UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ - ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc "Cybersecurity and Data Science"

ΠΜΣ 'Κυβερνοασφάλεια & Επιστήμη Δεδομένων'

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Evaluating Fault Attack Vulnerabilities in Lightweight Cryptographic HLS Implementations.** Αξιολόγηση Ευπαθειών Επιθέσεων Σφαλμάτων σε Ελαφροβαρείς Κρυπτογραφικούς Αλγορίθμους Σχεδιασμένους για HLS.
Student's Name - Suranme: Ονοματεπώνυμο Φοιτητή:	**Theodoros Kollias** Θεόδωρος Κόλλιας
Father's Name: Πατρώνυμο:	**Eleutherios** Ελευθέριος
Student's ID No: Αριθμός Μητρώου:	**MPKED21024** ΜΠΚΕΔ21024
Supervisor: Επιβλέπων:	**Michael Psarakis, Professor** Μιχαήλ Ψαράκης, Καθηγητής

Piraeus, November 2024 / Πειραιάς, Νοέμβριος 2024

3-member Examination Committee

Τριμελής Εξαταστική Επιτροπή

Panagiotis Kotzanikolaou **Professor**	**Michael Psarakis** **Professor**	**Athanasios Papadimitriou** **Assistant Professor**
Παναγιώτης Κοτζανικολάου Καθηγητής	Μιχαήλ Ψαράκης Καθηγητής	Αθανάσιος Παπαδημητρίου Επίκουρος Καθηγητής

Acknowledgements

I would like to sincerely thank my thesis advisor, Assistant Professor Athanasios Papadimitriou, for his guidance and support throughout my postgraduate studies. His feedback and advice were essential in helping me complete my thesis.

I am also very grateful to PhD candidate Amalia Koufopoulou for her assistance and valuable input during the research process. Her help was crucial in overcoming many challenges along the way.

Lastly, I want to thank my family for their constant support, understanding, and patience. Their encouragement has meant a lot to me during this journey.

Abstract

Today's exponential demand for lightweight applications requires the reduction of time-to-market while maintaining high security standards in hardware-oriented applications, particularly within embedded systems. High-Level Synthesis (HLS) tools have emerged as essential enablers in this process, allowing developers and designers to utilize High-Level Languages (HLL) alongside optimization strategies to produce Hardware Description Language (HDL) code. This approach accelerates hardware development, facilitates early verification and validation. Ultimately, it contributes to a more efficient design flow. However, the security implications of HLS-generated designs, especially their vulnerability to fault attacks, remain under-explored. This thesis delves into the fault attack vulnerabilities of four(4) prominent lightweight cryptographic algorithms ("GIFT-64-128", "LED-64", "KATAN32" & "SIMON64/128") implemented through the HLS workflow. Through a series of statistical fault injection experiments, we evaluate the robustness of these algorithms. The findings provide insights into the trade-offs between design efficiency and security in HLS-generated cryptographic hardware. Although HLS tools can boost the design process, consideration of synthesis configurations is essential to mitigate vulnerabilities and ensure robust protection against hardware-based attacks.

Keywords:
Hardware Security, High-level Synthesis (HLS), Fault Injection Attack, Lightweight Cryptography

Περίληψη

Σήμερα, η εκθετική ζήτηση «ελαφρών» εφαρμογών απαιτεί μείωση του χρόνου διάθεσής τους στην αγορά, διατηρώντας παράλληλα υψηλά επίπεδα ασφαλείας σε εφαρμογές προσανατολισμένες στο υλικό, ιδιαίτερα σε ενσωματωμένα συστήματα. Τα εργαλεία Σύνθεσης Υψηλού Επιπέδου (HLS) έχουν αναδειχθεί ως βασικοί κινητήρες σε αυτή τη διαδικασία, επιτρέποντας σε προγραμματιστές και σχεδιαστές να χρησιμοποιούν γλώσσες υψηλού επιπέδου παράλληλα με στρατηγικές βελτιστοποίησης για την παραγωγή κώδικα γλώσσας περιγραφής υλικού. Αυτή η προσέγγιση επιταχύνει την ανάπτυξη υλικού, διευκολύνει την έγκαιρη επαλήθευση και επικύρωση. Συνολικά, συμβάλλει σε μια πιο αποτελεσματική ροή σχεδιασμού. Ωστόσο, οι επιπτώσεις στην ασφάλεια των σχεδίων που δημιουργούνται από το HLS, ειδικά η ευπάθειά τους σε επιθέσεις εισαγωγής σφαλμάτων, παραμένουν αρκετά ανεξερεύνητες. Η διατριβή αυτή προσπαθεί να εμβαθύνει στα ευπαθή σημεία επίθεσης σφάλματος τεσσάρων(4) ελαφροβαρών κρυπτογραφικών αλγορίθμων ("GIFT-64-128", "LED-64", "KATAN32" & "SIMON64/128") υλοποιημένους μέσω HLS. Μέσω πειραμάτων επιθέσεων στατιστικής εισαγωγής σφαλμάτων, αξιολογούμε την ευρωστία αυτών των αλγορίθμων. Τα ευρήματα παρέχουν πληροφορίες για τις ανταλλαγές μεταξύ αποδοτικότητας σχεδιασμού και ασφάλειας σε κρυπτογραφικό υλικό που δημιουργείται από HLS. Όσο και αν τα HLS εργαλεία μπορούν να ενισχύσουν τη διαδικασία σχεδιασμού, η εξέταση των στοιχείων σύνθεσης είναι απαραίτητη για τον μετριασμό των ευπαθιών και τη διασφάλιση ισχυρής προστασίας από επιθέσεις σε υλικό.

