov *A New 128-bit Key Stream Cipher LEX* ECRYPT, 2005

[16] J Daemen, V Rijmen *The Pelican MAC Function* IACR Cryptology ePrint Archive, 2005

[17] H. Krawczyk *The order of encryption and authentication for protecting communications (or: How secure is SSL?)* Springer-Verlag, Berlin Germany, 2001.

[18] Albrecht, Martin R., Kenneth G. Paterson, and G. Watson. *Plaintext recovery attacks against SSH.* Security and Privacy, 2009 30th IEEE Symposium on. IEEE, 2009.

[19] Duong, Thai, and Juliano Rizzo. *Here come the ⊕-ninjas.* Unpublished manuscript (2011).

[20] Bellovin, Steven M. *Problem Areas for the IP Security Protocols.* USENIX Security. 1996.

[21] Paterson, Kenneth G., and Arnold KL Yau. *Cryptography in theory and practice: The case of encryption in IPsec.* Advances in Cryptology-EUROCRYPT 2006. Springer Berlin Heidelberg, 2006. 12-29.

[22] Degabriele, Jean Paul, and Kenneth G. Paterson. *Attacking the IPsec Standards in Encryption-only Configurations.* IEEE Symposium on Security and Privacy. 2007.

[23] Kohno, Tadayoshi, John Viega, and Doug Whiting. *CWC: A high-performance conventional authenticated encryption mode.* Fast Software Encryption. Springer Berlin Heidelberg, 2004.

[24] Yang, Bo, Sambit Mishra, and Ramesh Karri. *A High Speed Architecture for Galois/Counter Mode of Operation (GCM).* IACR Cryptology ePrint Archive 2005 (2005): 146.

[25] Rogaway, Phillip, Mihir Bellare, and John Black. *OCB: A block-cipher mode of operation for efficient authenticated encryption.* ACM Transactions on Information and System Security (TISSEC) 6.3 (2003): 365-403.

[26] Bellare, Mihir, Phillip Rogaway, and David Wagner. *The EAX mode of operation.* Fast Software Encryption. Springer Berlin Heidelberg, 2004.

[27] Moise, Avygdor, et al. *EAX'Cipher Mode (May 2011).*

[28] Minematsu, Kazuhiko, et al. *Attacks and Security Proofs of EAX-Prime.* Preproceedings of Fast Software Encryption. 2013.

[29] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, Martin Schläffer *Ascon v1* Submission to the CAESAR Competition, 2014

[30] Kazuhiko Minematsu on behalf of NEC Corporation, Japan *AES-OTR v1* Submission to the CAESAR Competition, 2014

[31] De Canniere, Christophe, and Bart Preneel. *Trivium.* New Stream Cipher Designs. Springer Berlin Heidelberg, 2008. 244-266.

[32] S. Babbage and M. Dodd. *The MICKEY Stream Ciphers.* In M. J. B. Robshaw and O. Billet, editors, The eSTREAM Finalists, volume 4986 of LNCS, pages 191–209. Springer, 2008.

[33] M. Hell, T. Johansson, A. Maximov, and W. Meier. *The Grain Family of Stream Ciphers.* In M. J. B. Robshaw and O. Billet, editors, The eSTREAM Finalists, volume 4986 of LNCS, pages 179–190. Springer, 2008

[34] G. Leander, C. Paar, A. Poschmann, and K. Schramm. *New Lightweight DES Variants.* In A. Biryukov, editor, FSE, volume 4593 of LNCS, pages 196–210. Springer, 2007.

[35] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. *HIGHT: A New Block Cipher Suitable for Low-Resource Device.* In L. Goubin and M. Matsui, editors, CHES, volume 4249 of LNCS, pages 46–59. Springer, 2006.

[36] C. H. Lim and T. Korkishko. *mCrypton - A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors.* In J. Song, T. Kwon, and M. Yung, editors, WISA, volume 3786 of LNCS, pages 243–258. Springer, 2005.

[37] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. *Spongent: A Lightweight Hash Function.* In B. Preneel and T. Takagi, editors, CHES, volume 6917 of LNCS, pages 312–325. Springer, 2011.

[38] J. Guo, T. Peyrin, and A. Poschmann. *The PHOTON Family of Lightweight Hash Functions.* In P. Rogaway, editor, CRYPTO, volume 6841 of LNCS, pages 222–239. Springer, 2011.

[39] J.-P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia. *Quark: A Lightweight Hash.* In S. Mangard and F.-X. Standaert, editors, CHES, volume 6225 of LNCS, pages 1–15. Springer, 2010.

[40] M. Agren, M. Hell, T. Johansson, and W. Meier. *Grain-128a: a new version of Grain-128 with optional authentication.* IJWMC, 5(1):48–59, 2011.

[41] D. W. Engels, M.-J. O. Saarinen, P. Schweitzer, and E. M. Smith. *The Hummingbird-2 Lightweight Authenticated Encryption Algorithm.* In A. Juels and C. Paar, editors, RFIDSec, volume 7055 of LNCS, pages 19–31. Springer, 2011.

[42] Andreeva, Elena, et al. *APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography.* IACR Cryptology ePrint Archive 2013 (2013): 791.

[43] ARM Ltd. *AMBA APB Protocol Version 2.0* 2010.

[44] Marsaglia, George *Xorshift rngs.* Journal of Statistical Software 8.14 (2003): 1-6.

[45] Saito, Mutsuo and Matsumoto, Makoto. *XORSHIFT-ADD (XSadd): A variant of XORSHIFT* 2014

[46] Bogdanov, Andrey, et al. *PRESENT: An ultra-lightweight block cipher.* Springer, 2007.

[47] Rolfes, Carsten, et al. *Ultra-lightweight implementations for smart devices–security for 1000 gate equivalents.* Smart Card Research and Advanced Applications. Springer, 2008

[48] De Canniere, Christophe, Orr Dunkelman, and Miroslav Knežević *KATAN and KTANTAN—a family of small and efficient hardware-oriented block ciphers.* Springer, 2009

[49] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi *A testing methodology for side-channel resistance validation.* NIST Non-invasive attack testing workshop. 2011.

[50] NIST table for t-test Evaluation thresholds *http://www.itl.nist.gov/div898/handbook /eda/section3/eda3672.htm* NIST

[51] Kocher, Paul, Joshua Jaffe, and Benjamin Jun. *Differential power analysis.* Advances in Cryptology—CRYPTO'99. Springer Berlin Heidelberg, 1999.

[52] Dhem, Jean-Francois, et al. *A practical implementation of the timing attack.* Smart Card Research and Applications. Springer Berlin Heidelberg, 2000.

[53] Research Center for Information Security, National Institute of Advanced Industrial Science and Technology (RSIC) *Side-channel Attack Standard Evaluation Board SASEBO-G Specification.*

[54] Comprehensive TEX Archive Network (CTAN) *http://www.ctan.org*

[55] Shamir, Adi, and Eran Tromer. *Acoustic cryptanalysis.* presentation available from http://www.wisdom.weizmann.ac.il/ tromer (2004).

[56] Bilgin, Begül, et al. *A more efficient AES threshold implementation.* Progress in Cryptology–AFRICACRYPT 2014. Springer International Publishing, 2014.