UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Cybersecurity and Data Science»

ΠΜΣ «Κυβερνοασφάλεια και επιστήμη δεδομένων»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Implementation and Evaluation of Side-Channel Attack Countermeasures for FPGAs** Υλοποίηση και Αξιολόγηση Αντιμέτρων Επιθέσεων Πλευρικού Καναλιού για FPGAs
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Eirini-Eleni Galidaki** Ειρήνη-Ελένη Γαλιδάκη
Father's name: Πατρώνυμο:	**Michail** Μιχαήλ
Student's ID No: Αριθμός Μητρώου:	ΜΠΚΕΔ2309
Supervisor: Επιβλέπων:	**Athanasios Papadimitriou, Professor** Αθανάσιος Παπαδημητρίου, Διδάσκων ΠΜΣ

March 2026/ Μάρτιος 2026

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Panagiotis Kotzanikolaou **Professor**	**Michael Psarakis** **Professor**	**Athanasios Papadimitriou** **Professor**
Παναγιώτης Κοτζανικολάου Καθηγητής	Μιχαήλ Ψαράκης Καθηγητής	Αθανάσιος Παπαδημητρίου Διδάσκων ΠΜΣ

Abstract

To this day, side-channel attacks are a well-known critical threat to cryptographic implementations on FPGAs, exploiting various physical effects such as power consumption and routing activity, which are often responsible for unintentionally leaking sensitive information of cryptographic operations. Therefore, a practical and tool-oriented methodology is proposed in the present thesis, which aims to enhance the side-channel resistance of an AES design implemented on a Xilinx Artix-7 FPGA. Firstly, an AES core implementation is floor planned inside a dedicated area on the FPGA board to contain its routing, thus creating a stable baseline AES implementation, which will pose as a reference for building structured noise generators. All noise generators and their variants are constructed from Look-Up Table (LUT) primitives, utilized as Shift Register LUTs (SRLs) and are inserted into pre-allocated available regions in between AES operations using RapidWright. In total, four different spatial placement patterns, each with three arbitrary noise variants have been produced, amounting to twelve individual noise generators. Subsequently, each implementation is programmed on a CW312-A35 board, where thousands of power traces are captured using a ChipWhisperer Husky for the purpose of a series of side-channel attacks. Eventually, Correlation Power Analysis (CPA) further demonstrates that structured placement of noise increases the number of traces required for a successful key recovery, thus proving their effective use as hiding countermeasures, that fit seamlessly in any existing FPGA design and improving hardware security in real-world applications.

Keywords: Side-Channel Attacks, FPGA Security, AES, Noise Generators, Side-Channel Attack Countermeasures, Correlation Power Analysis (CPA)

Περίληψη

Μέχρι και σήμερα, οι επιθέσεις πλευρικού καναλιού αποτελούν μια κρίσιμη απειλή για τις κρυπτογραφικές εφαρμογές σε FPGA, εκμεταλλευόμενες διάφορα φυσικά φαινόμενα, όπως η κατανάλωση ισχύος και η δραστηριότητα δρομολόγησης, τα οποία συχνά ευθύνονται για την ακούσια διαρροή ευαίσθητων πληροφοριών κρυπτογραφικών λειτουργιών. Ως εκ τούτου, στην παρούσα διατριβή προτείνεται μια πρακτική και εργαλειοκεντρική μεθοδολογία, η οποία στοχεύει στην ενίσχυση της αντοχής πλευρικού καναλιού ενός σχεδιασμού AES που υλοποιείται σε ένα FPGA Xilinx Artix-7. Αρχικά, μια υλοποίηση πυρήνα AES σχεδιάζεται σε μια ειδική περιοχή στην πλακέτα FPGA για να περιέχει τη δρομολόγησή της, δημιουργώντας έτσι μια σταθερή βασική υλοποίηση AES, η οποία θα αποτελέσει σημείο αναφοράς για την κατασκευή δομημένων γεννητριών θορύβου. Όλες οι γεννήτριες θορύβου και οι παραλλαγές τους κατασκευάζονται από πρότυπα Look-Up Table (LUT), που χρησιμοποιούνται ως Shift Register LUTs (SRLs) και εισάγονται σε προδιατεθειμένες διαθέσιμες περιοχές μεταξύ των λειτουργιών AES μέσω RapidWright. Συνολικά έχουν παραχθεί τέσσερα διαφορετικά χωρικά μοτίβα τοποθέτησης, το καθένα με τρεις τυχαίες παραλλαγές θορύβου, που αντιστοιχούν σε δώδεκα μεμονωμένες γεννήτριες θορύβου. Στη συνέχεια, κάθε εφαρμογή προγραμματίζεται σε μια πλακέτα CW312-A35, όπου καταγράφονται χιλιάδες ίχνη ισχύος χρησιμοποιώντας ένα ChipWhisperer Husky με σκοπό μια σειρά επιθέσεων πλευρικού καναλιού. Εν τέλει, η Ανάλυση Ισχύος Συσχέτισης (Correlation Power Analysis CPA) αποδεικνύει περαιτέρω ότι η δομημένη τοποθέτηση θορύβου αυξάνει τον αριθμό των ιχνών που απαιτούνται για την επιτυχή ανάκτηση κλειδιών, αποδεικνύοντας έτσι την αποτελεσματική χρήση τους ως αντίμετρα απόκρυψης, που ταιριάζουν άψογα σε οποιοδήποτε υπάρχον σχέδιο FPGA και βελτιώνουν την ασφάλεια του υλικού σε πραγματικές εφαρμογές.

Λέξεις-κλειδιά: Επιθέσεις πλευρικού καναλιού, ασφάλεια FPGA, AES, Γεννήτριες θορύβου, Αντίμετρα επιθέσεων πλευρικού καναλιού, Ανάλυση Ισχύος Συσχέτισης (Correlation Power Analysis CPA)

Contents

Table of Figures

1. Introduction

In today's world, critical security domains increasingly require the deployment of Field-Programmable Gate Arrays (FPGAs) for cryptographic operations, with a range covering embedded payment systems and automotive control units, to Internet of Things (IoT) ecosystems. Despite software being the main target in our digital era, the aforementioned fields intrigue attackers for more sophisticated infiltrations, which require more advanced knowledge of hardware devices and exploit information leakage produced by either power consumption or electromagnetic emissions. Said attacks are not unpreventable yet can be hindered by utilizing various countermeasures against an abundance of exploitations. In bibliography, numerous studies and papers demonstrate that even optimized hardware AES designs produce exploitable leakage patterns, when implemented on mid-range devices and platforms, such as Xilinx Artix-7. Side-Channel Analysis (SCA) is particularly preferred as a convenient and straightforward attack vector by adversaries, due to the low cost and accessibility of the equipment required for it. [1],[2],[3],[4],[5]

Usually, countermeasures against SCA focus on masking or shuffling techniques provided mainly for logic level, but their various limitations in dedicated areas on boards, in latency and cost deem them less suitable for industrial deployments, where timing and resource management is inflexible. As a consequence, there is a growing attention to more lightweight alternatives, such as noise generators, routing obfuscation, non-deterministic processors and resistant logic designs, which keep the cryptographic logic intact and manage to alter their leakage distribution with the least invasive methods possible. [6],[7]

In spite of this progress, there exists a significant research gap in automatically generated noise countermeasures in the post-implementation level, which stems from prior work that focuses more on either injecting noise broadly or, in most cases, requiring considerable Register Transfer Level (RTL) refactoring. Recently, in a research by Güneysu and Moradi on "Generic Side-Channel Countermeasures for Reconfigurable Devices", a series of generic countermeasures that demonstrate the feasibility of hiding critical operations is proposed, without fully addressing neither the structures used for noise generators or routing constraints applied during the construction of them. [6]

The objective of this thesis is to design multiple noise generator structures, implement them on a baseline AES core design on post-implementation level, and evaluate whether this noise injection methodology can pose as an effective countermeasure technique. [6],[7],[8]

2. FPGA Architecture Fundamentals

Working with cryptographic designs on FPGAs inevitably means working with the fundamental structures beneath them. That being said, no matter how carefully an AES core is defined on algorithmic or RTL level, its real behavior arises once it is mapped on a physical level, particularly when slices, LUTs, wires and buffers are occupied. Hardware features along these lines are rarely designed with side-channel security in mind, therefore when these features switch, they leak slight electrical footprints which can be observed and interpreted by attackers. It is crucial that a strong theoretical foundation is established to assist in understanding how these footprints occur, why their existence matters and how they can be exploited before any countermeasure is discussed.

2.1 Xilinx 7-Series Logic Architecture

In the following section, all relevant Xilinx architecture needed to further understand some of the core elements of this thesis will be expanded into two subchapters, one explaining slices, LUTs, SRLs and SLICEM areas (low level) and a second one explaining long wires and their potential leakage.

2.1.1 Slices, LUTs, SRLs, SliceM

Xilinx 7-series FPGAs, including Artix-7 used for this thesis, typically organize their logic into Configurable Logic Blocks (CLBs), each containing two sets of similar components within a block, called slices. Slices are categorized in "SLICEL" and "SLICEM" variants, where the former are limited to combinational and registered logic, and the latter configure their LUTs as distributed RAM or shift register LUTs, as shown in Fig.1(a). [9],[10] A slice can contain four 6-input LUTs, eight flip-flops and dedicated carry/control logic, summing up to a total of 8 LUTs and 16 flip-flops per CLB. [9] Fig.1(b) depicts the

structure of each slice and the carry chain that is composed of four 2-to-1 multiplexers [9], where each multiplexer output is connected to its adjacent multiplexer. Furthermore, Figure 2 depicts slices in CLBs from Vivado's floorplanning utility. [11]

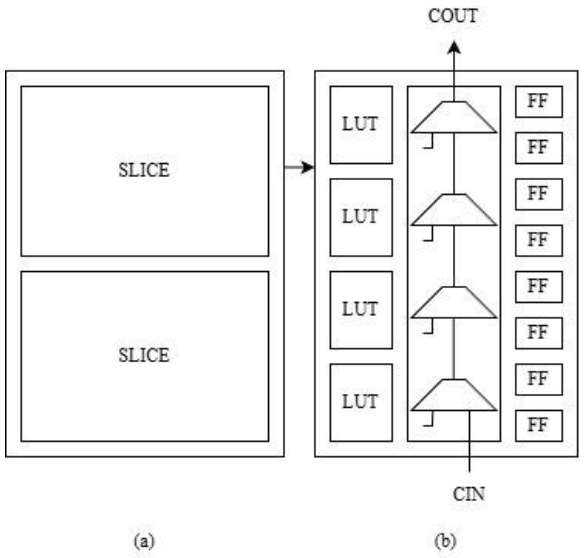


Figure 1: Logic Block Architecture. (a) CLB. (b) Slices.

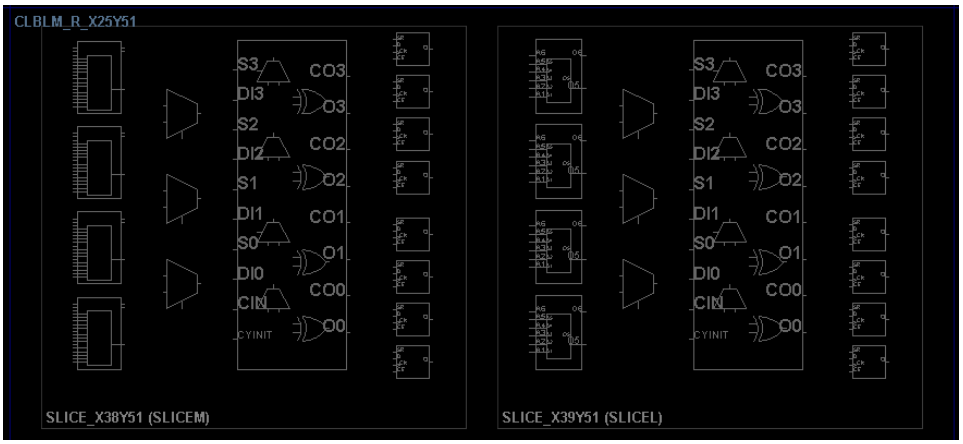


Figure 2: Vivado floorplanning with CLB containing Slices

SRLs provide a well-planned way to build high activity shift registers and ring oscillators (circuits consisting of cascaded chains of inverters arranged in a ring), as explained later in subchapter 6.4.1. Therefore, noise generators can be implemented by configuring a LUT as an SRL16 [12],[13], toggling at any frequency selected with minimal additional resources, making SRLs useful for structured noise injection around the AES core.

2.1.2 Long wire leakage

While the logical behavior of an FPGA design is defined by LUTs and slices, routing and interconnection between those logic units greatly impact the device's physical leakage characteristics. Routing in modern FPGAs is hierarchical and is composed of short, medium and long interconnection wires, each with their own leakage attributes. As discussed in "Leakier Wires: Exploiting FPGA Long Wires for Covert- and Side-Channel Attacks", long wires traverse multiple logic blocks to provide connectivity for timing-critical

or high-fanout signals. As a result, in contrast to local interconnection segments, longer wires require higher capacitance, due to their physical length. When a long wire is utilized, every transition triggers larger dynamic power events, because drivers must charge or discharge greater capacitive loads. More prominent fluctuations in the device's power distribution network or local EM emissions are then visible, due to the increased switching energy, exposing long wires in side-channel measurements. [9],[14],[15],[16]

Furthermore, according to the aforementioned Oxford report, long wires resemble "antennas" which propagate switching related disruptions across large sections of the FPGA boards, an effect that is amplified by using buffered routing switches. Said switches instigate additional paths and timing dependencies which correlate with specific signal transitions. It is also mentioned that when neighboring nets switch in correlated patterns, the electrical coupling between adjacent routing channels can further generate delay or amplitude variations, thus leaking information even when the embedded logic is properly implemented. Sensitive AES core information, such as S-Box outputs, round-key updates and state transitions are inevitably a segment impacted by the traversal of such long wires and naturally become prime targets for power attacks based on correlation. Figure 3 depicts an example of how short and long wires are routed between logic and switching blocks:

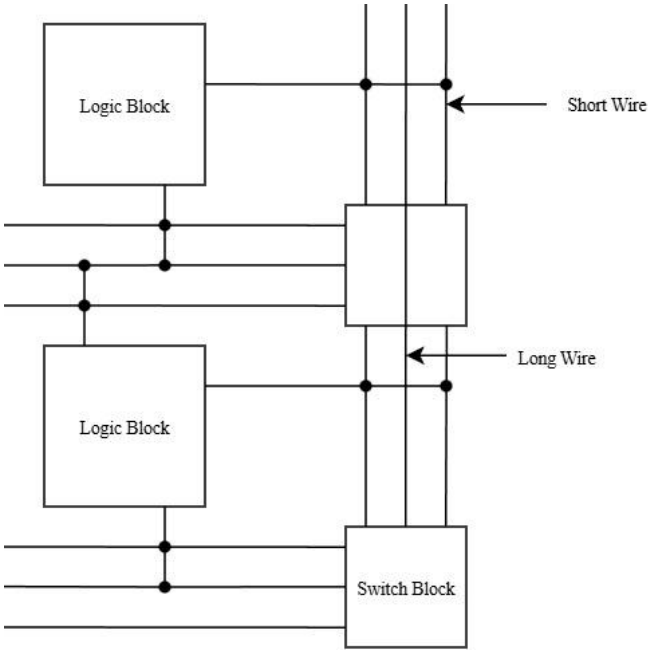


Figure 3: Routed Short and Long wires

2.2 AES Algorithm Overview

2.2.1 AES-128 Round Structure

Advanced Encryption Standard (AES) is a symmetric block cipher, which was standardized by NIST in FIPS 197 [1] and operates on 128-bit data blocks, with supported keys of 128, 192 and 256 bits. In this thesis, AES-128 is used, where the cipher executes 10 rounds to a 4x4 byte state (or 128-bits) called plaintext, until the final encryption product is finalized, which is called ciphertext (128-bits). All rounds, with the exception of the last round, apply specific operations sequentially, separated by registers, each usually within a single clock cycle. These operations are (as shown in Figure 4):

- SubBytes, a non-linear bytewise S-Box,
- ShiftRows, a sorting of state bytes
- MixColumns, a linear diffusion over Galois Field
- AddRoundKey, XOR with the round key

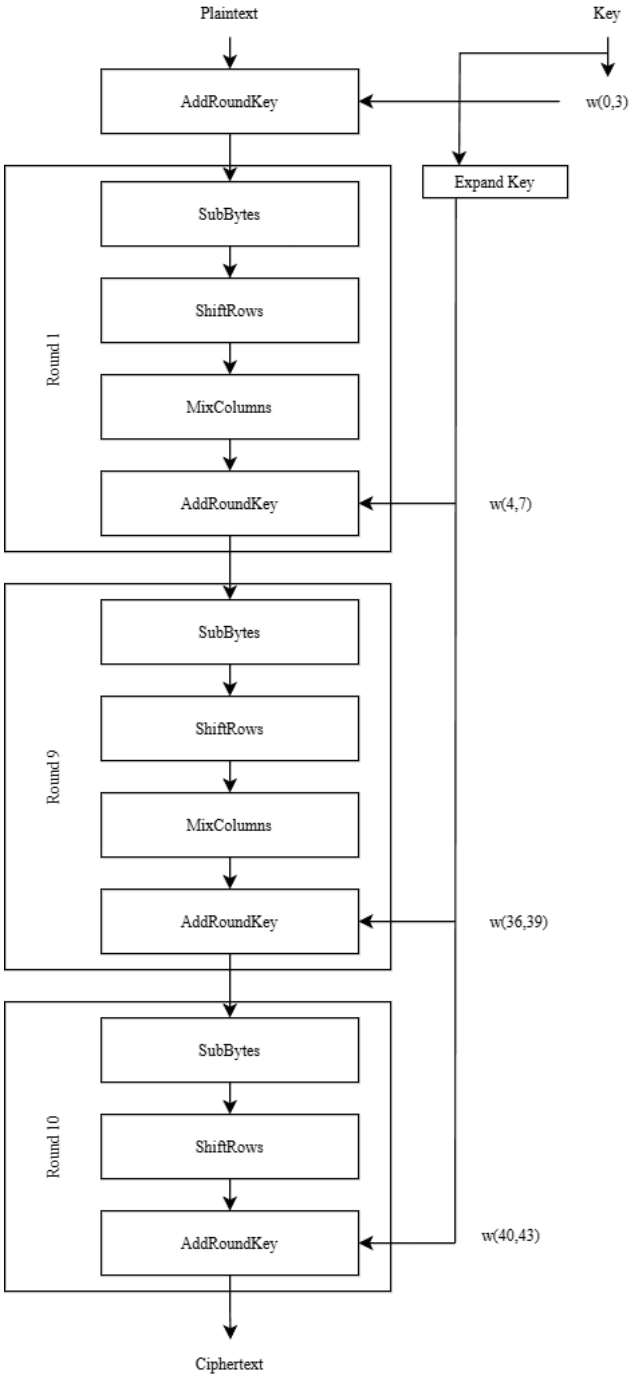


Figure 4: AES Rounds and Key Expansion

The key difference of last round is that MixColumns is omitted. More analytically, the SubBytes step applies and 8 to 8 bit substitution, called AES S-Box, which derives from a multiplicative inversion in GF(2^8) (Galois Field), followed by an affine transformation. Its purpose is to mix key and data bits in the cipher. Consequently, such an operation is more likely to be targeted by attackers during a Side-Channel attack, simply by making key hypothesis on single bytes and predicting S-Box inputs and outputs after correlating them with a specified and pre-measured leakage. FPGA implementations typically include S-Box designs that are either LUT-based read-only memory (ROMs) or logic networks, which in both cases are dominant in the key dependent leakage profile due to their internal toggling and routing.

