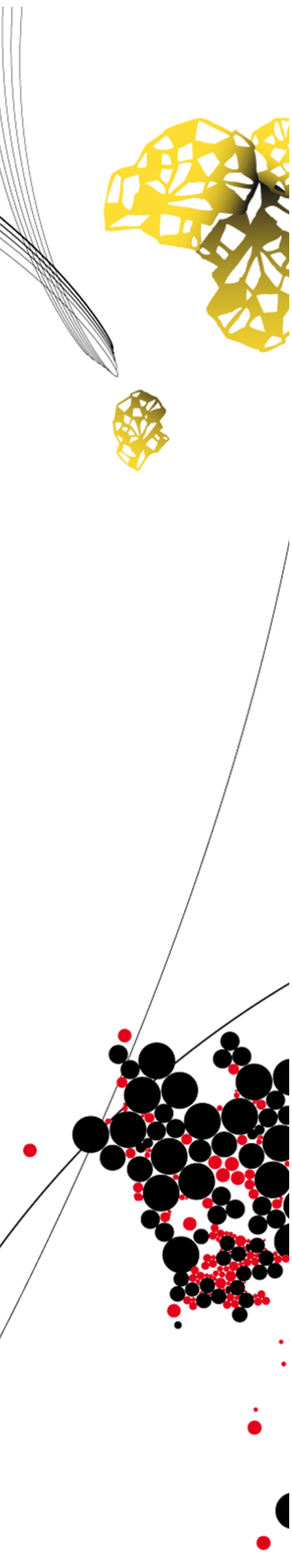

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

FPGA-based AES Variants Against Side-Channel Attacks

Emil Kerimov

M.Sc. Thesis
June 2020

Supervisors:
dr. ir. D. M. Ziener
M.Sc. Ali Asghar

Computer Architecture
for Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Nowadays, encryption is used as a means to protect one's personal information from unauthorized parties. Advanced Encryption Standard (AES) that is commonly used in various communication and storage systems is practically immune to mathematical cryptanalysis. However, it has been shown to be susceptible to side-channel attacks (SCA). Regular cryptanalysis methods exploit weaknesses of the encryption algorithm itself. SCAs, instead, rely on the phenomenon that power consumed by an electronic circuit is correlated to the data being processed by that circuit. Statistical analysis of repeated measurements of consumed power leads to exposing information that was meant to be hidden.

Field Programmable Gate Array (FPGA) devices are widely used to implement a variety of systems, including those operating on sensitive data. These devices are capable of Dynamic Partial Reconfiguration (DPR) that allows run-time switching between various configurations of the FPGA. This feature gives rise to a class of countermeasures against SCAs based on the implementation diversity of the system to be protected. Rather than modifying its algorithm or system-level architecture, variants of the system are generated by manipulating its physical implementation on the FPGA. These variants maintain functional integrity of the original design but may exhibit varying power consumption profiles. If the variants are interchanged in run-time, when the system is active, then the correlation between power and data can be distorted when compared to a static implementation. If the correlation is reduced, it may lead to an increase in the number of measurements, time, or processing necessary for analysis, and therefore improve SCA immunity.

The following work explores synthesis-level design modifications to generate variants of an FPGA-based AES implementation. First, existing countermeasures found in relevant literature sources are studied. Based on the identified methods a number of design variation concepts are proposed and evaluated. Generated variants are verified for functional integrity and implemented into a system prepared for measurements of power consumption. The effectiveness of the variants on the immunity against SCAs is measured by performing Correlation Power Analysis (CPA) on the acquired power consumption traces. Variants are first evaluated individually, followed by trace shuffling to emulate the DPR effect. Results show up to 12x incremental improvement in number of traces required to extract secret information compared to 2-3x of existing approaches.

Contents

Chapter 1

Introduction

1.1 Motivation

The following work will explore safeguards in the context of embedded systems that are a combination of hardware and software resources implementing a specific function or a set of functions. In the modern world, smart embedded systems accessing and managing personal data capable of intercommunication over the internet are integrated into our lives. Examples of these systems are smart house appliances, mobile devices, etc. Due to the nature of the information contained within these systems, they are frequent targets of attacks from third parties seeking access to the hidden information. Thus, privacy and security are important metrics of these systems among others, such as performance, reliability, cost, etc. Attacks on systems that contain personal, confidential government, or commercial data are increasing in number as the systems get more complex by design and may have more vulnerabilities exposing what should have been kept hidden [1]. These cyberattacks are usually carried out with malicious intent by parties with conflicting interests on random or targeted victims. The attackers are looking to gain advantage by extracting proprietary or secret information [2], [3]. Damage inflicted reflects itself in personal data being compromised or can be quantified in monetary value [4] for commercially valuable and confidential information. It is thus in the interest of system designers to implement features that mitigate the effects of these attacks or prevent them altogether.

In particular, the following discussion is on Field Programmable Gate Arrays (FPGAs) running an implementation of an Advanced Encryption Standard (AES) and increasing immunity of such a system against a type of cyberattacks known as the side-channel attacks. Synthesis-level modifications are explored within this work to generate alternative configurations (variants) for the dynamic partial reconfiguration (DPR) feature of modern FPGAs. This feature allows to interchange the physical configuration of the system while maintaining its functional integrity and availability. Proposed variants are evaluated using an experimental test setup and compared against relevant published studies.

1.2 Report Organization

The remainder of this report is organized as follows. The current chapter continues with a brief introduction of underlying concepts such as security, encryption, and FPGAs. In Chapter 2, Advanced Encryption Standard will be introduced in detail, along with Correlation Power Analysis and reconfigurable modules for FPGAs. Countermeasures to CPA are studied in Chapter 3 referring to relevant state-of-the-art literature. The report continues with Chapter 4, presenting the countermeasure proposals of the current work. Evaluation method and results from the power consumption measurements are discussed in Chapter 5. The report is concluded in Chapter 6, followed by Appendices.

1.3 Main Concepts

Dependability and Security

As systems increase in their complexity [5], their dependability becomes more important. Dependability, as defined by the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, is a term used to describe "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers" [6]. To design a dependable system several attributes have to be satisfied, with the following list defined by J. C. Laprie in the book "Dependability: Basic Concepts and Terminology" [7]:

- Reliability	- Confidentiality
- Maintainability	- Integrity
- Safety	- Availability

Reliability is the ability of a system to perform its required function with the emphasis on continuity of the service. Maintainability is the ability to undergo modifications and repairs, while safety is the prevention of catastrophic events on the environment. Security can be used as an umbrella term for the three remaining attributes defined above – confidentiality, integrity, and availability. First is the absence of unauthorized information access, second is the absence of improper system alteration, and the last attribute is the readiness for usage by authorized parties. Thus, to maintain its security, a system must deliver these three attributes [8]. Furthermore, to implement a secure system, three main properties of security need to be established – goals, threats, and means. Goals define what is to be protected, such as the system under observation. Threats are defined as against what or whom this system should be protected, such as malicious parties or attackers. Means define how this protection will be carried out [9]. With a system in mind to protect against malicious parties as discussed above, safeguards or countermeasures are used as the means to protect this system. While safeguards are meant to prevent an attack, the latter are

used to detect and respond to an attack – a priori and a posteriori defences, as per explanation by Stajano in [9].

Encryption

As mentioned earlier, modern systems are often equipped with data encryption – a method to transform the original piece of information into a secret code such that its original meaning is hidden. Access to the original data and the system is only available per request of the authorized parties that have a key to unlock the meaning of the secret code. Thus, all three attributes defining the security of a system can be achieved through the means of encryption. Several different methods and types of encryption exist, allowing users to protect their data from unwanted third parties [10]. Examples include widely known encryption algorithms such as Rivest-Shamir-Adleman (RSA) [11] and Data Encryption Standard (DES) [12]. Advanced Encryption Standard (AES) is a successor to the latter [13], defined by National Institute of Standards and Technology (NIST) of the United States of America. This algorithm is used within the current work and will be explained in detail in the following chapter.

Field Programmable Gate Arrays

The next concept to be introduced is the Field Programmable Gate Array, known as FPGA. These devices are configurable arrays of transistors interconnected to each other. Designers can implement any type of functionality required, including encryption using AES. Flexibility and low design time make FPGAs a preferred choice over Application-Specific Integrated Circuits (ASICs) for a lot of applications [14], [15], [16]. ASICs can deliver better energy efficiency and performance due to circuit optimizations, however, they require more time for development and manufacturing. A change in the functionality of ASICs necessitates redesign and production of the device from scratch.

To configure an FPGA, desired functionality is described using a Hardware Description Language (HDL) such as VHDL or Verilog HDL. Then, the design is passed through several transformations (synthesis, placement, and routing) and a file known as bitstream is generated at the output [17]. This file is used to program the device, specifying which logic blocks in the FPGA will be utilized and how they will be interconnected to each other. The configuration of the FPGA then remains static until it is powered off or reconfigured. An important feature of modern FPGAs is the Dynamic Partial Reconfiguration (DPR). Selective areas of an FPGA can be reconfigured in run-time by using DPR, while the rest of the system remains unchanged [18]. To achieve this functionality, FPGA is split into regions defined as static and partially reconfigurable (PRR). Two or more partially reconfigurable modules (PRM) can be loaded onto each PRR [19]. These modules can be designed to enable additional functionality which is not part of the original design or carry an alternative implementation of it.

Side-channel Attacks and Countermeasures

AES maintains its performance and efficiency throughout implementations in both hardware and software and is known to be among the most secure encryption algorithms [20]. Currently, the only known feasible attacks that can extract the secret key of AES are side-channel attacks (e.g. power analysis, electromagnetic analysis, etc.), as well as fault injection attacks. Power analysis attacks were pioneered by P. Kocher in the late 1990s and shown to be successful [21], [22]. The underlying principle of side-channel attacks is exploiting weaknesses in the system that implements the encryption algorithm, rather than the weaknesses in the algorithm itself. Typical properties of a system that may leak information about its operation are power consumption, electromagnetic radiation, timing information, etc. Power consumption analysis is among the most widely used and studied approaches and has been proposed by Kocher et al. in [23]. This method relies on collecting measurements of power consumed by the system during its normal operation. Then, statistical methods can be used to analyze acquired data and extract the secret information. Several variations of this attack are available among which simple [24], differential [23], and correlation power analysis [25] are the best known (see also IPA [26]). They rely on the principle that electronic circuits consume power proportional to the data being processed, due to the nature of transistors used to implement these circuits. Attackers can analyze and find the secret key with a leakage model based on certain properties of the data processed [27].

An important factor in the success of power analysis attacks is repeated measurements of power consumed under normal operation of the device. If the configuration of the device is static and does not change over time, then the statistical methods used by the attackers will be successful. Since PRMs can be alternative implementations of the existing functionality, they could be used to increase immunity of the design against side-channel attacks as described earlier. DPR could be used to switch between configurations of the design that maintain logical functionality but differ in physical implementation. Alternating between these configurations introduces deviations in power consumption measurements, thus, reducing the effectiveness of statistical analysis. A common criterion to evaluate the immunity of a design against power analysis attacks is the number of repeated measurements required. Increasing the number of these measurements is considered to make the attack less feasible due to more time necessary to acquire and more processing power to analyze collected data [28].

Several methods and approaches exist to design PRMs that introduce implementation diversity by using different logic blocks, interconnect routing, or adding extra circuitry. Approaches vary based on what level of the design is considered for the modifications – algorithmic, synthesis, or placement and routing. Several studies have explored the first and last methods of generating variants and will be introduced later. Only one recent work has been identified as using the synthesis-level modifications. This work also proposes variants on that level, discussing them and presenting experimental results for evaluation.

Chapter 2

Background

Three main concepts were introduced in the previous chapter – encryption (AES), reconfigurable modules, and side-channel attacks (SCA). The following sections will describe each of these concepts in further detail.

2.1 Advanced Encryption Standard

Advanced Encryption Standard is a method to encode data such that its true meaning is hidden and is accessible with a secret key that should be possessed by authorized parties only. This standard was defined by the National Institute of Standards and Technology of the United States of America in 2001 and is based on the Rijndael Block Cipher algorithm [13], [29]. AES is widely used around the world in different applications and is implemented both in hardware and software, or a combination of the two. It was developed as a successor to Data Encryption Standard (DES) that was in use at the time but was becoming susceptible to brute-force attacks [30]. The Rijndael algorithm is a block cipher, meaning that it accepts data in larger groups of bits, rather than one bit (or a byte) at a time like stream ciphers [31], [32]. Input data is known as the plaintext and a secret key is used to encode the incoming plaintext and produce a ciphertext as the output. Rijndael algorithm was designed to support varying data and secret key length, however, for AES, the secret key has been defined as 128-bit, 192-bit, or 256-bit long string and the data block length is fixed at 128 bits. The same key can be used to encode the plaintext or decode the ciphertext, meaning that AES is a symmetric block cipher [13].

Rijndael algorithm (further referenced as AES) works based on the iterative confusion and diffusion principle for cryptographic systems described by C. Shannon in [33]. The underlying principle of AES is also referred to as a substitution-permutation network [34]. Confusion ensures that every bit of the output data (ciphertext) is represented by multiple parts of the secret key, thus, lessening the connection between the two and increasing ambiguity of the ciphertext. AES replaces every bit of the input from a non-linear look-up table to eliminate patterns that might make

it easier to deduce the plaintext or the secret key. Diffusion is intended to spread the dependency of one part of the input on the output and vice versa. AES performs both of these operations multiple times in stages known as rounds. For three defined key lengths of 128, 192, and 256 bits there are 10, 12, and 14 rounds of transformations, respectively. Operations performed within these rounds are discussed next. The steps of the algorithm are summarized below.

- **Key expansion:** round keys are derived from the secret key for every round; an additional round key is derived for the final round
- **Initial round:** plaintext arranged as a 4x4 byte matrix is added to a round key using bitwise-XOR operation for every byte – this operation is known as AddRoundKey
- **Intermediate rounds:** 9, 11, or 13 rounds (for 128, 192, 256-bit keys) of the following operations:
  - *SubBytes:* each byte of the state matrix replaced with a byte from a non-linear look-up table
  - *ShiftRows:* last three rows of the state matrix are cyclically shifted to the left by 1, 2, and 3 positions
  - *MixColumns:* each column of the state matrix is linearly transformed using matrix multiplication
  - *AddRoundKey:* state matrix is combined with a round key using bitwise-XOR
- **Final round:** SubBytes, ShiftRows, and AddRoundKey operations are repeated for the last time and the output (ciphertext) is produced.

First, AES begins with arranging the plaintext into a 4x4 byte matrix, known as the state. The secret key is also arranged in such a matrix. This key is used to derive several round keys based on a schedule defined in the Rijndael algorithm. Each round key is an expanded version of the original key, configured as a 4x4 byte matrix, and thus longer 192 and 256-bit keys require more rounds to complete all transformations required for the encryption. For AES-128, AES-192, and AES-256 a total of 11, 13, and 15 keys are required, respectively. After initializing the state with the plaintext the first round key is added to it by performing a bitwise XOR operation. This is an operation that outputs a logical “1” only if one of the inputs is “1” and the other is “0”. Otherwise, output is set to “0”. Figure 2.1 depicts the AddRoundKey operation.

The next operation is SubBytes, shown in the Figure 2.2. Here, each byte in the state matrix will be replaced with a byte from the look-up table known as the S-box which is defined by the Rijndael algorithm. It is designed to prevent bit pattern propagation of the current state to the next one. This achieves high non-linearity of the output and thus makes AES a strong encryption algorithm [35]. Subsequently, the highest number of bits changing at the same time happens during this operation. This nature should be kept in mind when the operating principle of side-channel attacks will be explained in the next section.

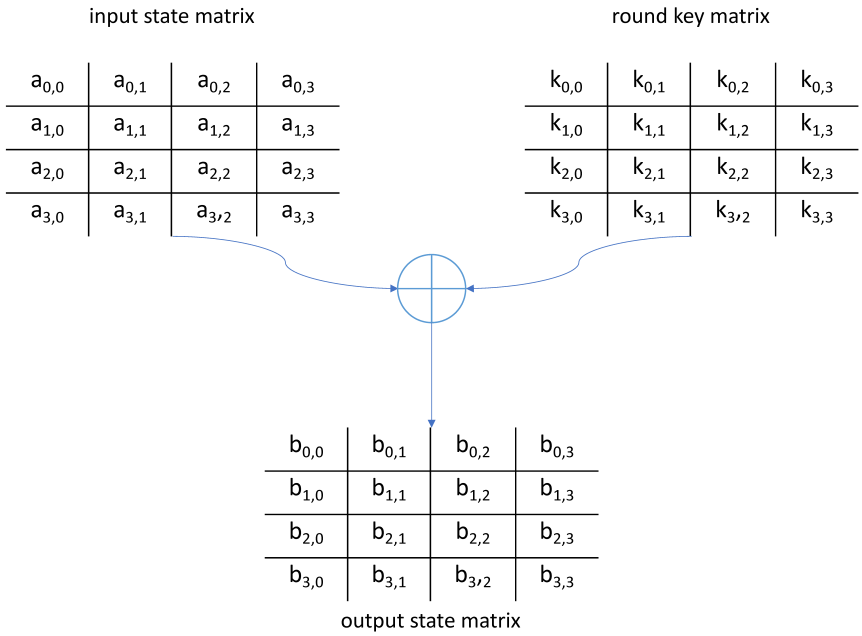

Figure 2.1: AddRoundKey function of AES

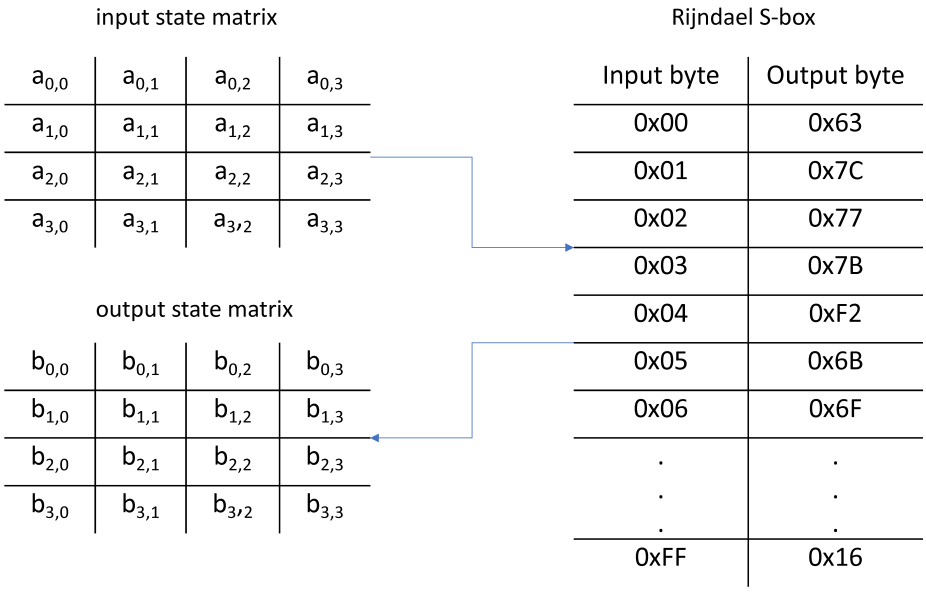

Figure 2.2: SubBytes function of AES

The third operation within AES is ShiftRows. Here the second, third, and fourth rows of the state matrix are cyclically shifted to the left by one, two, and three positions, respectively. The first row is left unchanged. This operation contributes to the diffusion principle. Refer to Figure 2.3 for a visual representation.

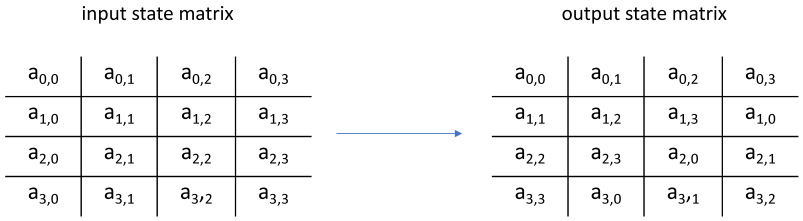

Figure 2.3: ShiftRows function of AES

The last operation is MixColumns, where each column of the state matrix is multiplied with a 4x4 matrix. Each byte of this matrix is based on the four-term polynomial $a(x) = 3x^3 + x^2 + x + 2$. This operation produces a new column where each byte is now dependent on the original input bytes, further contributing to the diffusion principle. See Figure 2.4 for reference.

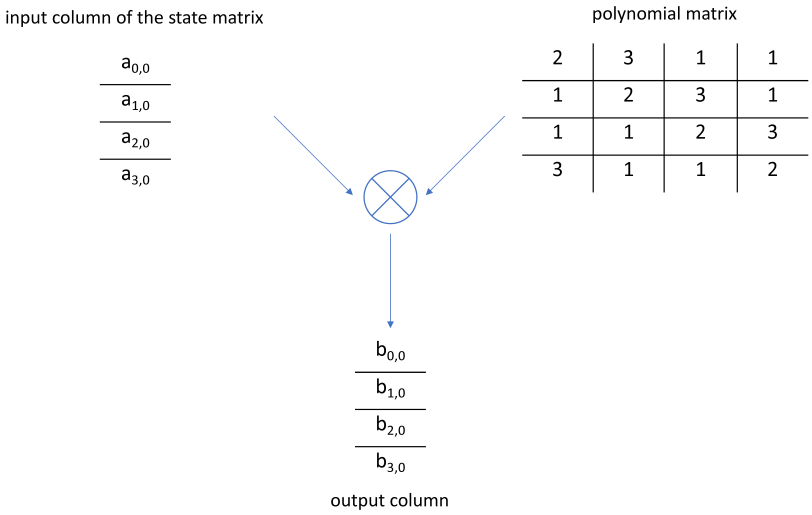

Figure 2.4: MixColumns function of AES

2.2 Correlation Power Analysis

Cyberattacks can be categorized as invasive and non-invasive. Each group can have logical and physical types of attacks [36], [37]. Logical attacks usually exploit vulnerabilities in software or by phishing users to trick them into giving attackers access to their private data. Software attacks attempt to bring the system to a state where it will perform functions it was not supposed to or completely halt its operation (denial of service). Physical attacks, on the other hand, imply that malicious parties have direct access to the device or system that they are interested in exploiting. Attackers can use reverse engineering, circuit modifications, microprobing, and simply damaging the device and these would be considered as invasive physical attacks [38], [39].

Side-channel attacks are known as non-invasive physical attacks. A malicious party does not damage or interfere with the system but rather exploits its physical properties and tries to use

found vulnerabilities to their benefit. Attackers can introduce faults or glitches in the data stream or the operating clock. Details about the secret data could be revealed due to the system propagating those intentional faults and the attackers observing the behaviour and the outputs of the system. Operating frequencies, temperature, or power supply levels could manipulated to trigger improper functioning of the system and induce a fault or a glitch. These types of attacks can be classified as active. Passive attacks, in contrast, are carried out by analyzing power consumption, electromagnetic radiation, or timing of the operations. The malicious party does not interfere and only observes the system collecting information about its behaviour. Then, performing statistical analysis of the acquired data can reveal the secret information that the attackers were seeking [40], [21], [22].

Among the aforementioned passive attacks power consumption analysis is of the main interest to the discussion in this work. It starts with the attacker measuring how much power is consumed by the system during its normal operation and once enough data points have been acquired, they can use statistical methods to analyze that data and extract what they were looking for. The underlying principle of these attacks is based on the fact that when an electronic circuit comprised of transistors processes a logical value “1” it will consume more power than if it was processing a “0” [41]. This is due to the structure of a CMOS transistor - the basic building element of most digital electronic circuits available today. These transistors act as electrical switches by allowing current to pass when they are switched “on” (by applying voltage to the control gate) or blocking current by removing voltage applied to the gate and switching the device “off”. This switching results in variations in the instantaneous power consumed by the overall circuit depending on what data is being processed at that given moment. Therefore, it is possible to observe power consumed and deduce what kind of operations are being performed by the circuit or what data is used during these operations. As an example, AES has a power consumption profile similar to the Figure 2.5, referred from [42]. Ten rounds of encryption can be identified as the ten lowest points, due to the way voltage levels were measured. Systems with a lot of switching activity, such as AES, are susceptible to power consumption analysis and may leak sensitive data despite being immune to regular differential and linear cryptanalysis methods [43].

The power analysis attack was first introduced by P. Kocher and demonstrated to be successful [41]. Depending on the method of analyzing data, three main types of power analysis attacks can be identified:

- Simple power analysis (SPA)
- Differential power analysis (DPA)
- Correlation power analysis (CPA)

Details of the analysis methods presented next are based on [42] and [44]. All power analysis methods begin with acquiring measurements of consumed power (traces) by monitoring supply

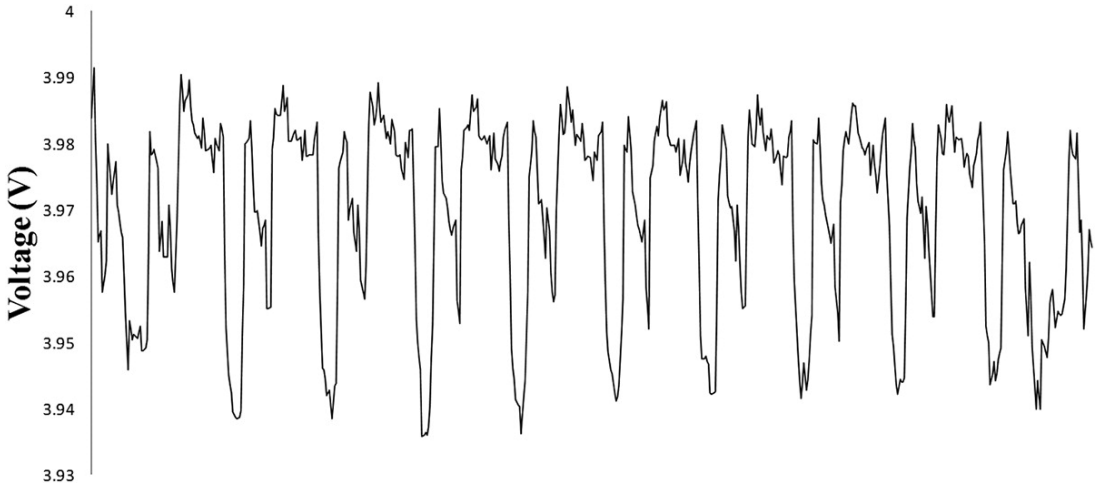

Figure 2.5: Example power profile of AES-128, as measured from Arduino Uno [42]

line voltages or current. Traces can be collected and displayed using an oscilloscope or transferred to a computer for further detailed inspection. Simple power analysis (SPA) implies that the attacker looks at the profile of acquired traces and tries to identify what kind of operation was performed at a given moment. Functions that require more power will appear as spikes. This method works well for systems with little background noise from operations in parallel or in the absence countermeasures such as intentionally adding noise or equalizing power consumption. SPA is thus more useful to identify design features, such as what kind of encryption algorithm is used and any other information about timing and power levels that might be helpful for a more in-depth analysis.

Differential and correlation power analysis techniques (DPA and CPA, respectively) are more thorough and require a higher number of traces compared to SPA. The underlying principle of both of these methods is similar but differs how the correlation between the hypothetical (guessed) and the actual measured power consumption is calculated. For DPA, the attacker uses a selection function $D(C_i, K_s)$ where C_i is the i^{th} ciphertext that was recorded during the acquisition of power consumption traces and K_s is the attacker's guess of a subset of the secret key with $0 \leq K_s < 2^8$ for a byte-long subset. The attacker then computes the difference between the average of the traces for which $D(C_i, K_s)$ is equal to one and the average of the traces for which selection function D yields a zero. If the guess K_s was incorrect, then the difference between the two sets of averages will approach zero as the number of points approaches infinity since there is no correlation. If the guess K_s was correct, then there will be a spike in the plot of the computed difference and the attacker knows that they have found a subset of the key. Repeating this procedure for the remaining subsets will reveal the entire key.