Λέξεις Κλειδιά:
Ασφάλεια Υλικού, Σύνθεση Υψηλού Επιπέδου (HLS), Επίθεση Εισαγωγής Σφαλμάτων, Ελαφροβαρής Κρυπτογραφία

Glossary

AE: Authenticated Encryption
AEAD: Authenticated Encryption with Associated Data
AES: Advanced Encryption Standard
ASIC: Application Specific Integrated Circuit

CAD: Computer Aided Design
CDFG: Control/Data-Flow Graph

DC: Differential Cryptanalysis
DES: Data Encryption Standard
DFG: Data-Flow Graph

ECAD: Electronic Computer-Aided Design
EDA: Electronic Design Automation

FIA: Fault Injection Attack
FIPS: Federal Information Processing Standards
FPGA: Field Programmable Gate Array

HDL: Harware Desciption Language
HLL: High-level Languages
HLS: High-level Synthesis

IC: Integrated Circuit
I/O: Input/Output
IEEE: Institute of Electrical and Electronics Engineers
IoT: Internet of Things

LC: Linear Cryptanalysis
LED: Light Encryption Device
LFSR: Linear Feedback Shift Register
LSB: Least Significant Bit
LUT: LookUp Table

MAC: Message Authentication Code
MDS: Maximum Distance Separable
MSB: Most Significant Bit

NIST: National Institute of Standards and Technology
NSA: National Security Agency

PCB: Printed Circuit Boards

PLL: Phase-Locked Loop
PoP: Package-on-Package

RFID: Radio Frequency Identification
RTL: Register Transfer Level

SDC: Silent Data Corruption
SWiFI: Software Implemented Fault Injection
SFI: Statistical Fault Injection
SPN: Substitution-Permutation Network

TDEA: Triple Data Encryption Algorithm

VHDL: VHSIC Hardware Description Language
VHSIC: Very High Speed Integrated Circuit
VLSI: Very Large Scale Integrated

XOR: Exclusive OR

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security, robustness and the resilience of a circuit to disruptions can be assessed through different methods after production. However, evaluating potential impact of soft errors during the design stage to prevent encountering undesirable behaviors later during qualification, plays crucial role. If the safety and reliability requirements are not taken into account during the initial stages of the design, its modification at a later stage of the production process requires additional study and correction time, resources and materials. This greatly increases the fixed cost of production. Especially in cases where the hardware cannot be modified and the manufacturer is asked to choose between keeping the vulnerable hardware or redesigning it at a cost.

In particular, security and reliability are related with various stages of the production process. From design, to testing, to implementation to its final release. Whether the predefined controls are applied during the production process largely determines the level of security and reliability that will be achieved. As at the software level, access to critical system data should also be prohibited at the hardware level.

Ensuring hardware integrity is another pillar of system's security. It is crucial that it is designed in such way that it recognizes its elements and in case of any change (e.g. attempt to insert a fake part) to alert and stop its operation. Hardware availability is an indicative element of system's security as well. This implies unhindered access to necessary system data as well as the internal code of the hardware by the system owner or authorized user. The data to which access is allowed, are determined according to the restrictions set by the system's security policy.

However, in order to mitigate the threats, it is necessary to know what are the main vulnerabilities of the hardware. As mentioned previously, there are many stages of the production chain and all of them can have vulnerabilities. Hardware architecture plays a key role. During design, failure to apply predefined and standardized methods results in the subsequent creation of functional errors. These errors become apparent through the observation of the operation, where any anomalies that occur are sensitive points, potentially vulnerable. Deliberately driving the system into abnormal operating conditions may cause the system to crash or reveal internal information. Nevertheless, many times, this can happen unintentionally from the system.

Fault Injection is now a standard technique for conducting dependability analysis. A fault injection campaign involves comparing the circuit's expected behavior for a specific workload (i.e., the correct behavior verified by the designer) with the behavior observed when each fault from a predefined set is introduced.