For this thesis, a slight modification in the AES core implementation has been made. Instead of a typical AES core, "Google Project Vault" AES was utilized [17], whose RTL is publicly available. Google Vault AES is a secure computing environment which was implemented on a microSD form factor card and embeds its own AES-128 implementation as part of a smaller SoC, which protects credentials and other sensitive operations on various untrusted hosts. This particular design has also been adopted by NewAE as their reference AES implementation on CW305 and CW312/CW313 FPGA platforms, due to its iterative round-based architecture with separate key and data paths, in addition to its lack of side-channel countermeasures, making it a realistic and intentionally unprotected baseline for any power analysis experiment. Additionally, previous side-channel work on CW305 indicates that CPA is able to recover the full-length key of such a cryptographic algorithm, with a smaller number of traces when unprotected.

2.2.2 Key Expansion Implementation

Another important module of Project Vault AES worth discussing is key expansion, which is located in aes_ks.v and its role is to expand the original cipher key (master key) into a sequence of round keys. Those keys will in turn be used in the AddRoundKey step of each round by the encryption core. An initial cipher key (key_i) is firstly accepted by key schedule, as well as a selected key size (size_i), control signals for loading (load_i) and enabling the expansion process (en_i). The output is in this case a 128-bit round key (ks_o), which will further be consumed by the encryption datapath at each cycle.

Whenever a new key is loaded, the key expansion module initializes an existing internal state, setting the round constant (Rcon) to 1. For this purpose, a Finite State Machine (FSM) is executed by the module, which implements the AES key expansion rules for all variants, including 10 rounds for AES-128, 12 rounds for AES-192 and 14 rounds for AES-256. Moreover, standard AES transformations are used to derive new 32-bit words from previous round keys, such as RotWord, SubWord, Rcon and XOR chaining. SubWord also applies the AES S-Box, which, as previously mentioned, is using a Galois Field based S-Box via the SBOX_GF flag:

- RotWord: byte rotation of the previous key word
- SubWord: applies the AES S-Box on every byte
- Rcon: round constant XOR applied to the first word
- XOR chaining: derives the remaining words from previous ones

In the case of AES-128, key schedule expands the master key into 44 words consisting of 32 bits each (W_0 to W_{43}), out of which, the first four words (0 to 3) are taken directly from the key input, while every subsequent word is derived from the previous ones, based on the following recurrence:

$$W_i = \begin{cases} W_{i-4} \oplus SubWord\big(RotWord(W_{i-1})\big) \oplus Rcon_{\frac{i}{4}}, & if\ i\ mod\ 4 = 0 \\ W_{i-4} \oplus W_{i-1}, \ otherwise \end{cases}$$

Simply, this recurrence means that for every four words the key schedule performs SubWord and RotWord, then mixes in the round constant Rcon and finally performs XOR on the result with the word, four positions earlier. The remaining three words of each round undergo another XOR operation. The first 16 words are generated and each group of four creates one 128-bit round key, which is then used in AddRoundKey for rounds 1 to 10. Figure 5 shows in detail how key expansion works:

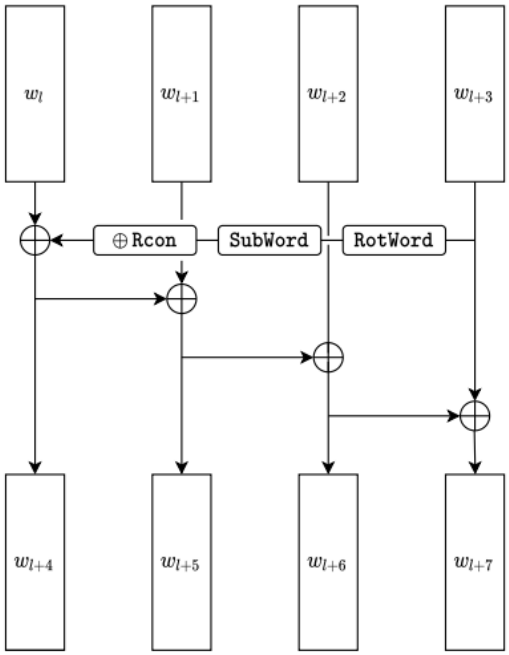


Figure 5: Key Expansion steps

The importance of the key expansion lies in one single fact: power measurements are directly affected by its existence and transitions during the creation of subkeys. Each AES round uses one 128-bit subkey (round key) during the AddRoundKey step, which XORs the subkey with the internal state. Therefore, mixing key and data directly affects the next S-Box input or ciphertext and creates visible power consumption leakage. Prior FPGA SCA research has shown that round keys often contribute to this leakage due to switching activity on state registers, key schedule output registers and long interconnection wires which carry key-dependent signals. For this thesis, it was observed that the presence of such leakage was prominent during last-round transitions between state_9 and state_10 (ciphertext), which further indicated that the most effective leakage model to be utilized was last-round Hamming Distance.

2.3 Side-Channel Leakage Models and Correlation Power Analysis

2.3.1 Dynamic Power and Leakage Modeling

CMOS technology, as well as logic and routing of FPGAs produce power during active computations, which is called dynamic power and is caused by charging and discharging of internal capacitances. An approximation used in bibliography is:

$$P_{dyn} = \alpha \cdot C_{load} \cdot V^2 \cdot f$$

where α is the switching activity factor, meaning the average number of transitions per clock cycle, C_{load} is the capacitance from a gate or interconnect net, V is the supply voltage and f is the clock frequency. Both α and any active capacitances depend on the bits of the internal state and the key, which in turn create data-dependent power signatures, especially during processing of secret data. Such power signatures are observed with either an external oscilloscope or a side-channel capture platform, such as ChipWhisperer's Husky.

Commonly, attackers might not have access to transistor-level details of their target FPGAs, hence they rely on leakage models which approximate the relationship between measured power and data. If the model is well-fitted to the actual hardware behavior, then the attack has increased chance of success. Prior work has shown that oftentimes changing the statistical measure of an attack is less effective than improving the leakage model. [18]

2.3.2 Hamming Weight and Hamming Distance Models

In power side-channel analysis attacks, two leakage models are widely preferred, Hamming Weight (HW) and Hamming Distance (HD) [4],[21].

- Hamming Weight (HW) model: assumes that the power leakage at a specific moment is roughly proportional to the number of 1 bits inside an internal value x:

$$L \approx a \cdot HW(x) + b + noise$$

where a is a scaling factor, meaning how strongly a device's power reacts to each 1 bit, b is a fixed offset, which is the baseline power the device always consumes and noise is everything that cannot be modeled, such as environmental noise, jitter, unrelated events and other complications. HW is typically applied to microcontrollers or read-memory operations, due to the dynamic power caused by charging and discharging of word and bit lines based on their stored value.

- Hamming Distance (HD) model: assumes that the leakage is proportional to the number of bit transitions between two successive values, an old one (x_{old}) and a new one (x_{new}):

$$L \approx a \cdot HD(x_{old}, x_{new}) + b + noise = a \cdot HW\ (x_{old} \oplus x_{new}) + b + noise$$

HD is typically used for pipelined hardware and FPGAs, where dynamic power is caused by register or interconnection switching.

There has been a general confirmation in prior studies that HD-based models are achieving improved performance in power measurements in hardware AES implementations. The explanation for this observation is that flip-flops and long wires draw current when their outputs toggle. For instance, Repka et al, proved that simulating power by HD yielded closer matches to the measured FPGA consumption than HW in an AES multiplier core. Similarly, Wang et al. noted that HD is more suitable for hardware, where there is a clear dominance of dynamic power.

It is worth mentioning that real devices are unpredictable and neither of the models explained previously are able to be perfectly followed in every case. However, HW/HD models are computationally cheap and accurate enough to penetrate many implementations. Furthermore, these models are useful evaluation tools for various countermeasures, especially for cases where countermeasures reduce correlation under a HW/HD model, which is considered a meaningful improvement in side-channel resistance methods.

2.3.3 Correlation Power Analysis (CPA)

Power traces from cryptographic devices can be statistically processed in order to recover secret keys, as introduced by Kocher, Jaffe and Jun in their first demonstration for Differential Power Analysis (DPA), where traces were partitioned according to a predicted bit and a computation of differences of means. Later on, Brier, Clavier and Olivier proposed a reformulation of this idea, using the Pearson correlation coefficient [2] between leakage model and the measured traces, thus proposing an alternative analysis called Correlation Power Analysis (CPA). CPA framework is standard in the modern hardware security era, due to generally fitting to multi-bit models like HW/HD, as well as measuring how effectively a key hypothesis explains the observed power consumption.

An attacker can, for a given set of N traces $L_i(t)$, where i are the encryptions and t is the time sample, select and intermediate value $v_i(k)$ which depends on either the plaintext or the ciphertext and a key hypothesis k. This is usually the output of S-Box in the first or last round of AES. The attacker computes a model value $M_i(k)$, for example either $HW(v_i(k))$ or HD ($v_i^{old}(k)$,$v_i^{new}(k)$), evaluating the Pearson correlation for every time sample t:

$$\rho(k,t) = corr(L_i(t), M_i(k))$$

Higher absolute correlation at time indices where the target operation occurs indicates that the key hypothesis is most likely correct and is furthermore producing a correlation peak in the ρ(k,t) map, which is also the main signal to be exploited in this thesis during the comparison of the baseline AES implementation and the proposed noise generators.

2.3.4 Last-Round Hamming Distance leakage model

As previously explained, the AES core used in this thesis (Project Vault AES) is integrated in the CW305/CW312 target and it is an iterative hardware design with dedicated output registers. Such a design is prominently leaking at register level, specifically in the final round, where the internal state is related to the ciphertext in a direct way. Therefore, in this thesis, a last-round HD model is utilized. For each key-byte hypothesis the internal state before and after the last SubBytes/AddRoundKey operation is reconstructed from the ciphertext, via the inverse S-Box. The HD between successive encryptions is used as the hypothetical leakage, which mirrors the number of bit flips occurring on the final state registers and on the long wires that feed them, resembling the distinguishable 0 to 1 and 1 to 0 transitions. [19],[20]

In practice, this model is incorporated into a custom CPA script, built on top of ChipWhisperer's analysis framework, and computes the inverse S-Box of ciphertext_byte $\oplus$ key_guess for multiple consecutive traces and afterwards applies a precomputed HD lookup which produces a leakage prediction per trace and per key guess. Subsequently, the Pearson correlation is computed between those predictions and the time sample from the measured traces.

2.4 Side-Channel Countermeasures and Related Work

Countermeasures against side-channel attacks can be grouped in three major categories: algorithmic, implementation-level and physical/hiding techniques [21],[22]:

1. Masking (algorithmic / data randomization): low-cost mitigation methods, where sensitive variables are split into multiple partitions using random numbers, therefore obscuring the correlation between physical leakage and secret data.
2. Hiding via noise and timing desynchronization: reduced signal-to-noise ratio instead of eliminating leakage. Usually involve methods such as adding dummy operations, inserting random delays, shuffling operation order, adding switching noise to make the key-dependent components indistinguishable. Approachable techniques such as hiding tend to be more attractive in industry, with minimal interferences to the RTL, however their effectiveness depends on their intensity and noise strength.
3. Randomization of LUT / memory values: Recent studies suggest that randomized LUT contents and various primitives for S-Boxes can be implemented as countermeasures, specifically targeting (but not limited to) Xilinx devices. Due to the unpredictable physical effects such methods might have, which can in turn be influenced by either subtle leakage introduced by LUTs themselves or by their implementation challenges, such randomization methods are still not fully proven to be effective and further studies are needed.
4. Dynamic partial reconfiguration (DRP): a method proposed for FPGAs to hinder DPA/CPA. Such algorithms reduce any signals related to cryptographic keys, by changing hardware implementations while they are actively operating.

3. Hardware and Software Setup

In this chapter, the basic tools and setup will be discussed, as well as the design workflow that is generated. At a high level, Vivado is used to implement the AES design and to generate design checkpoints (DCPs), which will afterwords be used by RapidWright to insert noise generators in specified available areas. This new design will be programmed on a CW312-A35 target and power traces from the bitstream are collected using a ChipWhisperer Husky. All scripts and relevant programming have been developed using IntelliJ for Java and Visual Studio Code for Python.

3.1 RapidWright

Before introducing the specific steps of the post-implementation flow of this thesis, it is important to highlight why a tool such as RapidWright [23],[24],[25],[26],[27] is convenient, combined with Vivado. While Vivado excels at synthesis, placement and routing, its flexibility for modifying already implemented designs is low. When it comes to adding strategic structures without disturbing the baseline AES implementation, a fine-grained access to the physical layout is needed, meaning access to slices, BELs, routing wires and pins. This access is not granted at post-route implementation using Vivado alone, therefore AMD/Xilinx Research Labs have created an open-source Java framework called RapidWright

(RW) in order to fill this gap. Any programmable interface is exposed as a DCP, which enables RW to add new logic and manipulate placement, or adjust routing in precise and reproducible ways.

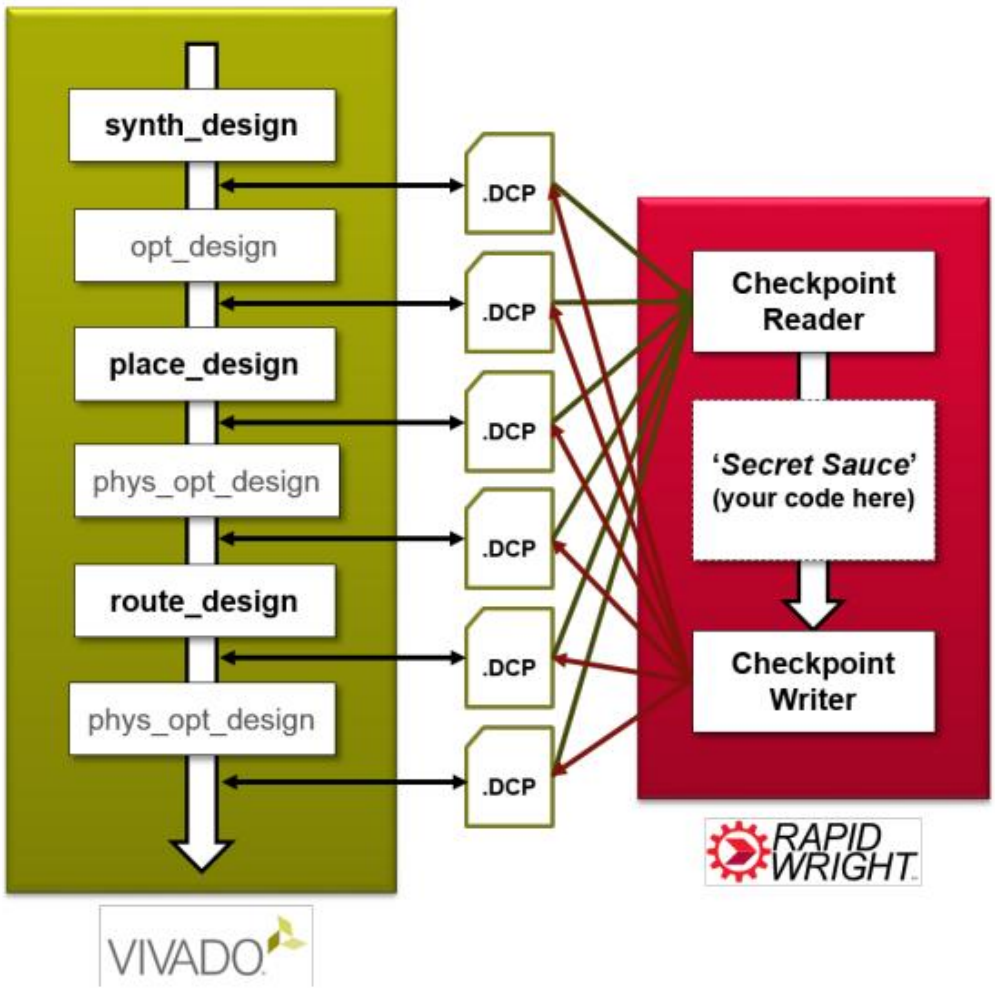


Figure 6: RapidWright Flow

For the context of this study, the flow is as follows:

- Vivado synthesizes and implements the baseline AES inside a pre-allocated constrained Pblock, and a DCP using Transaction Control Language (TCL) commands is generated, keeping the design at the current implementation stage intact.
- Another short TCL script is responsible for extracting all available (empty) SLICEM sites from all Pblocks, which were previously created to occupy space for noise generators, and stores them in a text file.
- The generated DCP is loaded by a custom-made script to RW, which reads the exported SLICEM site list and instantiates LUTs as SRLs to be used as noise generators in these empty locations. Afterwards, RW connects LUTs with local routing resources, without interfering with the AES implementation.
- Finally, the exported DCP is re-opened in Vivado for a final routing check and a bitstream generation, in order to be loaded onto the FPGA.

3.2 ChipWhisperer Husky

ChipWhisperer (CW) played an important role in capturing traces for the side-channel analysis studied in this thesis. It is widely used as an open-source platform for side-channel analysis and fault injection on

embedded or hardware targets. Husky is part of CWs ecosystem, primarily designed as a high-end capturing device [28],[29],[30],[31], with compact structure and includes high-speed ADC, programmable clocks, low-jitter trigger for input or output and additional digital I/O for glitching. Furthermore, Husky offers high sampling rates and flexibility in triggering, a trait that is particularly useful in experiments which require thousands of traces with consistent timing.

For this particular experiment, Husky was utilized as a capturing device, with the following properties:

- High resolution ADC: measures immediate voltage drops across shunt resistors on the CW312-A35 board.
- Configurable sampling clock and trigger alignment mechanism: utilizing the SS2 trigger signal from the FPGA. SS2 will further be explained in another chapter.
- Interface controlled by Python through CW's API: allows configuration, capturing and storing of traces for each noise variant produced.