In the case of a correlation power analysis (CPA), the attacker also begins with measurements and guessing a subset of the key. However, a power model will be used to calculate the correlation

between the guess and the actual measurement. A power model can be the Hamming weight H_w of the binary value of interest, which is defined by the number of bits set to "1" [45]. For example, for a byte value "011101001", $H_w = 5$ as there are five bits set to "1". As mentioned, the number of the bits changing their value is correlated to the power consumed – the more bits are switching, the more power would be required at that instant. Therefore, it is possible to show that the Hamming weight of a binary value is related to the power consumed to process that value. To quantify this relationship, CPA uses Pearson's correlation coefficient commonly used in statistics to evaluate the correlation between two variables [46]. Expressed as $\rho(X, Y)$, it is defined as the ratio of covariance $cov(X, Y)$ of two variables X and Y to the product of their standard deviations σ_X and σ_Y. Since covariance is defined by the mean and probability values of X and Y. Pearson's correlation coefficient can be re-written as in the Equation 2.1.

$$\rho(X, Y) = \frac{cov(X, Y)}{\sigma_X * \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X * \sigma_Y} \tag{2.1}$$

$$\mu_X = E[X] \tag{2.2}$$

$$\mu_Y = E[Y] \tag{2.3}$$

$$\sigma_X^2 = E[X^2] - E[X]^2 \tag{2.4}$$

$$\sigma_Y^2 = E[Y^2] - E[Y]^2 \tag{2.5}$$

Equations 2.2 - 2.5 define the components of $\rho(X, Y)$. μ_X and μ_Y are the means, and E is the expectation or probability of X and Y, respectively. Further expansion of the formula for $\rho(X, Y)$ is possible but not relevant to the discussion in this work. Pearson's correlation coefficient can take values in the range [-1, +1], where:

- $\rho(X, Y) = -1$: Y always increases when X decreases
- $\rho(X, Y) = +1$: Y always increases when X increases
- $\rho(X, Y) = 0$: Y is independent and not correlated to X

It is now necessary to apply the principle of CPA to the AES algorithm described earlier. As mentioned, the attacker would be interested in the point where intermediate values change the most – that is the point in the algorithm where the highest number of bit swapping happens and subsequently results in higher power consumption. For AES, this is the SubBytes operation where all 16 bytes of the state will be replaced with different values from a pre-defined look-up table. This table, known as the S-box, is designed to be non-linear and prevent propagating bit sequence patterns of the input data to the output. As such, a design choice for AES that protects it from regular

cryptanalysis techniques makes it vulnerable to side-channel attacks.

The attacker picks a point in the algorithm where part of the intermediate value is known, such as the input of the S-box in the first round (alternative locations, such as the output of the last round, are also possible). Recall that at this point in time, the original input plaintext is bitwise XOR'ed with the secret key and passed as the input to the SubBytes function. For this intermediate value the power estimate can be modelled as $H_w[Sbox(p_d \oplus i)]$. Here, p_d refers to a byte of the known input plaintext and i refers to attacker's guess for a byte of the secret key. The attacker can then calculate the Hamming weight of each of the $2^8 = 256$ possible combinations for each key-byte guess. Assuming that the attacker has obtained a sufficient number of points with a sufficiently high sampling rate, it is now possible to define two variables $t_{(d,j)}$ and $h_{(d,i)}$, for which correlation would be calculated:

- $t_{(d,j)}$ - point j in trace d for D power traces with T points, given that:

$$1 \leq d \leq D \text{ and } 0 \leq j < T$$

- $h_{(d,i)}$ - point i in estimates for a trace d with I guesses, given that:

$$1 \leq d \leq D \text{ and } 0 \leq i < I$$

Having these two variables it is possible to calculate their correlation $r_{(i,j)}$ for each measurement j and each guess i for D traces in total. Then, the attacker will pick the highest values of $|r_{(i,j)}|$ for every i, using absolute values since only the presence of correlation is important. The highest r_i will indicate the guess with the highest correlation which corresponds to a byte of the secret key. Performing this procedure for the remaining bytes will yield the full secret key used for the encryption. Considering AES-128, brute-force check of every possible key combination would require 2^{128} tries for all 16 bytes of the secret key. This is a very large number with an order of magnitude equal to 38. Using CPA, the number of guesses is equal to $16 * (2^8) = 2^{12}$, or just 4096 tries, which is a substantial reduction in the attack complexity and these guesses can be analyzed in a shorter amount of time. Countermeasures to prevent or mitigate this type of power analysis attacks are discussed in the next chapter.

2.3 Field Programmable Gate Arrays

As it was mentioned previously, Field Programmable Gate Arrays (FPGAs) are devices that can be programmed to execute any function in hardware by configuring the connections between basic computational logic blocks (CLBs) which contain look-up tables and flip-flops to store data [47], [14], [15], [16]. This configuration is usually loaded when the device is first powered and retained until it is either overwritten or the power is removed. During this process, the entire FPGA is programmed, even if only a part of CLBs have changed their configuration. Dynamic Partial

Reconfiguration, shortened to DPR, is a feature that allows reprogramming pre-defined parts of FPGA, referred to as reconfigurable regions, rather than the entire device [48], [49]. This enables the ability to implement additional functions that are not always required and can be replaced by different functional blocks, rather than being disabled or consuming residual power even when not being actively used [50], [51], [52], [53]. Examples can be an arithmetic logic unit (ALU) that is only required to execute simple mathematical operations such as addition and subtraction. When the system is required to perform a more complex operation, this ALU can be swapped with an alternative, more functional one. Once the result is ready, the simpler ALU can be brought back. Overall, this swapping may lead to better efficiency and performance of the system. Another example can be demonstrated in the context of approximate computing where calculations are executed with intentional deviations or inaccuracies that allow for much faster, simpler circuits with lower footprint and energy consumption. When the result is ready to be sent to the output, DPR allows loading a functional block that will correct intermediate deviations and produce a precise result. Thus, the circuit can produce results faster and use less energy compared to only using the precise functional block for the whole operation. The tradeoff is some inactivity time while DPR is in progress and the extra memory that is required to store the configuration files used to re-program parts of the FPGA. During this process, the rest of the FPGA remains functional and executes its originally programmed functionality. This region is referred to as static and usually includes, but not limited to, the logic that controls the reconfigurable regions.

2.3.1 FPGA Design Flow

To program an FPGA, it is necessary to develop the desired functionality in a hardware description language (HDL) such as VHDL or Verilog HDL, similar to developing software in programming languages like C or C++. The HDL sources are then passed through several transformations to generate an output file called the bitstream. This is a binary sequence that will be used to configure CLB interconnects, look-up table initialization values, and any other parameters that will enable the FPGA device to execute the functionality described by the HDL source. This bitstream can be loaded to the FPGA board via a computer and a wired connection or stored in an on-board memory chip and used by a dedicated control circuit to configure the FPGA every time it is powered on. The process is similar for DPR, however, as there is now a requirement to have alternative configurations that could be loaded to a reconfigurable region, multiple bitstream files have to be generated. These bitstreams are then stored in an on-board memory or some external location that is accessible to the DPR controller that takes care of reading and loading new configuration to the reconfigurable region on the FPGA [54], [55], [56]. To summarize, the levels are listed below:

- Algorithm level development
  - Defining the architecture, i.e. system-level design

- HDL level development
  - Implementing functionality in VHDL or Verilog, i.e. register-transfer-level design
- Logic synthesis
  - Processing abstract descriptions of HDL sources to produce the gate-level design
- Implementation
  - Mapping the gate-level description to the FPGA fabric, i.e. switch-level design
- Generating bitstream
  - Producing a file that is used to load results of the implementation stage to the FPGA

Development starts at the system and HDL design levels, where desired functionality and system behaviour is described using certain keywords and constructs that can be translated to simpler logical operations, such as storing a value in a register, performing an arithmetic operation, etc. The process of translating an HDL source into a lower level is called synthesis and there exist several tools capable of this process, usually provided by the vendors of the FPGA device. During synthesis, constructs in the HDL source are identified and represented at the gate-level. For example, the HDL level may describe an entity that stores a multi-bit value on a certain periodic event by latching the value available at its input. That value is then available as the output from this entity until it is overwritten with a new value or it is reset by an external trigger signal. This entity is recognized as a register and at the gate-level it is implemented as a special primitive called a flip-flop. Primitives are basic unit blocks that can implement a certain type of logic. Usually, FPGA devices will contain a matching hardware resource that is capable of executing the same behaviour. During synthesis, besides translating to gates (simple Boolean functions such as XOR or flip-flops), the logic described in the HDL is also optimized to produce a circuit that uses the least amount of resources, delivers the best performance, or attempts to achieve a trade-off in between the two extremes. Once the synthesis procedure is completed, the design is described as a netlist file in a specific format and syntax understood by the implementation stage.

The implementation stage consists of further logic optimizations and mapping a set of primitives to the available resources of the intended FPGA device. Most Boolean functions are mapped to look-up tables, however, if the device provides a digital signal processing unit (DSP), or any type of a dedicated functional unit (FIFO, interface controllers, etc.) and if the original design contained a description matching the functionality of these blocks, this stage will attempt to make use of these resources to deliver the most optimal solution. The implementation stage places all the functions in the CLBs available and also connects them using the interconnect resources of the FPGA device. This process is generally referred to as placement and routing, often shortened to P&R (or PnR). The output of the implementation stage is a netlist that contains information about which CLBs on the FPGA should be used, what values they should be initialized with, and how they are interconnected. This netlist is then used to generate a bitstream file that will include any

other information necessary to program the FPGA. Generally, all of the stages introduced above are contained within the FPGA vendor-provided software suite, such as Intel Quartus, Xilinx Vivado, or Lattice Diamond [57], [58], [59]. It is also possible to use third-party tools from different vendors, as well as solutions developed by the open-source community. However, their availability and functionality might be limited compared to vendor-provided tools [60].

2.3.2 Design of Partially Reconfigurable Modules

As mentioned, additional functions that will be loaded into partially reconfigurable regions (PRRs) may be completely new to the system or implement an existing function differently, e.g. energy efficiency or performance-focused variations of the same function. In both cases, it is first necessary to develop these partially reconfigurable modules (PRMs). While for a new function the design process will start at the HDL level or higher, such as the abstract system-level, alternative implementations of an existing function can be generated from the lower levels too. Each level has its disadvantages and advantages over the others. Depending on the requirements, any level may provide the best result. Generally, the flexibility and available design space freedom decreases from the algorithm to the implementation level. When the design is at the architectural or system-level, developers may change the order of the operations, use different functions that give the same result, etc. Once the design is described in an HDL, its implementation details start to become more visible. Although, it is still possible to alter minor details to produce the desired result, such as performance-optimized or area-efficient multiplication. At the output of the synthesis process, the design is described as a set of primitives (i.e. basic logic blocks) and is less open to modifications. After the implementation level the design is fully constrained to the physically available resources of the selected FPGA device. Depending on the original requirements, it might be possible to introduce changes to the design only after P&R – an example would be protected source files, proprietary pre-synthesized or pre-implemented IP cores, etc. This would require extra effort from the designer as at this stage the design is not easily read or understood by a human since it's intended for the machine processing. Changes on the architectural level could introduce deviations from the intended functionality or standards. Additional time and effort may be required to identify what parts of the design are more suitable for changes, such as understanding the theoretical background of an algorithm (complex signal processing, encryption algorithms, etc.).

HDL level changes may require the least amount of extra effort from the designer, as this is the primary level at which most projects are developed [61]. However, in the case of protected sources, as mentioned already, this might not always be an available option. It is also possible that by the project requirements, the HDL sources are to be kept unchanged, even though they are openly available. Another option is to generate alternative implementations of the original design during the synthesis process. Designers could introduce modifications if they have control over

this process. The representation is relatively high-level and changes can be propagated through the logic optimization, mapping, and further down to the P&R level. However, this approach is limited in options when using a vendor-specific software suite. Algorithms and functions to synthesize a design are proprietary intellectual properties of FPGA vendors. Usually hidden from the end user, only certain options are available that will operate on the entire design. Open access to the HDL sources may be required to split the project into smaller ones and target a specific part of the design. Some examples of synthesis options available within a vendor tool are restricting the size of the look-up tables or changing logic optimization goals (area- or performance-driven). In certain cases, these options may suffice, however, it would be preferred to have a greater degree of control to produce a well-optimized alternative design variant [62], [63].

2.3.3 Third-party tools for FPGA Development

Third-party FPGA development tools may be useful for cases when the vendor-specific tool is not sufficient for the designer's needs. These tools provide greater flexibility to the synthesis process by having free, unrestrained access to all of its parameters. Although they may be based on algorithms less optimized for a particular FPGA platform when compared to the vendor-supplied tools [64], [65], [66]. This means that an open-source synthesis tool could deliver a netlist that will use more resources than a vendor-generated one under the same optimization goals (area or performance). In practice, however, these differences are not significant enough, as will be shown by analyzing resource utilization of variants proposed within this work. If the implementation is done by the vendor tool, it can optimize the design further and minimize deviations from a design completely processed by the vendor's tool.

Out of several options available, some tools are either obsolete, e.g. SIS [67], or aim to be replacements to vendor-specific tools and rely on academia developed algorithms, e.g. ODIN [68] and Verilog-to-Routing projects [69]. Other examples include RapidSmith [70], Torc [71], IceStorm [72], etc. Another option is to use an open-source tool such as Yosys, originally developed by Clifford Wolf [73]. Yosys performs synthesis and links other open-source driven tools for logic optimizations (ABC from Berkeley Logic Synthesis and Verification Group [74]) and is also available as a commercial software suite including tools for P&R [75]. The following discussion is about the open-source version available on an online code sharing and version control platform GitHub [76]. Yosys can be modified by any user to their needs and it provides a greater degree of control over the synthesis process when compared to vendor-specific tools. A special internal structure is used to translate and transform an input design. Yosys produces synthesized netlists for the implementation step in formats compatible with most open-source and vendor-specific tools (EDIF or BLIF). Some parts of the inner workings of this tool are described in Chapter 4.

Chapter 3

Countermeasures to CPA

This chapter discusses countermeasures to the correlation power analysis attack described earlier. Masking and hiding techniques using dynamic partial reconfiguration and published research papers related to this work will be introduced.

3.1 Methods Overview

As it has been discussed in the previous chapter, correlation of processed data and power consumed is one the main factors in the success of side-channel attacks. Therefore, to protect the system and avoid compromising the privacy of sensitive data, it is necessary to minimize this correlation. Countermeasures can be divided into two main categories – masking and hiding. While the former attempts to manipulate the data being processed, the latter aims to affect the power consumed during the operation of the system [77]. In the following sections selected papers are used to demonstrate these approaches.

3.1.1 Masking

The idea behind masking type countermeasures is to not use the true value of the data for processing. Instead, values are masked through mathematical transformations that can be reversed when the true value is requested. The attackers will collect and analyze power consumption of the system while it processes false values. Thus, minimizing correlation between the true values and power consumed reduces effectiveness of the attacks. This approach has been first introduced by Messerges in their work on analyzing AES candidate algorithms [78]. The concept of manipulating data to increase side-channel attack immunity was already proposed in the works of Chari et al. [79] and Goubin et al. [80] for DES (the predecessor to AES), both of which are referenced by Messerges. These studies suggested a method to mask data using some secret scheme that would distribute the intermediate values during processing. It would require attackers to perform a more thorough statistical analysis and look at multiple points in the measured traces. It was proposed to only mask some of the intermediate values, such as the first and few of the last rounds

of DES. Inferential power analysis (IPA) was developed by Fahn et al. [26] and demonstrated the possibility of attacking middle rounds of the encryption algorithm and thus avoid the implemented countermeasure. Messerges avoided this drawback by proposing to mask all of the intermediate values for all rounds of encryption. Data is masked by applying either a Boolean transformation (such as bitwise XOR proposed by Messerges) or an arithmetic operation (addition, subtraction, modulo) to the intermediate values during encryption. However, not all of the operations within the encryption algorithm can use both methods. Boolean masking is not supported by addition and multiplication [78] and arithmetic masking is used for these. Then, it would also be necessary to convert between the two approaches from one operation to another. The mask itself is randomized and applied to both the plaintext and the secret key prior to the execution of the algorithm. Thus, the encryption process is completed with randomized masked values and simpler statistical analysis methods will not be successful. Attackers will need to perform higher-order power analysis and use joint probability distribution functions to analyze multiple points in the measured traces.

3.1.2 Hiding

The idea behind hiding is to directly influence the power consumed by the system and thus reduce the correlation of the processed data to the power consumption [77], [81]. Several methods have been proposed, such as reducing signal to noise ratio [82] or implementing dual-rail logic technique such as the Wave Dynamic Differential Logic (WDDL) [83]. The latter requires a secondary, copy circuit of the design which will operate on data that is complementary to the main, true values. In the context of FPGAs, this method was first presented by Yu and Schaumont as part of their proposal of a symmetrical routing method. This technique was found to be useful in increasing immunity against side-channel attacks [83]. The idea is based on the following expectation - that the total power consumed over time would be equalized if true and complementary values of data are processed in parallel by two identical circuits. When the true value is a logical “1”, the secondary copy of the circuit will operate on a logical “0”, and vice versa. Thus, power consumption profiling and data to power correlation would be minimized or eliminated. However, achieving this effect requires an exact copy of the main circuit. The discrepancy between the two circuits has to be minimized to achieve equal power consumption levels. This is a straightforward task in the context of ASICs where the circuit design can be copied to maintain its symmetry. For the FPGAs, however, this approach requires inferring the same placement and routing for both parts of the whole design. To achieve this, Yu and Schaumont [83] have developed a custom routing algorithm which controls the implementation stage of the FPGA design flow. The symmetry between the two circuit copies is achieved and discrepancies that lead to the success of statistical analysis are minimized.

While the dual-rail complementary logic approach has been shown to be successful, it also

comes with a significant penalty to performance – the authors reported 20.88 ns delay for the secure implementation, compared to 3.99 ns delay for the unmodified design. Area overhead is also significant – 818 LUTs in the case of Double Wave Dynamic Differential Logic (DWDDL) and 70 LUTs for the unmodified circuit. Note that DWDDL is essentially a double copy of WDDL. The latter approach utilized 409 LUTs and was not found to be secure against the DPA attack performed by Yu and Schaumont. As expected, doubling the logic to achieve full symmetry and parallel processing results in twice the amount of area and resources utilized. Thus, the decision to use this approach should be made with caution during design planning.

3.2 Literature Review

Considering the concepts introduced above and in the previous chapter this section will present how they can be combined by referring to relevant literature.

First, the work of Mentens et al. will be introduced [84]. Published in 2008 by the International Association for Cryptologic Research (IACR), this paper presents dynamic reconfiguration to increase resistance against power and fault analysis. The proposed technique is based on temporal and spatial jitter introduced on the architectural level. The second work to be reviewed is by Sasdrich et al. published by IACR 2015, where authors proposed to use run-time configurable look-up tables to achieve dynamic reconfigurability of the original design [85]. This work can be categorized under implementation level changes when considering design levels introduced in the previous chapter. In 2018, Bete et al. published their work at the IEEE International Symposium on Hardware Oriented Security and Trust (HOST) [86], with an extended publication in April of 2020 [87]. Their method is based on introducing HDL level datapath modifications to allow reconfiguring S-box implementations generated from the synthesis level. Additionally, extra logic was introduced to create a randomized delay in latching the data by the state register. Lastly, the paper published by Hettwer et al. in 2019 for Design, Automation & Test in Europe (DATE) conference will be presented [88]. Here, the authors are generating alternative implementations of an FPGA-based AES by randomized constraining of routing and placement process for a single synthesized netlist as the base design. The next chapter will present a countermeasure that is an alternative to this approach, due to their similarities in the underlying principle of preserving the original HDL level implementation, as well as shared measurement setups. Table 3.1 relates selected papers to the design flow levels presented earlier.

In summary, all studies reviewed demonstrate improvements in immunity against the common types of side-channel attacks. A characteristic shared by these approaches is that when countermeasures are implemented individually, their benefits are negligible. Thus, authors recommend combinations of their proposals to achieve a higher degree of immunity. Another shared characteristic is increased resource usage and introducing latency to the encryption process. This is an

Table 3.1: Papers reviewed and corresponding design levels

	Algorithm	*HDL*	*Synthesis*	*PnR*
Mentens [84]	■			
Sasdrich [85]				■
Bete [86], [87]		■	■	
Hettwer [88]				■
Proposed method			■	

expected effect, as the countermeasures proposed are based on extra circuitry introduced to the system, modifications of the existing data path, as well as the overhead from using the dynamic partial reconfiguration system itself. All of these methods are dependent on hardware resources. In general, increased security is shown to come at an increased cost of resources [89], [90].

3.2.1 Mentens et al. in IACR 2008

Power and Fault Analysis Resistance in Hardware through Dynamic Reconfiguration

The work of Mentens and others is aimed at reducing the effectiveness of power analysis and fault injection attacks. The concept of power analysis has been introduced in the previous chapter and fault analysis is based on forcing the system into a state where it does not function as intended. This can be done by exerting physical stress such as temperature changes, out-of-spec power supply levels, slower operating clock frequencies, etc. The authors identify three categories of side-channel attack countermeasures. The first category is for attacks that are meant to make fault injections difficult by spreading the time instant t_c or the physical location at which an operation is executed. The second category is for the detection of an attack, which requires redundancy of execution or comparison of results. The last category covers detection of an attempt to attack, usually done by sensors or special circuits built into the system. The authors consider countermeasures that fall under these categories and are implemented for reconfigurable devices.

To evaluate the countermeasures, Mentens et al. refer to the following two criteria, as defined by Lemke-Rust et al. and their work on An Adversarial Model for Fault Analysis Against Low-Cost Cryptographic Devices [91]. An attack will be considered successful if enough data has been collected to extract the secret key:

- Spatial resolution: p_{volume} defined as the probability of stimulating the correct volume/area of the chip to induce a fault attack
- Temporal resolution: p_{time} defined as the probability of injecting a fault at a time instant that leads to a successful attack.

To proceed further with the countermeasures, the baseline system configuration has to be defined first. Figure 3.1 shows the prototype implementation of AES-128 proposed by the authors

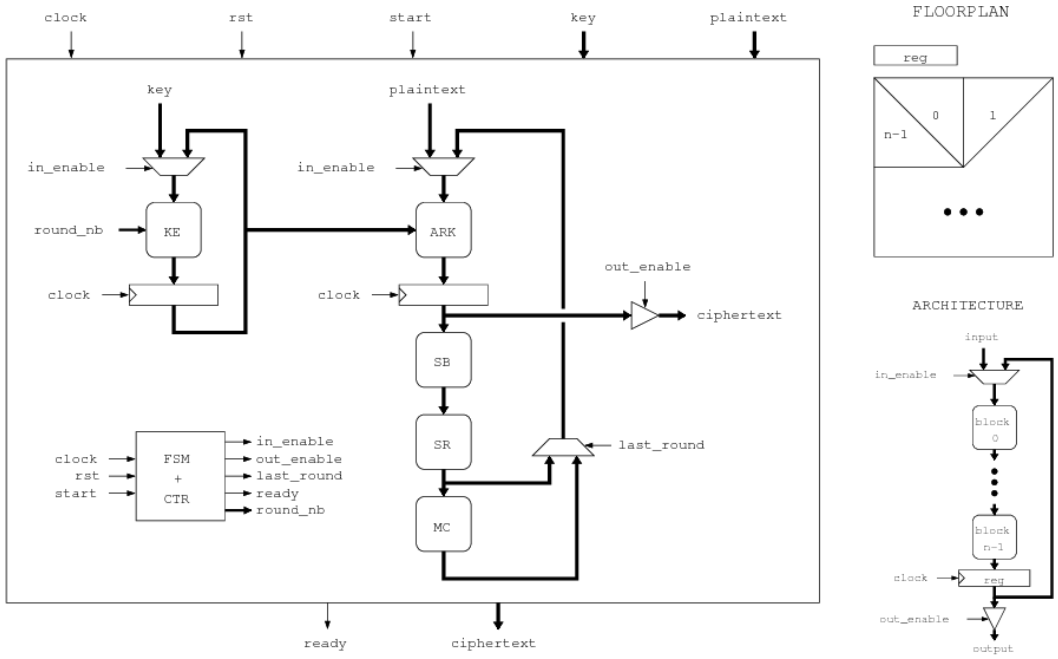

Figure 3.1: Reference design (left); floorplan (top right); general archictecture (bottom right) [84]

(left), floorplan on the top right, and the overall algorithm architecture on the bottom right. The floorplan refers to the spatial arrangement of n functional blocks used to implement subfunctions described in the architecture diagram. Each of these blocks corresponds to repetitive instructions that can be used to implement the encryption algorithm itself.

The first countermeasure under review is called temporal jitter and it is based on spreading calculations in time. For hardware implementations this is achieved by inserting registers between the functional blocks and using multiplexers to choose if a particular register is used. A single register corresponds to a single clock cycle delay, thus de-synchronizing the observations of the attackers. However, as the authors note, having registers in between all blocks would result in a significant resource and area overhead. Thus, a dynamically reconfigurable switch matrix is proposed. Depicted in Figure 3.2, this matrix randomly inserts one or more registers into the datapath. It connects the inputs and outputs of the consecutive functional blocks or re-routes them through the inserted registers. The total number of possible configurations is $m * n$, where m is the number of the registers and n is the number of functional blocks. Allowing to cascade the registers increases the total number of possibilities. Thus, the number of distinct configurations is $c = \binom{n+m-1}{m}$ or the number of combinations of m elements out of n. A random number generator is used to pick one of the possible matrix configurations. Its implementation is assumed to be secure and resistant against the side-channel attacks.

- Motivation: spread calculations over time to reduce the efficiency of power and fault analysis attacks
- Method: dynamically reconfigurable switch matrix that randomly inserts registers between

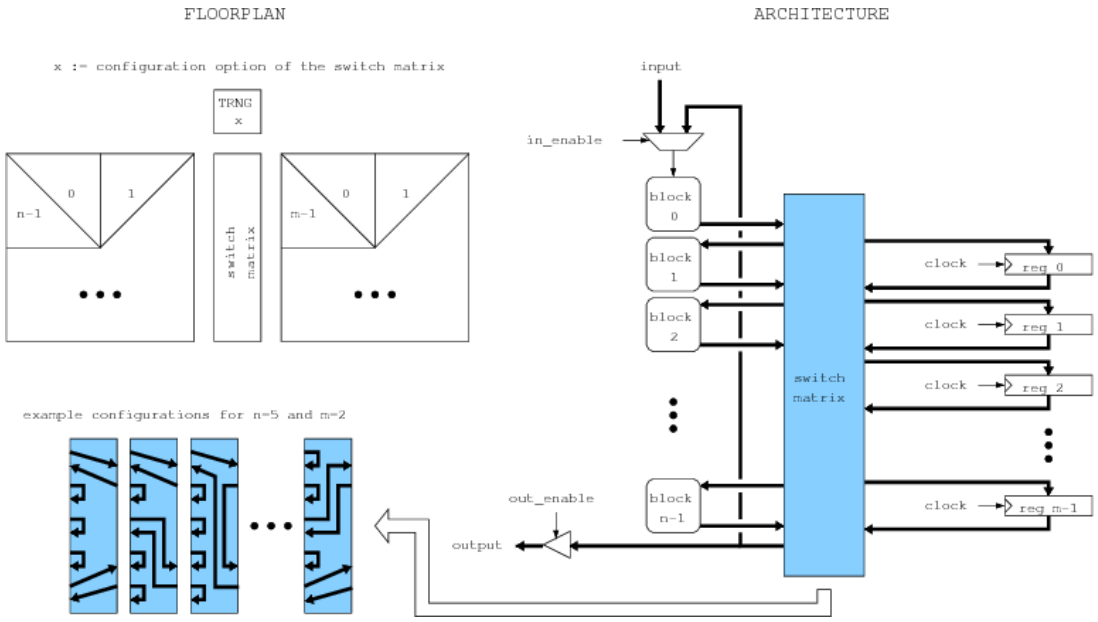

Figure 3.2: Proposed DPR switch matrix architecture to introduce temporal jitter [84]

the functional blocks

- Implementation:
  - Select the number of functional blocks n to be used and registers m to be inserted
  - Generate configurations of the switch matrix
  - Use DPR to configure the next matrix selected by the true random number generator
- Benefits: timing of the operations is now less predictable and thus power and fault analysis attacks will be less successful
- Downsides: resource overhead, output latency proportional to the registers inserted

The authors also present experimental results of using this countermeasure on their fully parallel implementation of AES-128. The following four functional blocks are identified – AddRoundKey (ARK), SubstituteBytes (SB), ShiftRows (SR), and MixColumns (MC). Two registers were set to be randomly inserted between the functional blocks, allowing cascading the registers. Then, the total number of distinct configurations is equal to $c = \binom{4+2-1}{2} = 10$. DPR latency of the matrix was found to be around 3ms. Depending on technological improvements this can be reduced significantly. The operational clock frequency of the system was reduced by approximately three times compared to the original design – 111MHz vs 33MHz, respectively. Splitting the design into static and dynamic parts requires additional control logic, communication between the two domains, as well as an extra 128-bit register. Combined, this leads to a static region that is larger than a fully static unmodified design and slower operating clock frequency. Resource usage overhead was 3251 logic slices (23% of available slices) of the Virtex-II Pro FPGA versus 685 slices (5%) for the original design.