Due to the very large size of the contemporary designs, it is often impractical to inject all potential errors across every location and clock cycle within a reasonable time frame. As a result, most research studies rely on Statistical Fault Injection (SFI). While in a SFI campaign, only a random portion of potential errors is introduced, with the selection based on the target of the injection.

1.1 Thesis Objective & Outline

This Thesis's objective, titled *"Evaluating Fault Attack Vulnerabilities in Lightweight Cryptographic HLS Implementations"*, is to investigate the fault resilience of lightweight cryptographic algorithms when synthesized through High-Level Synthesis (HLS) workflows. In response to the rising demand for efficient yet secure cryptographic solutions in embedded systems, this research aims to examine fault attack vulnerabilities of four(4) selected algorithms—GIFT-64-128, LED-64, KATAN32, & SIMON64/128—implemented using HLS. Through a series of Statistical Fault Injection (SFI) experiments, the Thesis evaluates each algorithm's security, identifying critical errors, hang errors, and silent errors that may arise under fault conditions. To describe the implementations and analyze the results, this Thesis is divided in nine(9) chapters, as follows.

• In Chapter 1, titled *"Introduction"*, research objectives, scope and structure of the Thesis are outlined.
• In Chapter 2, titled *"High-Level Synthesis (HLS)"*, it is provided an overview of HLS and Electronic Design Automation (EDA) tools. There also basic knowledge about Hardware Description Languages (HDL), specifically VHDL, with an emphasis on Vivado HLS as a tool used for generating HDL code.
• In Chapter 3, titled *"Cryptographic Acceleration"*, the Thesis examines basic characteristics of Lightweight Cryptography, covering both Stream and Block Ciphers.
• In Chapter 4, titled *"NIST Cryptographic Standards & Guidelines"*, NIST's influence on Lightweight Cryptography, particularly their Block Cipher essential standards for secure cryptographic design in constrained environments, are reviewed.
• In Chapter 5, titled *"Hardware Attacks & Techniques*, there is a categorization of Hardware Attacks, focusing on Fault Injection Attacks, including simulation-based and statistical methods, which are relevant to this Thesis.
• In Chapter 6, titled *"Ciphers Overview"*, there is a detailed analysis of the four(4) selected cryptographic algorithms, covering their specifications and their security posture.
• In Chapter 7, titled *"Ciphers Implementation"*, the HLS-based implementation of each cipher is presented, focusing on their slice logic, memory and power usage.
• In Chapter 8, titled *"Results Analysis"*, the results from these Fault Injection tests are presented and documented. There are data and analysis for each cipher.
• In Chapter 9, titled *"Conclusion"*, Thesis summarizes key findings, discussing observed Critical Errors, Hang Errors, and Silent Errors across the implementations. Chapter 9 closes with recommendations for further future research.

The final section, titled *"Bibliography"*, lists all the references of this Thesis.

Chapter 2

High-Level Synthesis (HLS)

2.1 Overview

An **High-Level Synthesis (HLS)** tool takes a program written in a High-Level Language (HLL), typically C/C++, and **automatically generates a design** in Register Transfer Level (RTL) written in a Hardware Description Language (HDL) that **retains the same functionality**. High-Level Synthesis has gained popularity by bringing the **simplicity of high-level abstraction** into the design process through behavioral synthesis. This approach allows individuals with basic HLL knowledge to leverage hardware programming benefits without needing to delve into implementation details. Additionally, HLS significantly **reduces human error** which results in more flexible programs that are much easier to debug and maintain [1].

Automatically converting from a HLL to a HDL, shifts tasks away from designers that would otherwise be their responsibility. The specific tasks and their sequence can vary significantly between tools. The fundamental tasks are outlined below, as described in [2].

***Data-Flow Graph (DFG)*:** A general representation of the design's inputs and outputs, the operations and their interoperability, and the shaping of data-flow based on data dependencies. This step can be expanded to a Control/Data-Flow Graph (CDFG) to incorporate control flow, including conditional cases.

***Resource Allocation*:** Determines the resources necessary to implement the design, based on information from the previous step. These resources should be sufficient to perform each defined operation but optimized to minimize reliance on more complex components.

***Scheduling*:** Assigns operations to clock cycles based on the timing characteristics of both the hardware and the design. This is a crucial step in behavioral synthesis which can be categorized as either *Scheduling for unconstrained designs* or *Scheduling for constrained designs*, depending on the chosen methodology.

***Register Allocation*:** Analyzes scenarios where registers are necessary, such as when data dependencies extend beyond a single cycle. Lifetime analysis may also be applied to assess data validity over time, enabling register sharing to minimize the number of required registers.