3.3 ChipWhisperer CW313 FGPA Board and CW312T-XC7A35 target

The AES design (baseline and protected) runs on a "UFO" FPGA CW312T-XC7A35 board, created by NewAE, featuring Xilinx Artix-7 and it consists of the Xilinx Artix-7 CW312T connected with a CW312 edge [32],[33],[34],[35]. The design uses the SimpleSerial v2 (SS2) protocol as interface between the host and the AES core and it is a text based serial protocol, commonly used by ChipWhisperer. Primarily, the host sends plaintexts or commands over a link resembling UART and the FPGA subsequently returns a ciphertext, after processing them.

The described setup, along with Husky, provides a stable power supply for power measurements, a trigger signal to be shared between the FPGA and Husky for aligned traces, and an interface which makes experiments on other AES cores or countermeasures fairly reproducible.

3.4 Development Environment

Software for all stages of the workflow was developed utilizing various commonly used tools, such as:

- Vivado 2023.2: used for synthesis, implementation and bitstream generation. TCL scripts were additionally used to create Pblocks and retrieve empty slices, ready to be used. Other relevant operations involve the application of routing containment and placement exclusion from specified areas during the implementation level, as well as exporting DCP files.
- IntelliJ IDEA: used as a Java development environment [36] for RapidWright 2024.1.3, specifically for a custom Java script, which builds the SRL noise ring generators.
- Python with ChipWhisperer 6.0.0 and Visual Studio Code: used to control Husky, program the FPGA, capture traces and perform CPA attacks. Custom scripts were also developed for each of the aforementioned stages, while also including saving all traces, plaintexts and ciphertext of each individual AES implementation.

4. Baseline AES Implementation

For the purpose of any experiment, a baseline or reference design is needed, in order to be compared with any subsequent results. Therefore, in this chapter, the baseline implementation of AES will be discussed, as well as the traces captured and the CPA attack performed on this reference design.

4.1 Pblock Floor Planning

During the implementation of the AES core, the design is optimized, and its position can vary, due to those optimizations in routing and placement. It should be reminded that the countermeasures designed for this thesis are area-sensitive, which practically means that the generators need to be placed near the areas to be protected, such as state registers or long wires. Therefore, it is important to create a dedicated area on the floorplanning, in order to have a fixed and specific placement of the design, which will then allow for the countermeasures to be placed at the appropriate spots. [37], [38]

In this case, the following Pblock area has been preserved to contain all of the AES core, through the project's XDC file, using the declaration "create_pblock" and the name of the pblock to be created. The size of the pblock can further be customized by "resize_pblock", adding the name of the pblock to be resized, as well as the slices needed to be included, as shown in the figure 7 below:

```
create_pblock aes_pblock
resize_pblock [get_pblocks aes_pblock] -add {SLICE_X12Y0:SLICE_X65Y49}
resize_pblock [get_pblocks aes_pblock] -add {DSP48_X1Y0:DSP48_X1Y19}
resize_pblock [get_pblocks aes_pblock] -add {RAMB18_X1Y0:RAMB18_X2Y19}
resize_pblock [get_pblocks aes_pblock] -add {RAMB36_X1Y0:RAMB36_X2Y9}
set_property CONTAIN_ROUTING 1 [get_pblocks aes_pblock]
```

Figure 7: Pblock declaration for AES baseline design

The region that is finally created spans a moderate portion of the device, where Vivado has enough freedom to meet timing restrictions, as well as still implement AES spaciously enough to fit in that particular area. Moreover, routing is contained inside the pblock using "set_property CONTAIN_ROUTING", a command that forces Vivado to route all AES nets only using wires inside that particular region, preventing any unintentional long-wire usage across the chip. Figure 8 shows the final result, containing AES core inside the designated pblock:

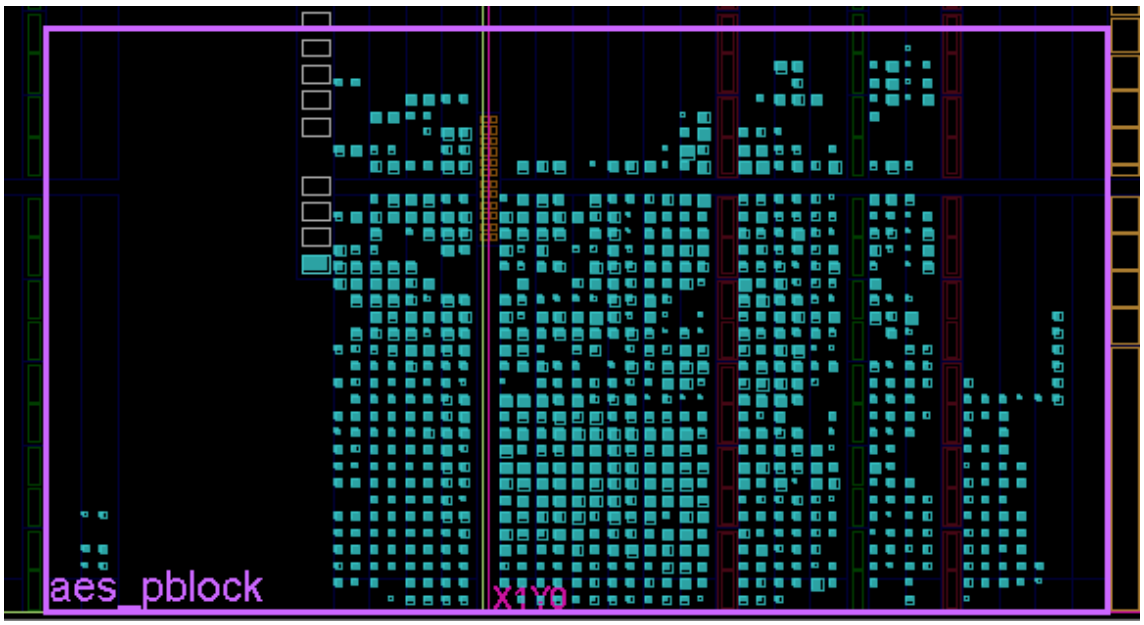


Figure 8: Baseline AES in Pblock implementation

Lastly, a bitstream is generated using "write_bitstream" in the TCL command line, in order to be programmed on the target board via the ChipWhisperer interface. At this point, a DCP file is not yet needed, as the workflow for the baseline AES implementation is directly following Vivado's workflow. [39]

4.2 Trace Capture

Trace capturing is fully automated using a custom Python script, which configures CW Husky device, programs the CW312T-XC7A35 and stores power traces with AES encryptions, which will later be used for CPA. For baseline AES, a total of 2000 power traces were approximately captured and used, using a fixed key (FC * 16), along with fully randomized plaintexts. [40],[41]

Key components of the capture chain that will be further discussed are:

- ADC sampling rate: Analog-to-digital converter, which determines the number of points per microsecond needed to be collected by Husky and is typically set to 100-250 MS/s, ensuring that the final AES round is clearly visible in the trace.

- Gain (scope.adc.advanced.gain): amplifies the analog signal. When set too high clipping might appear, whereas setting it too low might reduce signal-to-noise ratio (SNR).
- XADC / Power measurement channel: Husky has a dedicated power measurement channel implemented, which is routed to the shunt signal on the CW312 board.
- Trigger (scope.trigger): a digital edge generated by the FPGA AES wrapper, usually from SS2's "processing start" or "done" signal, which ensures that all traces are aligned to the beginning of the encryption, thus properly aligning them and mitigating any jitter produced during the capturing.

The following code snippet describes the scope and clock configuration used for Husky:

```
scope = cw.scope()
scope.gain.db = 45            # analog gain
scope.clock.clkgen_freq = 7370000  # 7.37 MHz FPGA clock
(CW313/CW312T-A35)
scope.io.hs2 = "clkgen"
scope.clock.adc_mul = 4       # ADC runs at 4× clkgen
scope.adc.samples = num_samples
scope.adc.offset = 0
scope.adc.basic_mode = "rising_edge"
scope.trigger.triggers = "tio4"
```

Due to restrictions from CW Husky, the setup is clocked at 7.37 MHz and Husky's ADC is set to sample at 4× of this frequency, producing several samples per AES clock cycle. Analog gain is also set at 45 dB, as suggested by CW's documentation, which is a value that provides a better resolution and avoids saturation. Trigger source tio4 is a digital I/O pin connected to the AES start signal, where ADC is armed before each individual encryption. Sampling then begins on the rising edge of this specified trigger, in order to capture aligned traces.

Afterwards, a custom key (FC*16) is setup and loaded on the FGPA. A function named "setup_target()" is responsible for configuring the SS2 platform and I/O output. Once it is programmed, the script subsequently loads the fixed key directly in the FPGA registers:

```
key = cw.bytearray([0xFC] * 16)  # default key: FC FC ... FC
target.fpga_write(0x0a, bytes(key)[::-1])
target.fpga_write(0x0c, [1])     # key valid
target.fpga_write(0x0f, [1])     # start key expansion
```

- 0A is the AES key input register (write_only), where the 128-bit AES key is written into the internal key buffer
- 0C is the key valid / key load confirm (or bit flag), where the value 1 indicates that the key written to 0a is valid and capturing can begin, otherwise AES core will continue using the previously written key in memory.
- 0F is the start key expansion / key schedule trigger register, which indicates that the AES-128 key expansion process can begin. It is crucial that this step is added before any encryption command, in order to complete the round key process, therefore producing correct encryption output.

Furthermore, "capture_traces()" collects the number of traces specified by the user, where each corresponds to an AES encryption, using the same key, as previously loaded, and a new random plaintext. Each trace is captured by collecting samples of datapoints, which depict the length of the trace in time. It was estimated after testing that each trace needed approximately 60 datapoints to be fully depicted, including both some pre-encryption noise (right before triggering starts) and after encryption noise:

```
scope.adc.samples = num_points

for i in range(num_traces):
    pt = np.random.randint(0, 256, size=16, dtype=np.uint8)
    plaintexts[i] = pt

    scope.arm()                        # arm ADC before starting
AES
    target.fpga_write(0x06, pt[::-1])  # write plaintext (reversed)
```

```
    target.fpga_write(0x0e, [1])        # start core
    target.fpga_write(0x05, [1])        # trigger encrypt

    target.fpga_write(0x09, [1])        # request ciphertext
    ct = target.fpga_read(0x09, 16)
    ciphertexts[i] = ct[::-1]

    traces[i, :] = scope.get_last_trace()[:num_points]
```

Some key notes for this code snippet:

- Plaintexts (pt) are fully random for each trace (np.random.randint(0, 256, size=16)), an essential step for CPA, in order to ensure that key-dependent leakage is diverse across all traces.
- Scope is always armed first, then a plaintext is written and finally the AES start registers are set. These steps trigger both the encryption process and Husky capturing via tio4.
- Bytes in plaintexts are written in reverse byte order ([::-1]), due to the little-endian interface on the FPGA. To be further explained, little-endian is a storing or transmitting method, specified for multi-byte data, where the least significant byte (LSB) comes first in memory on on wire.

The final step of this capturing process is storing data consistently in the project. All data, plaintexts, ciphertexts and traces, are stored in .npy files, which can later be loaded directly into the CPA script without any additional parsing. To ensure that no communication, encryption or indexing errors occurred during the capturing, random subsets of saved pt-ct pairs are checked, using "verify_saved_data()", which re-calculates the encryption results utilizing a software AES implementation, using the same key. Therefore, data integrity is ensured, in order to retrieve reliable CPA results.

A visualization of 100 traces out of 2000 that were initially retrieved is finally generated, which clearly depicts all rounds of AES, where the last round is roughly located around sample (datapoint) number 50 out of 60 initialized:

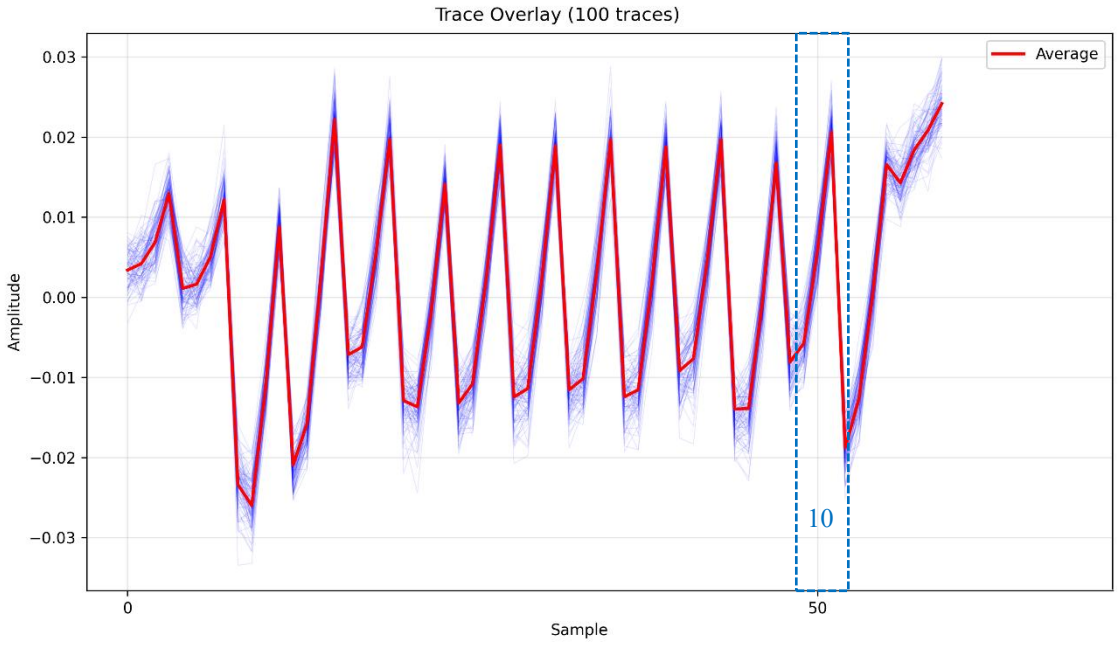


Figure 9: Baseline AES traces and visible rounds

4.3 Baseline CPA Attack

As previously mentioned in Chapter 2, in order to evaluate the side-channel leakage of the unprotected AES implementation, a CPA attack is applied, utilizing the last-round HD leakage model [42]. ChipWhisperer provides several built-in leakage models for AES, spanning from first round HW, to last-round HD,

pipelined variants and intermediate HD models. In this thesis, the leakage model implemented, as previously described, is constructed based on the "LastRoundHD". For this purpose, a custom script was created, that includes a ciphertext based last round attack, which simply means that the internal state before last AddRoundKey is reconstructed, using the AES inverse S-Box. The original encryption computation of last round is:

$$S(y_i) = C_i \oplus k_i$$

where S is S-Box, y_i is the internal byte value before the last S-Box, C_i is the ciphertext byte and k_i is the true last-round byte. The inversed equation for a key hypothesis is:

$$x_i = S^{-1}(C_i \oplus k'_i)$$

where x_i is the reconstructed internal state byte, S^{-1} is the S-Box inversion, C_i is the ciphertext byte and k'_i is the last-round key byte guess.

The corresponding code lines for this inversion are:

```
inv_prev = INV_SBOX[ct_prev ^ key_guess]
inv_curr = INV_SBOX[ct_curr ^ key_guess]
```

For the leakage model used, HD is computed:

```
hd_val = HD[inv_prev, inv_curr]
```

which predicts the number of toggles in the AES output registers, between two consecutive encryptions.

Afterwards, Pearson's correlation coefficient [43] is computed between the predicted leakage (leakage_centered) and the power measured at a specific sample (trace_centered), by dividing their covariance by the product of their standard deviations. Correlations[i] is the result, which indicates how strong of a correlation this specific time sample has with the leakage model for the given key:

```
cov = np.mean(leakage_centered * trace_centered)
correlations[i] = np.abs(cov / (std_hyp * std_trace))
```

Furthermore, the master key recovery depends on determining the round-10 key bytes from correlation peaks, using the line:

```
key_guess[byte_idx] = np.argmax(byte_corrs)
```

and then the script applies AES key schedule inversion with the following process:

```
master_key = key_schedule_rounds(inv_key_guess, rounds=N_ROUNDS,
inverse=True)
```

After experimenting with the number of traces needed to be captured to recover the full AES master key, it was noted that in 2000 traces the recovery ratio was 7/16 bytes, whereas at 2500 the same ratio was 16/16.

```
Recovered Key: A5 A0 A5 A2 A5 FC A5 FC FC FC A5 A2 FC FC A5 FC
Expected Key:  FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC

[*] Verification: 7/16 bytes correct (failed)

Per-byte correlations:
  Byte  0: 0xA5  (corr=0.1184) X
  Byte  1: 0xA0  (corr=0.1152) X
  Byte  2: 0xA5  (corr=0.1400) X
  Byte  3: 0xA2  (corr=0.1003) X
  Byte  4: 0xA5  (corr=0.1205) X
  Byte  5: 0xFC  (corr=0.1306) ✓
  Byte  6: 0xA5  (corr=0.1802) X
  Byte  7: 0xFC  (corr=0.1357) ✓
  Byte  8: 0xFC  (corr=0.1673) ✓
  Byte  9: 0xFC  (corr=0.0991) ✓
  Byte 10: 0xA5  (corr=0.1322) X
  Byte 11: 0xA2  (corr=0.1397) X
  Byte 12: 0xFC  (corr=0.1271) ✓
  Byte 13: 0xFC  (corr=0.1109) ✓
  Byte 14: 0xA5  (corr=0.1383) X
  Byte 15: 0xFC  (corr=0.1290) ✓
```

(a)

```
Recovered Key: FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC
Expected Key:  FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC FC

[*] Verification: 16/16 bytes correct ✓ PERFECT!

Per-byte correlations:
  Byte  0: 0xFC  (corr=0.1216) ✓
  Byte  1: 0xFC  (corr=0.1136) ✓
  Byte  2: 0xFC  (corr=0.1514) ✓
  Byte  3: 0xFC  (corr=0.0921) ✓
  Byte  4: 0xFC  (corr=0.1349) ✓
  Byte  5: 0xFC  (corr=0.1297) ✓
  Byte  6: 0xFC  (corr=0.1673) ✓
  Byte  7: 0xFC  (corr=0.1293) ✓
  Byte  8: 0xFC  (corr=0.1687) ✓
  Byte  9: 0xFC  (corr=0.0973) ✓
  Byte 10: 0xFC  (corr=0.1335) ✓
  Byte 11: 0xFC  (corr=0.1277) ✓
  Byte 12: 0xFC  (corr=0.1343) ✓
  Byte 13: 0xFC  (corr=0.1142) ✓
  Byte 14: 0xFC  (corr=0.1219) ✓
  Byte 15: 0xFC  (corr=0.1239) ✓
```

(b)

Figure 10: Key recovery after CPA attack. (a) 2000 traces. (b) 2500 traces.