The next countermeasure is presented as a combination of spatial and temporal jitter. In this case, the intent is to protect the design against localized attacks. An example of these would be optical fault injection using a laser-focused on a particular area of the chip, thus triggering an incorrect state of one or more bits of the intermediate data [92].

- Motivation: protect the design against localized fault inducing attacks
- Method: randomly re-locate the subfunctions of the algorithm to introduce spatial jitter
- Implementation:
  - For a given order of execution $[f_0, f_1, ..., f_{n-1}]$, f_i is the function implemented by the block i. Then, the possible number of positions for every block is n.
  - Define regions in the FPGA, where each region can be configured to execute all subfunctions of an AES algorithm (ARK, SB, SR, or MC).
  - Randomly load each of the regions with one of the functions. Order of SR and SB can be swapped without repercussions on the algorithmic level; therefore, the number of possible configurations is higher than the number of allocated regions.
  - Combining this method with the previously introduced one results in the combined spatial and temporal jitter countermeasure.
- Benefits: both timing and location of operations are now less predictable, thus making the design more immune against side-channel attacks. Additionally, implementing all of the subfunctions of the algorithm in the reconfigurable area of the FPGA allows it to recover from attack attempts and overcome short-term effects.
- Downsides: similar to the previous countermeasure – resource overhead.

The proposed architecture for the spatial jitter countermeasure can be seen in the Figure 3.3. Floorplan figure shows the division of the logic area into subfunction and the architecture shows the execution order. Each block drives an input of an OR gate, however, only the last block sends actual value while the rest send zeros. The output of the gate is then the output of the algorithm.

Remaining countermeasure is the detection of injected faults. The authors proposed a method where several bitstreams (one to all) are read back and compared with the reference bitstreams stored in a block RAM resource. However, this implies that the chosen FPGA device can read back bitstreams. The process for comparison is executed by using protected logic gates or by looking at the cyclical redundancy check bits that vendors provided with the bitstream generation. Faults in the bitstreams in the reconfigurable area can be detected only if the reference bitstreams are assumed to be intact and unmodified. The fault model chosen by the authors does not leave room for such an attack and the probability of compromised reference is thus negligible. It is also possible to duplicate and execute the algorithm twice. Done either in parallel or sequentially, this approach comes with large overhead of utilized resources and output latency.

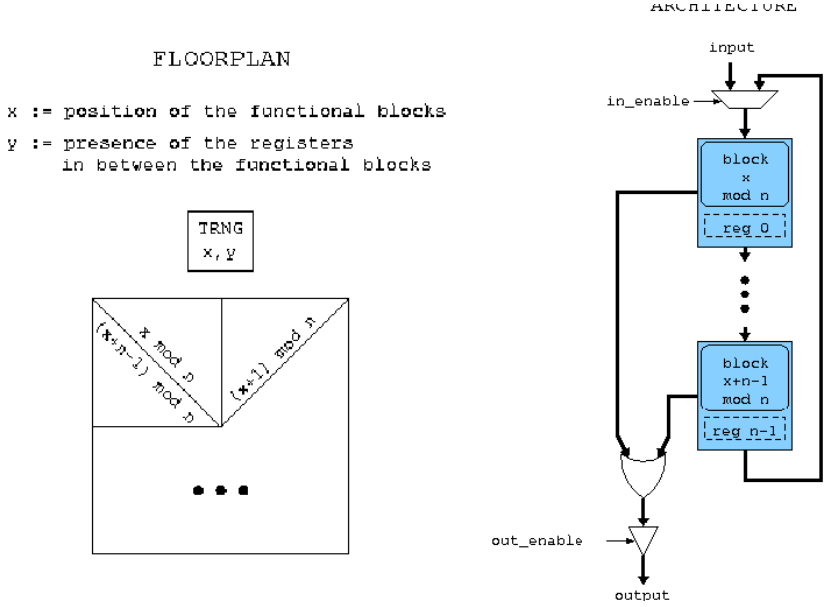

Figure 3.3: Proposed spatial jitter architecture [84]

3.2.2 Sasdrich et al. in IACR 2015

Achieving Side-Channel Protection with Dynamic Logic Reconfiguration on Modern FPGAs

This paper presents a method to increase immunity against side-channel attacks by using configurable look-up tables (CFGLUT) found in Xilinx Spartan-6 FPGAs. This type of LUTs can change its configuration during run-time, as opposed to standard LUTs that are initialized when the FPGA is programmed [93]. The authors demonstrated that the secret key used by the encryption algorithm PRESENT [94] was not found after performing a DPA attack on 10 million traces of power consumption. This algorithm is more lightweight but similar to AES in structure. 64-bit blocks of input plaintext are encrypted by a substitution-permutation network using 4x4 S-boxes. For this countermeasure, the S-box was implemented by using the CFGLUT primitives explained further below.

The authors of this paper also refer to a method based on block RAM scrambling proposed by Guneysu and Moradi [82]. In this case, S-box contents are dynamically randomized in memory. The S-box is stored in a dual-port BRAM, where one side A is denoted as the active context and the other side B is the inactive context. The latter is used to apply a Boolean mask m on the S-box while the A side stores the most recently masked copy of the S-box. The algorithm then interacts with the active side during the encryption process that has the current masked version of the S-box. If the S-box changes during an active encryption cycle it may lead to data mismatch on the output. Scrambling is in parallel with the encryption algorithm and context switching is controlled by an FSM as shown in Figure 3.4.

In their work, Guneysu and Moradi list several possible countermeasure methods, including

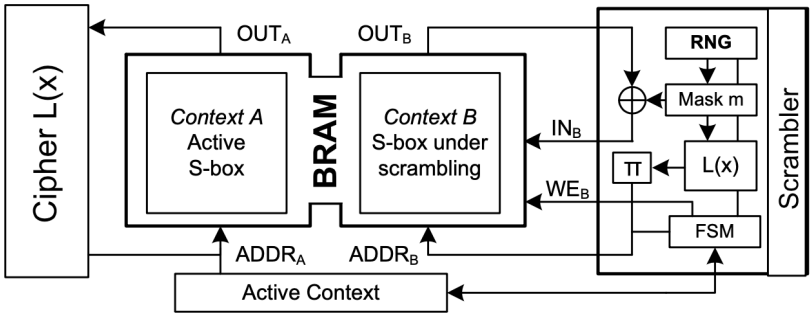

Figure 3.4: BRAM scrambling [82]

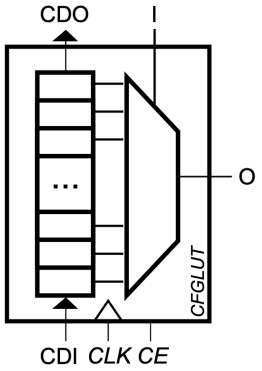

Figure 3.5: CFGLUT [85]

shift-register based LUTs. These look-up tables are similar to the CFGLUTs used by Sasdrich et al. in their work. In fact, for devices that do no support this primitive it is possible to implement the same behaviour by using shift-registers and some additional adjustments. Dedicated configurable look-up tables can be represented as a multiplexer with inputs driven by a shift-register (see Figure 3.5). Using a control signal new data can be loaded into the shift-register, thus changing the configuration of the LUT.

These primitives are limited to 16-bit memory blocks and at most they can only implement a 4-input and 1-output Boolean function. Therefore, the authors presented a Reconfigurable Function Table (RFT) that can implement any Boolean n x m function using CFGLUTs. Boolean $nx1$ functions are built by cascading multiplexers and using CFGLUTs as the inputs. To build any n x m function, it is possible to replicate this structure m times and share the inputs. The resulting structure is a basic Reconfigurable Function Table with $m*2^{n-4}$ CFGLUTs. Referring to the example by the authors, an 8x8 S-box of the AES algorithm can be created as an 8x1 Boolean function instantiated eight times. Sixteen CFGLUTs and fifteen 2-to-1 multiplexers are used to build one 8x1 function and subsequently 128 CFGLUTs sharing their inputs correspond to a full AES S-box. An example of a 6x4 RFT can be found in the Figure 3.6. It should be noted, however, that this method is not efficient for such large S-box configurations. Only 32 6-input LUTs are necessary to implement a standard AES S-box, compared to mapping 128 CFGLUTs to 64 6-input LUTs.

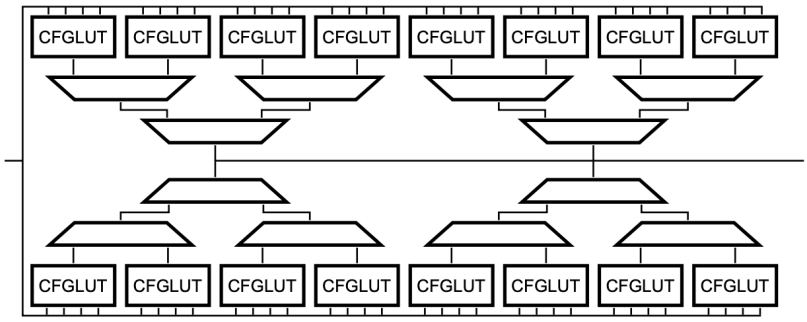

Figure 3.6: 6x4 Reconfigurable Function Table (RFT) built using CFGLUTs [85]

Table 3.2: Implementation statistics for [85]

Design component		**Unprotected**		**Protected**	
		LUT	**FF**	**LUT**	**FF**
Key schedule		48	85	48	85
Round function		128	64	224	128
Countermeasure instance		N/A	N/A	1236	338
Latency	**s-box decomp.**	N/A		16 (avoided if in parallel)	
	reconfiguration	N/A		16	
	encryption	31		62	

To avoid costly doubling of the resources, the authors recommend using this method for more lightweight algorithms, such as PRESENT and its 4x4 S-box.

The structure of the PRESENT algorithm is shown in the Figure 3.7. The round-based implementation of the algorithm uses two clock cycles per round. A total of 31 rounds of operations are performed on a 64-bit block of input data with 32 subkeys derived from the initial 80-bit or 128-bit secret key. Substitution is performed by 16 parallel S-boxes. The authors provide implementation statistics for the unprotected and protected versions of the algorithm, a copy of which can be found in Table 3.2.

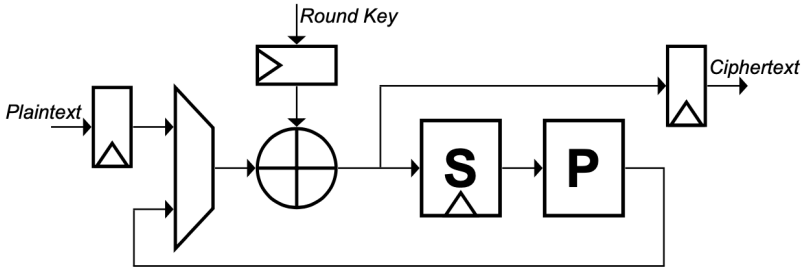

Figure 3.7: PRESENT encryption algorithm datapath [85]

Several countermeasure methods are proposed, as listed below and summarized in the block diagram of an S-box in the Figure 3.8. First one involves decomposing S-box into stages:

- Motivation: avoid storing S-box outputs in registers as they are common targets for SCA
- Method: split S-box into two random mappings storing incorrect outputs in registers
- Implementation:
  - using two 4x4 RFTs configure the first RFT to implement a random bijection $R1$
  - second RFT is configured such that for all values of x, $R2(R1(x)) = S(x)$
  - correct S-box output value is found only by executing both mappings
  - $R1$ and $R2$ are computed before the encryption cycle in which they are used
- Benefits: only intermediate and randomized output of $R1(x)$ is stored in the memory, hiding the true value of the correct S-box output

The next countermeasure is Boolean masking of the S-box contents:

- Motivation: masking intermediate values to decrease data to power consumption correlation exploited by SCA
- Method: implement two random Boolean masks m_1 and m_2 to maintain compatibility with S-box decomposition countermeasure
- Implementation:
  - configure new RFT $R1'$ based on $R1$: $R1'(x) = R1(x \oplus m1) \oplus m2$
    - $R1$ is computed with data masked by m_1
    - $R1'$ then stores output of $R1$ masked by m_2
  - configure new RFT $R2'$: $R2'(x) = R2(x \oplus m2) \oplus P - 1(m1)$
    - $R2$ is computed with data masked by m_2
    - $R2'$ then stores output of $R2$ masked by the inverse of m_1, which keeps the final output of every round masked by m_1
  - 4-bit masks m_1 and m_2 are recalculated for every round to avoid reusing
  - masking is disabled by setting m_1 and m_2 to zero
- Benefits: in addition to S-box decomposition, intermediate values are masked, decreasing efficiency of SCA

The last countermeasure proposed by the authors is register precharging:

- Motivation: the same mask is applied to both input and outputs of a round and a single register stage is used between the consecutive rounds, thus making the design vulnerable
- Method: split the single register stage into two
- Implementation: a dummy encryption round is necessary to propagate through two registers between the rounds
- Benefits: single state register leakage is avoided at the cost of doubling the latency

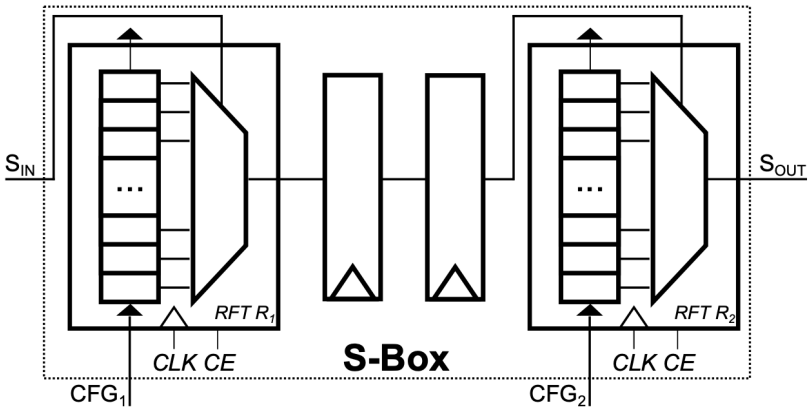

Figure 3.8: S-box using RFTs, masking, and register precharge [85]

Table 3.3: Evaluation setup for [85]

Processing platform	Xilinx Spartan-6 FPGA
Power trace collection	PicoScope 6402B monitoring voltage drop over 1 Ohm resistor connected to Vdd line
Attack and analysis method	Test vector leakage assessment by Goodwill et al. [95] using specific t-test method

The setup described in the Table 3.3 was used for evaluations of countermeasures. The authors employed a statistical t-test [95], rather than the full attack. Known as Welch's t-test or unequal variances test, it calculates the equality of means of two data sets. Using this method, the authors analyzed collected traces and if the t-test confirms the null hypothesis, then the two groups of data are not correlated. For values of t, where $|t| > 4.5$, it shows that an actual DPA attack would be successful. In total, eight evaluations profiles were set up. Each included a combination of the countermeasures described earlier and 1,000,000 encryption runs with random plaintexts and a fixed key. The profiles are grouped in three categories based on the point of attack: (1) S-box output bits of one round, (2) XOR-result bits of the inputs of two rounds, (3) 4-bit value of two S-box outputs. Profiles are listed below:

- Profile 0: Reference measurement
- Profile 1: S-box random decomposition
  - Leakage detected for S-box outputs
- Profile 2: Boolean masking
  - Leakage detected for consecutive round value
- Profile 3: Register precharge
  - XOR between rounds leakage is reduced, but not the S-box output
- Profile 4: S-box decomposition and Register precharge

  - Significantly reduced, but some leakage is detectable
- Profile 5: Boolean masking and Register precharge
  - Significantly reduced, but some leakage is detectable
- Profile 6: S-box decomposition and Boolean masking
  - Leakages not compensated sufficiently
- Profile 7: S-box decomposition, Boolean masking, and Register prechage
  - Proposed countermeasures all together withstand 10 million traces collected

In conclusion, the work presented by Sasdrich et al. on using reconfigurable look-up tables as a method to employ a type of DPR is successful when all proposed countermeasures were combined. However, it should be noted that this work depends on a special type of a resource primitive, which may not be available on all platforms. Additionally, resource usage overhead is significant. Taking into account additional logic required to mimic CFGLUTs on platforms that do not have them, this method is useful for only certain applications. Moreover, Roy et al. demonstrated a big security risk of using CFGLUTs [96]. They showed that it is possible to inject a so-called Trojan-attack in applications that use reconfigurable look-up tables and thus undermine immunity introduced by using these primitives in the first place.

3.2.3 Bete et al. in HOST 2018

Side-channel Power Resistance for Encryption Algorithms using Dynamic Partial Reconfiguration (SPREAD)

The countermeasure presented by the authors of the SPREAD system is the only one based on synthesis level modifications, other than the proposal of this work. Differences between the two approaches will be highlighted in the next chapter. Additionally, modifications introduced by the SPREAD system require HDL level modifications to the source code to implement the intended control logic. At the moment, the SPREAD is the most recent work published, although it was initially conceived in 2017 [97], details of their approach were not available until publication in [87].

SPREAD is aimed to reduce the data to power correlation, similar to most countermeasures proposed. However, as the authors note, their approach does not strictly fall under “noise enhancing” or “signal reducing” categories (hiding and masking, respectively, within the context of the current work). The authors describe their approach as the “moving target architecture”. While a standard AES-128 implementation requires 16 Sbox modules working in parallel, this work introduces an extra module that can be swapped by the DPR controller in run-time. This work has been extended and in a recent publication from April 2020 [87], the authors introduce two extra Sbox modules. Alternative implementations of the modules are derived in two methods labelled

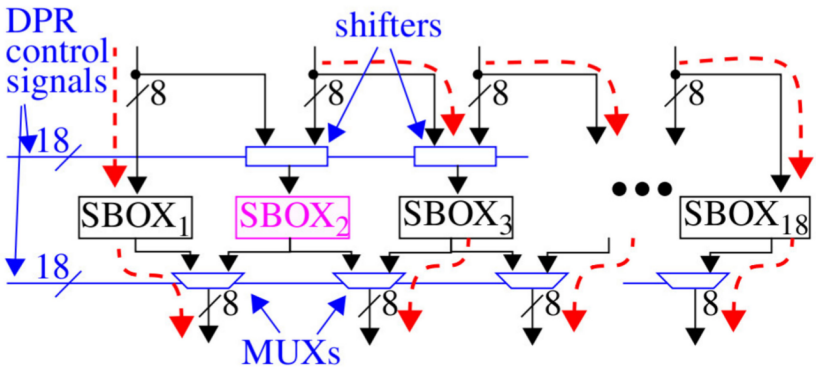

Figure 3.9: 16 Sboxes plus two redundant modules for reconfiguration [87]

as the synthesis- and circuit-directed, which will be described below.

Overall, since each Sbox module is a reconfigurable region, there are 18 reconfigurable regions comprised of original 16 parallel Sboxes and two new modules. This allows for spatial diversity as the Sbox modules can be moved by the DPR controller and placed in any of the regions. A system of shifters and multiplexers is reconfigured to reflect the new path for data, as shown in the Figure 3.9. Additionally, the SPREAD system supports multiple alternative implementations, however, for the demonstration authors generate 16 variants of the Sbox modules. Synthesis and circuit directed methods are used to generate one half each, respectively. The two extra regions allow creating a “hole” where no active Sbox is present or to use it as the reconfiguration target for an alternative implementation in the case of more than 18 variations. Having fewer variations than the regions available allows avoiding using external memory. Partial bitstreams can be embedded within the system and the main bitstream to be stored within the on-chip memory. The controller can read out the configurations during the start-up and store them in BRAM for subsequent reconfigurations. It should be noted that other functional blocks of AES, such as the AddRoundKey, ShiftRows, and MixColumns, were not manipulated, although, they can also be implemented within the reconfigurable regions. The delay and skew variations caused by the Sbox manipulation are expected to propagate through the whole system even when these modules remain static. Additionally, all Sbox regions support relocatable partial bitstreams and this reconfiguration approach requires excluding the static routing paths from the dynamic regions. This is not supported by the default Xilinx Vivado tool flow which will attempt to place static paths (e.g. controller, shifters, and MUXs connections) within the relocatable dynamic regions. Thus, the authors develop a custom tool flow to support their approach and will present it in the future.

The synthesis-directed approach is based on using the Cadence RTL compiler and targeting different standard cell libraries to generate a netlist, which is then used by Xilinx Vivado for the P&R process. This approach is in contrast to the traditional device-oriented synthesis process, where the synthesis tool (e.g. Xilinx Vivado itself), will map the logic of the system to the cells

available within the target architecture. SPREAD synthesis-directed method relies on modified cell libraries which vary in terms of available logic gates and would have an impact on the output netlist. A VHDL behavioural description of the Sbox module was passed through the Cadence RTL compiler using eight different cell libraries. The authors reported the total number of gates used by the first four versions, citing that the remaining four had similar results, shown in Table 3.4. Cell library variations include limiting available cells, such as removing different widths of AND, OR, AND-OR gates. Detailed overview of the libraries can be found in Table 1 in [87].

Table 3.4: Gate usage of alternative Sbox netlists generated by the Cadence RTL compiler [87]

Variant	Total gates
Design 1	300
Design 2	312
Design 3	312
Design 4	317

The total number of gates used is similar for the alternative designs, however, the resulting netlists are structurally different. After the P&R process, the authors observed the number of LUT2 to LUT6 primitives used to be between 128 and 206 and 13 to 57 of MUX7 and MUX8 primitives across all eight design variations. This countermeasure can be summarized below:

- Motivation: introduce structural diversity to the Sbox implementation
- Method: synthesize the VHDL description of the Sbox to different standard cell libraries using the Cadence RTL compiler
- Implementation: cell libraries vary in logic gates available for mapping
- Benefits: an automated and fast way of introducing various implementations of the design, delay paths ranging from 100ps to 5ns
- Downsides: limited variety and non-standard cell libraries

The circuit-driven method is meant to introduce delays to the buffers or other internal signals. The authors exploit increasing fan-out by using dummy loads and manipulating the clock network. The 128-bit state register used to store the intermediate results between the rounds is driven by a network of an AND gate, a buffer, and a 2-to-1 MUX to the clock input. The overview of the delay logic introduced can be seen in the Figure 3.10. Multiplexer select signal is derived from a different set of bits of the plaintext passed through the XNOR gate, thus each one of the flip-flops has a different clock delay. This introduces distortion of input latching among the bits of the whole state register. Two further modified versions of this clock delay circuit were introduced, giving a total of three circuit-driven design variations. This countermeasure is summarized below:

- Motivation: introduce temporal jitter in the system
- Method: randomize clock delay of the state register

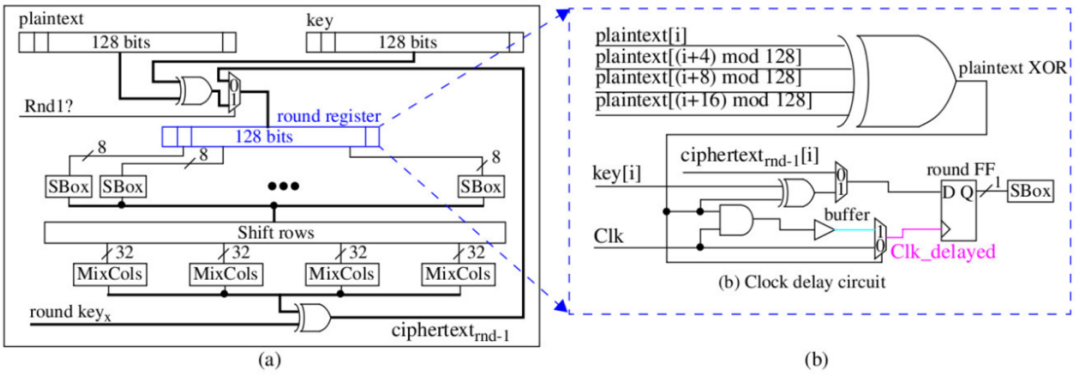

Figure 3.10: Delay circuit driving the clock input of the AES state register [87]

- Implementation: extra combinational logic to derive dynamic, data-dependent delay to the clock signal
- Benefits: more switching nodes introducing distortion to the power consumption and creating time offsets to the switching events
- Downsides: possible faults introduced due to the desynchronized individual bits of the same register

The evaluation method used for SPREAD was to measure the power consumption of different variations separately, rather than a full DPR implementation. Each of the reconfigurable regions is still defined but configured once and statically. A total of twelve variations were generated from the methods explained above. Three reference versions (as per authors notation) included only the synthesis-directed approach with 16 unique Sbox implementations each. Further, using these three variations, nine more were generated by incorporating the aforementioned three circuit-directed modifications. 30,000 power traces for each of the twelve variants were recorded and then randomly mixed to create an effect similar to DPR switching between these implementations. Power traces from different implementations are mixed, thus affecting the average of the statistical analysis. The authors also analyzed the variants individually using the CPA method described earlier and observed that the secret key can be extracted easily. The circuit-driven variants could expose the true value of Byte 3 of the secret key after analysis of fewer than 1000 traces.

An equal number of traces for each variant were combined, to give a total of 8192 traces (682 traces for each variant) as the smallest set and 360,000 traces as the biggest set (30,000 traces for all 12 variants). Increments were in powers of 2, for a total of 7 sets of traces. In the Figure 3.11, the authors demonstrate the effectiveness of the attacks by plotting Pearson's correlation coefficient (PCC) difference against the attacked secret key byte. PCC difference is defined as $PCC_{correctkey} - PCC_{largestincorrectkey}$, and its positive value corresponds to a successful guess. Correct key guess rank is an additional metric defined to order $PCC_{correctkey}$ by their magnitudes for all sets of traces analyzed. As expected, a higher number of traces results in higher

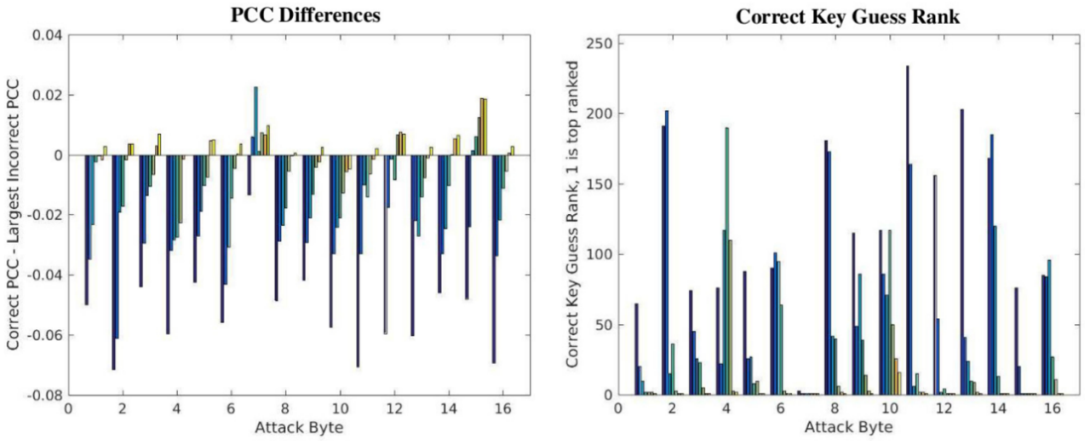

Figure 3.11: CPA efficiency for 7 sets of traces collected for 12 variants [87]

correlation, with the exclusion of Byte 7. The authors were not able to find an explanation as to why this particular byte has a decreasing first, then increasing correlation profile. Bytes 4 and 10 never reported positive PCC difference, even for the maximum amount of 360,000 traces. Most remaining bytes did not expose their true value until 262,144 power traces were analyzed, which is the second biggest set.

By a worst-case estimation from the authors, a 1ms period would be necessary for a complete reconfiguration of the system and a maximum of about 5000 traces for each variant can be collected before the countermeasures start to fail against the analysis. Attackers can acquire up to 313 traces within this reconfiguration period and even for a set of 16 variations of the Sbox module, the total number of traces that could be acquired stays within the limit of 5000 (313*16=5008). These results are based on extrapolation of the available data per each variant and a future report is expected where the system will include a true DPR implementation. This will allow for a more detailed analysis and accurate evaluation of the countermeasures proposed.