***Binding*:** Allocates scheduled operations to corresponding hardware resources.

***State Machine Extraction*:** Derived from the scheduling step, defines how data is routed into components based on the current state.

***Netlisting*:** Produces the synthesized design, which can also be performed alongside other steps to provide a form of verification.

Interface Synthesis determines the structure of the top module's interface within a design, encompassing both I/O and control signals necessary for communication with other modules and peripherals. It organizes internal functions into distinct blocks, establishing a hierarchical arrangement of modules and entities. While the process is largely automated, most HLS tools offer designers some control to optimize the design and fully leverage hardware capabilities. This approach allows minimal effort early-stage testing which facilitates design exploration.

However, it is important to emphasize that while most HLS tools help designers in navigating the complexities of hardware application development, they must not lose sight of the fact that the **final implementation will occur in hardware**, which comes with its own specific limitations. Designers must carefully consider the **transition from sequential to concurrent logic** throughout the design process. Also, **memory management** differs in hardware. For instance, high-level techniques like dynamic memory allocation which are advantageous in software, do not synthesize well or not at all in hardware. This includes methods such as dynamic programming, pointer usage, recursion, and system calls [1].

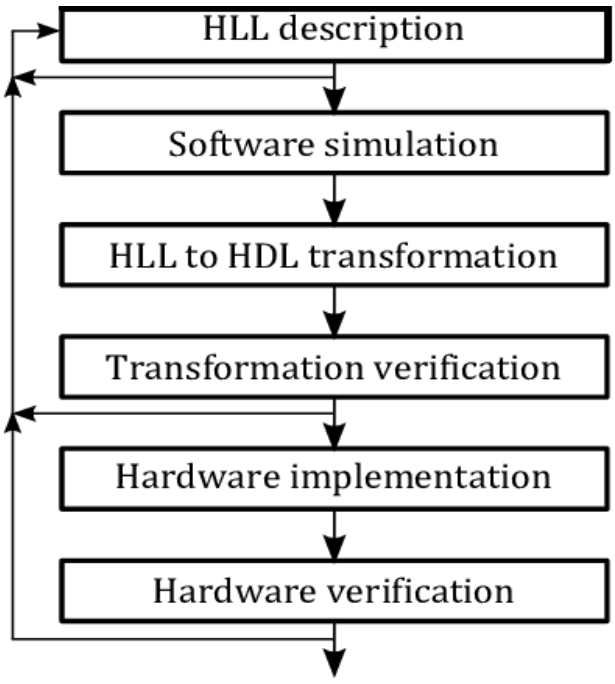

Figure 2.1: Generic HDL Design Flow [3]

2.2 EDA Tools

Electronic Design Automation (EDA), also known as Electronic Computer-Aided Design (ECAD), is a specific category of software tools and services that enable engineers to design electronic systems and products, such as Printed Circuit Boards (PCB) and Integrated Circuits (IC). EDA tools collaborate within a design flow that chip designers utilize to create and evaluate complete semiconductor chips. Given that modern semiconductor chips can contain billions of components, these tools are critical for their design. They enable designers to **model, simulate, test and analyze circuit designs** to detect potential problems before production begins.

Prior to the development of EDA, Integrated Circuits (IC) were designed by hand and manually laid out [4]. By the mid-1970s, developers began automating circuit design alongside drafting, leading to the creation of the first placement and routing tools. As designs have become more complex, EDA software has gained significant importance for developers and designers working on Printed Circuit Boards (PCB) and other types of circuit boards. EDA software tools primarily focuses on [5]:

$$\Rightarrow \textit{Design}$$
$$\Rightarrow \textit{Simulation}$$
$$\Rightarrow \textit{Analysis \& Verification}$$
$$\Rightarrow \textit{Manufacturing Preparation}$$

Nowadays, EDA has become increasingly important due to the **continuous scaling of semiconductor technology** [6]. Modern digital design flows are highly modular, where front-end processes generate standardized design descriptions that compile into units resembling cells, irrespective of their underlying technology. These cells perform logic or other electronic functions using specific IC technologies. Manufacturers typically offer component libraries tailored to their production processes, complete with simulation models compatible with standard simulation tools. Finally, EDA increases **design reusability** simplifying the design process.