Moreover, a correlation and key guess plot was visualized, which clearly depicts the correlation "spikes" for all 16 bytes individually:

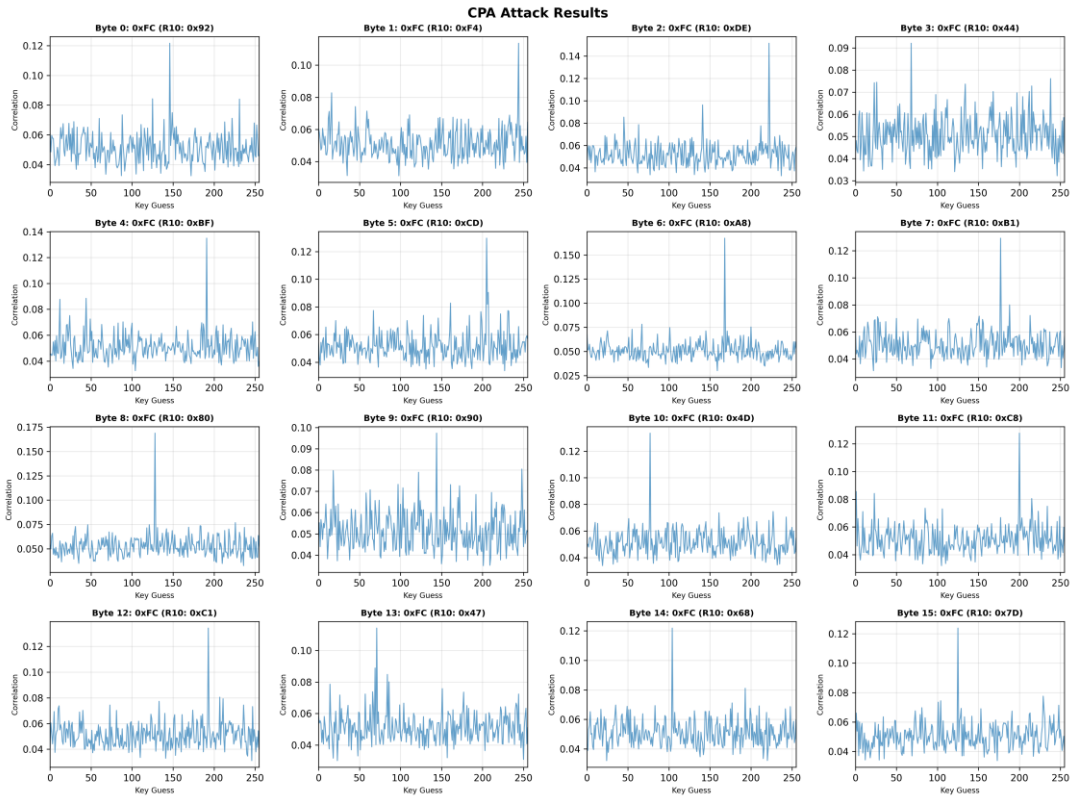


Figure 11: Correlation - Key Guess CPA Attack result plots per byte

5. Noise Generators Structure

Noise generators created for this thesis have a particular structure, area of implementation, spatial arrangement and logic contained. The goal of this chapter is to explain the logic behind each pattern's placement and how it possibly protects sensitive operations, long wires and registers on the FPGA.

5.1 Noise regions

To control where noise can be added [43],[44],[45],[46] during the floorplanning level, noise regions should firstly be defined as empty Pblocks in Vivado. These areas could surround or intersect routing used by the AES core and are constrained on the XDC using the "EXCLUDE_PLACEMENT", so that the baseline design leaves them empty and all other logic is built around them. It should be noted that Plocks strictly include SLICEM sites, so that LUTs can be utilized as SRL primitives. Afterwards, as previously mentioned, a Vivado TCL script scans the Pblocks in the baseline design and outputs a list of eligible SLICEM sites, which will later be utilized by RW as an "allowed map" for noise placement on the DCP.

An example of how noise generator placeholder areas are implemented is shown in Figure 12:

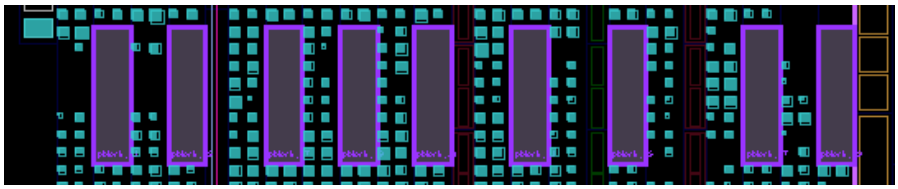


Figure 12: Noise generator placeholders

5.2 Placement patterns

When it comes to creating placement patterns for noise generators, countless options exist and the combinations are as many as the available SLICEM slices on the board. In this thesis, four different spatial patterns are explored, which were produced and placed experimentally near various points of interest [47],[48].

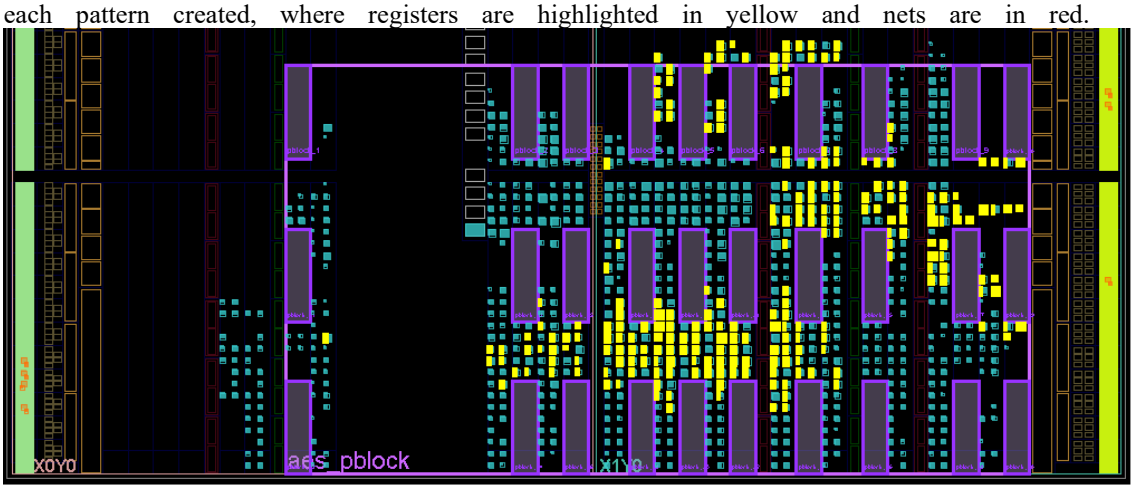
- Pattern 1: 30 empty areas, each 8 slices long with a total of 960 ready to use LUTs.
- Pattern 2: 5 empty areas, each 34 slices long with a total of 680 ready to use LUTs.
- Pattern 3: 20 empty areas, each 25 slices long with a total of 2000 ready to use LUTs
- Pattern 4: 19 empty areas, each 16 slices long with a total of 1216 ready to use LUTs

Table 1: Critical Registers & Nets

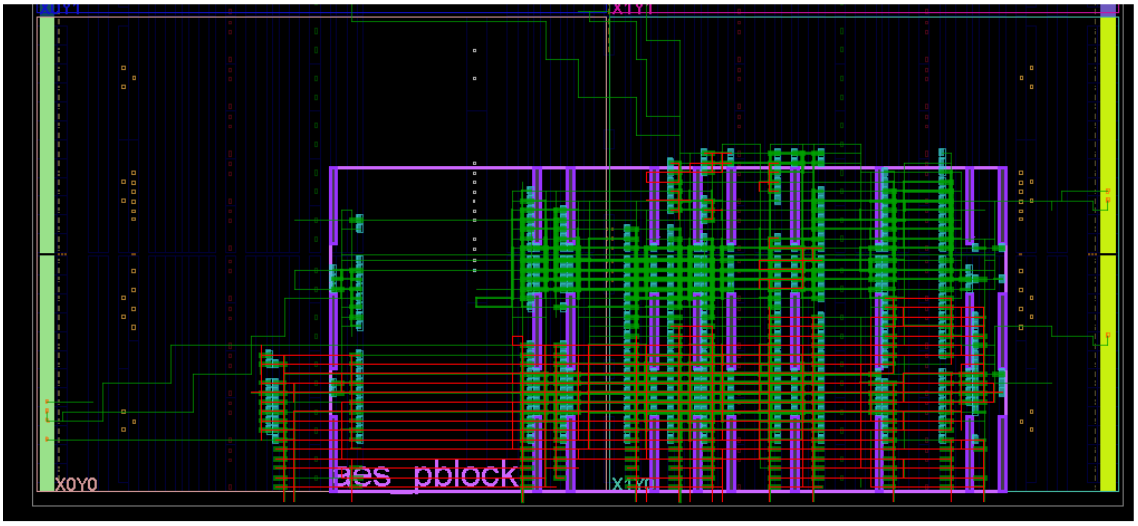
Registers	Key & Data	reg_crypt_key reg_crypt_textin reg_crypt_cipherout reg_crypt_textout
	Core AES State	state_reg round_reg fsm_reg sbb_i_reg sbb_o_reg
	Key Schedule	ks_mem ks_reg round_max_reg Rcon_reg size_r_reg
	Pipeline & Control	go_pipe_reg busy_pipe_reg
Nets	Key Schedule	ks_val ks_next ks_rotword ks_subword
	State Transformation	state_new sbox_out mixcolumns_out shiftrows_out addroundkey_out
	Control Signals	aes_round_start aes_round_done aes_final_round aes_round_trigger
	Data Path	pt_xtime1/2/3 sbox_in sbox_out_delayed
	Key Expansion	key_expand_round key_schedule_wire rcon_wire
	S-Box Related	sbox_internal sbox_out_wire sbox_mul2 / sbox_mul3
	Power Analysis Critical	hamming_weight parity_bit key_byte_xor
	Synchronization	crypt_key_sync crypt_textin_sync crypt_cipherout_sync

	Control & Status	key_schedule_enable mixcolumns_enable subbytes_enable shiftrows_enable

With this reference in mind, the following 4 figures (13,14,15,16) depict the placed Pblocks for each pattern created, where registers are highlighted in yellow and nets are in red.

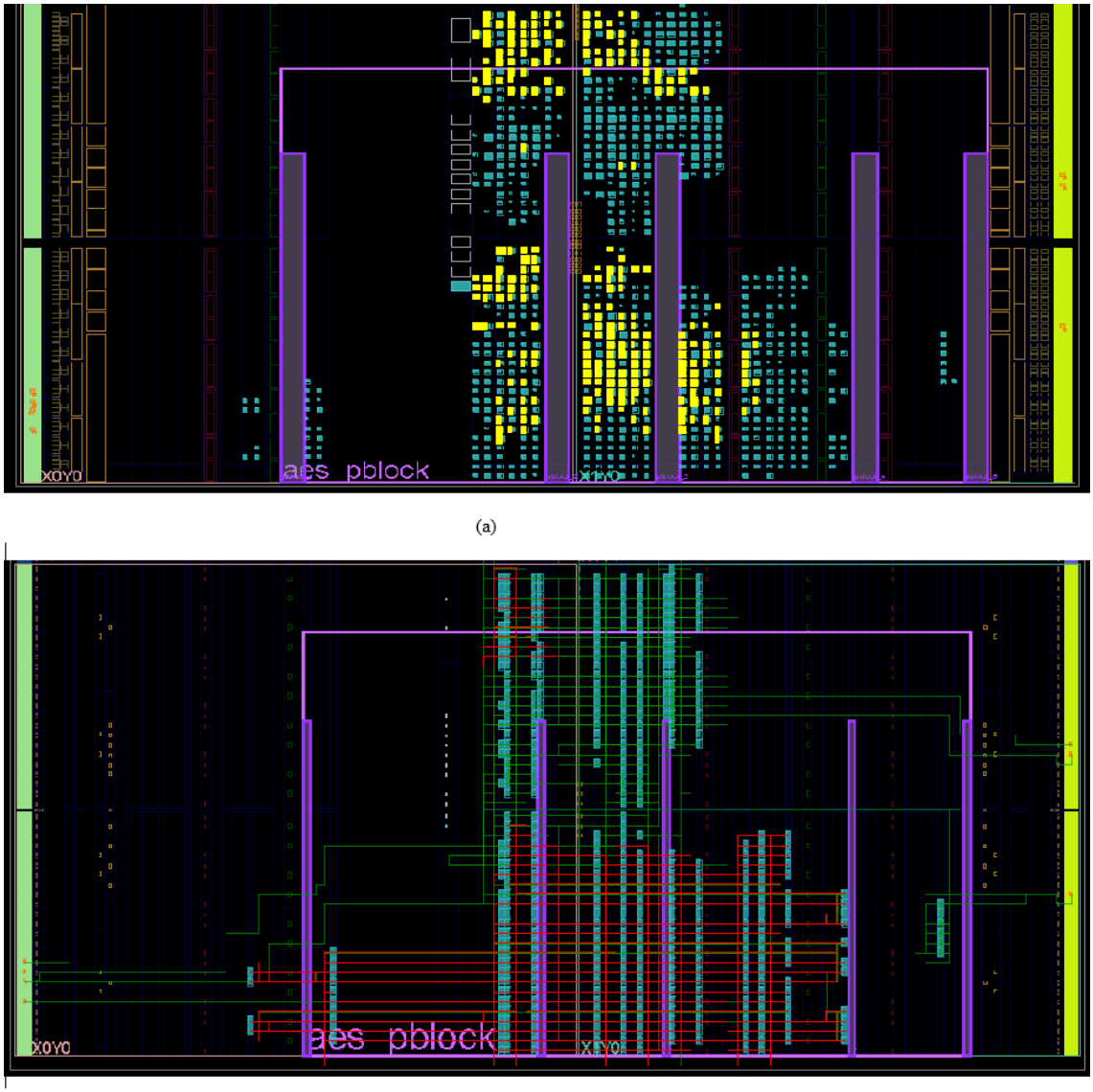


(a)



(b)

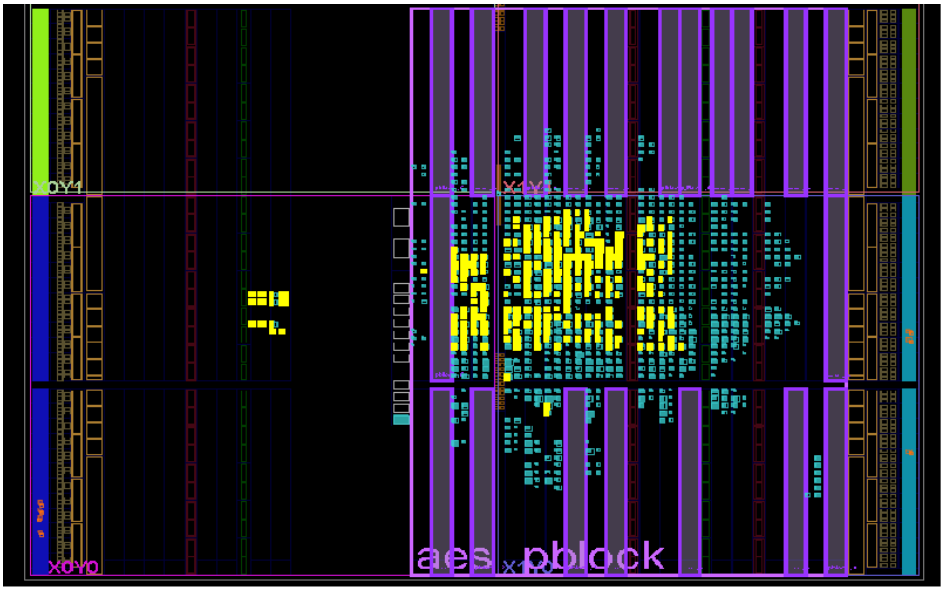
Figure 13: Pattern 1 (a) highlighted registers (b) highlighted nets.



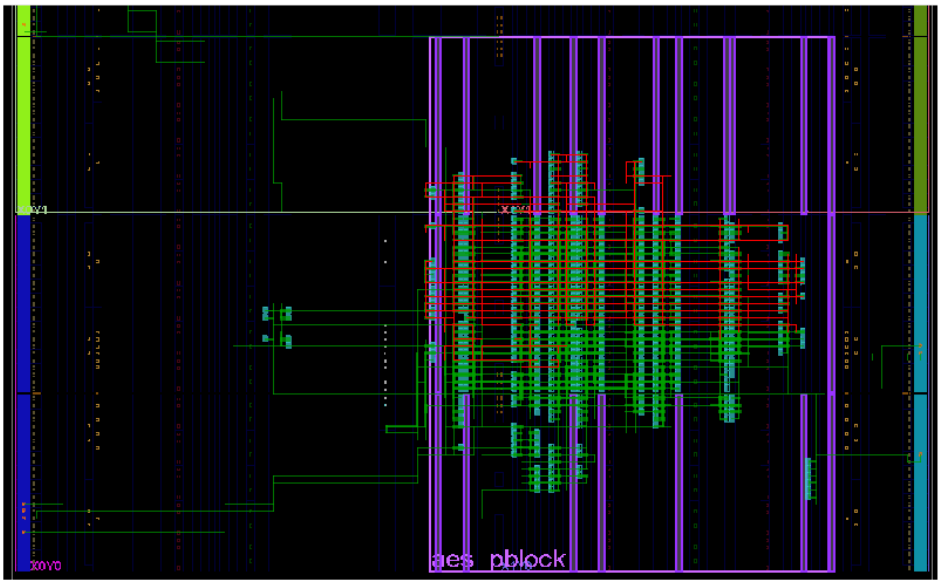
(a)

(b)

Figure 14: Pattern 2 (a) highlighted registers (b) highlighted nets.