3.2.4 Hettwer et al. in DATE 2019

Securing Cryptographic Circuits by Exploiting Implementation Diversity and Partial Reconfiguration on FPGAs

In this work, the authors present a countermeasure that focuses on the implementation of AES on an FPGA. This approach is generic and thus applicable to other applications and algorithms. As in previously presented papers, Hettwer et al. explore methods of generating implementations of the AES algorithm on an FPGA that are more secure against side-channel attacks. Based on a single input design, this work focuses on generating multiple variations of the same synthesized netlist on the implementation level. The approach is based on applying random resource constraining during the placement and routing process. Various generated implementations of the AES algorithm are then used for dynamic partial reconfiguration. The authors achieve the

following benefits by using this approach:

- Varying dynamic power consumption – countermeasure to power analysis attacks
- Location of logical operations is not static – countermeasure to fault analysis attacks
- Generic methodology of circuit manipulation allows applying it to a variety of designs

This method has been verified on a Xilinx ZYNQ Ultrascale+ board using a serial implementation of AES-128. Dedicated Partial Reconfiguration Controller (PRC) IP core is used to do the switching of partial bitstreams. The authors present their work for self-adaptive and self-healing hardware concepts, which are based on using partial reconfiguration [98]. As such, the overhead of introducing the concept of reconfiguration into a static design is reduced when compared to previously discussed approaches. Constraining the placement and routing process allows controlling the physical and timing aspects of a design. Physical constraints allow restricting locations of logical blocks to a certain cell on the FPGA, thus affecting the layout and routing within a design. Timing constraints have similar effects but achieve it through a more fine-grained approach by specifying path delays for individual signals and thus affecting their routing options. The size of a partial bitstream and the time required to execute DPR are among the main factors in deciding what parts of the design to reconfigure. Including only sensitive blocks such as the S-box leads to smaller bitstreams and faster reconfiguration time but leaves most of the algorithm in the static region. Alternatively, the entire design can be located in the dynamic region but at the price of higher latency and memory space. This countermeasure can be summarized as follows:

- Motivation: a generic approach to countermeasure power, FA, and EM attacks
- Method: random placement and routing constraining of a single synthesized netlist input
- Implementation:
  - Dynamically switching the complete AES with static control logic using Tool Command Language (Tcl) script automation
  - Define 25% of the FPGA fabric as reconfigurable and randomly constrain 80% of the slices available for placement and routing
  - Generate 128 variations with their partial bitstream files
- Benefits: high efficiency against EM and FA attacks, generic approach applicable to different designs, and a possibility to combine with other countermeasures
- Downsides: higher resource utilization, need to store a large number of partial bitstreams in external flash memory (approx. 790MB)

The authors report that unprotected AES implementation was compromised after collecting 8250 traces, in contrast to more than 24500 traces necessary for the protected version with 128 variations. For reference, see Figure 3.12. Linear decrease of correlation when the number of variations was increased from 1 to 128 was not observed. However, the maximum correlation decreased by a factor of 2.2 between the two extremes. The method of attack was CPA with

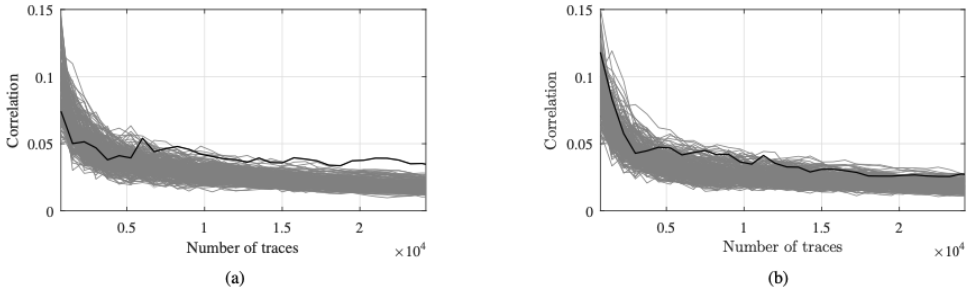

Figure 3.12: Correlation for unprotected (a) and countermeasure (b) implementations [88]

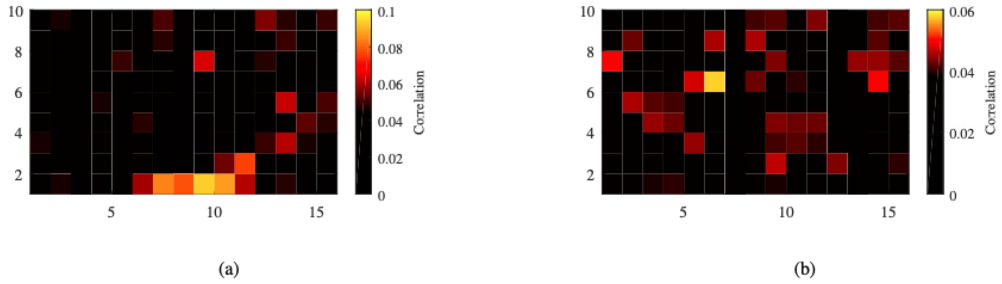

Figure 3.13: Correlation heat maps for unprotected implementation (a) and first configuration of the countermeasure (b) [88]

the Hamming distance model that was described earlier in this work. Two consecutive outputs of the S-box from the first round were the targets in this attack. The authors measured the current flowing through decoupling capacitor of the power supply to the system, based on the proposal in the work of O'Flynn and Chen [99]. This approach is an alternative to measuring voltage drop over a shunt resistor, as the high-frequency components present in the supply line flow through the decoupling capacitor, rather than the shunt, when both are present. These high frequency components prove to be useful for SCA and increase the success rate of a CPA attack. Stronger effects of the proposed countermeasure were found for electromagnetic and fault injection attacks. See Figure 3.13 for correlation heat maps. Unprotected and proposed countermeasure implementations are presented, with the latter showing considerable changes in the spatial distribution of the sites with higher leakage. On average, the number of measurements increased by n times for n different variations of the design when analyzing EM attacks. In the case of FA, the authors observed less than 1% probability of successful fault injection with the maximum number of variations enabled.

Chapter 4

Proposed Variants

This chapter introduces the countermeasures proposed within this work. Discussion includes motivation, methods, and technical details of the implemented designs.

4.1 Proposal Discussion

Given the information and concepts introduced in previous chapters the following conclusions could be made about countermeasures for side-channel attacks. Effectiveness depends on how much the design changes will affect the correlation between data processed and the power consumed by the system. The flexibility of the changes in terms of their variety and options, as well as keeping resource usage overhead to the minimum are also important. As it was discussed in the Section 2.3.3, this work chooses to employ a third-party synthesis tool Yosys [73]. This decision is based on the openness of the design space during the synthesis process when compared to a vendor tool, such as Xilinx Vivado. Although, the latter still includes many useful options to control different stages of the design flow. During synthesis, options such as retiming of the logic, thresholds for net fanouts or comparator sizes, LUT input size, etc. are available [100]. Vivado does also provide the granularity to control which entity or instance will be affected by user-defined constraints. However, these options are still limited. LUT sizes are restricted from 4 to 6 input bits and cannot assign cost values to different LUT sizes. Cost values allow the mapping tool to prefer or avoid certain LUT sizes based on the value assigned to them, rather than completely excluding them from the logic mapping process. ABC mapping tool invoked within Yosys employs several priority mapping methods based on area or delay costs [101], [102]. These values can be modified using the command invoking ABC within Yosys [103]. In the case that the source implementation is in some way protected and cannot be modified at the HDL level, Yosys allows for logic manipulation through its built-in commands. This opens possibilities to add or remove logic or create structures within the project that serve a specific purpose to the designer. Moreover, being an open-source and a platform-independent tool, Yosys can be utilized for the synthesis process by anyone and for any target architecture. However, as it was already mentioned in Sec-

tion 2.3.3, such flexibility comes at a cost of less optimized output netlist for a vendor-specific target FPGA device.

Countermeasure variants that are proposed within this work rely on the concepts that were already defined by previous publications and studies – masking the true value of data and influencing switching activity of the system. Detailed implementations will be described shortly but they can be generalized as the following three ideas:

(1) Manipulating data between rounds or submodules of AES

- *contributes to switching activity and noise in power consumption*

(2) Extra circuits running in parallel with AES

- *contributes to noise in power consumption*

(3) Manipulating the logic to primitives mapping

- *contributes to a different implementation and possibly power consumption profile*

Within the scope of the current work, the first idea is realized as the inversion of state values being stored in registers that are either part of the source implementation or have been introduced on top of the logic. This contributes to additional switching activity and allows to mask intermediate state values. Inversion could be replaced or accompanied by other types of Boolean transformations for a greater diversity of variants. The second one is presented in the form of register chains that include a series of flip-flops connected through inverters. This contributes to an increase in the noise and overall power consumption. This method can be extended to implement different types of Boolean operations between the registers. The last method is based on controlling the process of mapping the logic to primitives such as the look-up tables of an FPGA. Implementing the same logic on an alternative set of primitives would result in a different final configuration of the device. Thus, it would be expected to have influence on the power consumption profile of the system and the data to power correlation. These ideas can be extended and implemented in a variety of ways as part of a future study. However, the current work serves as a proof-of-concept experiment. It aims to show the possibility of using an open-source tool like Yosys and to determine whether countermeasures produced provide any benefit over existing approaches.

As it will be introduced in the Chapter 5, proposed ideas are implemented as standalone variants of the AES encryption algorithm. Two versions of the algorithm have been modified - round-based and serial-based. The outlined countermeasure concepts are applicable to both versions with slight modifications due to their differences in the implementation. A partial reconfiguration system is not included in this work and is intended for future projects. Each generated variant is then individually measured and analyzed to determine the effect of introduced changes. Additionally, recorded power consumption traces are shuffled to emulate the effect of a DPR-capable

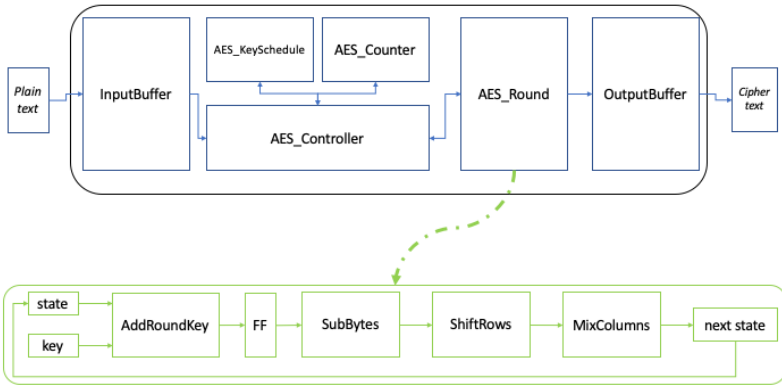

Figure 4.1: Round-based AES: full system (top) and expanded AES_Round module (bottom)

system and set expectations of a fully reconfigurable system with countermeasures against side-channel attacks. This approach has also been used for the SPREAD system [86], [87].

This work is compared and related to the recent work of Hettwer et al. Both studies are using the same source implementation of the AES algorithm and they share the measurement setup. While in [88] the authors are using only one, serial-based implementation, this work develops variants for both serial and an alternative round-based version of the algorithm. Due to certain timing constraints, unfortunately, only the latter one has been measured and results presented within this report. Due to this, it was not possible to include a baseline unmodified (unprotected) AES in the measurements. Variants of the serial implementation are expected to be tested in the future and results will be reported. These variants form a more complete set of implementations ranging from the unmodified baseline to multiple variations. They demonstrate randomization and diversity capabilities of the approach presented in this work. In addition to [88], during evaluation comparisons are also made to the SPREAD system [86], [87]. This is the only other publication that explicitly aims at the synthesis-level manipulations. Several similarities can be observed, such as using a third-party synthesis tool, manipulating the mapping process, and tempering with the switching activity of the encryption algorithm. The results, which will be discussed later, also exhibit similarity in the number of power traces to be collected until the secret key is exposed to the attackers.

4.2 Variants of round-based AES implementation

This section introduces three variants generated for the round-based AES. Labeled as Variant Class 1, 2, and 3, these correspond to the three ideas presented earlier. Figure 4.1 demonstrates the block diagram of the round-based AES implementation referred by the following explanations.

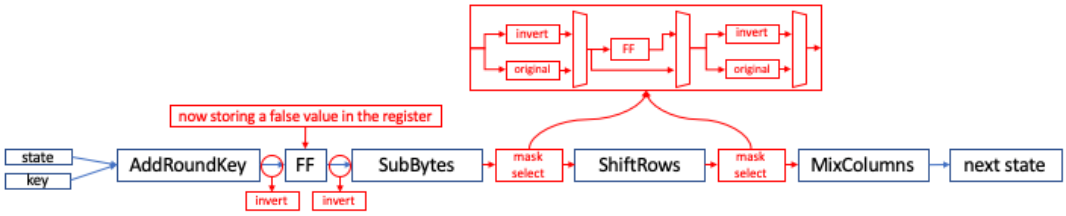

Figure 4.2: Variant Class 1 concept diagram

4.2.1 Variant Class 1 for round-based AES

The first countermeasure proposed by this work is referred to as Variant Class 1 (VC1). In this approach the aim is to introduce background switching activity that will impact the power consumption of the system while it is executing a round of encryption. This variant is implemented using two approaches that affect the system during both its active and inactive states. The system includes a register that stores the result of AddRoundKey operation (bottom half of the Figure 4.1) is the only point of the system that stores intermediate state values and thus it can be a point of interest for the attackers. An inverter is placed at the input of this register to avoid storing a true state value during the execution of a round. To avoid working with the now wrong value of the state, the output of this register is inverted again before it goes into the SubBytes module. By design, there cannot be a register inserted in between SB, SR, and MC modules (to avoid timing issues due to a cycle delay from a flip-flop), a block comprising of three multiplexers is constructed, called *mask select*. In this block the input signal can go through two paths - either retaining its original value or being inverted and then inverted back before the next block (i.e. SR after SB) processes it. This path is defined by the first and the third (last) multiplexers, which are controlled by a selector signal generated from a reduce-XOR operation of the state input. This operation takes 128 bits of the input and performs a bitwise XOR operation to produce a single bit output. The middle multiplexer of the *mask select* block introduces a register into the path of the inverted intermediate data. This MUX is operated based on an inverted `ENABLE` control signal supplied to the `AES_Round` module. In this manner, it is ensured that no extra delay is inserted when encryption is active. When the round is inactive, the alternative path with the register is selected and due to the clock signal still being active, this part of the system will exhibit some switching activity. This changing operation of the circuit will contribute to both the active and inactive states of the round function and affect the observed power consumption profile. The expectation would be that power spikes correlated to the active round execution will be minimized compared to the drops in between the rounds. The concept of this variant can be summarized in the Figure 4.2. The actual implementation that has been measured for the evaluation is shown in the Figure 4.3. The realization slightly differs from the concept and is expected to be finalized in the future. Namely, the *mask select* block will be added in between SR and MC blocks, as well as at the output of MC. At the moment, these blocks only have the register part of the *mask select* block inserted between them. Code excerpts for this variant can be found in the Appendix.

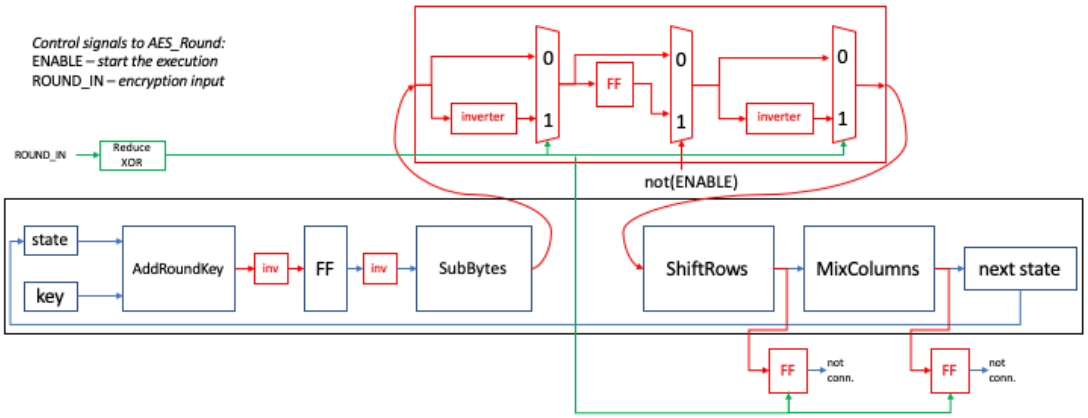

Figure 4.3: Variant Class 1 implementation diagram

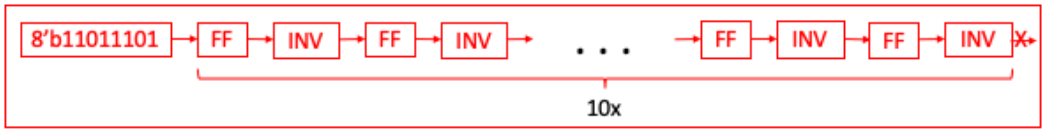

Figure 4.4: Variant Class 2 implementation diagram

4.2.2 Variant Class 2 for round-based AES

Variant Class 2 (VC2) is based on a straightforward idea to increase background noise in power consumption. This is expected to reduce the correlation of data to power consumption. For this purpose, a chain of registers and inverters is added to the system. This chain can be seen as a pair of a register driving an inverter. It is easy to waste hardware resources with this countermeasure and therefore, the chain in Variant Class 2 is limited to 10 pairs of registers and inverters. The first input is set to a static value and the output of the chain is left unconnected for the evaluated version. This idea has been visualized in the Figure 4.4. It is possible to extend this idea further by implementing a dynamically changing input value, as well as swapping the inversion with other types of Boolean operations, which will differently contribute to the switching activity and power consumption profile. The chain has been inserted into 10 modules of the round-based AES system: `AES_Core`, `AES_Controller`, `AES_Counter`, `AES_Round`, `AES_SubBytes`, `AES_ShiftRows`, `AES_MixColumns`, `AES_MixSingleColumn`, `AES_SBox`, and `AES_KeySchedule`. All of these modules have been introduced, except for `AES_MixSingleColumn` and `AES_SBox`, which are simply the building blocks of the `AES_MixColumns` and `AES_SubBytes` modules, respectively. Since there are four columns within a 4x4 matrix representing a 128-bit state value in AES, there are 4 parallel instances of `AES_MixSingleColumn` in `AES_MixColumns` and 16 instances of `AES_SBox` (each referring to a single byte) in `AES_SubBytes`. Code excerpts for this variant can be also be found in the Appendix.

4.2.3 Variant Class 3 for round-based AES

The remaining variant (VC3) for the round-based AES is directly manipulating the mapping process during synthesis. The command responsible for invoking the ABC tool within Yosys was modified to implement this variant. Originally, this command is as follows:

original: `abc -luts 2:2,3,6:5,10,20`
modified: `abc -luts2:2,3,5 AES_Round AES_SBox AES_Core`

Here every number refers to the cost associated with a LUT of an input width that corresponds to the position of the number. Respectively, ”2:2” refers to one and two input LUTs costing ”2” units, ”3” refers to three input LUTs costing ”3” units, and ”6:5” refers to LUT4, LUT5, and LUT6 each costing ”5” units. Lastly, ”10” and ”20” correspond to seven and eight input LUTs with considerably higher cost values. This aligns with the fact that 7-series Xilinx FPGAs have at most 6-input LUTs available and functions that require a larger number of inputs are constructed using multiple LUT6 and multiplexers [104]. Therefore, it is expected that they will cost more than already available LUT sizes. In its original form the command is applied to the entire design with all possible LUTs and therefore it has been modified to limit mapping to a maximum of 4-input LUTs, as shown above. This setting is applied to only three of the modules `AES_Round`, `AES_SBox`, `AES_Core`. The SBox module was chosen as it is the most attractive point to attackers due to its high switching activity nature.

This subpar (less than optimal) mapping method is expected to influence the power consumption profile. However, it is also expected to have the highest negative impact on resource utilization, since the best available resources of the FPGA are excluded from the mapping process. This variant has some underlying similarity to the synthesis-driven approach used by Bete et al. in [87]. However, in this work the design is mapped to LUTs directly, while the SPREAD system relies on a cell library consisting of logic gates that is later synthesized and placed by Vivado. This method of mapping is also available through ABC within Yosys. During the development of the proposed variants this approach was tested as an experiment but was not investigated further due to higher resource utilization during the Vivado P&R process. In contrast, the authors of the SPREAD system did not report higher utilization for their synthesis-driven approach. In their work, the netlist generated within Cadence RTL is passed through both synthesis and P&R processes within Vivado. It is possible that the Vivado syntesis algorithm performed an optimization of the logic and did not result in significant utilization overhead.

4.3 Variants of serial AES implementation

At the moment, this version of AES has not been measured or evaluated. However, it was originally developed as the main set of variants intended for measurements. It is also used as an ex-

ample of a broader implementation diversity that can be achieved through Yosys. The underlying ideas for the variants remain the same as already presented and therefore serial-based variants will be discussed briefly. Round-based AES introduced previously is, in fact, operating in a serial fashion: every round is executed one by one and no other encryption process is running in parallel. Therefore, the version of AES referred to as serial in this section is similar in its structure to the previous implementation. There is a top-level called `AES128` which includes the `ControlLogic` (state controller and counter), `RoundFunction` (implementing AES functions), and `KeySchedule` (key expansion) modules. `RoundFunction` implements the SubBytes, ShiftRows, MixColumsn, and AddRoundKey functions as expected. An extra module called `Cell` is introduced in this version of AES, which is an 8-bit register that includes a multiplexer. This register is instantiated within the `RoundFunction` module and corresponds to 16 bytes of the 128-bit state. All the subfunctions of AES operate on these 16 instances of the `Cell` module. Variants of this AES were generated from a C program that produced randomized Yosys scripts. Several parameters were considered for randomization (excerpts of the code can be found in the Appendix). Assuming that RoundFunction (RF) module is the main building block of the system, this module must undergo modifications for countermeasures. Then, KeySchedule (KS) and ControlLogic (CL) modules are chosen randomly, either both or one of them, and modifications are applied. The modifications based on the three ideas presented earlier are parametrized and listed below. Five sets of variants were generated with each corresponding to one or a combination of these ideas. For each set, multiple variations are possible, since each variant can be generated with randomly chosen parameters. The list of variants intended for measurements is shown in the Appendix Table D.1.

- Idea 1: Value stored in a `Cell` module is inverted (one, some, or all bits).
- Idea 2: The length of the register-inverter chain is set randomly (50 to 100 pairs).
- Idea 3: LUT mapping is applied to different modules (maximum size varied from 3 to 6).

4.4 Yosys Overview

Given the technical details of the variants, a brief overview of the tool that allowed to generate them is presented below.

Yosys has been developed as an alternative, free, and open-source tool for situations where a custom synthesis flow or algorithm cannot be used with the existing proprietary tools [105]. It supports Verilog HDL based designs in its open-source version. However VHDL support can be added through external translators available within the commercial edition of Yosys [75]. It can output the design in several netlist formats, such as BLIF or EDIF. Support for synthesizing and mapping the input design to a specific cell library or some FPGA devices, as well as mapping and final logic optimizations are available through the ABC tool included in all versions of Yosys [74]. As can be seen in the Figure 4.5 among the levels of the FPGA design flow, Yosys targets levels

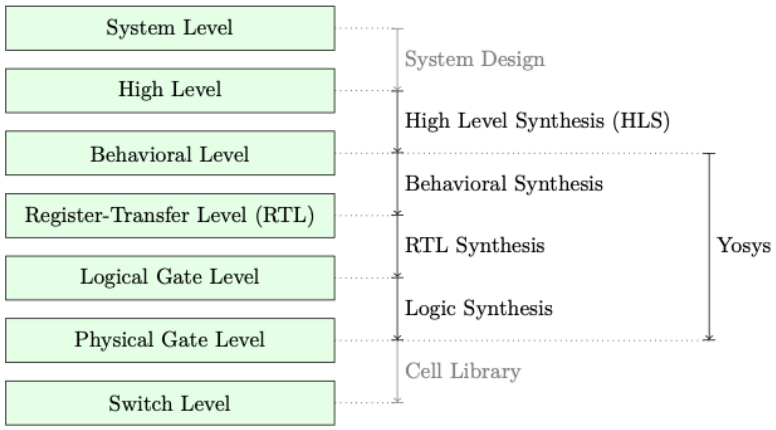

Figure 4.5: Yosys and design flow levels [105]

from HDL down to the physical gate level. Meaning that the output from this tool is a netlist that refers to hardware resources available on the target architecture, such as FPGA primitives. This netlist is then available for P&R process by tools such as nextpnr [106] or Xilinx Vivado.

Yosys processes the input HDL design into an internal format that allows further modifications to the design without referring to the original HDL. The internal representation is using the Register-Transfer-Level-Intermediate-Language (RTLIL). By using a unified internal language, Yosys processes an input design in ”passes”, which refer to various transformations and manipulations that could be done. These include logic optimizations, insertion and removal of cells or gates, and mapping of the design to hardware resources. The advantage of Yosys is that it operates on the design on both coarse (RTL) and fine (Logic Gate) grain levels. Objects such as coarse multi-bit wide cells are used to describe the input design initially. Later, they are converted to fine-grain objects such as bit-wide gates. These level changes allow for a greater variety of logic optimizations and design manipulations. After this, the ABC tool is invoked to transform the Gate Level representation of the design to the Physical Gate Level that refers to target specific resources.

Internally, Yosys supports a variety of cell types to describe an HDL source and it is possible to add custom objects as well. Registers, wires, RAM blocks, multiplexers, arithmetic units, logic gates, etc. are among many available cell types. There is also a variety of commands available, such as adding or removing logic, visualizing the design, solving Boolean Satisfiability (SAT) problems, equivalence checking, etc. For a full list, refer to the Yosys Manual [105]. Yosys also supports scripting, meaning that it is possible to create and automate custom synthesis flows. Some synthesis flows are built-in, such as targeting certain Xilinx, Intel, Anlogic, and Gowin FPGAs. Although limited support is available for proprietary platforms, as their cell structures are not publicly available and thus logic mapping cannot be performed to the best of the tool’s ability.

However, it is possible to add custom cell libraries and allow for well-optimized output netlists. Among the available commands, the following two are of interest to the current discussion:

- `add` - adds objects to the design
- `synth_xilinx` - synthesis script targeting Xilinx FPGAs

While the first command is self-explanatory, the second one is a script that performs synthesis for Xilinx FPGAs. It performs mapping of sequential processes and finite state machines from the input HDL to gates, as well as logic optimizations. If the original design described a structure that refers to dedicated resources such as Digital Signal Processing (DSP) units or First-In-First-Out (FIFO) memory structures, these can be detected by the internal solver. They are then mapped to the corresponding primitives available within the Xilinx FPGA library built into Yosys. It is possible to prevent such optimization and use regular LUT primitives for resource-intensive logic. This would be equivalent to synthesis for a generic target architecture, where no special primitives are available. Xilinx cell library used by Yosys has been collected from the design guides available from the vendor itself, as the full libraries are not public. Therefore, only 7-Series Xilinx devices are supported at the moment [107]. In the end, "`synth_xilinx`" invokes ABC to perform mapping of the remaining design to LUTs to prepare the design for placement and routing on an FPGA.