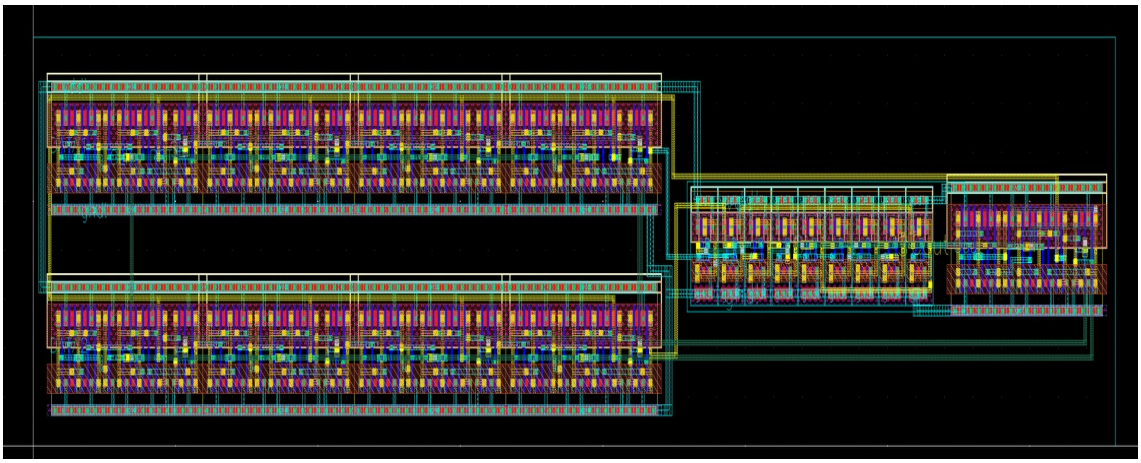

Figure 2.2: 4-Bit Adder Layout with “Cadence” EDA tool

2.3 Hardware Description Languages

In computer engineering, a **Hardware Description Language (HDL)** is a specialized language for describing the structure and behavior of electronic circuits. HDLs provide **text-based descriptions** of the architecture and functionality of electronic systems, allowing designers to specify how the hardware should operate over time. Unlike most software programming languages, their syntax and semantics include **explicit notations** for expressing concurrency and incorporate the concept of time, a key characteristic of hardware. They can **describe designs at various abstraction levels**, such as structural, behavioral or Register Transfer Level (RTL), while maintaining the same circuit functionality.

Some points worth keeping in mind about HDLs, as described in [7]:
• **Hierarchical Design**: Complex circuits can be designed in sections, consisting of modules and sub-modules.
• **Modular Architecture**: Modifying a fixed module should not affect other modules within the design.
• **Text-based Explanation**: Standard parsing concepts can be used to handle depiction rather than relying on pictorial representation.
• **Reprocess of Obtainable Resources**: Existing resources should be reused to save energy and time.

An HDL provides a **precise and formal way to describe an electronic circuit**, enabling **automated analysis and simulation**. It also supports the synthesis of an HDL description into a netlist, specifing electronic components and their connections. This netlist can then be used for placement and routing, ultimately producing the masks required to fabricate an Integrated Circuit (IC).

Worth noting that HDLs are an essential component of EDA systems, particularly for designing ASICs, FPGAs, and microprocessors, which are regarded as complex circuits.

2.3.1 Hardware Description Language VHDL

VHDL is a Hardware Description Language (HDL) which is used to develop integrated digital systems and circuits. The term VHDL is short for "VHSIC Hardware Description Language", where VHSIC is an acronym for "Very High Speed Integrated Circuit". It was developed in the early 1980s with funding from the US Department of Defense, and was standardized by the IEEE Institute as IEEE 1076 in 1987.

When it was created, it was intended for **modeling and simulating circuits** but later it was also used as a **synthesis tool**. During synthesis, the compiler constructs a circuit that accurately corresponds to the logic and timing description that the code models.

Today, VHDL code is the **primary input file type** that digital circuit design (e.g. CAD) software accepts, enabling the creation of complex integrated circuits. In addition, VHDL language is widely used for describing and implementing digital systems in reconfigurable logic devices, such as Field Programmable Gate Arrays (FPGAs).

Finally, an important advantage of VHDL is it's independence from companies and technologies [8]. This essentially means that VHDL code can be "ported" without limiting designers on how to implement his system.

2.4 Vivado HLS

Vivado HLS, version 2020.1, is the used tool for the needs of this Thesis. It is included in the **Vivado Design Suite** developed by Xilinx, enabling developers and designers to implement high-level algorithms in programmable logic through an advanced design flow. Vivado HLS incorporates key technologies from processor compilers for interpreting, analyzing, and optimizing C/C++ programs. The primary distinction lies in the target platform for executing the application.

To guide the Vivado HLS compiler in generating an optimal processing architecture, a solid understanding of hardware design principles is essential and needed. A Vivado HLS project includes the **high-level code**, a **testbench file** that provides the inputs for code testing , and at least one **solution** (e.g. sol_0) where the project's constraints and optimization details are specified. The solutions can be then synthesized, generating the corresponding functionality in VHDL or Verilog files, along with various performance reports.