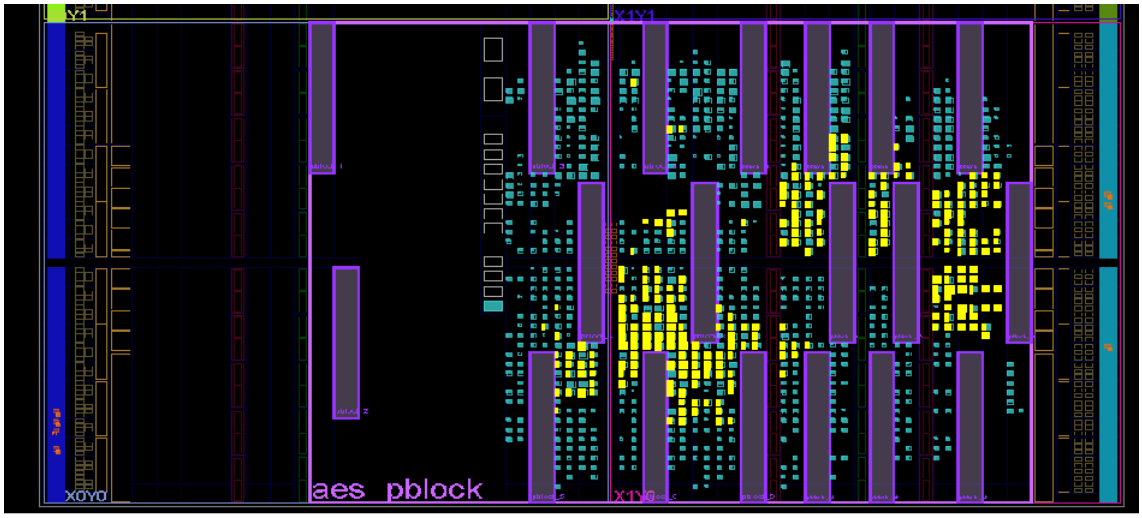


(a)

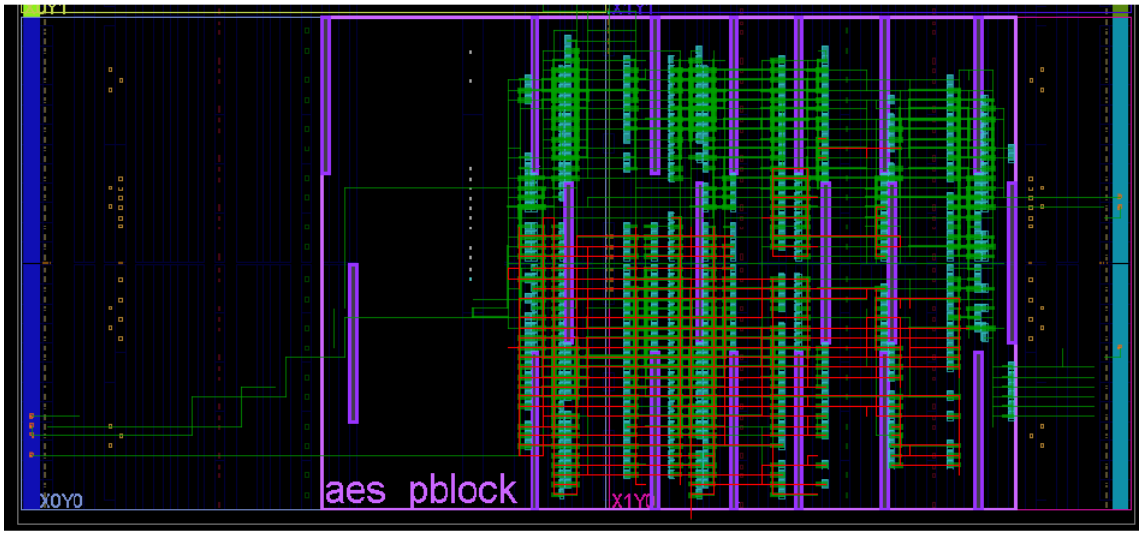


(b)

Figure 15: Pattern 3 (a) highlighted registers (b) highlighted nets.



(a)



(b)

Figure 16: Pattern 3 (a) highlighted registers (b) highlighted nets.

5.3 Noise generator contents

Next step in noise generator creation is adding appropriate content to maximize the switching activity within the unused FPGA patterns, while also preserving the AES core after implementation [49],[50],[51]. For the purposes of this thesis, three different variants are explored, where each is toggled at every cycle, and all of them contain INIT values, randomly assigned to each LUT, as further explained in subchapter 6.3.2 as well:

1. Pblock level noise (N1): a number of ring oscillators is placed per noise region, solely created by multiple LUTs (SRLs) connected in a feedback loop to sustain constant toggling.
2. Slice-level noise (N2): each SLICEM is occupied by one noise source, implemented as a short SRL chain (SRL16E).
3. LUT-level noise (N3): each LUT within a SLICEM is configured independently as an SRL. This particular level of noise is considered more aggressive than the previously mentioned ones, due to all LUTs activating simultaneously, creating the highest switching density.

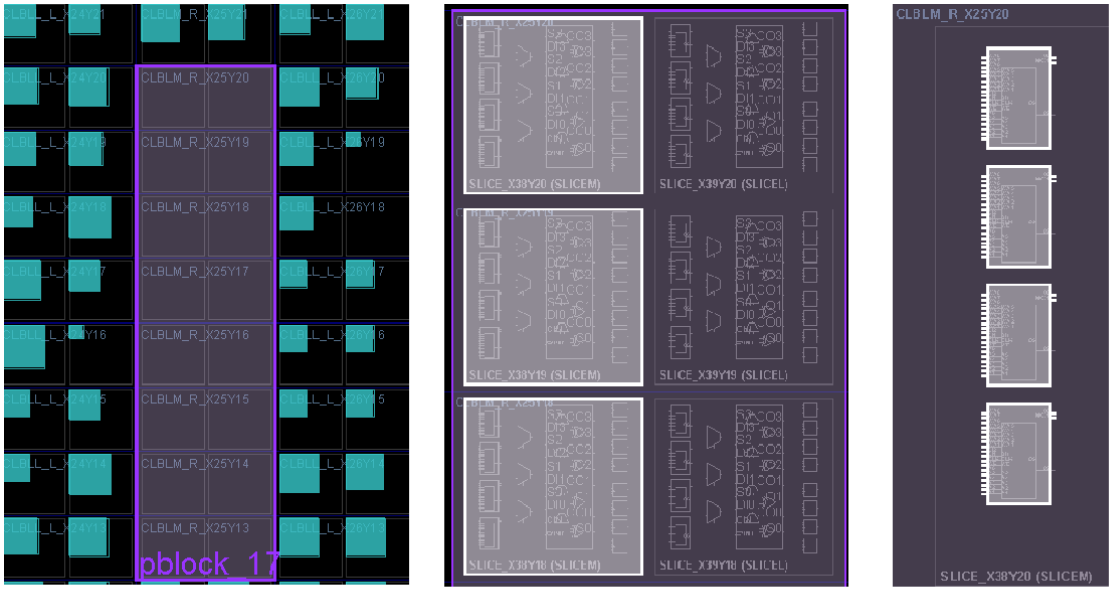


Figure 17: Noise content areas (a) N1, (b) N2, (c) N3

6. RapidWright Implementation of Noise Generators

Noise structures described in Chapter 5 are implemented on the FPGA using a post-implementation flow based on Vivado DCP files and RapidWright. As previously explained, AES core is firstly implemented normally in Vivado, then a TCL script extracts all candidate SLICEM sites from empty Pblocks and finally a custom-made RW script instantiates SRL-based noise rings and routes them using the existing clock and power. Lastly, the DCP is returned to Vivado for the final routing checks and bitstream generation.

6.1 RapidWright Workflow Overview

The workflow consists of four basic steps [23],[24],[25]:

1. Baseline Implementation in Vivado: the AES and SS2 core is floorplanned into a designated Pblock, while all surrounding Pblocks are reserved for noise and therefore are kept empty. Afterwards, Vivado writes a baseline DCP, which contains both logical and physical information about the current design and is ready to be used by RW.
2. Slice discovery utilizing in Vivado: A short TCL script was created for the purpose of mass retrieval of all empty SLICEM sites inside existing Pblocks, which stores them in a text file. The format in which those sites are stored is a physical coordinate of the region, for example SLICE_X30Y144.
3. Netlist Augmentation in RW: a custom script created using RW is responsible for loading the baseline AES as a DCP, reading the SLICEM list from the text file and creating SRL16E cells in selected LUTs. These LUTs are then connected into rings, attaching their CLK pins to the AES clock net “clk_buff”, while all control inputs (A0-A3, CE) are further tied to VCC and finally local nets for the SRL outputs are created and routed.
4. Finalization in Vivado: RW generates the DCP containing noise, which is re-opened in Vivado for a final “route_design” in TCL command line, which makes sure that all nets are connected properly, right before the bitstream file is generated.

6.2 TCL Automation for Pblock & SLICEM Extraction

When the user restricts areas in Pblocks, multiple available SLICEM sites need to be extracted from the design, right before the DCP is generated in Vivado [9],[11]. Therefore, a short TCL script automates this process as follows:

```
set all_pblocks_raw [get_pblocks pblock_*]
if {[regexp {pblock_(\d+)} $pb -> num]} {
    lappend num_name_pairs [list $num $pb]
}
set sorted_pairs [lsort -integer -index 0 $num_name_pairs]
``` :contentReference[oaicite:2]{index=2}
```

Firstly, the TCL script collects all Pblocks whose names match pblock_*, it extracts them numerically and sorts them, so that they can be occupied with noise in the same order.

```
set slices [get_sites -of_objects [get_pblocks $pbname] -filter
{SITE_TYPE == SLICEM}]
set slices [lsort $slices]
foreach s $slices {
    set placed_here [get_cells -quiet -of_objects [get_sites $s]]
    if {[llength $placed_here] == 0} {
        puts $fh "\"$s\","
    }
}
``` :contentReference[oaicite:3]{index=3}
```

The script also enumerates all sites inside each Pblock, picks the empty areas (with no cells placed) and writes the final list of available locations in a text file, one per line, which will be used by RW.

6.3 Java Noise Generator Construction

The RapidWright framework proved to play a significant role in noise generation, since the is capable of accessing designs at a lower level by using the device's primitives and routing resources in order to insert SRLs acting as countermeasures, during post-implementation by instantiating new cells, custom net creation, and placement within pre-specified SLICEM sites, as previously described. By deploying RW's capabilities, the patterns analyzed in Chapter 5 can be physically translated into hardware structures without disturbing the AES core implementation while at the same time effectively protecting the cryptographic core. [9],[12],[23],[24-27].

6.3.1 Design Setup and Inputs

The script begins by constructing a RapidWright Design object, targeting the Artix-7 device, which loads the architectural model used (BELs, Programmable Interconnect Points, sites and wires) and prepares the design to accept a DCP:

```
Design d = new Design("SRLMultiSliceRing", "xc7a35tcsg324-1");
d.setAutoIOBuffers(false);
d.setDesignOutOfContext(true);
Design.readCheckpoint(d, inputDCP);
```

To ensure that RW does not modify any of Vivado's I/O placement, "setAutoIOBuffers" is set to false, an action which makes the AES block electrically identical to the baseline AES design.

Following, the SLICEM sites previously extracted using TCL are loaded in the script:

```
String[] availableSlices =
SliceReader.readSlicesFromFile(sliceFilePath);
```

A quick sanity check ensures that there are enough resources for the script to use, particularly enough slices for the configured number of rings, enough LUTs for the noise modes (N1-N3), as well as valid SLICEM locations on the target device:

```
if (slicesPerRing * numRings > availableSlices.length) {
    throw new IllegalArgumentException("Not enough slices
available.");
}
```

6.3.2 SRL Instantiation & INIT Pattern Assignment

The basis core of the noise generator construction involves the conversion of every available LUT inside a SLICEM site into an SRL, specifically an SRL16E, a conversion process which starts when RW allows direct creation and placement of a UNISIM primitive, given the input of BEL locations. Each slice has four valid LUT BELs, which are initialized early inside the custom Java script, and afterwards each LUT inside every slice is instantiated as an SRL16E, as can be seen below:

```
String[] lutNames = {"/A6LUT", "/B6LUT", "/C6LUT", "/D6LUT"};
String srlName = "Ring0_SRL" + cellIndexGlobal;
String fullLocation = sliceName + lutName;
Cell srl = d.createAndPlaceCell(srlName, Unisim.SRL16E,
fullLocation);
```

In this way, it is possible to bypass logic optimization, forcing the exact placement of the noise generators. Moreover, it is important to note that continuous toggling patterns are diversified across LUTs, by assigning a unique 16-bit INIT pattern to each SRL:

```
String initLiteral = makeInitLiteral(); // e.g., "16'hA5F0"
EDIFTools.createAndSetInitOfSRL(allCells[cellIndexGlobal],
initLiteral);
```

For this thesis, and as previously explained, three different noise variants have been generated (N1, N2, N3) whose noise is achieved by modifying either which LUTs or how many slices the RW script iterates over. The aforementioned flow of assignment is common in all three noise pattern variants that were produced for this experiment, with the main differences between them being how many slices or LUTS are used per ring and how INIT values are chosen and reused per ring, slice or LUT.

Firstly, the common instantiation skeleton is discussed, as it is shared in all three patterns (N1, N2, N3). This skeleton includes looping over rings, then over slices within each ring and lastly over LUTs inside each slice. For each LUT that is selected, a new SRL is created and an INIT value is assigned to it:

```
for (int ringIndex = 0; ringIndex < numRings; ringIndex++) {
    int cellIndex = 0;

    for (int sliceIndex = 0; sliceIndex < slicesPerRing;
sliceIndex++) {
        String sliceName = availableSlices[ringIndex *
slicesPerRing + sliceIndex];

        for (String lutName : lutNames) {
            String srlName      = "Ring" + ringIndex + "_SRL" +
cellIndex;
            String fullLocation = sliceName + lutName;

            Cell srl = d.createAndPlaceCell(srlName, Unisim.SRL16E,
fullLocation);

            // INIT selection for N1, N2, N3:
            String initValue = /* depends on noise mode */;
            srl.addProperty("INIT", initValue);

            cellIndex++;
        }
    }
}
```

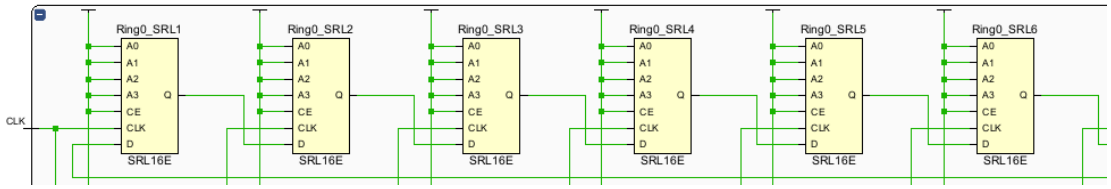


Figure 18: LUT Ring example

A LUT ring (or SRL ring) can be depicted in Figure 18 and further information about the connections of each LUT will be discussed in section 6.4.

Slight differences across all three variants can greatly affect the final countermeasure that is created, due to the different densities and patterns of noise that are applied to each generator. To be more precise, the differences between these variants are as follows:

1. N1 – Pblock Noise: In this mode, each ring (or Pblock noise region) utilizes a single INIT pattern shared by all the SRLs contained in it, which is achieved by indexing "initPatterns" only by "ringIndex":

```
String[] initPatterns = {
    "16'h9E37","16'hD3A9","16'h7C96","16'h2B1D",
    "16'hE4C3","16'h4DB2","16'h1F79","16'hB6E5",
    "16'hCAF3","16'h3FAC","16'hF33C","16'h3CF3"
};

// N1: one INIT per ring (per Pblock)
String initValue = initPatterns[ringIndex % initPatterns.length];
srl.addProperty("INIT", initValue);
```

   What this design achieves is that all SRLs in one region will toggle with the same pattern, but are scattered across the board, thus noise is coarse because it is applied in different regions. Out of the three variants, N1 is the least aggressive one, mostly due to the noise being similar to multiple areas.

2. N2 – Slice Noise: The INIT pattern for this variant is selected on a per-slice basis, rather than per-ring one. In this case, all LUTs within a given slice would have the same pattern for INIT, but other slices can use different ones, even if they belong to the same Pblock area.

```
// extended pattern list
String[] initPatterns = { "16'h9E37","16'hD3A9", ..., "16'h4110" };

// N2: one INIT per slice
int patternIndex = (ringIndex * slicesPerRing + sliceIndex) %
initPatterns.length;
String initValue = initPatterns[patternIndex];
srl.addProperty("INIT", initValue);
```

   In this specific variant, "patternIndex" is dependent on both the ring and the slice within it, making each slice behave as a small "noise cell", while all four LUTs inside still toggle, thus creating denser noise across the board.

3. N3 – Lut Noise: In the last variant, the INIT pattern is assigned to each individual SRL, using the global cell index. For the purpose of this countermeasure, due to the large number of LUTs for each design, a Pseudorandom Number Generator (PRNG) was implemented in the script, to generate different 16-bit patterns for each SRL, using an xorshift generator:

```
// xorshift32-based PRNG for 16-bit INIT values
private static int rngState = 0xA5A5F00D;

private static int nextRand16() {
    rngState ^= (rngState << 13);
    rngState ^= (rngState >>> 17);
    rngState ^= (rngState << 5);
```

```
    return rngState & 0xFFFF;
}

private static String makeInitLiteral() {
    int val16 = nextRand16();
    String hex = String.format("%04X", val16);
    return "16'h" + hex;
}
```

In this way, all LUTs (even neighboring ones, within the same slice) will be occupied with a unique INIT value and due to them being simultaneously active on all slices, a high-density noise will be produced, thus establishing N3 as the most aggressive and diverse out of the three variants.

6.4 Connectivity and Routing Integration

RapidWright is useful for configuring how primitives operate by defining their shifting behavior, their connectivity and the clocking and routing that is required for them to be integrated into the baseline AES. This chapter explains how RW can manipulate pins, nets and routing resources to create noise generators, while keeping the original AES logic intact. [11],[25]

6.4.1 Shifting Behavior and Connectivity

The SRL16E primitive's exposed pins consist of a 4-bit address for selecting the output (A0 to A3), the clock enable (CE), a shift clock (CLK), serial data input (D) and shifted output (Q), all of which are responsible for controlling the shift register's internal behavior. Furthermore, each SRL instance has its A0-A3 and CE pins connected to the global VCC net, which forces the SRL to operate at a 16-bit depth, meaning that the value of A0-A3 is set to "1111" and CE is set to "1", thus continuously shifting the contents of each LUT to the next one. RapidWright is using a text-based format called Electronic Design Interchange Format (EDIF), which represents a design's logical netlist to read and write from. As it can be seen at the following code snippet, "vccNet.createPortInst" creates an EDIF port instance on the SRL, which connects each pin to the VCC net.

```
vccNet.createPortInst("A0", srl);
vccNet.createPortInst("A1", srl);
vccNet.createPortInst("A2", srl);
vccNet.createPortInst("A3", srl);
vccNet.createPortInst("CE", srl);
```

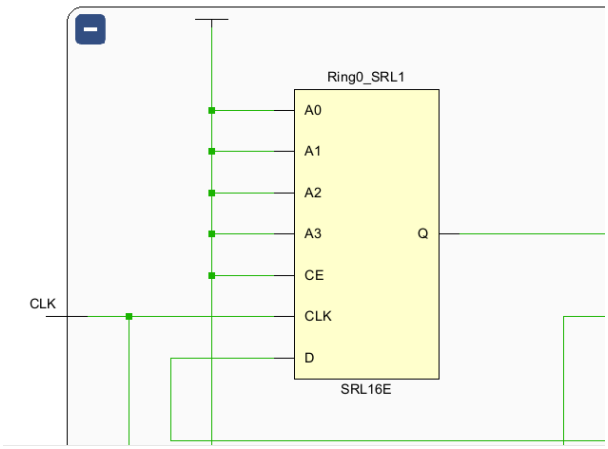


Figure 19: Pins and Ports of an SRL

First step to better conceive how SRL outputs drive subsequent SRL inputs is to understand how the connectivity between them is achieved. RapidWright is capable of allocating a dedicated net that is connected to SRL's Q pin, by utilizing "createNet":

```
allNets[cellIndexGlobal] = d.createNet("Ring0_Net" +
cellIndexGlobal);
allNets[cellIndexGlobal].connect(allCells[cellIndexGlobal], "Q");
```

When all SRLs have an output net assigned to them, the script then connects them into a single ring:

```
for (int i = 0; i < allCells.length; i++) {
    int nextIndex = (i + 1) % allCells.length;
    allNets[i].connect(allCells[nextIndex], "D");
}
```

As shown, this loop is responsible for the wiring between Q and D, where each SRL output Q drives the D input of the next SRL and the final SRL wraps back to the first one.