The open-source version of Yosys used within this work does not support all of the cell types that are available within the tool. Only wires could be added to the design out of the box. Modifications for the "add" command were necessary since generating any of the variants implied adding registers, logic gates, and re-routing some signals within the design. These cell types could be added with relative ease as the source code of Yosys is open and available in the C++ programming language. After analyzing the internal code structure of the tool, new cell types were added based on the existing example of the wires type. Added cells include unary (single input) and binary (two input) operators, multiplexers, and registers. These are coarse-grain type cells as they operate on multi-bit wide inputs. When the design is represented at the fine-grain level it is also possible to make manipulations using corresponding single-bit cells (such as a single D-type flip-flop). However, the scale of the design may be too large for manually introduced modifications. Pinpoint changes could be done to a small number of specifically targeted cells but with extreme care. The designers risk damaging the design unintentionally by removing a cell or connecting a wire to the wrong port. Therefore, global changes such as generating variants are better suited for the coarse-level representation. The second command is easier to modify since it is essentially a script that invokes several commands available within Yosys. Here, the modification is done to the "map_luts" process which is responsible for invoking the ABC tool with specific LUT cost values and size limits. By varying the parameters of the "abc" command called within the "map_luts" process, it is possible to specify the cost of 1 to 8 input LUTs in terms of area and delay. These values are processed by the ABC tool internally. It should be noted, however, that when the output netlist goes through the P&R flow, changes might be made to the LUT structures if no matching

primitive is available on the target FPGA device. Therefore, changes introduced here should be done with the platform in mind. In addition to the mentioned commands and changes introduced, producing the synthesized output netlist for the subsequent P&R process is also slightly modified. When generating the variants, special keywords (attributes) are used to prevent the tools from optimizing or combining the introduced logic [100]. If the modifications of the design do not include top-level input-output signals or are found to perform logic that does not affect the rest of the system, synthesis and implementation algorithm may eliminate these parts. For example, the proposed register-inverter chain would be an ideal target for removal during synthesis due to not having any impact of the logic of the system. Therefore, keywords such as `"DONT_TOUCH"` and `"KEEP"` are used to denote the blocks of logic that the tools should not attempt to optimize or affect in any other way. By default, these attributes do not carry over into the output netlist and thus, the command is modified accordingly (refer to code excerpts in the Appendices).

Chapter 5

Proposal Evaluation

This chapter compares round-based AES variants to the original, unmodified AES implementation in terms of functionality and resource utilization. Then, the results of the power consumption measurements are discussed and evaluated.

5.1 Functional Verification

One of the main conditions for reconfigurable variants of any design is that they must maintain the functional integrity of the system. The proposed changes cannot be used if they result in a wrong output or unintended behaviour of the system. First, Vivado can be used for both synthesis and P&R of the unmodified AES. This version could be referred to as the reference implementation. Then, unmodified AES synthesized by Yosys and implemented by Vivado is verified for correct functionality by comparing it to the reference. Once this is established, Yosys can be used further to modify AES and generate variants as explained in the previous chapter. AES implementation used in this work includes a testbench that compares output ciphertext with an expected ciphertext for a given plaintext input. By using this testbench, as well as third-party online resources such as [108] that can execute AES encryption on a given data, the results can be verified to be true. For testing purposes, the secret key, the input plaintext, and the expected ciphertext are given below in hexadecimal representation. Figure 5.1 shows the results from the online tool used to verify the correctness of AES implementation used in this work [108].

- Key: `2b7e151628aed2a6abf7158809cf4f3c`
- Plaintext: `3243f6a8885a308d313198a2e0370734`
- Ciphertext: `3925841d02dc09fbdc118597196a0b32`

Figure 5.2 shows the simulation output from the Vivado reference design. Figure 5.3 shows the simulation of unmodified AES synthesized by Yosys. The netlist in the EDIF format was imported to Vivado, where a post-synthesis functional simulation was performed using the same testbench as the reference. As can be seen, Yosys generated netlist behaves as expected. Round-based

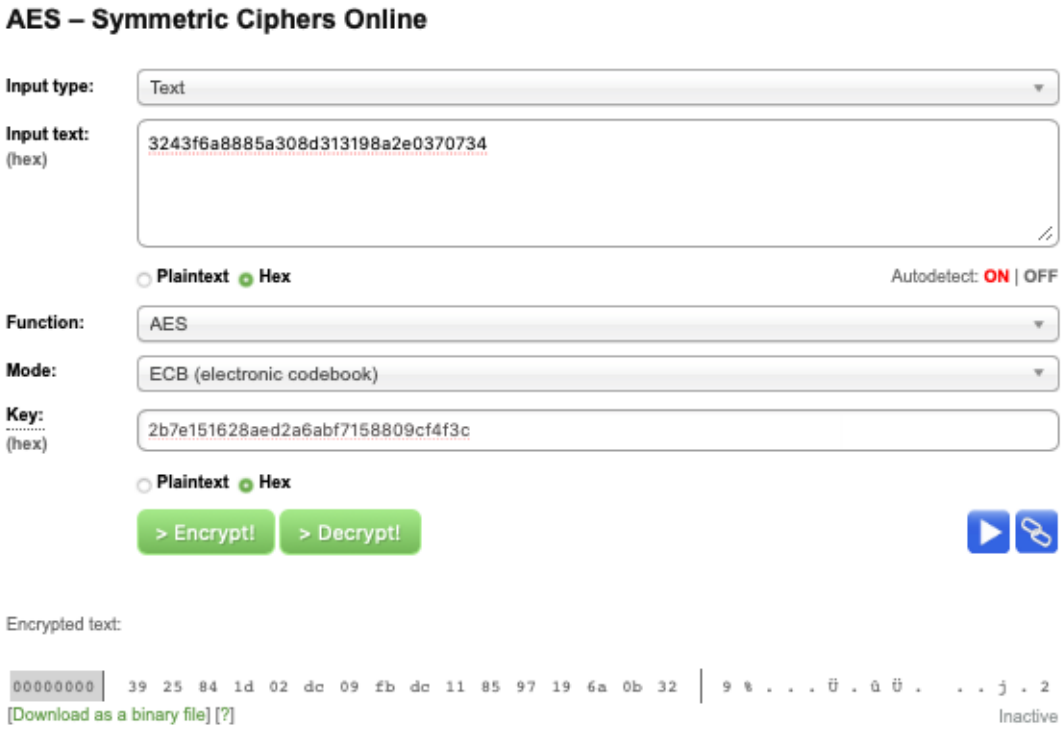

Figure 5.1: Verification of the output for given inputs [108]

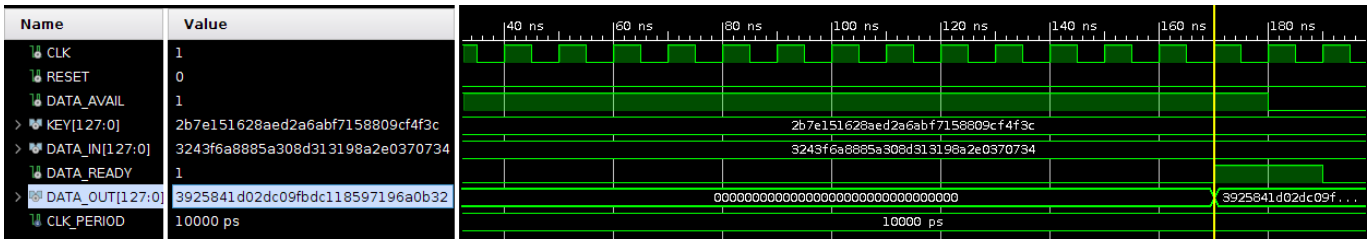

Figure 5.2: Reference design simulated in Vivado

AES variants generated from Yosys have also been verified for functional integrity. Corresponding simulation results can be found in the Appendix A. Three waveforms are presented for ”mod7”, ”mod8”, and ”mod9” corresponding to VC1, VC2, and VC3, respectively.

5.2 Resource Utilization

The Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [109] was used within this work for the test measurements and evaluation. It includes the Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC device - an FPGA with an integrated ARM processor, making it suitable for a variety of applications. This combination enables software access to the system and reserves the FPGA for the programmable logic functionality. This device will be referred to as ZCU102 for convenience. For this platform, resource utilization is reported as the number of look-up tables and flip-flops (reg-

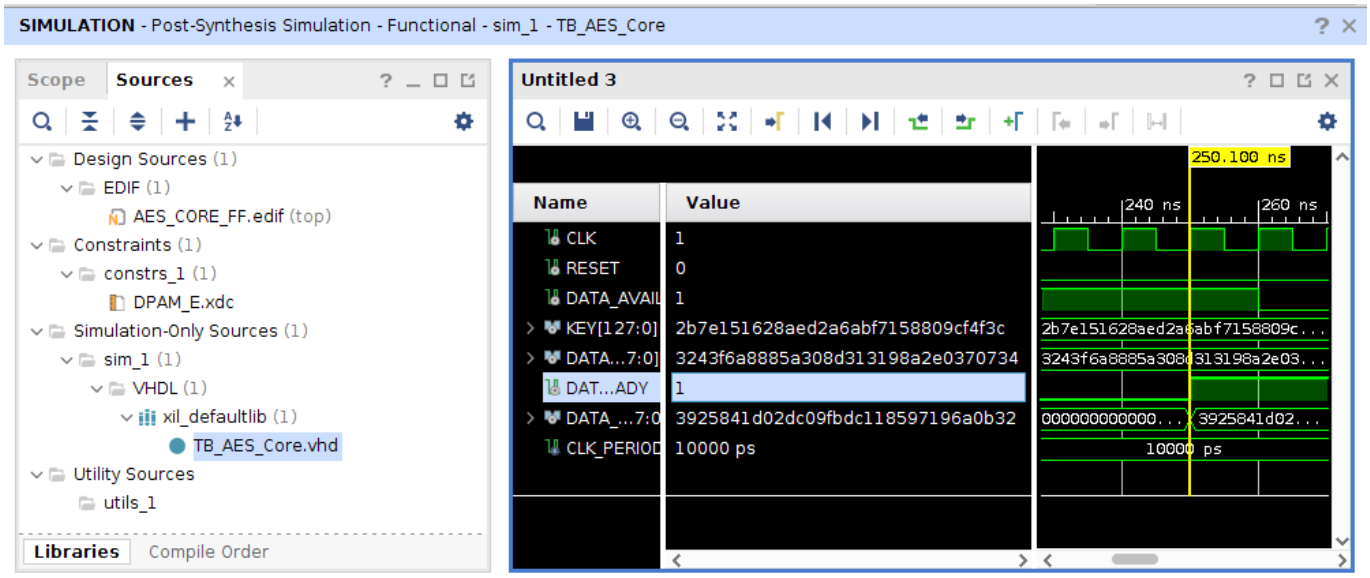

Figure 5.3: Unmodified AES from Yosys simulated in Vivado

isters) used. To successfully implement a design on an FPGA, some additional constraints, such as the clock frequency, have to be specified. These constraints are used by the P&R algorithm to physically place the logic within available resources such that the designer-specified path delays and timings are met [110]. If the design is constrained to run at higher clock frequencies it might require stricter allocation of resources (i.e. shorter interconnect paths). In certain cases, if the logic is too large, the P&R solver algorithm might not find an optimal solution for the given constraint and the process will exit with an error. In this work the system was successfully implemented and satisfied all design constraints with a 50MHz clock. The system, including the variants, were found to successfully pass timing analysis stage at up to 400MHz. Table 5.1 lists utilization for all configurations of the round-based AES.

As it can be seen, reference and Yosys synthesized unmodified AES are very close in the utilization numbers. This shows that a third-party synthesis tool can deliver netlists optimized close to what a vendor tool can produce. Resource utilization for the variants changes as expected. For example, VC1 causes an increase in the number of both flip-flops and look-up tables, as it introduces extra logic and registers into the system. Variant Class 2 exhibits a similar increase in the registers used, but with a smaller overhead on the LUT usage. Variant Class 3 is purely LUT driven modification and therefore its flip-flop usage does not change from the original version. However, the LUT usage spikes up by 349%, which is a considerable overhead. While it is still a relatively low usage of the resources available on the FPGA in total (2.64%), other variants show much lower overhead. For reference, target FPGA has 274080 LUTs and 548160 registers.

Table 5.1: Resource utilization for all configurations

Configuration	Utilization	Overhead
Reference	LUT: 1319 (0.48%)	-
(Vivado only)	FF: 533 (0.09%)	-
Unmodified	LUT: 1652 (0.6%)	25.24%
(Yosys and Vivado)	FF: 530 (0.09%)	-0.56%
Variant Class 1	LUT: 2281 (0.83%)	38%
	FF: 658 (0.12%)	24.15%
Variant Class 2	LUT: 1910 (0.69%)	15.6%
	FF: 658 (0.12%)	24.15%
Variant Class 3	LUT: 7254 (2.64%)	349%
	FF: 530 (0.09%)	-0.56%
Combination of	LUT: 7278 (2.65%)	351%
VC1, VC2, VC3	FF: 658 (0.12%)	24.15%

5.3 Measurements Setup

Once the functional integrity of the variants has been established, they can be integrated into the full system to be used for measurements. The round-based AES is a stand-alone hardware implementation of the algorithm but controlling it through software gives more flexibility. Therefore, it is packaged as an IP block which can be instantiated within a larger system. Xilinx provides an IP block that enables the use of the software and hardware combination within ZCU102 [111]. Figure 5.4 shows what the system looks like when all the necessary components have been instantiated. It contains the processor and its corresponding reset controller. Advanced eXtensible Interface (AXI) is used to communicate with the AES core. This is a flexible interface system that is part of the ARM-based microcontroller bus architecture [112]. It supports lite, streaming, and high-performance memory-mapped configurations to establish a link between the processor and a peripheral device. In this system, Zynq Ultrascale+ is the processor and AES Core is the peripheral device. The full system is then implemented and loaded onto the board. A software development kit (SDK) interface is available to control the system and can be launched through Vivado. A program is developed using the C programming language and loaded onto the processor through this interface. This program handles sending plaintexts to the AES core and reading the output ciphertext back over the AXI bus. Data being sent can be displayed in a built-in console of the SDK and used for verification.

In addition to the main system components, there is an external trigger signal that can be seen in the IP block diagram. This signal is used by the measurement setup to align and start collecting power consumption traces. It is generated by the `AES_Controller` module on purpose and facilitates easier trace collection. The result is that the traces are less likely to contain power con-

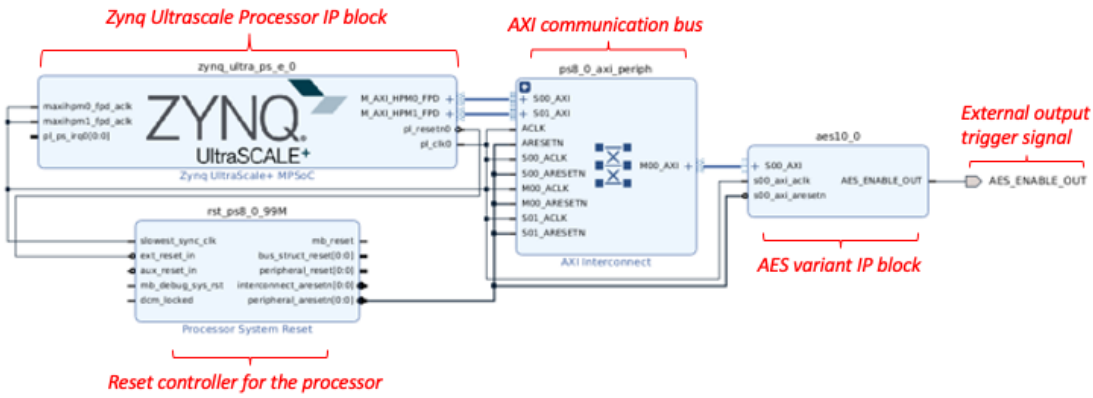

Figure 5.4: Block diagram of a complete system

sumption of the system when encryption was not running. This makes the analysis of the traces easier for experimenting. In reality, however, attackers are not likely to know the starting moment with such high precision and will need to spend more time measuring and analyzing the system. Before the packaged system was sent for measurements, this trigger signal was mapped to a general-purpose input/output (GPIO) pin of the Zynq board and observed using an oscilloscope. The functionality of the implemented system was also verified by applying a range of plaintexts through the SDK and cross-referencing the output using the online tool mentioned earlier. An example of the trigger signal can be seen in the Figure B.1 in the Appendix. This signal is set to ”high” while the AES encryption is running and is set to ”low” otherwise. The system was configured to run at 100MHz clock frequency (i.e. one clock cycle is 10ns) and this clock is also supplied to the AES core. As previously explained, one round takes one clock cycle to complete and it can be deduced that one full cycle of encryption should take about 100ns in total. This is confirmed by observing the trigger signal stay on ”high” for approximately 100ns on the oscilloscope screen.

To collect the traces, the current flowing through the decoupling capacitor of the FPGA power supply was measured, as it has been proposed in the Chapter 3. Langer ICR HV500-75 EM probe [113] was used to measure the current and a Picoscope 6404D USB oscilloscope [114] with 1.25GS/s sampling rate was used to capture the values and offload them to a PC. The measurement process consists of using the same plaintext input 250 times which are then averaged into 1 trace corresponding to 1 encryption. Each trace contains 2000 data points. A total of 10000 traces were collected in this manner for each variant of the round-based AES: VC1, VC2, and VC3. All encryption runs were performed using the same secret key. The set of 10000 traces for each variant is then analyzed individually. As mentioned previously, designing a fully reconfigurable system was not part of this work. Therefore, to emulate the effect of DPR, traces from different variants were randomly shuffled for the analysis, similar to the method used by the authors of the SPREAD system [87].

Additionally, the system was set to run with a 10MHz clock frequency during the measure-

ments. Configuration of the test setup used in this work provides some advantages from the attacker's point of view. Mainly, averaging each encryption cycle removes the noise in the recorded values and lower operating speed coupled with a high sampling rate form the oscilloscope allow for cleaner measurements and easier analysis.

5.4 Power Analysis Attack Model

To perform an attack on the collected traces, Hamming distance between the input of the Sbox during the last round and a byte of the ciphertext is calculated. Since the attack is performed using the ciphertext and not the plaintext as it was described in Section 2.2, the equation can be re-written as follows:

$$H_w[Sbox^{-1}(CT(i)(j) \oplus LRK(j)) \oplus CT(i)(k)] \tag{5.1}$$

Here, $CT(i)(j)$ corresponds to the j^{th} byte of the ciphertext for the i^{th} encryption. $LRK(j)$ refers to the j^{th} byte of the last round key, which would be the value the attackers will guess. $Sbox^{-1}$ is the inverse operation of the regular Sbox described earlier. $CT(i)(k)$ refers to the k^{th} byte of the ciphertext for the i^{th} encryption. It is necessary to refer to the same byte to find the Hamming distance between the output ciphertext and the input to the last round. Since AES shuffles the bytes of the state in each round, the j^{th} byte of the output state in each round is not the j^{th} byte of the input state to that round. Therefore, the round operations have to be traversed back to the start of the round. The last round of AES is different from the rest, as it skips the MixColumns operation. This means that its power consumption profile is expected to be different and it has a fewer number of operations to traverse back. As an example, see the Figure 5.5 for a visualization of the backtracking process for byte 11 of the last round output (essentially the ciphertext).

As shown in the figure, after traversing the round operations back to the start, the ciphertext *byte 11* is found to occupy the position of the ciphertext *byte 7*. Thus, the Hamming distance will be calculated between the bytes that are located in those positions. Hence why the reverse Sbox operation is performed on *byte 11* and its output is compared to *byte 7*. *Byte 11* of the last round key is the value to be guessed by the attacker and it corresponds to the *expanded* secret key, per the KeySchedule algorithm of AES. As further examples, guessing the secret key *byte 15* corresponds to ciphertext *byte 11*, while guessing the secret key *byte 4* to corresponds to ciphertext *byte 4*. This is because ShiftRows operation does not affect the first row and only modifies the bottom three rows.

By using this method, a hypothetical value of consumed power (the Hamming distance) is calculated for each guess (a total of 256 guesses) of each byte of the secret key. Then, correlation between each hypothetical power value and each point in a power consumption trace is computed

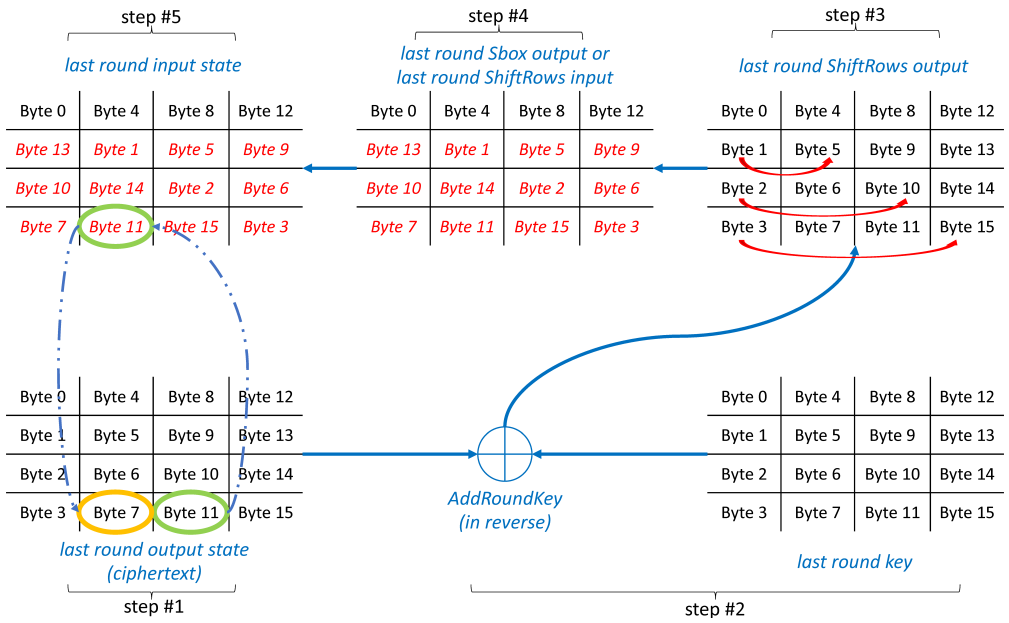

Figure 5.5: Reversal of last round operations in AES

by using the Pearson's Correlation Coefficient (PCC). The expectation is that for each guess, one PCC value will be larger than the rest. The guess with the highest PCC is most likely the correct value for that particular byte of the key. Performing these calculations for a larger set of power consumption traces recorded will increase the confidence in the guess. Repeating this method for all 16 bytes will reveal the entire last round key which can be converted to the initial secret key used by the algorithm. A simplified explanation for a case of one power trace recorded is presented below.

1. Given one trace of power consumption with 2000 data points
2. Perform the first guess of the 11^{th} byte of the last round key (i.e. 0)
3. Use the 11^{th} byte of the ciphertext and XOR it with the guess to get value X
4. Use the reverse Sbox to find the transformation of X into Y
5. Perform XOR of the 7^{th} byte of the ciphertext with Y to find the Hamming distance H_w
6. Calculate PCC value of H_w with each of the 2000 data points in the given trace
7. Record the highest correlation point and its absolute value, (i.e. C_0)
8. Repeat previous steps for all 255 remaining guesses of the 11^{th} byte of the secret key
9. Among all recorded peak correlation values (C_0 to C_{255}), pick the highest one (i.e. C_i)
10. Value guessed on the i^{th} iteration is the 11^{th} byte of the last round key
11. Perform key schedule reverse operation to get the 11^{th} byte of the initial secret key
12. Repeat this process for all bytes to get the full secret key

5.5 Correlation Analysis of Individual Variants

Having explained the procedures to collect power traces and perform the attack, the results of the analysis can be presented. System designers and attackers are the same parties and therefore the secret key is known in advance. For simplicity, the secret key was set to $[\mathrm{FF}]_{16}$ for all 16 bytes. This allows preemptive calculation of the secret key expansion and skips the two last steps of the attack procedure described earlier. For the given secret key, its expansion is presented in Table G.1. If the results of the analysis yield the values found in the ”last key” column, the attack will be considered successful and the secret key was found. For each of the AES variants submitted for measurements, three files have been obtained: input plaintexts (10000 entries with 16 bytes each), output ciphertexts (10000 entries with 16 bytes each), and trace values (10000 entries with 2000 points each). The latter two are of interest in this analysis. The plaintexts would be used to verify the secret key that was guessed by generating a ciphertext and comparing the value to those recorded during measurements. If the ciphertexts match, the secret key was correct. A Python program was developed to process the ciphertexts, generate a guess, calculate the hypothetical power value, and perform the correlation analysis. Generated values are written to a file for comparing and determining the best guess. Figure 5.6 shows the raw trace values for one of the measurements. This corresponds with the Figure 2.5 shown earlier and identifies the executed AES rounds. Figure 5.7 represents the peak correlation values of all guesses made for one of the bytes of the Variant Class 3. Plots for Classes 1 and 2 are added to the Appendix. Figure E.6 shows the output from the program determining the guess corresponding to the highest correlation among the maximums of each of the 10000 traces. Reported best guess for byte 11 is 85_{16} and the expanded round key byte 11 was calculated to be 85_{16} too. This coincides with the value shown in Table G.1. This experiment was repeated for other bytes and it was concluded that the full secret key can be revealed.

Table 5.2: Analysis summary of the variants and their combinations

Variant:	**VC1**	**VC2**	**VC3**	**VC1&VC2**	**All shuffled**
Peak Correlation (PC):	0.15081	0.16122	0.24252	0.05815	0.07364
PC factor (over VC3):	1.61x	1.5x	-	4.17x	3.29x
Traces to get the key:	1250	1350	750	9500	7500
Traces factor (over VC3):	1.66x	1.8x	-	12.67x	10x

Peak correlation values for a given byte for all variants are listed in the Table 5.2. Refer to Figure 5.8 and Table F.1 for an overview on all bytes. It can be observed that the VC1 and VC2 show lower correlation values when compared to Class 3. The subpar mapping of the logic to the LUTs does not have the desired effect of reducing or changing the switching activity profile. As a result, the correlation between data and power was found to be significantly high. Thus, VC3 was chosen as the comparison reference in the absence of the unprotected system measurements.