From **Synthesis** perspective, the code of any HLS-generated module requires a software testbench, providing also debug option. The software testbench serves the following important functions as mentioned in *UG998 (v1.1)* [9]:
• Ensuring that the software intended for FPGA implementation operates correctly and does not produce a segmentation fault. Segmentation faults are a concern in HLS just as they are with any other compiler. The most common cause of this error is a user program attempting to access a memory location linked to a pointer address before the memory has been allocated and associated with the pointer.
• Demonstrating the functional correctness of the algorithm. For the produced hardware implementation, the HLS compiler ensures only functional equivalence with the original C/C++ code. Consequently, a robust software testbench is essential to reduce the efforts required for hardware verification and validation.

Extra **Optimizations** can be also applied to the code, in the form of directives. **Directives** can specify various aspects, such as minimum or maximum latency for a code block, resources usage for variable handling (e.g., selecting array's implementation memory types), imposing limitations on resources, loops implementation (e.g., unrolled or merged), dataflow adjustments, I/O signals management etc.. The ultimate goal of all the aforementioned optimizations is the enhancement performance of the code. Finally, apart from user interface, Vivado HLS compilation can be managed through a Tool Command Language (Tcl) script file [9].

Chapter 3

Cryptographic Acceleration

3.1 Overview

Since ancient times, humans have needed to protect their communication by hiding their messages. A need arose for secure and reliable communication, especially when it came to military and administrative matters [10]. To this day, the core concept remains unchanged: to discover effective methods of concealing information transmitted by any means, while ensuring that only the intended recipients possess the knowledge to reveal it.

Key terms in cryptography include the **plaintext**, the **ciphertext** and **cryptographic key**. The plaintext is the original information intended to be transmitted, while the ciphertext is the encrypted version of that plaintext. Cryptographic key is a secret and crucial piece of information utilized as a parameter in a cipher to convert plaintext into ciphertext during the encryption process. The security of the entire process does not relies on the confidentiality of the algorithm, but on the secrecy of the key. The quality of the key, which depends on factors like its generation process and its size, are vital. The key is also essential for decryption, the reverse process that restores the original information.

In computing, a **cryptographic accelerator** is a co-processor specifically designed to execute cryptographic operations which are computationally intensive. Developers must also carefully consider how and where cryptographic algorithms are processed. If the system's main CPU handles the computationally heavy cryptographic tasks, it reduces the available processing power for user applications. Hardware-based accelerators dedicated to cryptographic tasks can offload most of this processing from the CPU, performing the cryptographic operations much more efficiently than a general-purpose CPU.

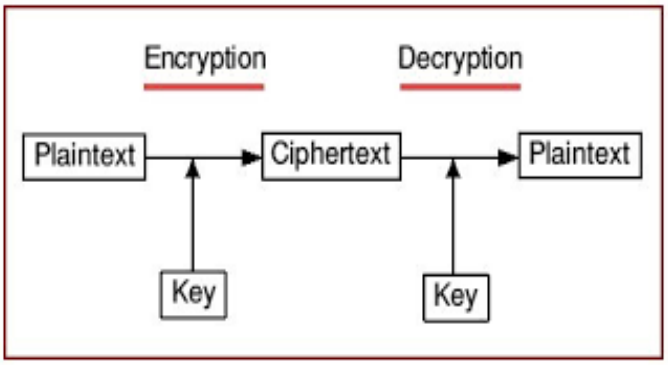

Figure 3.1: Encryption & Decryption Process [11]

3.2 Lightweight Cryptography

Cryptographic algorithms, based on the type of keys they use for encryption and decryption, are divided into two main categories. These are the following:

- ***Symmetric or Secret Key*:** In the case of Symmetric or Secret Key Algorithms, the sender and the recipient use the same key for encryption and decryption process respectively. The sender sends the ciphertext to the recipient, once he has encrypted the plaintext with the key. The recipient in turn, after receiving the ciphertext, applies the same key and decrypts the message, thus recovering the original text. An important part of the process is the security and secrecy of the key between sender and receiver. If by any means, an attacker succeeds and discovers the key and has knowledge of the algorithm, then he can gain access to all transmitted messages and data. Important subclasses of symmetric algorithms are Stream Ciphers and Block Ciphers [12].
- ***Asymmetric or Public Key*:** In Asymmetric of Public Key Algorithms, both the sender and the recipient have their own pair of keys. One "public" and one "private" key. The former is used during the encryption process and is known to everyone, while the latter is known only to the owner of the key itself and is used during the decryption process. In order to communicate, the sender uses the recipient's public key and encrypts the message. The recipient, after receiving the encrypted message, uses his own private key to decrypt the message and read the original text. The success of this method is based on the fact that each user's private key is mathematically linked to their public key. Therefore, the stronger the mathematical transformation, the harder is to reveal the original message without the right key.

Cryptographic algorithms, whether symmetric or public key, are vulnerable to Fault Injection Attacks (FIA).