6.4.2 Clock Coupling to AES

As anticipated, connecting generators through net instantiation and arranging them into usable, active rings is the initial phase of an effective countermeasure against side-channel attacks, when produced in the aforementioned way. Another essential detail that needs to be taken into consideration, is the timing of these countermeasures and how their behavior is changing based on the clock they are using. Therefore, all generators need to be connected with the same clock as the AES core, and in particular use the same clock net (clk_buf) as the cryptographic core. The process described is clear in the following code snippet:

```
EDIFNet clock_net = d.getTopEDIFCell().getNet(clkNetName);
if (clock_net == null) {
    clock_net = new EDIFNet(clkNetName, d.getTopEDIFCell());
    d.getTopEDIFCell().addNet(clock_net);
    EDIFPort clkPort = d.getTopEDIFCell().createPort("CLK",
EDIFDirection.INPUT, 1);
    clock_net.createPortInst(clkPort);
}
clock_net.createPortInst("CLK", allCells[cellIndexGlobal]);
```

The script firstly checks whether the “clk_buf” exists in the noise design and in case it does not, an EDIFNet is created with a top-level port “CLK”, using the “createPortInst”, which attaches the CLK pin to this specific net. All SRLs are clocked by the same clock tree as AES, therefore the switching activity achieves the same timing margins as the las AES round transitions.

When the noise design is later merged into the top level of the provided DCP, the same “clk_buf” is used, deploying RW’s Module/ModuleInst which is a mechanism that ensures the port targeted on the module instance is in fact named “CLK” and is mapped to the internal “clk_buf” net:

```
EDIFNet topClkNet = topDesign.getTopEDIFCell().getNet(clkNetName);
…
topClkNet.createPortInst("CLK", lutRingModuleInst.getCellInst());
```

6.4.3 Routing & Merging in the AES design

After defining all cells and nets at the netlist stage, RW carries out a physical routing in two different phases:

```
d.routeSites();
new Router(d).routeDesign();
```

“routeSites” manages the internal wiring, for example LUT outputs to SRL inputs withing the same slice, whereas “Router(d).routeDesign” assigns routing resources for inter-site purposes, such as INT tiles, switch boxes and long wires to all newly formed nets. As mentioned before, “CONTAIN_ROUTING” is already used in the previous step where the AES Pblock was implemented in Vivado, therefore all internal nets are already pre-routed and remain unchanged during this phase, meaning that RW only routes the new nets created for the generators.

Ultimately, the goal is to incorporate the noise generator into the already implemented AES architecture:

```
Design topDesign = Design.readCheckpoint(inputDCP);
ModuleInst lutRingModuleInst =
topDesign.createModuleInst("LUTRing", new Module(d));
lutRingModuleInst.placeOnOriginalAnchor();
…
topDesign.writeCheckpoint(outputDCP);
```

Here, “new Module (d)” encapsulates the noise configuration as a module, “createModuleInst” and “placeOnOriginalAnchor” inserts it into the same physical region matching the SLICEM coordinates previously provided, and finally exports an “outputDCP”, which generates the noise generator implementation. Moreover, the AES implementation is identical to the baseline, including its logic, placement and routing, the reserved empty Pblocks surrounding AES are occupied by noise generators, and all SRLs are clocked by “clk_buf” as previously described. Finally, the DCP is opened in the Vivado environment, where any unplaced nets are fixed using the TCL command “route_design” and a bitstream file is generated using the command “write_bitstream”.

7. Experimental Methodology and Results

The current chapter includes a complete evaluation of the baseline AES implementation and all the noise-enhanced variants created using RapidWright, as outlined in Chapters 5 and 6. Unlike the previous chapters which focused on the architectural design, the construction of the countermeasures, and the implementation details, this chapter only presents empirical results. It covers how each design performs under a CPA attack, how many traces are needed to recover the AES 128-bit key, how noise placement and density might influence correlation, and hardware overhead exists, based on Vivado built-in reports. Unless otherwise stated, each experiment starts with 2000 traces, and this trace count increases to 2500, 3000 or even 3500 only if the full 16-byte key is not recovered [9],[21],[39],[52].

An important clarification, at this point, is that the evaluation of the implemented designs was conducted in order to assess whether a countermeasure such as noise generators, is sufficient to protect any baseline AES implementation. This means, that the point of the final evaluation was to increase the effort needed for a single attack in a measurable way, rather than create a design which would require thousands of traces to break, using the tools available for this particular thesis.

7.1 Design Overview

Overall, a total of 13 AES variants were evaluated, one baseline design of the AES core which was placed inside a tightly constrained Pblock with no generators, and twelve enhanced implementations, consisting of four different placement patterns, each with three different noise densities (per Pblock, per slice, per LUT).

For the purposes of uniformity across all evaluations during the experiments, all variants were captured under identical measurement conditions, including same key, Husky ADC gain, sampling frequency, and identical triggering configurations. Apart from the consistency achieved by this means of capturing, the attack difficulty and the countermeasure effectiveness can also be studied, due to the only changes between designs being placement and content pattern.

7.1.1 Implemented Countermeasures on Designs

As described in subsection 5.2, the countermeasures were implemented in a series of patterns. In the following figures, the placement of LUTs is visible for every design created. A useful note is that the implementation of the countermeasures only happens once for each design and the difference between each pattern lies in the contents of each noise generator. Therefore, four visually different designs can be studied and later attacked:

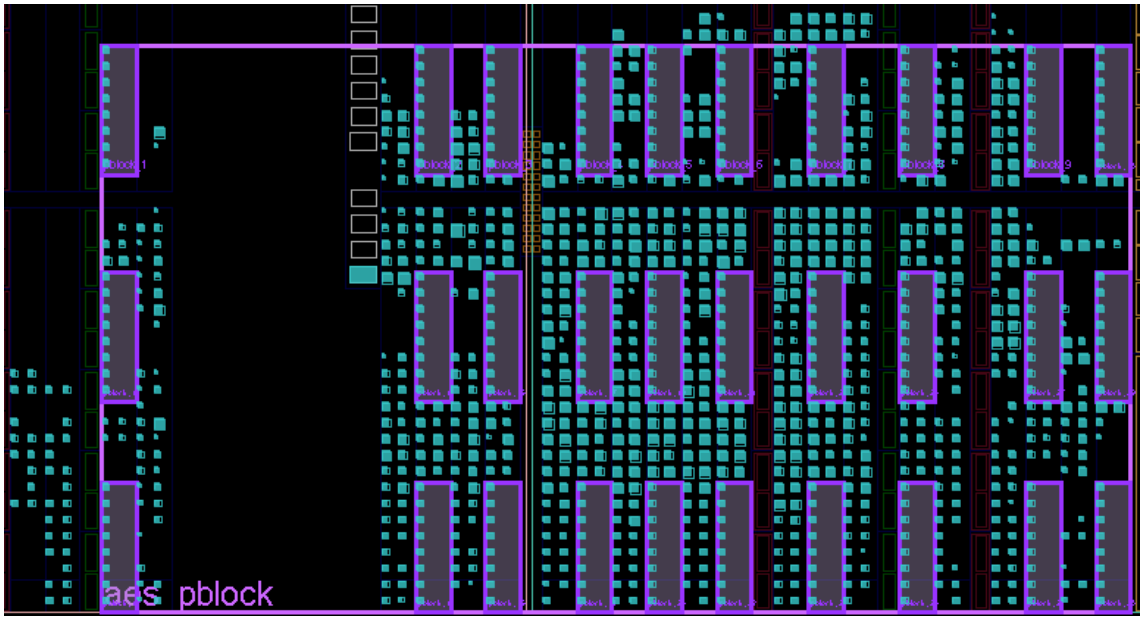


Figure 20: Pattern 1 (noise generators)

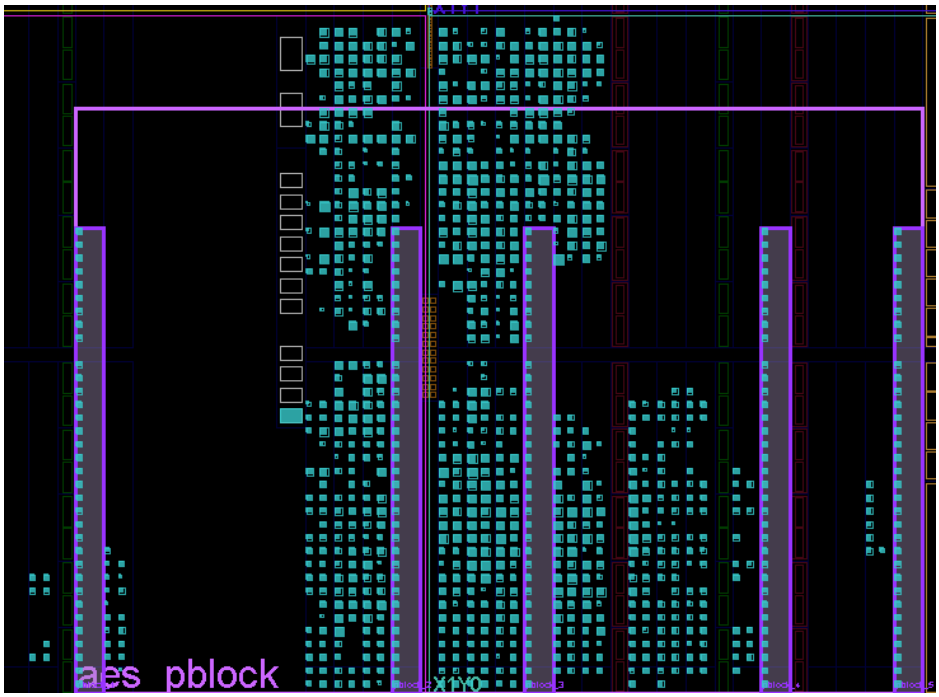


Figure 21: Pattern 2 (noise generators)

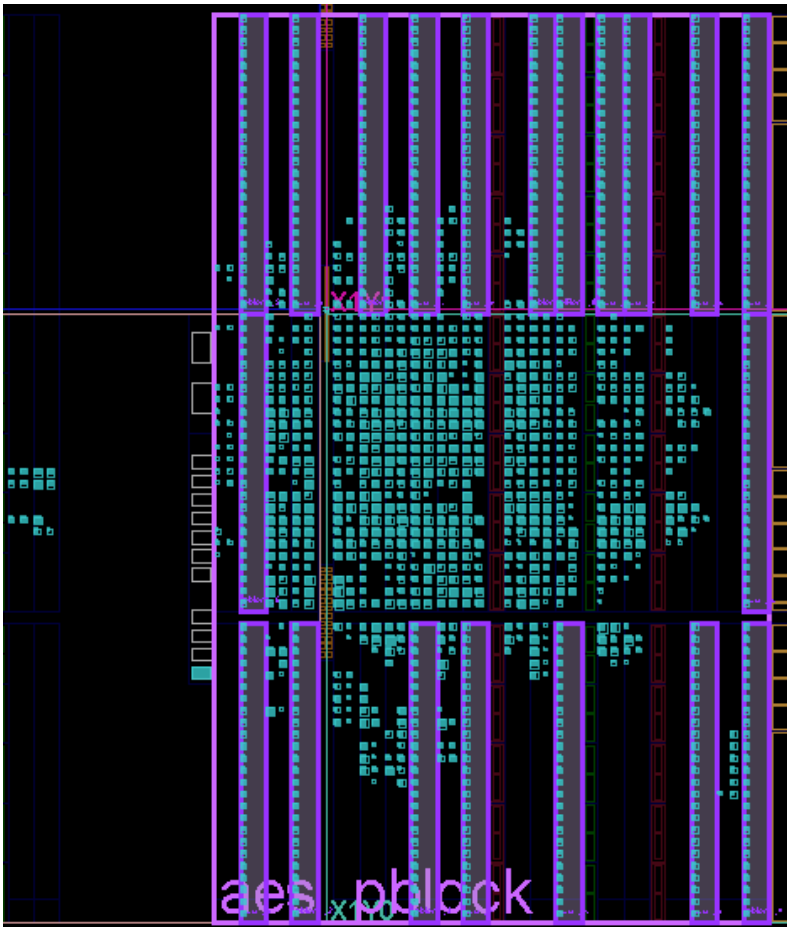


Figure 22: Pattern 3 (noise generators)

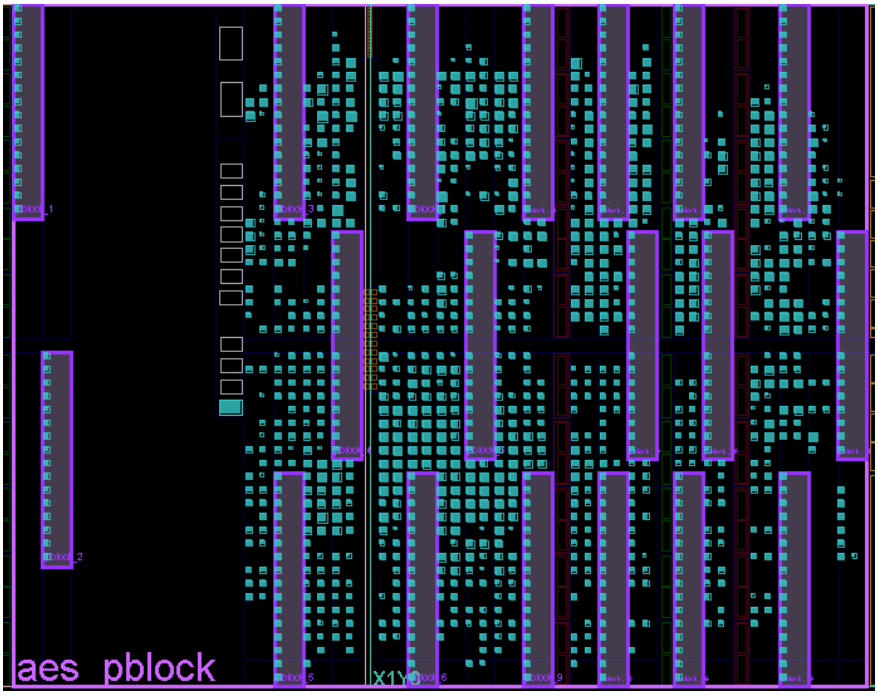


Figure 23: Pattern 4 (noise generators)

For more clarity, the following figures (24-28) depict how noise generators coexist in the same implementation, where sensitive registers are in yellow and sensitive nets are in red:

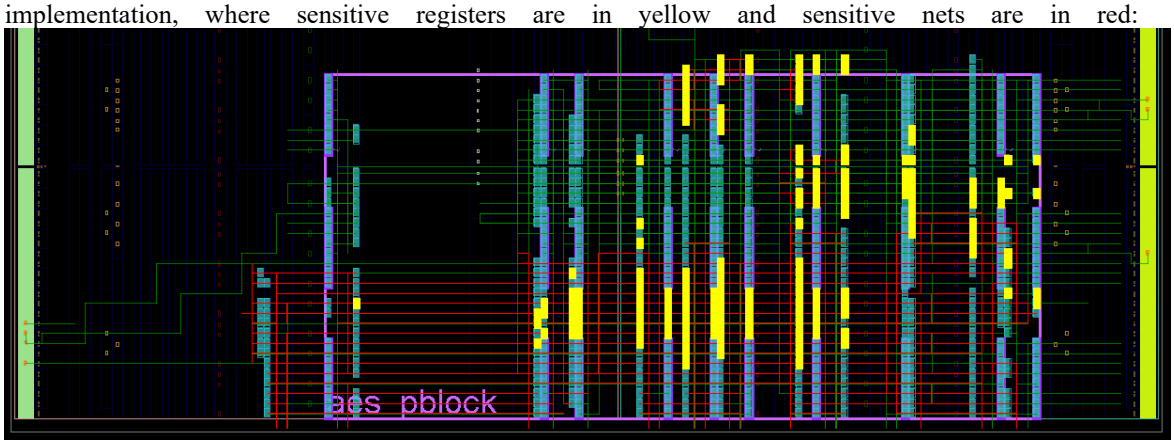


Figure 24: Pattern 1 Full Implementation

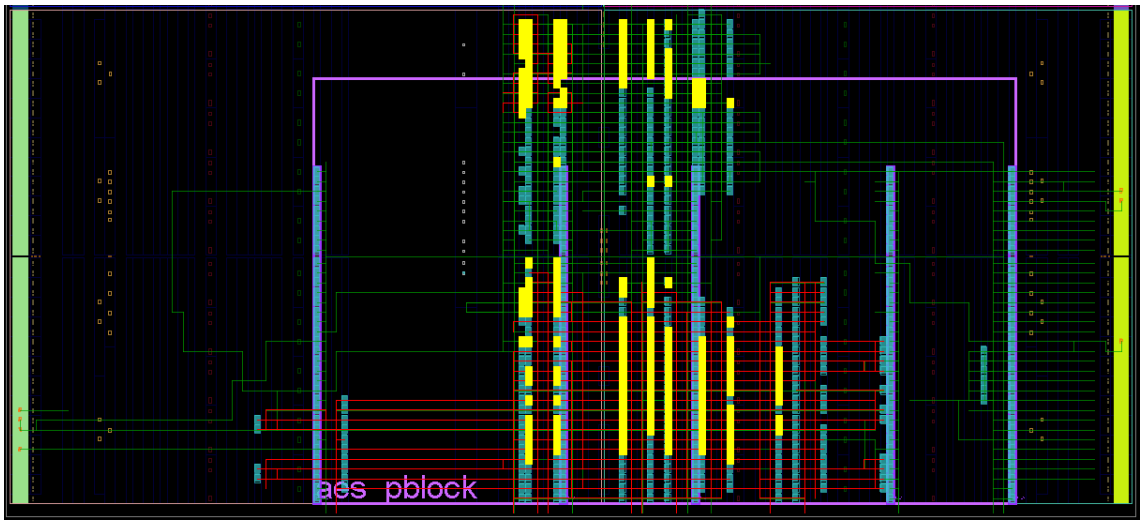


Figure 25: Pattern 2 Full Implementation

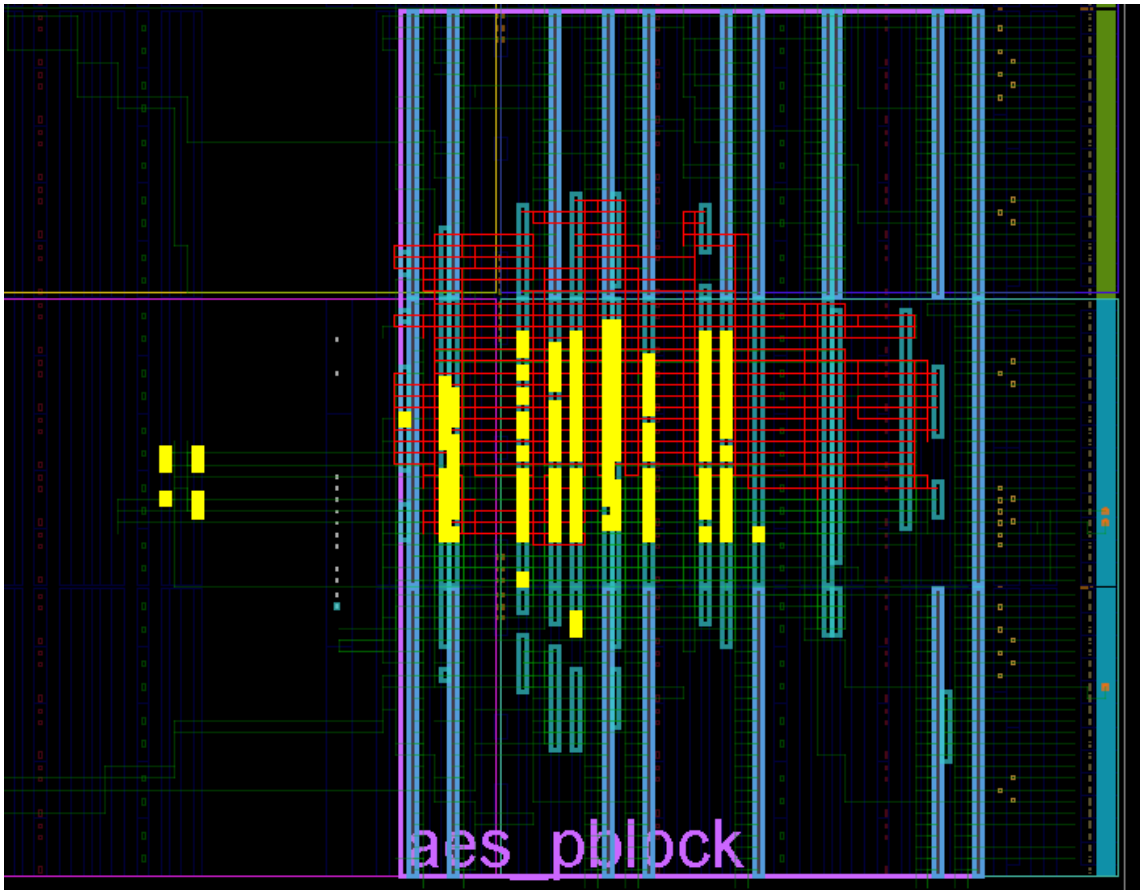


Figure 26: Pattern 3 Full Implementation

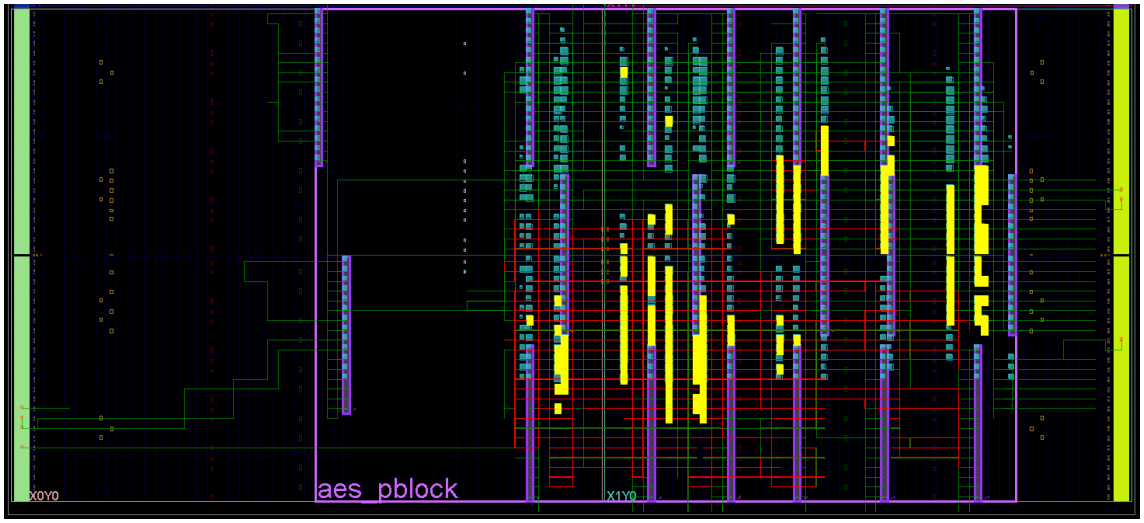


Figure 27: Pattern 4 Full Implementation

It is apparent that in patterns 1 and 4, most of the AES logic (if not all of it) is placed inside the pre-arranged pblock and that all sensitive registers and nets are routed inside it, whereas when it comes to patterns 2 and 3, some sensitive registers seem to be placed outside the pre-allocated space, which could later be interpreted as worse countermeasures compared to 1 and 4. An explanation for this phenomenon could be that the first pblock which contains all of the AES logic either is too restrictive and the implementation is forced to assign areas outside of it to match any timing restrictions or that due to the

usage of LUTs by the countermeasures, resources are less than expected, thus forcing the implementation again to use areas outside the AES pblock.

7.2 Key Recovery Performance

7.2.1 Trace Count for Successful Key Recovery

As previously mentioned, the four patterns based on their available empty Pblock areas are 30, 5, 20 and 19 (numbered from 1 to 4 in this order) and the three different noise contents are per Pblock, per slice and per LUT (numbered from N1 to N3 in this order). The following table depicts an overview of the attacks of all combinations of patterns and noise contents, as well as the baseline AES:

Table 2: Results overview of attacked designs

Design		Key Recovery (16 bytes)			
Pattern	**Noise**	**2000 traces**	**2500 traces**	**3000 traces**	**3500 traces**
Baseline	-	16/16	-	-	-
1	N1	1/16	16/16	-	-
1	N2	16/16	-	-	-
1	N3	1/16	2/16	2/16	16/16
2	N1	16/16	-	-	-
2	N2	5/16	16/16	-	-
2	N3	16/16	-	-	-
3	N1	16/16	-	-	-
3	N2	16/16	-	-	-
3	N3	16/16	-	-	-
4	N1	2/16	2/16	16/16	-
4	N2	6/16	16/16	-	-
4	N3	2/16	16/16	-	-

Regarding the baseline AES implementation, there is a clean leakage profile, where the key is fully recovered withing 2000 traces. It is evident that the most substantial improvements arise from Pattern 1 and 4, where in pattern 1 with N3 (per LUT noise) the traces needed to recover the full key are increased to 3500, 1500 traces more than the baseline, whereas pattern 4 with N1 and N3 results in 3000 and 2500 traces respectively. In contrast, patterns 2 and 3 offer little or no security benefit, with many of their variants being able to break at 2000 traces [21],[43].

7.2.2 Effect of Noise Placement

As observed by the results in Table 2, the attack resistance of the aforementioned designs (or the resistance in general) is directly influenced by the noise generator placement, an observation which also aligns with the FPGA leakage theory which was previously presented; isolated LUT activity could potentially dominate AES leakage much less than register switching and long interconnection wires that carry sensitive operational information, such as carry S-Box outputs, round key values, and final round state transitions. Therefore, placing multiple noise generators with various patterns of their own can overlap the sensitive nets that need to be protected, increasing local dynamic power and possibly managing to confuse CPA, due to noise introduction.

In figures 24 and 27, it is clear that patterns 1 and 4 seem to intersect the AES datapath more aggressively, as multiple long wires must physically pass through or near occupied by noise generators SLICEM rows. On the other hand, patterns 2 and 3 operate on areas that are spatially distant from registers or nets and seem to contain less long interconnection nets. Consequently, their noise does not greatly impact the AES timing-critical routes and leakage model remains stable, as is also evident by Table 1, where these specific designs totally break at around 2000 traces and do not exhibit any difference in comparison to the baseline AES design.

In short, placement is effective only when the noise generators physically intersect the real leakage sources, thus it is required to place them as near as possible to any sensitive points of interest [14-16],[21].

7.2.3 Effect of Noise Density

While structuring and placing noise generators around sensitive areas of a design can greatly impact the breaking point of it and increase the number of traces needed to achieve this result, noise density is of equal importance to the noise generation effectiveness. Noise density enhances countermeasures only in conjunction with good placement, but its internal structure is what differentiates why some noise contents outperform others.

Starting with Pblock noise (N1), only a few SRLs are able to toggle in each pblock area, therefore the noise generation is usually coarse and might also be sparse and thinly distributed across the FPGA. To be more precise, during implementation and routing, long wires have increased chances of not passing close to the slices that were selected for the current countermeasure, leading to the wire left unprotected and the AES leakage undisturbed. Such a phenomenon was observed in cases where N1 was helpful and increased traces required for the attack, such as P4, whereas in other patterns (e.g. P2 and P3) the protection collapses completely early on.

Similarly, regarding the per-slice noise (N2), each slice and its four LUTs share the same INIT patterns and different slices use different patterns. When compared to N1, N2 manages to introduce spatial diversity across slices, while maintaining the correlation within each slice, which briefly explains why N2 sometimes outperforms N1. Nevertheless, the same principle applies; multiple different patterns in less space manage to produce more unpredictable patterns, therefore resulting in better outcomes. A clear example is in P4 where N2 configuration is able to recover a full key at 2500 traces, whereas N1 seems to collapse sooner.

Finally, the per-LUT noise configuration (N3) assigns pseudo-random INIT values to every individual LUT (each SRL), and connects all SRLs into a single noise ring, where all neighboring LUTs never share the same noise pattern, therefore neither do their corresponding slices. Moreover, from the attacker's perspective, the AES transitions are now dependent on many overlapping switching sources in the same physical area. For example, this case is evident under P1, where N3 delays the full key recovery until 3500 traces, while on the other hand N1 and N2 already allow the key retrieval to happen at 2500 and 2000 traces respectively. To explain this further, N3 injects fully independent switching activity in every LUT and therefore creates a higher local entropy, while also disturbing AES long wire leakage much more efficiently than N1 and N2 [7],[21],[43].

7.3 Correlation Analysis

While it is apparent that key recovery success provides a clear and practical metric for the effectiveness of each noise configuration, the experiments should be accompanied by deeper analysis and insights into how and why noise suppresses information leakage. Therefore, it is important that the correlation analysis is also examined alongside the placement and density analysis of the implemented countermeasures. For this experiment, CPA was performed on the same traces used for key recovery attempts and for each design the CPA script computed correlation between the measured traces and the predicted Hamming Distance model for all 256 key guesses and all time samples. Only the maximum absolute correlation ($|\rho|$) per byte was recorded, regardless of whether the key was recovered, which enables an equal comparison between designs even when the key is not recovered under the same trace number.

In figure 28, CPA reveals clear patterns across varying noise levels and in particular, P1 has a mean correlation of 0.1281 ± 0.0180 and peaks at 0.1601 at sample 222. Similarly, P2 has a mean correlation of 0.1254 ± 0.0263 and has a higher peak at 0.1716 at 125. Moreover, P3 demonstrates an even higher mean correlation (0.1342 ± 0.0217) and reaches 0.1664 at sample 222, while P4 also has similar metrics with a mean correlation of 0.1266 ± 0.0240 peaking at 0.1638 at sample 77. In the same figure, it is observed that peak correlations have a fixed range from 0.160 to 0.172, which points to the fact that the number of areas does not contribute much to the maximum leakage of information. However, shifting peak locations may indicate differences in vulnerability instead.

As a conclusion, despite peak correlation values remaining withing a narrow range, the consistent downward shifts in mean correlation proves that noise generators do in fact produce amplified distortions into the leakage and that even small shifts in maximum correlation are meaningful, due to the fact that CPA success depends on the correct key guess producing a higher correlation peak, as opposed to the current example where the results indicate that active noise flattens or smears these peaks.

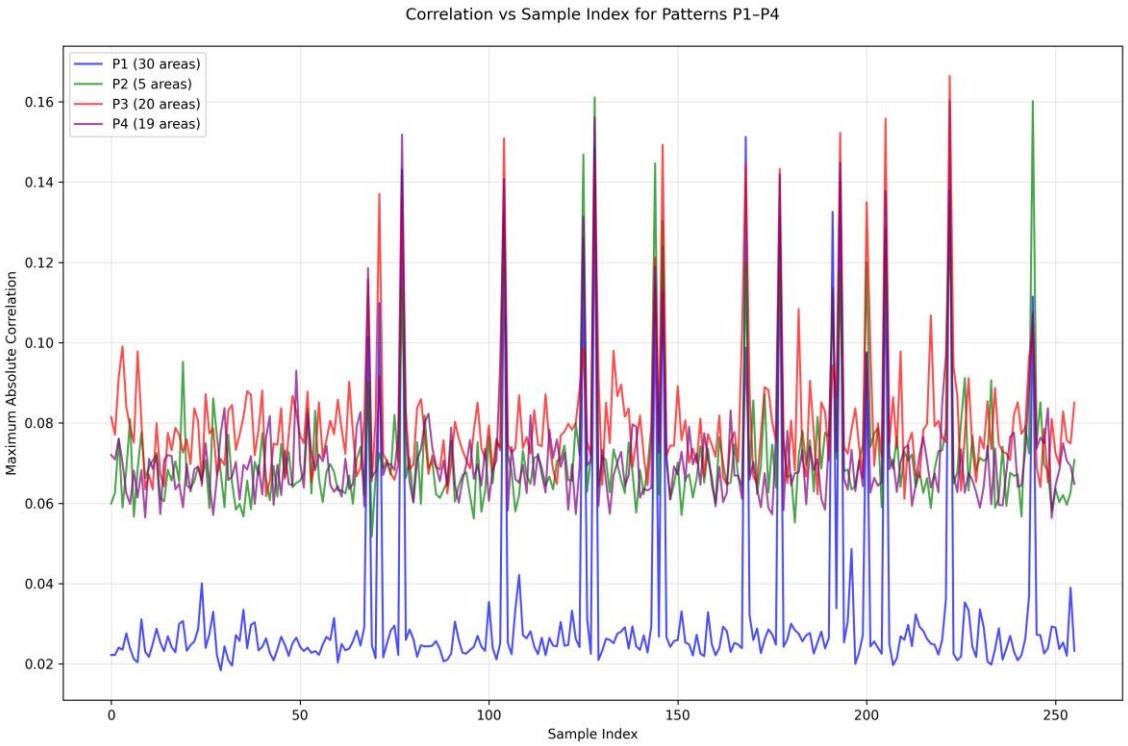


Figure 28: Correlation - Sample Index Plot

In figure 29, the 192 measurements (4 placement patterns x 3 noise densities x 16 key bytes) are analyzed, further showing that the three noise injection strategies exhibit similar behavior, though they still introduce consistent reduction of the CPA leakage signal. To expand more, the per Pblock configuration (N1) is able to achieve a mean maximum correlation of 0.1297 ± 0.0229, whereas the per slice (N2) and per LUT (N3) configurations have almost identical mean values, 0.1324 ± 0.0219 and 0.1320 ± 0.0236 respectively. Moreover, all three configurations share the same maximum correlation of 0.1810, indicating that perhaps the worst-case leakage is analogous to all cases, regardless of the noise variations. Even though these statistical differences might not be significant, the results actively prove that the produced noise generators are consistently lowering the average correlation compared to the baseline design and are capable of increasing variability. Small reductions in correlation amplitude could harden the correct key hypothesis (further proven by the increased number of traces required to retrieve the key), therefore not eliminating leakage completely, but weakening its exploitable structure [4].

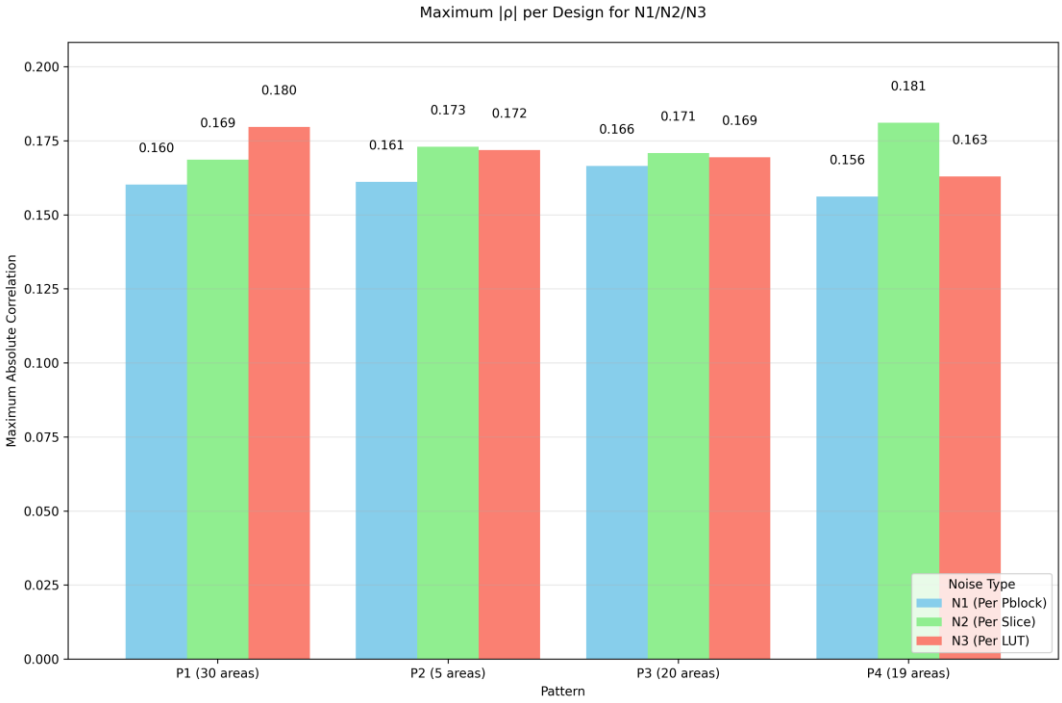


Figure 29: Maximum Correlation per Design

7.4 Vivado Resources Reports

Since all noise generators are additional logic implemented on the FPGA, the increase of resource usage and logic implementation is inevitable, thus making Vivado an important measurement tool for such utilities. This chapter contains all noise generator resources which were used, quantified by the built-in utilization reporting tools, as provided by Vivado. Moreover, the power report is also added in this section, to better understand how much power consumption each LUT ring for every design has and how much impact a potential noise generator might have in a cryptographic core such as AES. Table 3 contains all utilization report results for all designs (baseline as well) and their resources, while in Table 4 power consumption is analyzed and reports the percentage of the power consumption for each new noise implementation, compared to the overall power needed from the design. Note that Vivado power report is only an estimate of the values based on some metrics and might not be as precise as an actual physical power measurement [9],[11],[21].