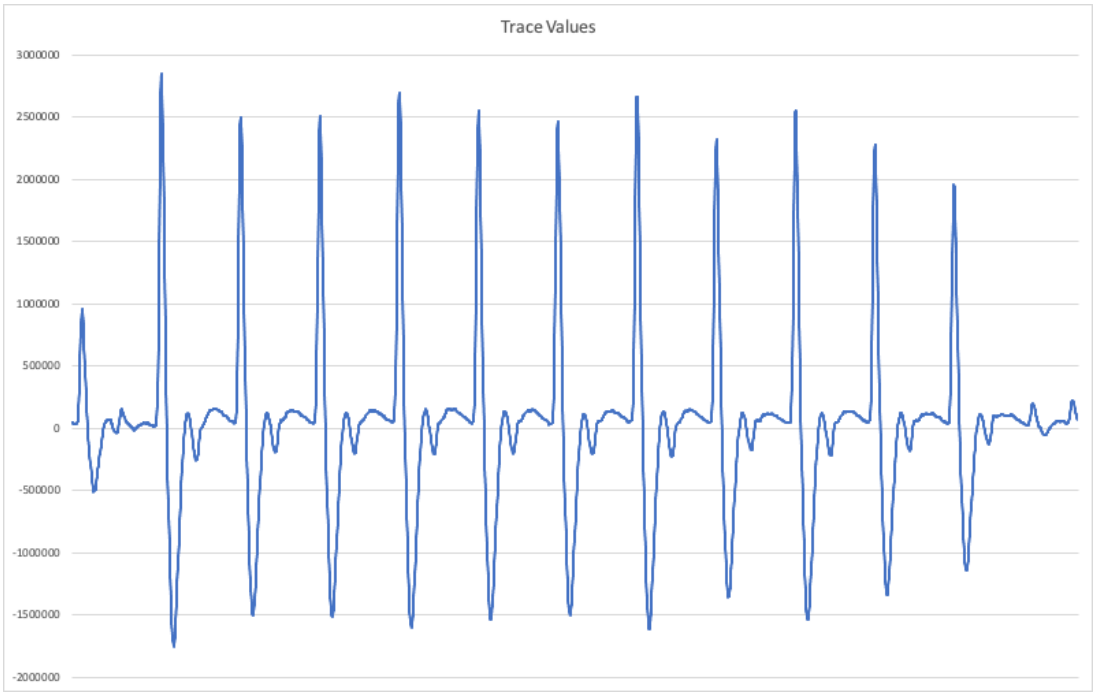

Figure 5.6: Power profile of a measured AES-128 encryption cycle

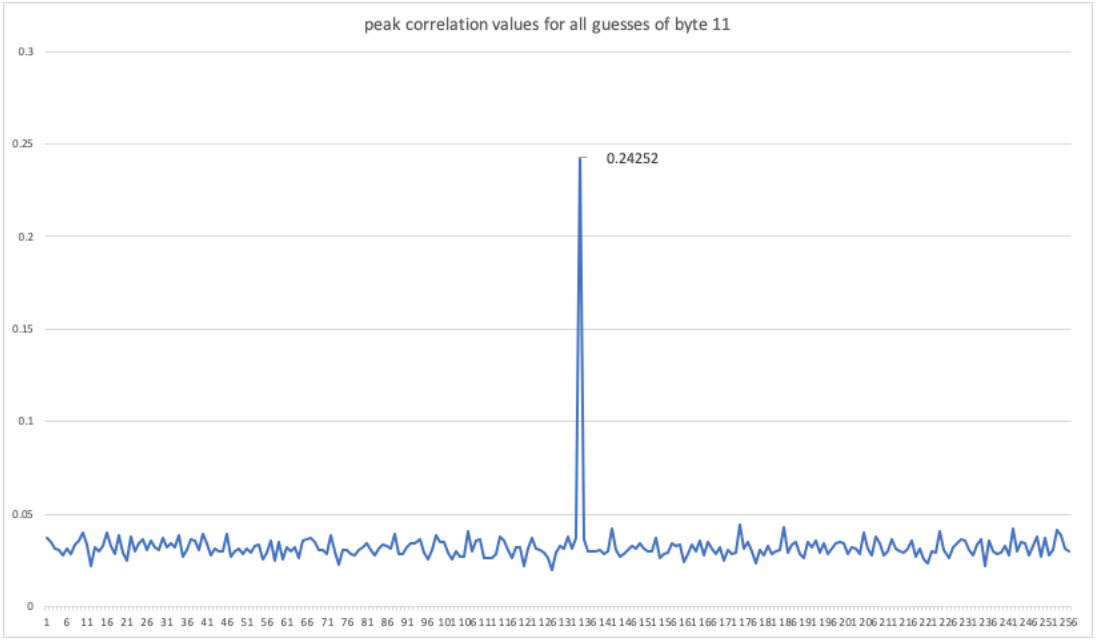

Figure 5.7: VC3 peak correlation values of all guesses for a given byte

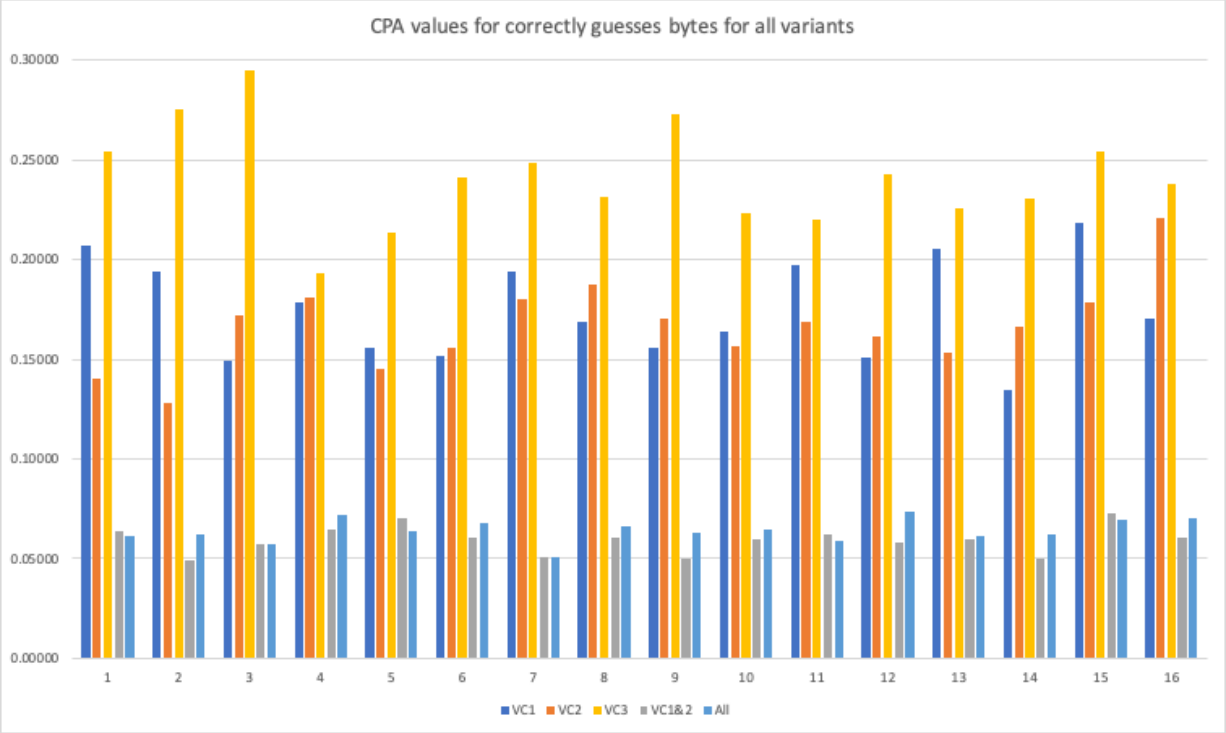

Figure 5.8: Plot of PCC values for each byte (1-16) for all variants using 10000 traces.

5.6 Correlation Analysis of Shuffled Variants

Collected individual traces were combined and shuffled to estimate how the variants would perform in a DPR-based system. A new set of 10000 traces was created using a number of the original traces from each variant. For a combination of VC1 and VC2 5000 traces from each set were put together. For a combination of VC1, VC2, and VC3, 3333 traces from each set were collected. These sets are then shuffled in random order and analysis is performed using the previously described method.

Shuffled traces of VC1 and VC2 show a significant improvement (drop) in correlation (see Figure 5.9). Maximum CPA values of less than 0.1 were reported for each guess. The new peak was recorded at just 0.05 - a 3x decrease from the best individual result (0.15 for VC1) and $\sim$4.2x from the worst (0.24 for VC3). In contrast, Hettwer et al. reported a maximum of 2.2x decrease in correlation when comparing 1 to all 128 variants generated in their work [88]. Shuffling all three classes of variants shows a similar, but slightly higher peak correlation at 0.07 (see Figure E.3). This corresponds to a 3.7x decrease in correlation from the worst individual result. The negative contribution of VC3 once again shows that it is not a good candidate for countermeasures. However, overall improvements over results presented by [88] are shown using a considerably smaller number of variants. Another downside of the approach used by Hettwer et al. is the total size of the bitstream files (approx. 790MB). For the variants generated within

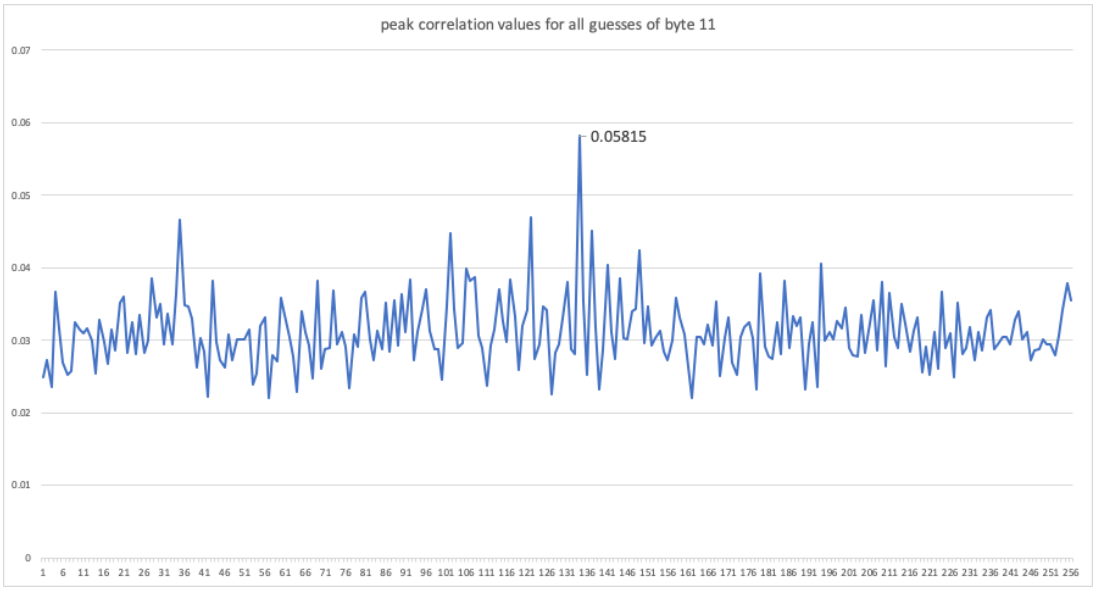

Figure 5.9: Peak correlation values for shuffled VC1 and VC2 for DPR emulation

this work the bitstream files for the full system were considered including the processor. Each configuration resulted in a ∼26MB bitstream file. Difference of 8 bytes was observed from the unmodified AES version that was synthesized and implemented by Vivado.

5.7 Number of Traces for CPA

Lastly, the number of traces required to find the correct guess is considered. In [88], the unprotected version was compromised at 8250 traces collected and the full reconfigurable system required 24500, yielding a 3x increase in immunity. The authors of the SPREAD system used a method of testing similar to the current work and reported ∼1000 traces for the weakest single variant configuration [87]. Applying the shuffling method improved the results by about 5x to 5000 traces using 12 variants. Note that for the evaluation of the SPREAD system the AES core was running at 3.33MHz and 16 iterations were averaged for each encryption with 5000 data points for a total of 30000 traces per each variant. AES core of this work was measured at 10MHz over 250 averaged runs with 2000 data points for a total of 10000 traces per each variant. During the analysis the number of traces was gradually increased in increments of 500 while observing the results. The analysis was stopped once all bytes were guessed successfully.

For VC1, a minimum of 1250 traces was necessary to guess all of the bytes of the last round key correctly, specifically bytes 13 and 15 were found last. Some of the bytes (e.g. 0 and 11) were guessed in as low as 250 traces and most bytes were guessed after analyzing 750-1000 traces. For a representation of the correlation coefficient values for VC1 when guessing byte 11 with 250 traces available, refer to Figure E.4. The peak is close to the rest of the values, but the key byte

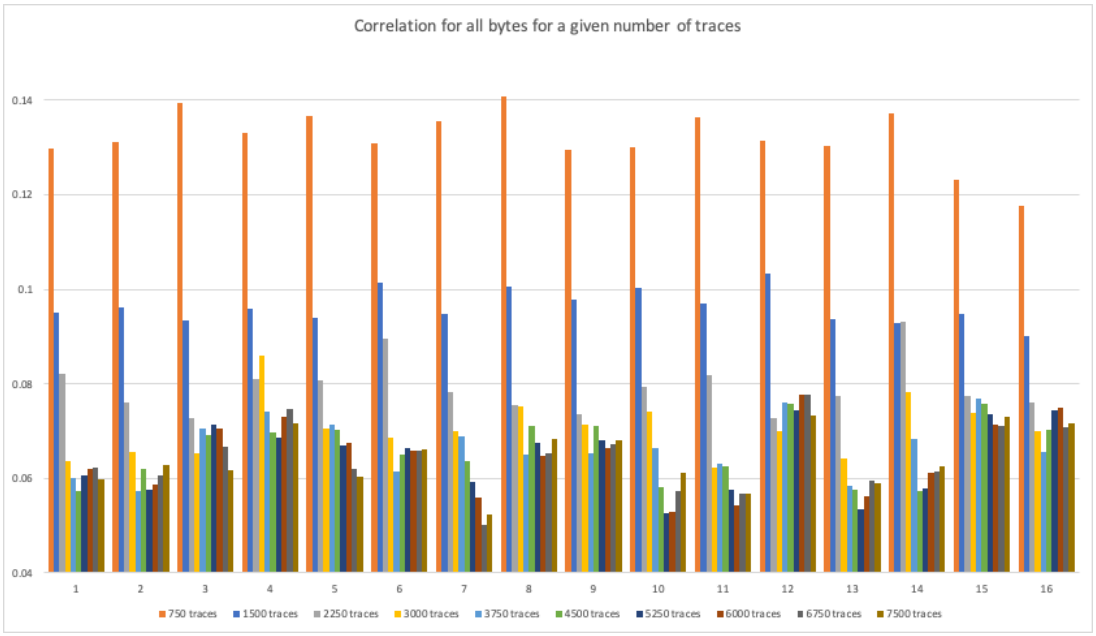

Figure 5.10: Peak CPA values for each byte (1-16) for sets of all shuffled traces

was guessed correctly. Increasing the number of traces will make the peak more prominent in comparison to other values. Variant Class 2 performed similarly to VC1 but byte 1 was guessed after analyzing 1750 traces. Bytes 0, 5, and 11 were exposed at 250 traces in the case of VC3 and the rest were found after 750 traces. Figure E.5 shows a plot of the correlation values for all guesses with increasing number of traces until all bytes were identified correctly. While correlation values for lower number of traces appear to be higher, they correspond to the incorrect guesses. As the number of traces increases and noise values are averaged out, the difference between the peak correlation value and the rest is emphasized, even though values themselves get smaller.

In the case of trace shuffling greater improvements were observed, as expected. All variants were mixed to form multiple sets of traces. The smallest set includes 250 traces from each variant for a total of 750 traces. 250 traces are incrementally added to each set until 3250 traces from each variant are included. In total there are 13 sets of 750 to 9750 traces from VC1, VC2, and VC3. Performing the same type of experiment as with the individual variants, the full key was revealed starting from the second largest set of 7500 traces. The CPA values for every byte for every set of traces are shown in the Figure 5.10. Overall, lower correlation values were observed than in the Figure E.5. Considering these results, the smallest increase in the number of required traces observed is ∼4.3x when compared to a single VC2 case, given that $CPA_{VC2}/CPA_{shuffled}$ is ∼2.2x. The highest increase in the number of traces is ∼10x when compared to VC3, given that $CPA_{VC3}/CPA_{shuffled}$ is ∼3.3x. These values are close to the expectations of [24], stating that a 2x decrease in correlation should require 4x more traces for a successful attack. Hettwer et al. also observed not quite, but a similar trend for their results.

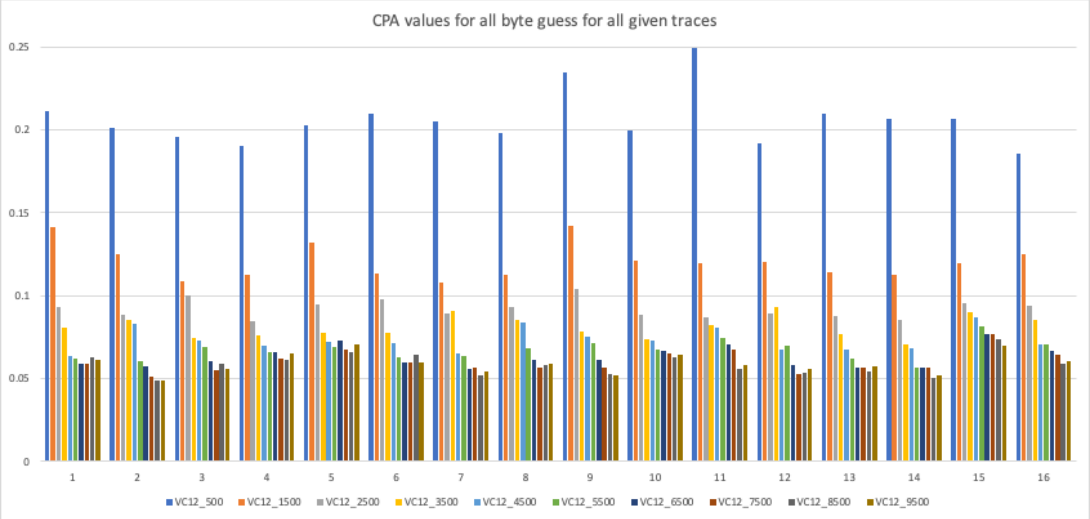

Figure 5.11: Peak CPA values for each byte (1-16) for sets of VC1&2 shuffled traces

When the shuffling method was applied to only Variant Classes 1 and 2, the number of traces required to extract the full key increased even further. Ten sets of traces ranging from 250 to 5250 traces per variant were analyzed. At 7500 and 8500 traces, four and two bytes out of the entire secret key were still guessed incorrectly, respectively. The full key was only revealed after using 9500 traces (4750 from each variant). This is considerably better than either of the variants individually and than the combination of all three. Improvements over individual cases range from $\sim$5.4x to $\sim$12.6x. See Figure 5.11 for the plot of correlation points against all sets of traces.

These results further prove the point that Variant Class 3 requires a conceptual redesign. It could be beneficial to explore the idea of subpar mapping based on the synthesis-driven approach presented for the SPREAD system. Yosys and the ABC logic mapping tools are capable of mapping a design to a variety of gates and primitives. By tweaking which modules are affected, to what extent, and what kind of resource manipulations are included, VC3 can be converted into a better alternative. As it stands, this variant should not be included in DPR-capable or stand-alone systems due to its high leakage and wasteful resource utilization characteristics.

In general, presented results are comparable to [87] and [88], especially considering that a smaller number of variants was generated - three static versions in contrast to twelve from [87] and 128 from [88]. Both the SPREAD system and the current work require further analysis based on full DPR integration but already show promising results. Proposals within this work reported the peak increase of about 12x for the number of traces necessary for analysis and peak reduction of 4x in correlation of data to measured power, when compared to the least protected variant. As such, the evaluation of the proposal of this work can be positively concluded.

Chapter 6

Conclusion

In conclusion, manipulating the implementation of an encryption algorithm using a third-party synthesis tool has been successfully demonstrated. Generated variants, together and separately, have shown improvements in the immunity of a design against side-channel attacks.

Each variant was evaluated individually to provide a detailed insight into the influence on immunity against correlation power analysis. Decrease in correlation was observed among the variants, however, these are intended to be part of a reconfigurable system. Thus, DPR effect was emulated using trace shuffling method and considerable improvements to immunity against CPA were demonstrated. Results were similar to the only other proposal available that uses synthesis-level modifications by Bete et al. The variants generated within this work are conceptually different from each other and thus come in lower quantity and lack implementation diversity. This is in contrast to the work of Hettwer et al., where a large number of variants is presented, but they lack conceptual diversity. Nonetheless, CPA values and number of traces were on par with both studies used for comparison. Proposal of this work can be used in a DPR system with none or minor tweaks.

Additionally, it was demonstrated that the effectiveness of the variants depends not only on their contribution to immunity but also on resource utilization. Among the proposed and evaluated variants, Variant Class 3 had the biggest utilization overhead and the worst CPA results. Thus, manipulating the logic mapping process should be considered with diminishing benefits in mind. Variant Classes 1 and 2 show that the established methods of countermeasures, namely masking and hiding, perform better for both immunity and utilization even outside of the DPR framework.

6.1 Recommendations and Future Work

The results of this work can be improved by following some changes to the work process as summarized below. Yosys is a synthesis tool with a broad range of options that requires an in-depth analysis to extract its full potential. Serial AES variants are an example of this approach and prove that a greater implementation variety can be achieved by adding randomization and generalization in variant designs.

- More extensive usage of the capabilities of Yosys
  - *side-loading designs developed in HDL, rather than manual modifications*
  - *scripting and automation*
- Designing variants with randomization and generalization in mind from the start
  - *to enable a broader range of supported systems*
  - *to have lesser dependence on the structure of the original design*

This study has fulfilled its original intent to explore synthesis level modifications and usage of a third-party tool. Potential in improvement of SCA immunity has been shown and the work can serve as a solid basis for future extensions, including, but not limited to:

- Implement and evaluate a full DPR system with the proposed variants
- Evaluation of the serial AES variants
- Evaluation of the variants against other side-channel attacks (e.g. fault analysis)
- Combination of the synthesis level approach presented here with the P&R method of [88]
- Improvement of VC3-based variants using resource mapping inspired by SPREAD [87]
- Exploration of the effect on other encryption algorithms
- Study of methods alternative to reconfiguration (e.g. high-level synthesis [115], [116])

Bibliography

[1] Cisco, “What are the most common cyber attacks?” https://www.cisco.com/c/en/us/products/security/common-cyberattacks.html.

[2] J. Hamer, R. van Est, L. Royakkers, and N. Alberts, “Cyberspace without conflict,” https://www.rathenau.nl/en/digital-society/cyberspace-without-conflict.

[3] CheckPoint, “What is a cyber attack?” https://www.checkpoint.com/definitions/what-is-cyber-attack/.

[4] K. Bissel, R. M. Lasalle, and P. D. Cin, “Ninth annual cost of cybercrime study,” 2019, https://www.accenture.com/us-en/insights/security/cost-cybercrime-study.

[5] P. Burr, “The evolution of embedded devices: Addressing complex design challenges,” 2018, https://www.embedded.com/the-evolution-of-embedded-devices-addressing-complex-design-challenges/.

[6] I. F. for Information Processing, “Ifip working group 10.4,” https://www.dependability.org/wg10.4/.

[7] J. C. Laprie, *Dependability: Basic Concepts and Terminology*. Vienna: Springer Vienna, 1992, pp. 3–245. [Online]. Available: https://doi.org/10.1007/978-3-7091-9170-5_1

[8] D. P. Tröger, “Dependable systems dependability attributes,” https://osm.hpi.de/teaching/depend/05_dep_attributes.pdf.

[9] F. Stajano, *Security Issues in Ubiquitous Computing**. Boston, MA: Springer US, 2010, pp. 281–314. [Online]. Available: https://doi.org/10.1007/978-0-387-93808-0_11

[10] M. Rouse, “What is encryption and how does it work?” https://searchsecurity.techtarget.com/definition/encryption.

[11] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[12] A. Barkati, “A complete description of data encryption standard (des),” 2019, https://medium.com/@ahsanbarkati/the-des-data-encryption-standard-16466b45c30d.

[13] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. D. Jr., “Advanced encryption standard (aes),” 2001, https://www.nist.gov/publications/advanced-encryption-standard-aes.

[14] R. Singh, “What is an fpga?” 2018, hhttps://numato.com/blog/differences-between-fpga-and-asics/.

[15] Anysilicon, “Fpga vs asic, what to choose?” 2016, https://anysilicon.com/fpga-vs-asic-choose/.

[16] C. P. SOLUTIONS, “Fpga vs. asic: Differences and choosing best for your business,” 2019, https://resources.pcb.cadence.com/blog/2019-fpga-vs-asic-differences-and-choosing-best-for-your-business.

[17] Altera, “An 311: Standard cell asic to fpga design methodology and guidelines,” 2009, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an311.pdf.

[18] W. Lie and W. Feng-yan, “Dynamic partial reconfiguration in fpgas,” in *2009 Third International Symposium on Intelligent Information Technology Application*, vol. 2, 2009, pp. 445–448.

[19] M. Stöttinger, S. Malipatlolla, and Q. Tian, “Survey of methods to improve side-channel resistance on partial reconfigurable platforms,” in *Design Methodologies for Secure Embedded Systems*, A. Biedermann and H. G. Molter, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 63–84.

[20] D. Crawford, “How does aes encryption work?” 2019, https://proprivacy.com/guides/aes-encryption.

[21] P. Kocher, “Differential power analysis,” 1998, https://cryptome.org/jya/dpa.htm.

[22] R. Press, “An introduction to side-channel attacks,” 2018, https://www.rambus.com/blogs/an-introduction-to-side-channel-attacks/.

[23] P. Kocher, J. Jaffe, and B. Jun, “Introduction to differential power analysis and related attacks,” 1998, http://web.mit.edu/6.857/OldStuff/Fall03/ref/kocher-DPATechInfo.pdf.

[24] S. Mangard, E. Oswald, and T. Popp, *Simple Power Analysis*. Boston, MA: Springer US, 2007, pp. 101–118. [Online]. Available: https://doi.org/10.1007/978-0-387-38162-6_5

[25] E. Brier, C. Clavier, and F. Olivier, “Correlation power analysis with a leakage model,” in *Cryptographic Hardware and Embedded Systems - CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–29.

[26] P. N. Fahn and P. K. Pearson, “Ipa: A new class of power attacks,” in *Cryptographic Hardware and Embedded Systems*, Ç. K. Koç and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 173–186.

[27] M. Joye and F. Olivier, *Side-Channel Analysis*. Boston, MA: Springer US, 2005, pp. 571–571. [Online]. Available: https://doi.org/10.1007/0-387-23483-7_394

[28] P. Sasdrich, A. Moradi, O. Mischke, and T. Güneysu, “Achieving side-channel protection with dynamic logic reconfiguration on modern fpgas,” in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 130–136.

[29] J. Daemen and V. Rijmen, “Aes proposal: RIJNDAEL,” 1999, https://cs.ru.nl/~joan/papers/JDA_VRI_Rijndael_V2_1999.pdf.

[30] B. Rothke, “Brute force: Cracking the data encryption standard,” 2010, https://www.rsaconference.com/industry-topics/blog/brute-force-cracking-the-data-encryption-standard.

[31] M. Bellare and P. Rogaway, “Introduction to modern cryptography,” 2005, https://www.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf.

[32] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, “Handbook of applied cryptography,” 1996, http://cacr.uwaterloo.ca/hac/.

[33] C. E. Shannon, “A mathematical theory of cryptography,” 1945, https://www.iacr.org/museum/shannon/shannon45.pdf.

[34] A. Biryukov, “Substitution–permutation (sp) network,” 2011, https://doi.org/10.1007/978-1-4419-5906-5_619.

[35] J. Daemen, “Annex to aes proposal rijndael, chapter 5: Propagation and correlation,” 1998, https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/PropCorr.pdf.

[36] D. Ziener, “Lecture notes: Security in embedded hardware.”

[37] H. Song, G. A. Fink, and S. Jeschke, *Security and Privacy in Cyber-Physical Systems: Foundations, Principles, and Applications*. Wiley-IEEE Press, 2017.

[38] J. Baehr, M. Brunner, and A. Hepp, “Invasive attacks,” https://www.ei.tum.de/en/sec/research/hardware-reverse-engineering/.

[39] M. Weiner, “Invasive attacks,” https://www.ei.tum.de/en/sec/research/invasive-attacks/.

[40] N. I. of Standards and Technology, “Security requirements for cryptographic modules,” 2019, https://csrc.nist.gov/publications/detail/fips/140/3/final.

[41] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” 2019, https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf.

[42] O. Lo, W. J. Buchanan, and D. Carson, “Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa),” 2016, https://doi.org/10.1080/23742917.2016.1231523.

[43] H. Kruppa and S. Umair, “Differential and linear cryptanalysis in evaluating aes candidate algorithms,” 1998, http://www.cs.cmu.edu/~hannes/diffLinAES.pdf.

[44] C. Wiki, “Correlation power analysis,” https://wiki.newae.com/Correlation_Power_Analysis.

[45] N. I. for Telecommunication Sciences, “Hamming weight. telecommunications: Glossary of telecommunication terms,” 1996, https://www.its.bldrdoc.gov/fs-1037/dir-017/_2530.htm.

[46] S. Solutions, “Pearson’s correlation coefficient,” https://www.statisticssolutions.com/pearsons-correlation-coefficient/.

[47] Xilinx, “What is an fpga?” https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[48] W. F.-y. Wang Lie, “Dynamic partial reconfiguration in fpgas,” *Third International Symposium on Intelligent Information Technology Application*, 2009.

[49] Intel, “Partial reconfiguration,” https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/features/partial-reconfiguration.html.

[50] E. Chen, V. Gusev, D. Sabaz, L. Shannon, and W. A. Gruver, “Dynamic partial reconfigurable fpga framework for agent systems,” http://www2.ensc.sfu.ca/~lshannon/file/edward_holomas_11.pdf.

[51] L. A. C. Cardona, “Dynamic partial reconfiguration in fpgas for the design and evaluation of critical systems,” 2016, https://www.tdx.cat/bitstream/handle/10803/386416/lacc1de1.pdf?sequence=1&isAllowed=y.

[52] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura, “Approximate computing: Challenges and opportunities,” in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8.

[53] D. Ziener, “Improving reliability, security, and efficiency of reconfigurable hardware systems,” *CoRR*, vol. abs/1809.11156, 2018. [Online]. Available: http://arxiv.org/abs/1809.11156

[54] Xilinx, “Fpga design flow overview,” https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm.

[55] Hardwarebee, “The ultimate guide to fpga design flow,” 2019, https://hardwarebee.com/ultimate-guide-fpga-design-flow/.

[56] K. Parikh, “Detailed design flow for partial reconfiguration,” *International journal of combined research and development*, vol. pISSN, pp. 2321–2241, 03 2014.

[57] Intel, “Intel quartus prime software suite,” https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[58] Xilinx, “Vivado design suite,” https://www.xilinx.com/products/design-tools/vivado.html.

[59] L. Semiconductor, “Lattice diamond software,” http://www.latticesemi.com/latticediamond.

[60] K. Morris, “The fpga tool problem,” 2016, https://www.eejournal.com/article/20161004-opensource/.