3.2.1 Stream Ciphers

Stream Ciphers are subclass of Symmetric Cryptographic Algorithms. Their operation is based on the encryption of the original text into separate individual pieces, usually in the size of bit or byte. Their main advantage is their **greater speed** compared to the rest of the symmetric algorithms, while maintaining relatively **simple hardware implementation**. Furthermore, the fact that the encryption is applied to independent pieces of the plaintext, makes them **less vulnerable to error propagation** [13].

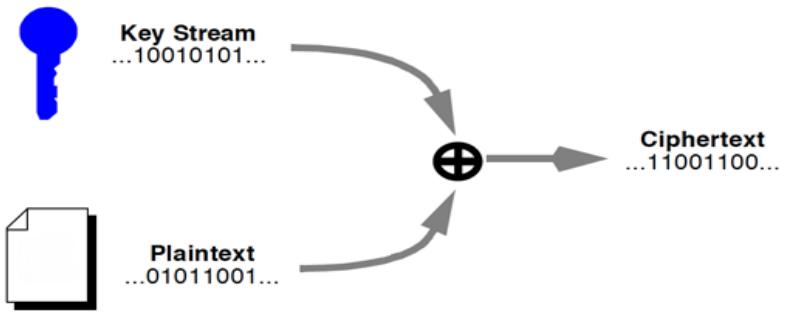

Figure 3.2: Stream Ciphers Encryption Process [14]

3.2.2 Block Ciphers

Block Ciphers are also subclass of Symmetric Cryptographic Algorithms. The plaintext is segmented into blocks of a fixed length which are inputted separately to the encryption function, and output blocks of usually the same length. A larger block size has **better resist** against statistical analysis. For the encryption process they use transposition techniques and they are **slower** than stream ciphers.

The two main properties of block ciphers are confusion and diffusion. **Confusion** is obscuring the relationship between plaintext and ciphertext, aiming to make statistical analysis difficult, even when an attacker has access to numerous plaintext/ciphertext pairs. It protects the link between the ciphertext and the key. **Diffusion**, on the other hand, ensures that the influence of individual plaintext characters spreads across much of the ciphertext. This means each output bit is affected by several input bits, helping to obscure the statistical relationship between the plaintext and ciphertext.

Operation Modes define how messages, larger than the block size, are encrypted. They are essential for ensuring the security of the encryption process. Each mode processes data differently, having its own advantages and disadvantages. Block Cipher Encryption Modes, as described by [15]:

$\Rightarrow$ *Electronic Code Book (ECB)*
$\Rightarrow$ *Cipher Block Chaining (CBC)*
$\Rightarrow$ *Cipher FeedBack (CFB)*
$\Rightarrow$ *Output FeedBack (OFB)*
$\Rightarrow$ *Counter Mode (CTR)*

Many algorithms in use today, implement multiple rounds of simple substitution and permutation, forming an iterated block cipher, called Substitution-Permutation Network (SPN). In SPNs, confusion is provided by substitution boxes [16].

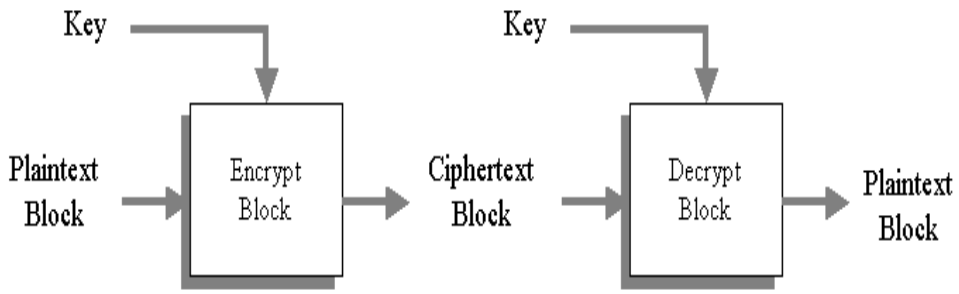

Figure 3.3: Block Ciphers Generic Flow [17]

Chapter 4

NIST Cryptographic Standards & Guidelines

4.1 Overview

NIST stands for *"National Institute of Standards and Technology"* part of the U.S. Department of Commerce. This government laboratory focuses on developing, testing, and recommending best practices for federal agencies and other organizations, particularly in areas like cybersecurity.

NIST creates cybersecurity standards, guidelines, best practices, and various resources to address the needs of U.S. industry, federal agencies, and the wider public [18]. It's activities range from generating specific information that organizations can implement right away to conducting longer-term research that anticipates advancements in technologies and future challenges. Priority areas to which NIST contributes include emerging technologies, identity and access management, risk management, trustworthy networks and platforms, education and workforce development, privacy, and of course cryptography. [18].