Table 3: Utilization Report for baseline and protected designs.

Design	Name	Slice LUTs	Slice Registers	Slices	LUT (Logic)	LUT (Memory)
Baseline	SS2 AES Wrapper	2606	1416	812	2557	49
P1	SS2 AES Wrapper	3566	1416	1052	2557	1009
	Noise Gen	960	0	240	0	960
P2	SS2 AES Wrapper	3285	1416	1009	2556	729
	Noise Gen	680	0	170	0	680
P3	SS2 AES Wrapper	4606	1416	1340	2557	2049
	Noise Gen	2000	0	500	0	2000
P4	SS2 AES Wrapper	3823	1416	1158	2558	1265
	Noise Gen	1216	0	304	0	1216

In Table 3, "SS2 AES Wrapper" is the top module in the hierarchy, which contains all of the AES core logic, as well as the noise generators ("Noise Gen") for each design. Therefore, the values reported in the table for the wrapper are the total amount of utilized resources in the design, where part of those values correspond to the noise generators, which are separately reported for each design as well. It is further evident, from Table 3, that the additional SRL based noise generators have predictably increased LUT and slice utilization, which are proportional to the size of the pre-allocated areas for each on the FPGA board, while at the same time leaving the AES core intact, thus proving that the countermeasures scale based on their placement.

Table 4: Power Consumption Report for baseline and protected designs.

Design	Name	Power Utilization (% of total)
Baseline	SS2 AES Wrapper	2
P1	SS2 AES Wrapper	4
	Noise Generator	1
P2	SS2 AES Wrapper	4
	Noise Generator	1
P3	SS2 AES Wrapper	5
	Noise Generator	2
P4	SS2 AES Wrapper	4
	Noise Generator	1

The percentage of power utilization in Table 4 represents the "SS2 AES Wrapper" total power utilization needed for the FPGA, a value that also contains the percentage of power used by the noise generators in each design. For example, when SS2 AES Wrapper uses 4% and the noise generator uses 1%, then this 1% is included in the total 4% of the wrapper. In Table 4, P3 exhibits more estimated power among the other designs, although being one of the least effective designs created for this experiment. While higher power overhead might sometimes appear as a better metric for an improved AES resistance against CPA attacks, in this case there is no apparent correlation with the previous results. Such a case might be interpreted as an insufficient placement of a noise generator, which cannot overlap with the AES leakage, for example long wires and state registers. Therefore, this particular result is an active highlight that increased power consumption alone cannot stand as a hiding countermeasure, though with alternative spatial alignment more effective results might arise.

8. Conclusion and future research

This thesis focused on designing, implementing and evaluating noise generators as countermeasures against side channel attacks on FPGA AES implementations [6],[8]. A thorough post-implementation design methodology was further described, which utilized the built-in Vivado floorplanning, as well as RapidWright's netlist engineering to embed shift register luts as noise generators into constrained areas in between the AES core [23],[24],[25]. This workflow corresponds to Artix-7 Xilinx FPGA [9],[10]. Contrary to conventional countermeasures which are based on RTL, the specified strategy in this thesis implements countermeasures at a physical implementation level, meaning that there is a further capability to apply existing cryptographic designs with modifications without any intervention in their logic [6],[47].

Next, an experimental assessment of the designs created was conducted using ChipWhisperer Husky [28],[29]. This experiment showed a significant increase in the complexity of Correlation Power Analysis attacks when noise structures are placed in specific locations [2],[21]. In particular, noise integrated in positions close to critical registers and long wires, is able to impact the number of encryption traces (samples) which are needed for a full key recovery [14],[48], for a total of 3500 traces compared to 2000 that are required for a baseline design. Furthermore, a correlation analysis was conducted as well, which proved that average correlation has a direct connection to smearing of peaks or variability in leakage per design [39],[43]. Simply, any increase in switching activity is not a stand alone and sufficient approach to enhancing the design's resistance, since it was shown that designs with large power overhead, but poorly placed generators did not increase the protection [21],[50].

Finally, in terms of hardware cost, the Vivado utilization and power consumption indicated that the countermeasures are able to introduce predictable and usually localized overhead, all of which were constrained within the allocated empty spaces at the beginning of the workflow [11]. An important conclusion at this point is that the AES core was not impacted even when countermeasures had increased power consumption or large noise footprints, as implied by Vivado's reports [9],[10]. Therefore, the proposed countermeasures have been proven to hide the sensitive AES cryptographic core, based on hardware requirements for real world FPGA implementations, while satisfying both timing restrictions and reusability, as well as simplicity of integration for end users [6],[47].

A number of other research directions could be considered, further utilizing similar countermeasures or different types of attacks. For example, the development of automatic placement algorithms can be enforced, which can detect leakage-sensitive regions such as wires or registers, and are capable of adding noise generators wherever needed [14],[15]. Moreover, evaluation of EM side-channel attacks and their leakage models would be fruitful, allowing a complete analysis and design of various countermeasures, which are mostly focused on timing restrictions [41],[54], [55] further expanding the nature of any possible hiding protection. Of course, all of the above can be reinforced by or combined with machine learning algorithms to enhance their capabilities and efficiency [5].

Additionally, an extension of this work would simply involve any exploration regarding the contents of the implemented noise generators, in order to better enhance the complexity of the noise that is directly affecting the AES implementation. For instance, SRLs could contain not only simple hexadecimal values per LUT, but also computations (either lightweight or more complex), such as logic operations, arithmetic units or even pseudorandom transformations.

In summary, this thesis shows that noise inserted at post implementation level requires a level of placement awareness and is a feasible and practical technique for enhancing side-channel protection for FPGAs, while at the same time taking into account architectural restrictions and other constraints that should be considered during the design process of countermeasures [6],[8],[47].

9. Bibliography

[1]. P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," *Advances in Cryptology — CRYPTO' 99*, pp. 388–397, 1999, doi: https://doi.org/10.1007/3-540-48405-1_25.

[2]. E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," *Lecture Notes in Computer Science*, pp. 16–29, 2004, doi: https://doi.org/10.1007/978-3-540-28632-5_2.

[3]. F. X. Standaert, F. Mace, E. Peeters, and J. -J. Quisquater, "Updates on the Security of FPGAs Against Power Analysis Attacks," *Lecture notes in computer science*, pp. 335–346, Jan. 2006, doi: https://doi.org/10.1007/11802839_42.

[4]. O.-X. . Standaert, E. Peeters, G. Rouvroy, and J.-J. . Quisquater, "An Overview of Power Analysis Attacks Against Field Programmable Gate Arrays," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 383–394, Feb. 2006, doi: https://doi.org/10.1109/jproc.2005.862437.

[5]. F.-X. Standaert, S. B. Örs, J.-J. Quisquater, and B. Preneel, "Power Analysis Attacks Against FPGA Implementations of the DES," *Lecture Notes in Computer Science*, pp. 84–94, 2004, doi: https://doi.org/10.1007/978-3-540-30117-2_11.

[6]. T. Güneysu and A. Moradi, "Generic Side-Channel Countermeasures for Reconfigurable Devices," *Lecture Notes in Computer Science*, pp. 33–48, 2011, doi: https://doi.org/10.1007/978-3-642-23951-9_3.

[7]. S. Mangard, E. Oswald, and T. Popp, "Power Analysis Attacks: Revealing the Secrets of Smart Cards", Springer, 2007, doi: https://doi.org/10.1007/978-0-387-38162-6.

[8]. R. Beat, P. Grabher, D. Page, S. Tillich, and M. Wojcik, "On reconfigurable fabrics and generic side-channel countermeasures," vol. 3659, pp. 663–666, Aug. 2012, doi: https://doi.org/10.1109/fpl.2012.6339147.

[9]. "AMD (Xilinx), 7 Series FPGAs Configurable Logic Block User Guide (UG474)" *Amd.com*, 2025. [Online]. https://docs.amd.com/r/en-US/ug474_7Series_CLB/About-This-Guide.

[10]. "AMD (Xilinx), 7 Series FPGA Overview (DS180)," *docs.amd.com, 2020*. [Online]. https://docs.amd.com/v/u/en-US/ds180_7Series_Overview

[11]. "AMD (Xilinx), Vivado Design Suite User Guide: Using the Vivado IDE (UG893)," *Amd.com*, 2025. [Online]. https://docs.amd.com/r/en-US/ug893-vivado-ide.

[12]. "AMD (Xilinx), Vivado UltraScale Libraries Guide — SRL16E Primitive (UG974)," *Amd.com*, 2025. [Online]. https://docs.amd.com/r/en-US/ug974-vivado-ultrascale-libraries/SRL16E.

[13]. "Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL Designs," *Amd.com*, 2013. [Online]. https://docs.amd.com/v/u/en-US/7series_hdl

[14]. I. Giechaskiel, K. Eguro, and K. B. Rasmussen, "Leakier Wires," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 3, pp. 1–29, Sep. 2019, doi: https://doi.org/10.1145/3322483

[15]. Ilias Giechaskiel and Jakub Szefer, "Information leakage from FPGA routing and logic elements," pp. 1–9, Nov. 2020, doi: https://doi.org/10.1145/3400302.3415695.

[16]. Ilias Giechaskiel, Kasper Bonne Rasmussen, and K. Eguro, "Leaky Wires," *arXiv (Cornell University)*, Nov. 2016, doi: https://doi.org/10.1145/3196494.3196518.

[17]. ProjectVault, "orp/hardware/mselSoC/src/systems/geophyte/rtl/verilog/crypto_aes at master · ProjectVault/orp," *GitHub*, 2025. [Online]. https://github.com/ProjectVault/orp/tree/master/hardware/mselSoC/src/systems/geophyte/rtl/verilog/crypto_aes

[18]. J. M. Rabaey, *Digital Integrated Circuits*. 2003.

[19]. A. Moradi, O. Mischke, and T. Eisenbarth, "Correlation-Enhanced Power Analysis Collision Attack," pp. 125–139, Aug. 2010, doi: https://doi.org/10.1007/978-3-642-15031-9_9.

[20]. A Highly Efficient Power Model for Correlation Power Analysis (CPA) of Pipelined Advanced Encryption Standard (AES)," 2025, doi: https://doi.org/10.1109/iscas45731.2020.9180778

[21]. I. Levi, D. Bellizia, D. Bol, and F.-X. Standaert, "Ask Less, Get More: Side-Channel Signal Hiding, Revisited," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4904–4917, Dec. 2020, doi: https://doi.org/10.1109/tcsi.2020.3005338

[22]. S. Sunkavilli, N. Chennagouni, and Q. Yu, "A New Dynamic Countermeasure to Strengthen Design Obfuscation in FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 3, pp. 1–25, Feb. 2025, doi: https://doi.org/10.1145/3716502.

[23]. C. Lavin and A. S. Kaviani, "RapidWright: Enabling Custom Crafted Implementations for FPGAs," *Field-Programmable Custom Computing Machines*, Apr. 2018, doi: https://doi.org/10.1109/fccm.2018.00030.

[24]. "RapidWright Documentation Release 2025.2.0-beta AMD Research and Advanced Development," 2025. [Online]. Available: https://www.rapidwright.io/docs/RapidWright.pdf

[25]. "Design Checkpoints — RapidWright 2025.2.0-beta documentation," *Rapidwright.io*, 2025. [Online]. https://www.rapidwright.io/docs/Design_Checkpoints.html

[26]. "Class Design – RapidWright Javadoc," *Rapidwright.io*, Dec. 2025. [Online]. https://www.rapidwright.io/javadoc/com/xilinx/rapidwright/design/Design.html

[27]. "Package Summary – com.xilinx.rapidwright.design," *Rapidwright.io*, Dec. 2025. [Online]. https://www.rapidwright.io/javadoc/com/xilinx/rapidwright/design/package-summary.html

[28]. "ChipWhisperer-Husky Care & Feeding Instructions Reduced Size Web Version," *NewAE Technology Inc, Mar. 2023.* [Online]. Available: https://media.newae.com/manuals/cwhusky_manual_28march2023.pdf

[29]. NewAE Technology Inc, "ChipWhisperer-Husky," *Newae.com*, 2015. [Online]. https://rtfm.newae.com/Capture/ChipWhisperer-Husky

[30]. "Scope API — ChipWhisperer Documentation," *Readthedocs.io*, 2025. [Online]. https://chipwhisperer.readthedocs.io/en/latest/scope-api.html

[31]. "Overview and Comparison — ChipWhisperer Documentation," *Readthedocs.io*, 2023. [Online]. https://chipwhisperer.readthedocs.io/en/v6.0.0b/Capture/overview.html

[32]. NewAE Technology Inc, "CW312T-XC7A35," *Newae.com*, 2015. [Online]. https://rtfm.newae.com/Targets/UFO%20Targets/CW312T-XC7A35T

[33]. "CW313 - 20-Pin to Card Edge (CW312 Style) Breakout/Adapter Board — ChipWhisperer Documentation," *Readthedocs.io*, 2023. [Online]. https://chipwhisperer.readthedocs.io/en/v6.0.0b/Targets/CW313.html

[34]. "Simpleserial Documentation — ChipWhisperer Documentation," *Readthedocs.io*, 2023. [Online]. https://chipwhisperer.readthedocs.io/en/latest/simpleserial.html

[35]. "Target API — ChipWhisperer Documentation," *Readthedocs.io*, 2023. [Online]. https://chipwhisperer.readthedocs.io/en/latest/target-api.html

[36]. "JetBrains Blog – Developer Tools for Professionals and Teams," *JetBrains Blog*. 2025. [Online]. https://blog.jetbrains.com/

[37]. P. Grabher *et al.*, "An Exploration of Mechanisms for Dynamic Cryptographic Instruction Set Extension," *Lecture Notes in Computer Science*, pp. 1–16, 2011, doi: https://doi.org/10.1007/978-3-642-23951-9_1.

[38]. A. Galip Bayrak, N. Velickovic, F. Regazzoni, D. Novo, P. Brisk, and P. Ienne, "An EDA-Friendly Protection Scheme against Side-Channel Attacks." 2018. [Online]. https://www.epfl.ch/labs/lap/wp-content/uploads/2018/05/BayrakMar13_AnEdaFriendlyProtectionSchemeAgainstSideChannelAttacks_DATE13.pdf.

[39]. A. Heuser, Olivier Rioul, and Sylvain Guilley, "Good Is Not Good Enough," *Lecture notes in computer science*, pp. 55–74, Jan. 2014, doi: https://doi.org/10.1007/978-3-662-44709-3_4.

[40]. "Comparison of side channel analysis measurement setups," *Research portal Eindhoven University of Technology*, 2015. [Online]. https://research.tue.nl/en/studentTheses/comparison-of-side-channel-analysis-measurement-setups/

[41]. F.-X. Standaert and C. Archambeau, "Using Subspace-Based Template Attacks to Compare and Combine Power and Electromagnetic Information Leakages," *Lecture notes in computer science*, pp. 411–425, Aug. 2008, doi: https://doi.org/10.1007/978-3-540-85053-3_26.

[42]. E. Käsper and P. Schwabe, "Faster and Timing-Attack Resistant AES-GCM," *Lecture notes in computer science*, pp. 1–17, Jan. 2009, doi: https://doi.org/10.1007/978-3-642-04138-9_1.

[43]. Y. Yano, K. Iokibe, Y. Toyota, and T. Teshima, "Signal-to-noise ratio measurements of side-channel traces for establishing low-cost countermeasure design," *2017 Asia-Pacific International Symposium on Electromagnetic Compatibility (APEMC)*, Jun. 2017, doi: https://doi.org/10.1109/apemc.2017.7975433.

[44]. F.-X. Standaert, "Introduction to Side-Channel Attacks," *Integrated Circuits and Systems*, pp. 27–42, Dec. 2009, doi: https://doi.org/10.1007/978-0-387-71829-3_2.

[45]. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A Survey of Lightweight-Cryptography Implementations," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 522–533, Nov. 2007, doi: https://doi.org/10.1109/mdt.2007.178.

[46]. H. Kim, T. H. Kim, D.-G. Han, and S. Hong, "Efficient Masking Methods Appropriate for the Block Ciphers ARIA and AES," *ETRI Journal*, vol. 32, no. 3, pp. 370–379, Jun. 2010, doi: https://doi.org/10.4218/etrij.10.0109.0181.

[47]. A. Barenghi, M. Brevi, W. Fornaciari, D. Zoni, and P. Di Milano, "Integrating Side Channel Security in the FPGA Hardware Design Flow." Accessed: Dec. 16, 2025. [Online]. Available: https://re.public.polimi.it/retrieve/e0c31c0f-7496-4599-e053-1705fe0aef77/main.pdf.

[48]. I. Giechaskiel, K. B. Rasmussen, and J. Szefer, "Measuring Long Wire Leakage with Ring Oscillators in Cloud FPGAs," *IEEE Xplore*, Sep. 01, 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8891998

[49]. J.-S. Coron, E. Prouff, and Matthieu Rivain, "Side Channel Cryptanalysis of a Higher Order Masking Scheme," *Lecture Notes in Computer Science*, pp. 28–44, Aug. 2007, doi: https://doi.org/10.1007/978-3-540-74735-2_3.

[50]. D. SUZUKI and M. SAEKI, "An Analysis of Leakage Factors for Dual-Rail Pre-Charge Logic Style," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E91-A, no. 1, pp. 184–192, Jan. 2008, doi: https://doi.org/10.1093/ietfec/e91-a.1.184.

[51]. F. Kocan, L. Li, and D. G. Saab, "Exact Path Delay Fault Coverage Calculation of Partitioned Circuits," *I.E.E.E. transactions on computers/IEEE transactions on computers*, vol. 58, no. 6, pp. 858–864, Jun. 2009, doi: https://doi.org/10.1109/tc.2008.205.

[52]. O. Choudary and M. G. Kuhn, "Efficient Template Attacks," *Lecture notes in computer science*, pp. 253–270, Jan. 2014, doi: https://doi.org/10.1007/978-3-319-08302-5_17.

[53]. R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, "Deep learning for side-channel analysis and introduction to ASCAD database," *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, Nov. 2019, doi: https://doi.org/10.1007/s13389-019-00220-8.

[54]. M. Wiener, *Advances in cryptology--CRYPTO '99 : 19th Annual International Cryptology Conference, Santa Barbara, California, USA August 15-19, 1999 proceedings*. Berlin: Springer, 1999.

[55]. A. Sayakkara, N.-A. Le-Khac, and M. Scanlon, "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics," *Digital Investigation*, vol. 29, pp. 43–54, Jun. 2019, doi: https://doi.org/10.1016/j.diin.2019.03.002.