[61] N. Instruments, “Fpga fundamentals,” 2019, https://www.ni.com/en-gb/innovations/white-papers/08/fpga-fundamentals.html#section-1863139369.

[62] Xilinx, “Constraints overview,” https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_using_design_constraints.htm.

[63] EDN, “Fpga constraints for the modern world: Product how-to,” 2016, https://www.edn.com/fpga-constraints-for-the-modern-world-product-how-to/.

[64] C. Souza, “Third-party synthesis tools free altera’s design resources,” 2000, https://www.eetimes.com/update-third-party-synthesis-tools-free-alteras-design-resources/#.

[65] K. Morris, “Fpga synthesis showdown,” 2016, https://www.eejournal.com/article/20160119-synthesis/.

[66] B. E. Nelson, “Third party cad tools for fpga design—a survey of the current landscape,” in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds. Cham: Springer International Publishing, 2019, pp. 353–367.

[67] E. M. Sentovich, K. J. Singh, L. Lavagno, A. Saldanha, R. K. Brayton, C. Moon, R. Murgai, H. Savoj, P. R. Stephan, and A. Sangiovanni-Vincentelli, “Sis: A system for sequential circuit synthesis,” 1992, https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/sis/.

[68] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, “Odin ii - an open-source verilog hdl synthesis tool for cad research,” in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 149–156.

[69] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, “The vtr project: Architecture and cad for fpgas from verilog to routing,”

in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 77–86. [Online]. Available: https://doi.org/10.1145/2145694.2145708

[70] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapidsmith: Do-it-yourself cad tools for xilinx fpgas," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 349–355.

[71] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 41–44. [Online]. Available: https://doi.org/10.1145/1950413.1950425

[72] C. Wolf, "Icestorm," http://www.clifford.at/icestorm/.

[73] ——, "Yosys: Yosys open synthesis suite," http://www.clifford.at/yosys/about.html.

[74] B. L. Synthesis and V. Group, "Abc: A system for sequential synthesis and verification," http://vlsicad.eecs.umich.edu/BK/Slots/cache/www-cad.eecs.berkeley.edu/~alanmi/abc/.

[75] S. EDA, "Symbiotic eda," https://www.symbioticeda.com.

[76] C. Wolf, "yosys – yosys open synthesis suite," https://github.com/YosysHQ/yosys.

[77] R. Chaves, Ł. Chmielewski, F. Regazzoni, and L. Batina, "Sca-resistance for aes: How cheap can we go?" in *Progress in Cryptology – AFRICACRYPT 2018*, A. Joux, A. Nitaj, and T. Rachidi, Eds. Cham: Springer International Publishing, 2018, pp. 107–123.

[78] T. S. Messerges, "Securing the aes finalists against power analysis attacks," in *Fast Software Encryption*, G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 150–164.

[79] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 398–412.

[80] L. Goubin and J. Patarin, "Des and differential power analysis the "duplication" method," in *Cryptographic Hardware and Embedded Systems*, Ç. K. Koç and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 158–172.

[81] M. Stöttinger, S. Malipatlolla, and Q. Tian, "Survey of methods to improve side-channel resistance on partial reconfigurable platforms," in *Design Methodologies for Secure Embedded Systems*, A. Biedermann and H. G. Molter, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 63–84.

[82] T. Güneysu and A. Moradi, “Generic side-channel countermeasures for reconfigurable devices,” in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–48.

[83] P. Yu and P. Schaumont, “Secure fpga circuits using controlled placement and routing,” in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 45–50. [Online]. Available: https://doi.org/10.1145/1289816.1289831

[84] N. Mentens, B. Gierlichs, and I. Verbauwhede, “Power and fault analysis resistance in hardware through dynamic reconfiguration,” in *CHES*, 2008.

[85] P. Sasdrich, A. Moradi, O. Mischke, and T. Güneysu, “Achieving side-channel protection with dynamic logic reconfiguration on modern fpgas,” in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 130–136.

[86] N. Bete, F. Saqib, C. Patel, R. Robucci, and J. Plusquellic, “Side-channel power resistance for encryption algorithms using dynamic partial reconfiguration (spread),” 2018, http://www.hostsymposium.org/host2018/hwdemo/HOST_2017_hwdemo_1.pdf.

[87] I. Bow, N. Bete, F. Saqib, W. Che, C. Patel, R. Robucci, C. Chan, and J. Plusquellic, “Side-channel power resistance for encryption algorithms using implementation diversity,” *Cryptography*, vol. 4, no. 2, p. 13, Apr 2020. [Online]. Available: http://dx.doi.org/10.3390/cryptography4020013

[88] B. Hettwer, J. Petersen, S. Gehrer, H. Neumann, and T. Güneysu, “Securing cryptographic circuits by exploiting implementation diversity and partial reconfiguration on fpgas,” in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 260–263.

[89] P. Sailer, C. Schmittner, and M. Tauber, “Managing the trade-off between security and power consumption for smart cps-iot networks,” 2019, https://ercim-news.ercim.eu/en119/special/managing-the-trade-off-between-security-and-power-consumption-for-smart-cps-iot-networks.

[90] K. Wolter and P. Reinecke, *Performance and Security Tradeoff*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 135–167. [Online]. Available: https://doi.org/10.1007/978-3-642-13678-8_4

[91] K. Lemke-Rust and C. Paar, “An adversarial model for fault analysis against low-cost cryptographic devices,” in *Fault Diagnosis and Tolerance in Cryptography*, L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 131–143.

[92] J. G. J. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2011, pp. 91–99.

[93] Xilinx, "Cfglut5 in ultrascale architecture libraries guide ug974," 2014, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug974-vivado-ultrascale-libraries.pdf.

[94] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[95] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "P.: A testing methodology for side-channel resistance validation, niat," 2011.

[96] D. B. Roy, S. Bhasin, S. Guilley, J.-L. Danger, D. Mukhopadhyay, X. T. Ngo, and Z. Najm, "Reconfigurable lut: A double edged sword for security-critical applications," in *Security, Privacy, and Applied Cryptography Engineering*, R. S. Chakraborty, P. Schwabe, and J. Solworth, Eds. Cham: Springer International Publishing, 2015, pp. 248–268.

[97] N. Bete, M. Nakka, F. Saqib, C. Patel, and R. Robucci, "Implementation diversity and dynamic partial reconfiguration for impeding differential power analysis attacks on fpgas," 2017, http://www.hostsymposium.org/host2017/hwdemo/HOST_2017_hwdemo_25.pdf.

[98] D. C. Keezer and J. Yang, "Biologically inspired hierarchical structure for self-repairing fpgas," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–8.

[99] C. O'Flynn and Z. Chen, "A case study of side-channel analysis using decoupling capacitor power measurement with the openadc," in *Foundations and Practice of Security*, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A. Miri, and N. Tawbi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 341–356.

[100] Xilinx, "Vivado design suite user guide. synthesis. ug901," 2017, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug901-vivado-synthesis.pdf.

[101] B. Verification and S. R. Center, "Command summary. mapping: Luts," https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm#_Toc179291839.

[102] S. Cho, S. Chatterjee, A. Mishchenko, and R. Brayton, "Efficient fpga mapping using priority cuts," 2007, https://people.eecs.berkeley.edu/~alanmi/publications/2007/fpga07_fast.pdf.

[103] C. Wolf, "abc - use abc for technology mapping," http://www.clifford.at/yosys/cmd_abc.html.

[104] Xilinx, “7 series fpgas configurable logic block user guide. ug474,” 2016, https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[105] C. Wolf, “Yosys manual,” http://www.clifford.at/yosys/files/yosys_manual.pdf.

[106] ——, “The nextpnr foss fpga place-and-route tool,” http://www.clifford.at/papers/2018/nextpnr/slides.pdf.

[107] ——, “synth_xilinx - synthesis for xilinx fpgas,” http://www.clifford.at/yosys/cmd_synth_xilinx.html.

[108] OnlineDomainTools, “Aes – symmetric ciphers online,” http://aes.online-domain-tools.com/.

[109] Xilinx, “Zynq ultrascale+ mpsoc zcu102 evaluation kit,” https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#overview.

[110] R. Cofer and B. F. Harding, “Chapter 9 - design constraints and optimization,” in *Rapid System Prototyping with FPGAs*, ser. Embedded Technology, R. Cofer and B. F. Harding, Eds. Burlington: Newnes, 2006, pp. 137 – 154. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978075067866750010X

[111] Xilinx, “Zynq ultrascale+ mpsoc processing system ip,” https://www.xilinx.com/products/intellectual-property/zynq-ultra-ps-e.html.

[112] ——, “Axi reference guide ug761,” https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.

[113] Langer, “Icr hv500-75 near-field microprobe 200 khz to 1 ghz,” https://www.langer-emv.de/en/product/near-field-microprobe-icr-hv-h-field/40/icr-hv500-75-near-field-microprobe-200-khz-to-1-ghz/115.

[114] P. Technology, “Picoscope® 6000c/d series,” https://www.picotech.com/oscilloscope/6000/picoscope-6000-overview.

[115] L. Zhang, W. Hu, A. Ardeshiricham, Y. Tai, J. Blackstone, D. Mu, and R. Kastner, “Examining the consequences of high-level synthesis optimizations on power side-channel,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1167–1170.

[116] S. T. C. Konigsmark, D. Chen, and M. D. F. Wong, “High-level synthesis for side-channel defense,” in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 37–44.

Appendix A

Verifying Implemented Variants

This appendix section includes results of functional simulations performed on variants of round-based AES implementation. These variants were generated through Yosys, exported as a netlist and simulated using Xilinx Vivado. The inputs and expected output of the system are shown below:

- Key: `2b7e151628aed2a6abf7158809cf4f3c`
- Plaintext: `3243f6a8885a308d313198a2e0370734`
- Ciphertext: `3925841d02dc09fbdc118597196a0b32`

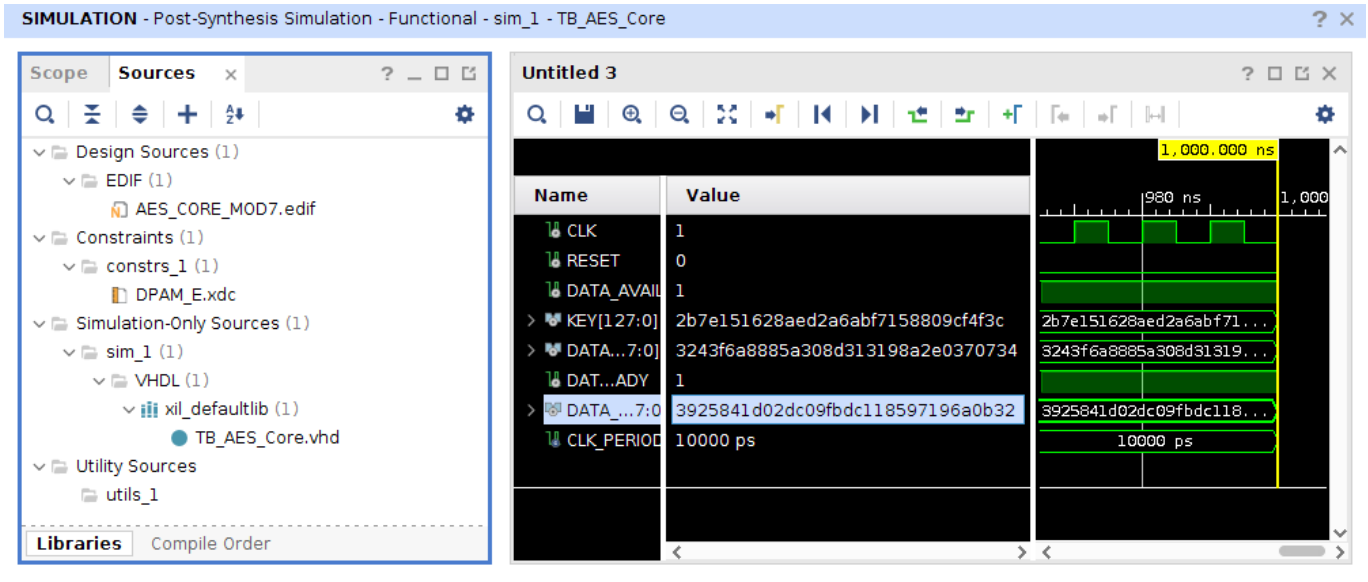

Figure A.1: Variant Class 1. Yosys generated netlist ”AES_CORE_MOD7.edif”.

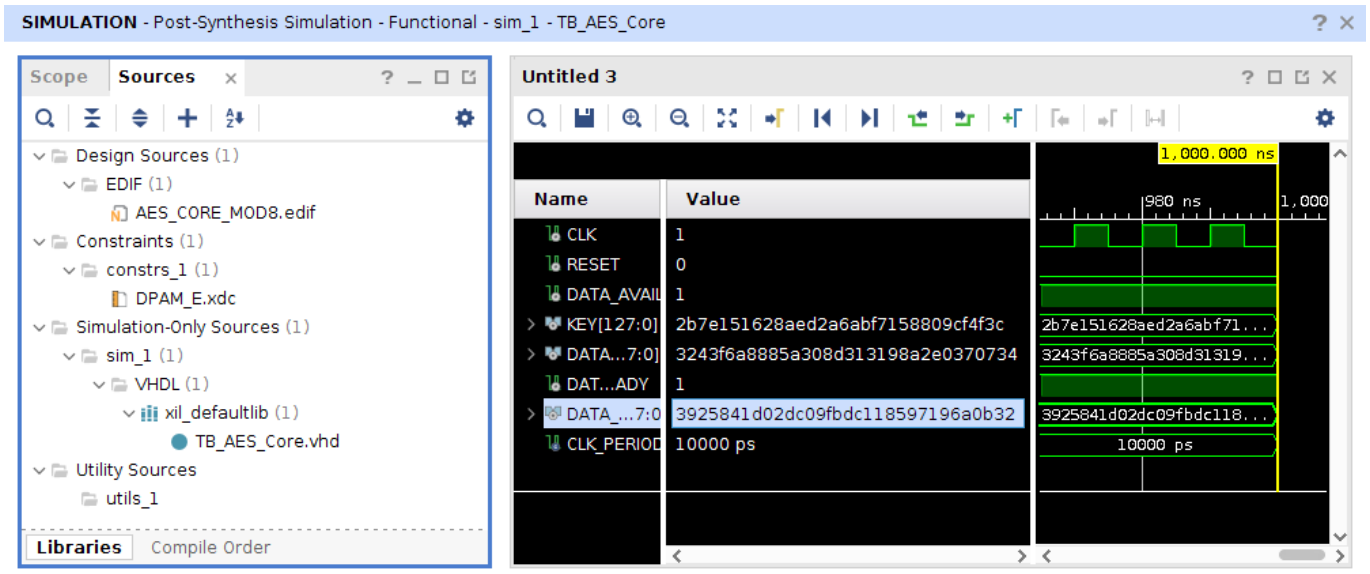

Figure A.2: Variant Class 2. Yosys generated netlist ”AES_CORE_MOD8.edif”.

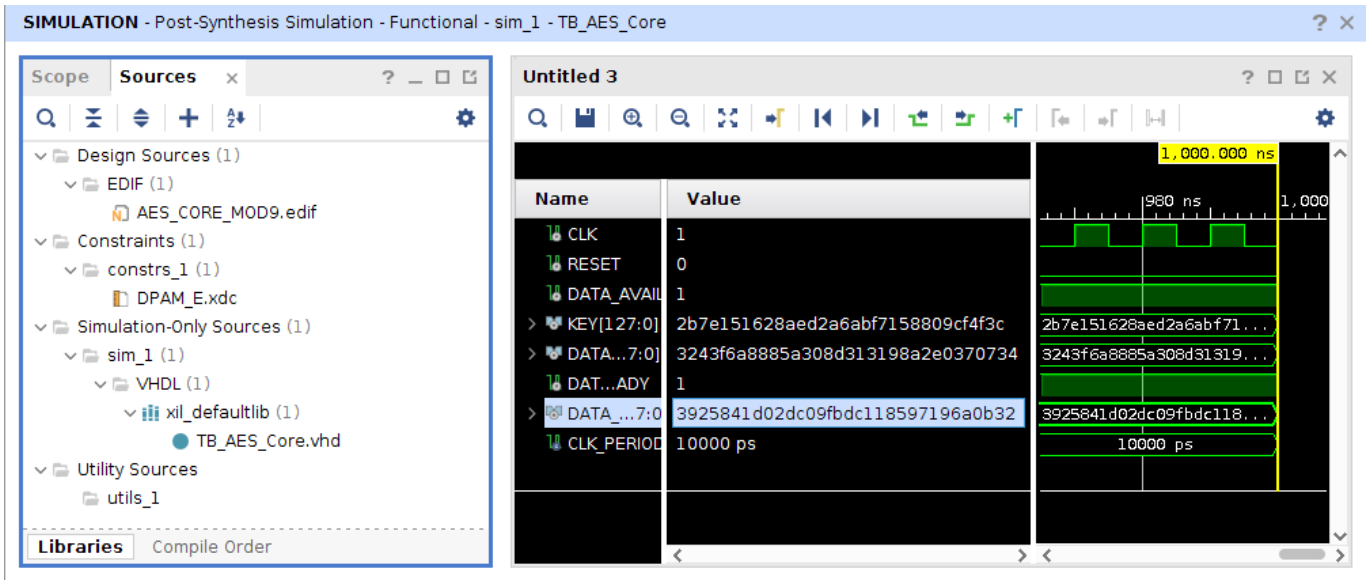

Figure A.3: Variant Class 3. Yosys generated netlist ”AES_CORE_MOD9.edif”.

Appendix B

Testing Implemented Variants

B.1 Trigger signal

Trigger signal as observed from the scope (the same signal was observed for all variants). Due to currently limited access to resources, input and output values from the AES core running in the SDK console are not available and could not be included to this Appendix. Excerpts of the code loaded through SDK are available below.

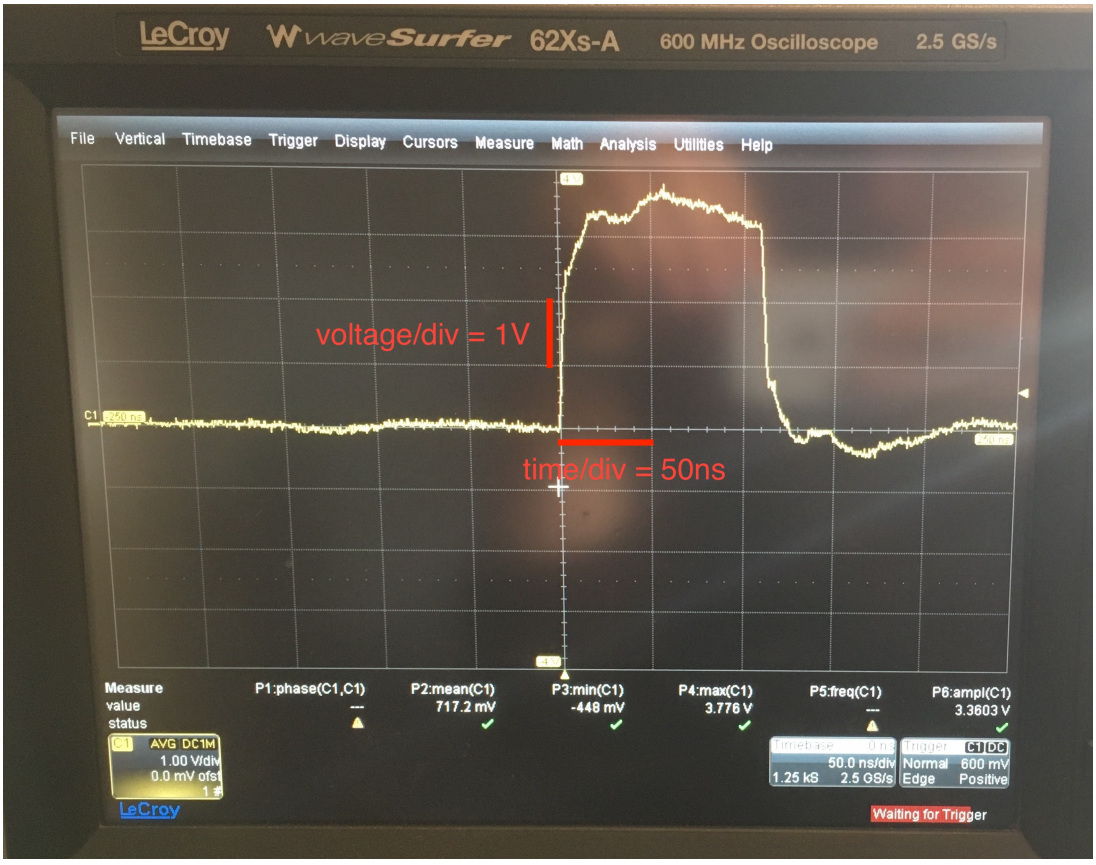

Figure B.1: Trigger signal observed using an oscilloscope.

B.2 Code to operate the AES core

```
/* note: this is an excerpt and not complete code */
/* plaintexts are generated randomly using an LFSR*/
/* values are written to corresponding registers */
/* and output is read back from the registers */

    t0 = lfsr_step(); //8
    t1 = lfsr_step(); //16
    t2 = lfsr_step(); //24
    t3 = lfsr_step(); //32
    t4 = lfsr_step(); //40
    t5 = lfsr_step(); //48
    t6 = lfsr_step(); //56
    t7 = lfsr_step(); //64

    Plain_1 = (uint64_t) t7 | ((uint64_t) t6 << 8) | ((uint64_t) t5 << 16) | ((uint64_t) t4 << 24) | ((uint64_t) t3 << 32) | ((uint64_t) t2 << 40) | ((uint64_t) t1 << 48) | ((uint64_t) t0 << 56);

    AES_BaseAddr_Plain_1 = (uint32_t)(Plain_1 >> 32);
    Xil_Out32(XPAR_AES10_0_S00_AXI_BASEADDR + 32, (uint32_t)(Plain_1 >> 32));
/* . . . */
/* . . . */
    Xil_Out32(XPAR_AES10_0_S00_AXI_BASEADDR + 44, (uint32_t)(Plain_2));
    // Data written to the core
    xil_printf("Data_In = %x%x%x%x\n\r", AES_BaseAddr_Plain_4, AES_BaseAddr_Plain_3, AES_BaseAddr_Plain_2, AES_BaseAddr_Plain_1);
/* . . . */
    {
      /* . . . */
      Xil_Out32(XPAR_AES10_0_S00_AXI_BASEADDR + 52, 0x00000001);
      data_avail = Xil_In32(XPAR_AES10_0_S00_AXI_BASEADDR + 52);

      //Reading DATA_OUT
      DATA_OUT1 = Xil_In32(XPAR_AES10_0_S00_AXI_BASEADDR);
      DATA_OUT2 = Xil_In32(XPAR_AES10_0_S00_AXI_BASEADDR + 4);
      DATA_OUT3 = Xil_In32(XPAR_AES10_0_S00_AXI_BASEADDR + 8);
      DATA_OUT4 = Xil_In32(XPAR_AES10_0_S00_AXI_BASEADDR + 12);
      xil_printf("DATA_OUT = %x%x%x%x\n\r", DATA_OUT4, DATA_OUT3, DATA_OUT2, DATA_OUT1);
      /* . . . */
    }
```

Appendix C

Yosys Commands

C.1 Modified ”add” command

```
/*** note: the following is an excerpt and not the complete code ***/

include "kernel/yosys.h"

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

// EK: function to add unary cells as per Section 5.1.1 Yosys Manual: name, sig_a
    , sig_y, is_signed
static void add_unary_cell(RTLIL::Design *design, RTLIL::Module *module, std::
    string name, std::string type, int width_a, int width_y, bool flag_signed_a=
    false)
{
  RTLIL::Cell *cell = NULL;

  name = RTLIL::escape_id(name);

  if (module->count_id(name) != 0)
  {
    if (module->cells_.count(name) > 0)
      cell = module->cells_.at(name);

    if (cell == NULL)
      log_cmd_error("Found incompatible object with same name in module %s!\n",
    module->name.c_str());

    log("Module %s already has such an object.\n", module->name.c_str());
  }
  else
  {
    if    (type == "$not")
```

```
28        cell = module->addNot(name, module->addWire((name + "_A"), width_a), module
      ->addWire((name + "_Y"), width_y), flag_signed_a, "");
29      else if (type == "$pos")
30        cell = module->addPos(name, module->addWire((name + "_A"), width_a), module
      ->addWire((name + "_Y"), width_y), flag_signed_a, "");
31 /* . . . */
32 /* . . . */
33      else
34      {
35        log("Cell type %s not supported or incorrect.", type.c_str());
36        return;
37      }
38
39      // log cell addition
40      log("Added cell %s to module %s.\n", name.c_str(), module->name.c_str());
41    }
42 /* . . . */
43 /* . . . */
44 }
```

C.2 Modified synth_xilinx command

```
1  /*** note: the following is an excerpt and not the complete code ***/
2      if (check_label("map_luts")) {
3  /* . . . */
4  /* . . . */
5        else {
6          if (nowidelut)
7            run("abc -luts 2:2,3,6:5" + string(retime ? " -dff" : ""));
8          else
9            run("abc -luts 2:2,3,6:5,10,20" + string(retime ? " -dff" : ""));
10       }
11       run("clean");
12 /* . . . */
13 /* . . . */
14
15 /* during synthesis, logic introduced is preserved by attributes */
16 /* by default, writing an EDIF netlist removes them */
17 /* therefore, the command has been modified to preserve these changes */
18     if (check_label("edif")) {
19       if (!edif_file.empty() || help_mode)
20         run(stringf("write_edif -attrprop -pvector bra %s", edif_file.c_str()));
21     }
22 /* . . . */
```

Appendix D

Serial-based AES Variants

D.1 Sample C code to generate a Yosys script

Automating and introducing randomization in the variants generated for the serial AES version.

```
// note: this is not a complete representation of the code
// details omitted for simplicity
int main() {
/* . . . */
    RF_cmd_gen (k, fp);
    Cell_cmd_gen (fp);
/* . . . */
    int module_pick_KS, module_pick_CL;
    module_pick_KS = rand()%3;
    if (module_pick_KS == 1)
      KS_cmd_gen (k, fp);
    module_pick_CL = rand()%3;
    if (module_pick_CL == 2)
      CL_cmd_gen (k, fp);
/* . . . */
    fprintf (fp, "cd; synth_xilinx -top AES128 -run :edif;;\n");
    fprintf (fp, "cd RoundFunction;\n");
    fprintf (fp, "setattr -set KEEP 1 *dffe* rOUT;\n");
    fprintf (fp, "setattr -set DONT_TOUCH 1 *dffe* rOUT;\n");
    fprintf (fp, "cd;\n");
/* . . . */
    fprintf (fp, "synth_xilinx -top AES128 -run edif: -edif ./circuits/
   SerialAES_serial_cellmod_%d.edif", k);
    // close the filename
    fclose (fp);
  }
  return 0;
}
```

D.2 Sample script to be imported into Yosys

```
design -reset; // clear the workspace
read_ilang ./circuits/SerialAES.ilang; // process the input design
hierarchy -check -top AES128; // checking design structure and top level entity

cd Cell(size=8); // select a submodule to work in

// add an inverter with 8-bit input, 8-bit output, unsigned
add -unary inv0 $not 8 8 0;
add -unary inv1 $not 8 8 0;
add -unary inv3 $not 8 8 0;
add -unary inv4 $not 8 8 0;

connect -set inv0_A DIN0; // set connections for the added cells
connect -set inv1_A DIN1;

connect -port $verific$mux_7$./circuits/s-aes/Cell.vhd:68$309 \A inv0_Y;
connect -port $verific$mux_7$./circuits/s-aes/Cell.vhd:68$309 \B inv1_Y;

connect -set DOUT inv3_Y;
connect -port $verific$DOUT_reg$./circuits/s-aes/Cell.vhd:70$313 \Q inv3_A;

connect -set inv4_A DOUT;
connect -port $verific$mux_9$./circuits/s-aes/Cell.vhd:69$311 \A inv4_Y;
connect -port $verific$mux_10$./circuits/s-aes/Cell.vhd:69$312 \B inv4_Y;

// prevent synthesis from optimising away the extra logic
setattr -set KEEP 1 inv*;
setattr -set DONT_TOUCH 1 inv*;
setattr -mod -set KEEP 1;
setattr -mod -set DONT_TOUCH 1;

// execute Xilinx oriented synthesis until saving a netlist
cd; synth_xilinx -top AES128 -run :edif;

// prevent P&R tool from optimising away the extra logic
cd Cell(size=8);
setattr -mod -set KEEP 1;
setattr -mod -set DONT_TOUCH 1;
setattr -set KEEP 1 *
setattr -set DONT_TOUCH 1 *;
cd;

// finalize the synthesis process and output a netlist
synth_xilinx -top AES128 -run edif: -edif ./circuits/SerialAES_cellonly.edif
```

D.3 List of generated Serial-based AES variants

Table D.1: Serial AES variants generated using randomized scripts

Variant Name	*Description*
randmod	FF+INV chains inserted in RoundFunction and ControlLogic modules
cellonly	Cell module modified to invert 8-bit values stored in a register (all bits 7:0)
cellmod v0	Cell module modified to invert 8-bit values stored in a register (only LSB 0:0) FF+INV chain inserted in RoundFunction
cellmod v1	Cell module modified to invert 8-bit values stored in a register (all bits 7:0) FF+INV chain inserted in RoundFunction and KeySchedule
cellmod v3	Cell module modified to invert 8-bit values stored in a register (bits 3:0) FF+INV chain inserted in RoundFunction, KeySchedule, and ControlLogic
cellmod v4	Cell module modified to invert 8-bit values stored in a register (bits 5:0) FF+INV chain inserted in RoundFunction and ControlLogic
cellmod v6	Cell module modified to invert 8-bit values stored in a register (bits 4:0) FF+INV chain inserted in RoundFunction, KeySchedule, and ControlLogic
lutonly	Constraining LUT mapping (max 4-input) during synthesis in RoundFunction, SBox, MixColumn modules
lutmod	Cell module modified to invert 8-bit values stored in a register (bits 2:0) FF+INV chain inserted in RoundFunction, KeySchedule, and ControlLogic Constraining LUT mapping (max 4-input) during synthesis in RoundFunction, SBox, MixColumn modules

Appendix E

Measurement Results

E.1 Peak Correlation Plots

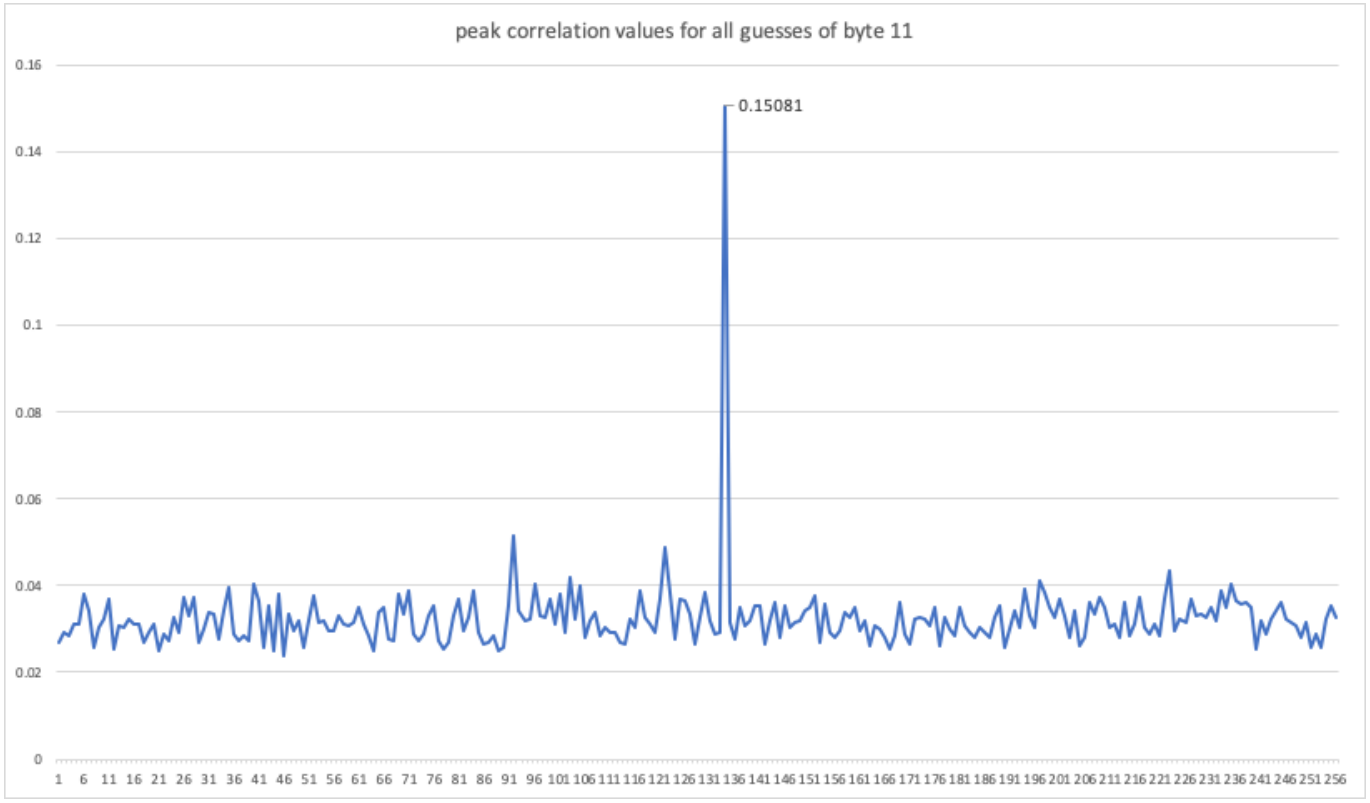

Figure E.1: Peak Correlation Values for Variant Class 1

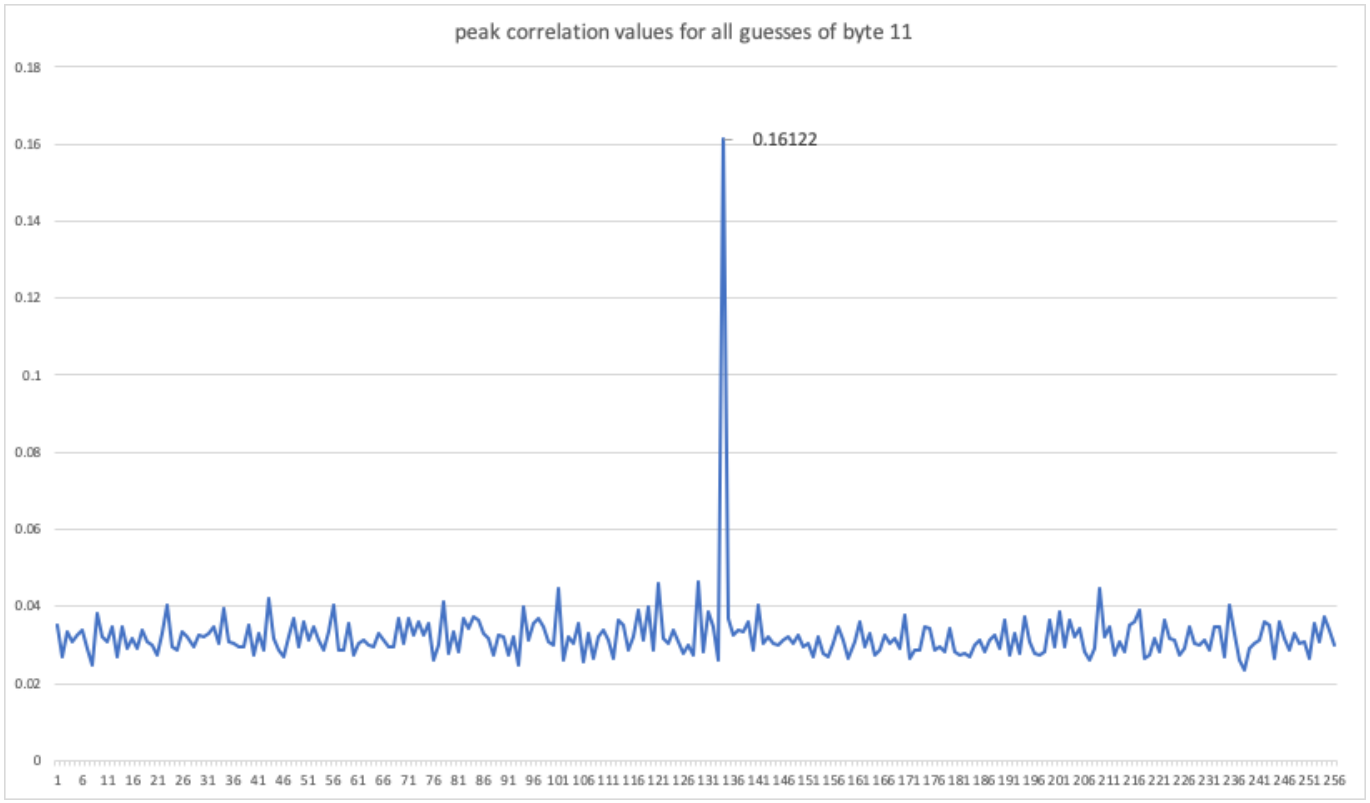

Figure E.2: Peak Correlation Values for Variant Class 2

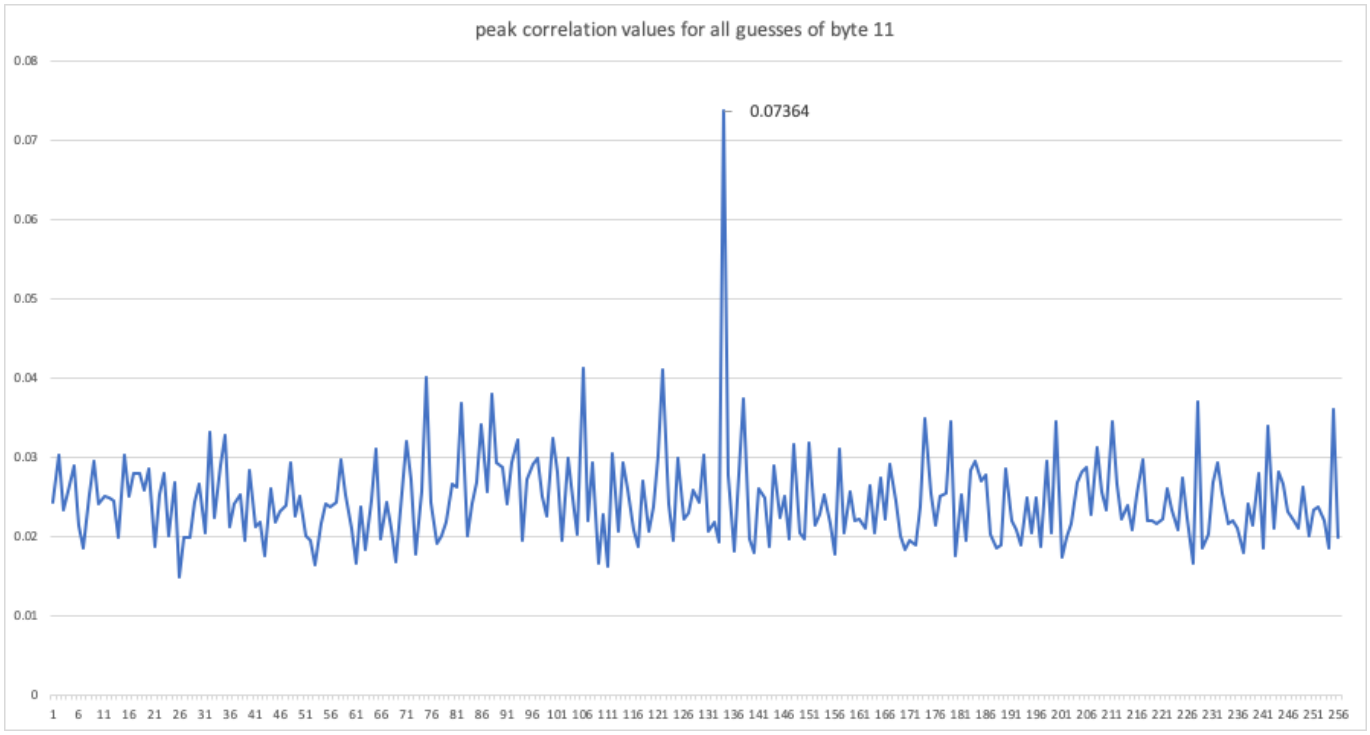

Figure E.3: Peak correlation values for all variants shuffled for DPR emulation

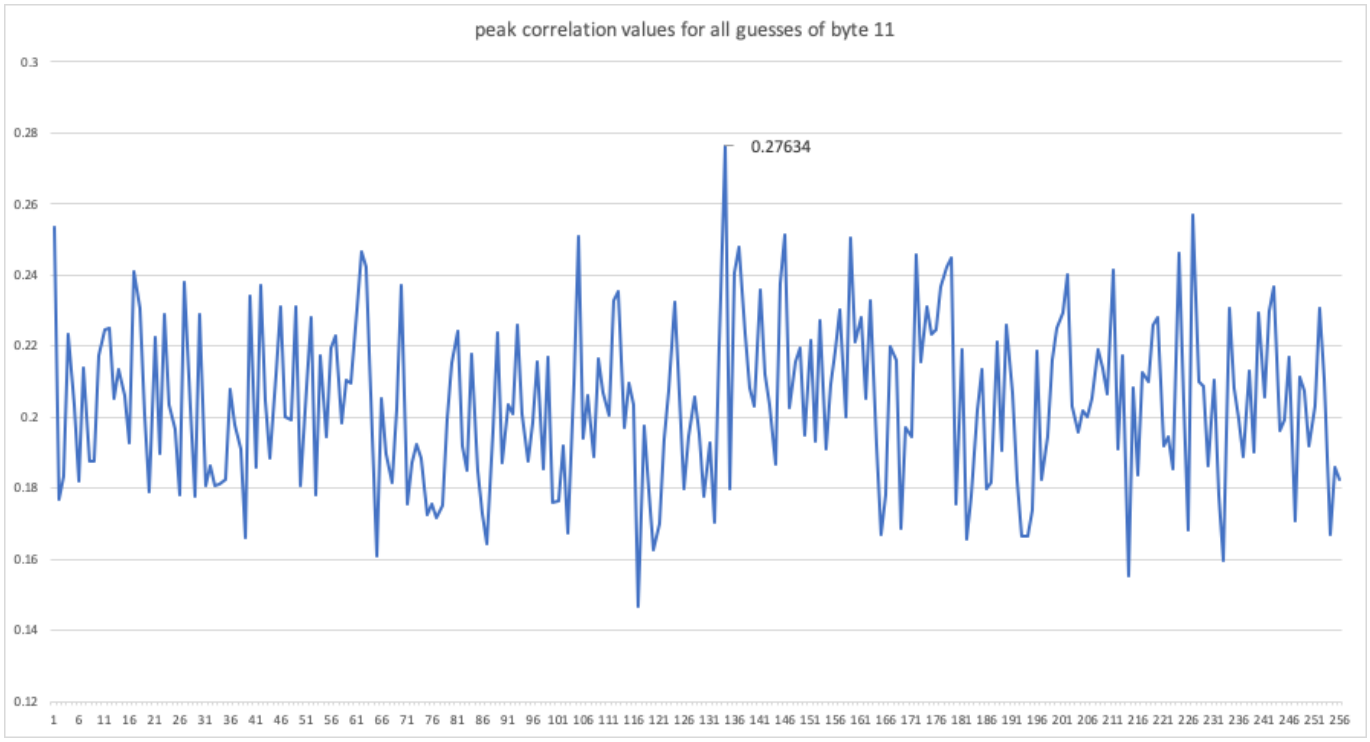

Figure E.4: Variant Class 1 peak correlation values for 250 traces

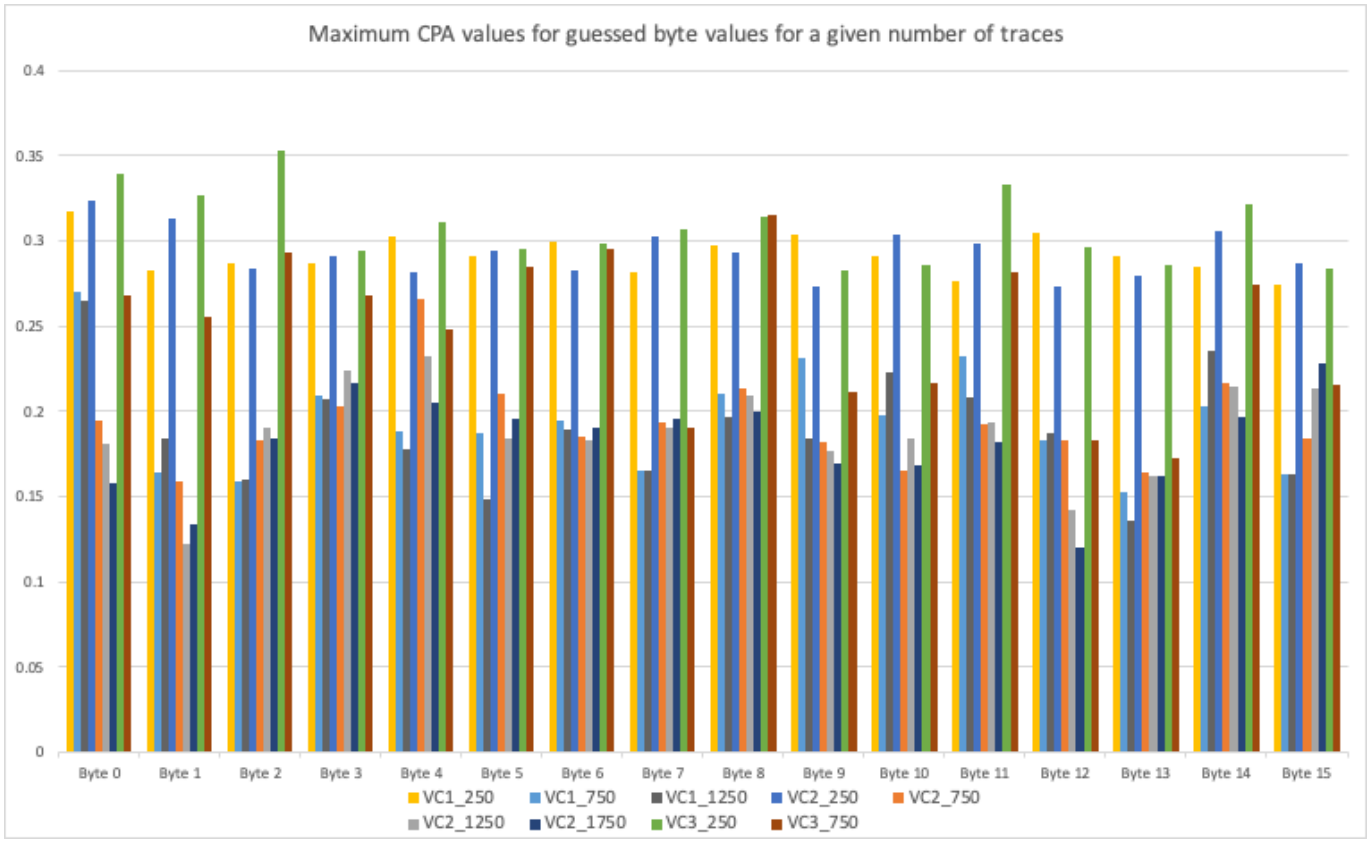

Figure E.5: Peak CPA values for all bytes and number of individual traces used

E.2 Code excerpt for key byte guessing

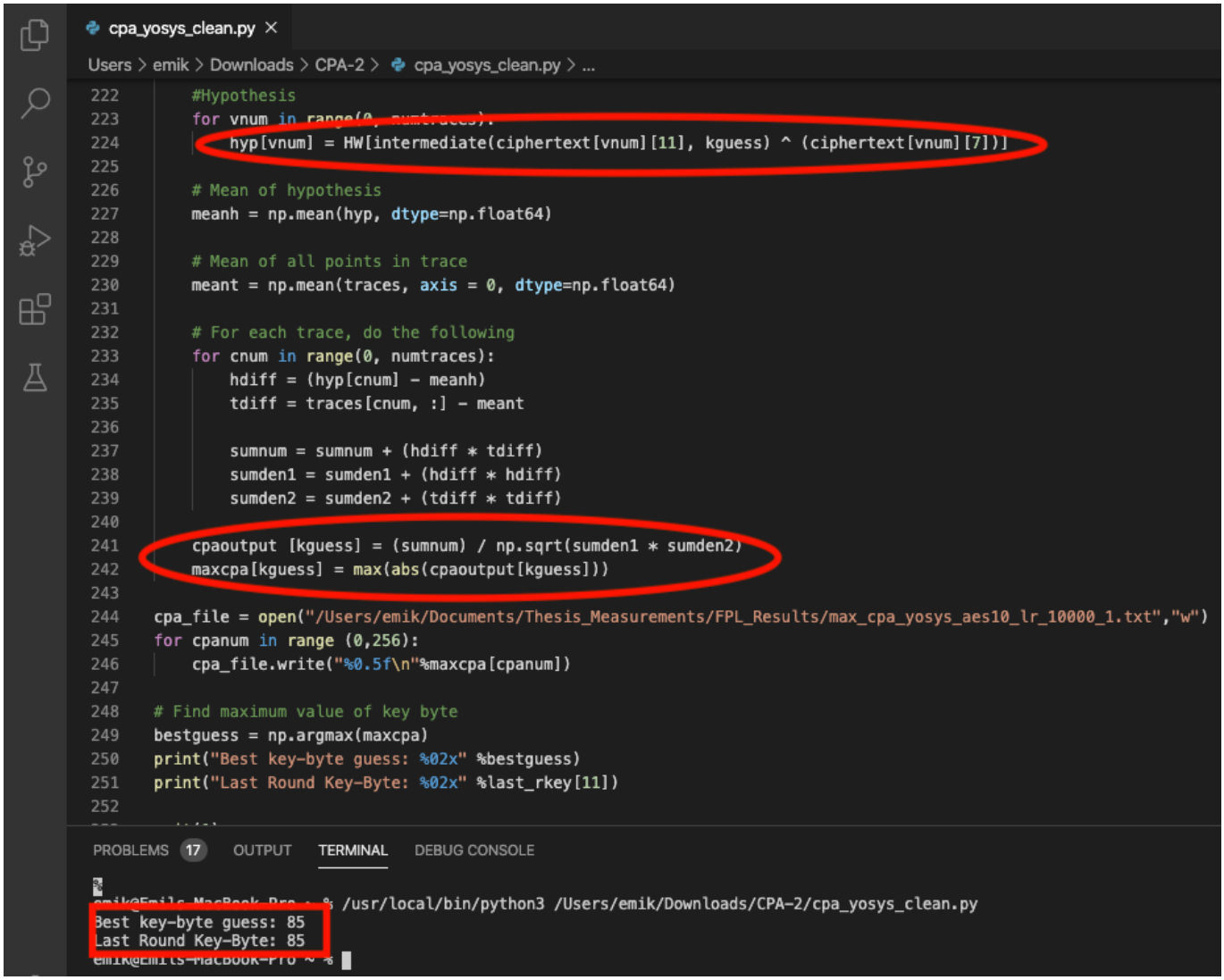

Figure E.6: Highlighted sections refer to the Hamming weight model, calculation of maximum CPA values, and the observed output for the correct guess.

Appendix F

Analysis for All Bytes

The Table F.1 (refer to Figure 5.8 for visual representation) shows PCC values for correctly guessed byte values when using 10000 traces for a correlation power analysis attack.

Table F.1: PCC for all bytes for each variant at 10000 traces

Byte #	Configuration					Average per byte
	VC1	***VC2***	***VC3***	***VC1&2***	***All***	
Byte 0	0.207	0.140	0.254	0.064	0.061	0.145
Byte 1	0.194	0.128	0.275	0.049	0.062	0.142
Byte 2	0.149	0.172	0.295	0.057	0.057	0.146
Byte 3	0.178	0.181	0.193	0.064	0.072	0.138
Byte 4	0.156	0.145	0.213	0.070	0.064	0.130
Byte 5	0.152	0.155	0.241	0.060	0.068	0.135
Byte 6	0.194	0.180	0.248	0.050	0.051	0.145
Byte 7	0.169	0.187	0.231	0.061	0.066	0.143
Byte 8	0.156	0.171	0.273	0.050	0.063	0.142
Byte 9	0.164	0.156	0.223	0.060	0.065	0.134
Byte 10	0.197	0.169	0.220	0.063	0.059	0.141
Byte 11	0.151	0.161	0.243	0.058	0.074	0.137
Byte 12	0.206	0.153	0.225	0.060	0.061	0.141
Byte 13	0.135	0.167	0.231	0.050	0.062	0.129
Byte 14	0.218	0.179	0.254	0.073	0.069	0.159
Byte 15	0.170	0.221	0.238	0.061	0.070	0.152
average	0.175	0.167	0.241	0.059	0.064	
minimum	0.135	0.128	0.193	0.049	0.051	
maximum	0.218	0.221	0.295	0.073	0.074	

Appendix G

Expanded Secret Key for AES-128

Table G.1: All round keys derived from the secret key used for measurements

Key:	*Initial*	*Round #1*	*Round #2*	*Round #3*	*Round #4*	*Round #5*	*Round #6*	*Round #7*	*Round #8*	*Round #9*	*Last*
Byte 0	**0xFF**	**0xE8**	**0xAD**	**0x09**	**0xE1**	**0xE5**	**0x71**	**0xE9**	**0x96**	**0x8B**	**0xD6**
Byte 1	**0xFF**	**0xE9**	**0xAE**	**0x0E**	**0x6A**	**0xBA**	**0xD0**	**0x0D**	**0x33**	**0xF0**	**0x0A**
Byte 2	**0xFF**	**0xE9**	**0xAE**	**0x22**	**0xBD**	**0xF3**	**0x7D**	**0x20**	**0x73**	**0x3F**	**0x35**
Byte 3	**0xFF**	**0xE9**	**0x19**	**0x77**	**0x3E**	**0xCE**	**0xB3**	**0x8D**	**0x66**	**0x23**	**0x88**
Byte 4	**0xFF**	**0x17**	**0xBA**	**0xB3**	**0x52**	**0xB7**	**0xC6**	**0x2F**	**0xB9**	**0x32**	**0xE4**
Byte 5	**0xFF**	**0x16**	**0xB8**	**0xB6**	**0xDC**	**0x66**	**0xB6**	**0xBB**	**0x88**	**0x78**	**0x72**
Byte 6	**0xFF**	**0x16**	**0xB8**	**0x9A**	**0x27**	**0xD4**	**0xA9**	**0x89**	**0xFA**	**0xC5**	**0xF0**
Byte 7	**0xFF**	**0x16**	**0x0F**	**0x78**	**0x46**	**0x88**	**0x3B**	**0xbB6**	**0xD0**	**0xF3**	**0x7B**
Byte 8	**0xFF**	**0xE8**	**0x52**	**0xE1**	**0xB3**	**0x04**	**0xC2**	**0xED**	**0x54**	**0x66**	**0x82**
Byte 9	**0xFF**	**0xE9**	**0x51**	**0xE7**	**0x3B**	**0x5D**	**0xEB**	**0x50**	**0xD8**	**0xA0**	**0xD2**
Byte 10	**0xFF**	**0xE9**	**0x51**	**0xCB**	**0xEC**	**0x38**	**0x91**	**0x18**	**0xE2**	**0x27**	**0xD7**
Byte 11	**0xFF**	**0xE9**	**0xE6**	**0x9E**	**0xD8**	**0x50**	**0x6B**	**0xDD**	**0x0D**	**0xFE**	**0x85**
Byte 12	**0xFF**	**0x17**	**0x45**	**0xA4**	**0x17**	**0x13**	**0xD1**	**0x3C**	**0x68**	**0x0E**	**0x8C**
Byte 13	**0xFF**	**0x16**	**0x47**	**0xA0**	**0x9B**	**0xC6**	**0x2D**	**0x7D**	**0xA5**	**0x05**	**0xD7**
Byte 14	**0xFF**	**0x16**	**0x47**	**0x8C**	**0x60**	**0x58**	**0xC9**	**0xD1**	**0x33**	**0x14**	**0xC3**
Byte 15	**0xFF**	**0x16**	**0xD0**	**0x6E**	**0xB6**	**0xE6**	**0x8D**	**0x50**	**0x5D**	**0xA3**	**0x26**