NIST Cybersecurity Framework can be implemented in these five(5) areas [19]:

$\Rightarrow$ *Identify*
$\Rightarrow$ *Protect*
$\Rightarrow$ *Detect*
$\Rightarrow$ *Respond*
$\Rightarrow$ *Recover*

NIST has promoted the advancement of cryptographic techniques and technologies through an open process that unites industry, government, and academia. This collaborative approach aims to develop practical cryptographic solutions that ensure effective security.

Today, NIST's cryptographic solutions are extensively utilized in commercial applications, ranging from tablets and smartphones to ATMs. They secure e-commerce globally, protect U.S. federal information, and even safeguard classified federal data. It is advancing lightweight cryptography to meet security needs for circuits that are smaller than what was imagined just a few years ago. It's validation of robust algorithms and implementations fosters confidence in cryptography, thereby enhancing its adoption to ensure the privacy and security of individuals and businesses [20].

NIST continues to spearhead public collaborations for the advancement of modern cryptography, including the following categories, as described in [20]:

- **Lightweight Cryptography.** Suitable for small devices like Internet of Things (IoT) devices and other resource-constrained platforms that may struggle with current cryptographic algorithms.
- **Block Ciphers.** Data encryption in block-sized chunks is effective when dealing with large amounts of data.

4.2 NIST's Lightweight Cryptography

NIST started exploring cryptography for constrained environments in 2013 [21]. Following two workshops and consultations with industry, government, and academic stakeholders, NIST launched a process to "solicit, evaluate, and standardize schemes" [21] that offer Authenticated Encryption with Associated Data (AEAD) and optional hashing capabilities for resource-limited environments where the performance of existing NIST cryptographic standards is inadequate. In 2018, NIST released a call for algorithms detailing the selection process, requirements and the evaluation criteria [21].

Round 1. The initial round of the NIST lightweight cryptography standardization process. In March 2019, NIST received fifty-seven(57) submissions for consideration. In April 2019, it commenced with the announcement of fifty-six(56) candidates and concluded in August 2019. "NISTIR 8268" details the evaluation of the first-round candidates and identifies thirty-two(32) algorithms that advanced.

Round 2. NIST's second round of the standardization process commenced with the announcement of thirty-two(32) candidates in August 2019. By March 2021, it was concluded with the announcement of the finalists. The evaluation of the second-round candidates and the ten(10) finalists are provided in "NISTIR 8369".

Final Round. The final round of the process began with the announcement of the ten (10) finalists. It was concluded with NIST's selection of the Ascon family in February 2023. "NISTIR 8454" outlines the evaluation of the finalists, providing extra details on the selection process.

4.3 NIST's Block Cipher Techniques

At present, there are two(2) approved block cipher algorithms that can be utilized for both applying or removing cryptographic protection, or verifying the previously applied protection. These are AES and Triple DES [22]. Before we explain more about the aforementioned algorithms, it would be useful to understand FIPS, short for "Federal Information Processing Standards". The FIPS of the United States comprise a set of publicly announced standards developed by NIST for use in computer systems of non-military U.S. government agencies and contractors [23]. These standards establish requirements for ensuring computer security and interoperability, nd are designed for situations where appropriate industry standards do not exist [23].

Advanced Encryption Standard (AES) is outlined in "FIPS 197", and was approved in November 2001. AES must be utilized with modes of operation that are specifically designed for block cipher algorithms [22]. This standard also identifies that AES, which is variant of the Rijndael algorithm, can be employed by U.S. Government organizations and others to protect sensitive information.

Triple DES is detailed in "SP 800-67 Revision 2", titled "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher," which received approval in November 2017 [22]. TDEA must be implemented using suitable modes of operation that are designed for block cipher algorithms [22].

Chapter 5

Hardware Attacks & Techniques

5.1 Hardware Attack Categories

Before we proceed, it is essential to examine the different types of hardware attacks. These attacks, based on the degree of physical intrusion, are fundamentally categorized into three categories [24]:

- **Invasive Attacks.** They involve physically opening the Integrated Circuit (IC) package to expose the silicon die using chemical techniques. This allows direct observation of signals and intermediate values through specialized instruments. However, such attacks demand costly equipment, technical expertise, and carry a significant risk of permanently damaging the device.

- **Semi-invasive Attacks.** They also involve opening the IC package, but without directly contacting the internal wires. Instead, methods such as optical lasers are used to manipulate intermediate values, lowering the risk of physical damage compared to invasive attacks. Certain fault injection techniques fall into this category

- **Non-invasive Attacks.** They exploit vulnerabilities without directly accessing the silicon die, making them less intrusive, yet effective in compromising security. Attackers maintain minimal contact with the IC chip pins, using techniques such as Side-Channel Analysis and certain Fault Injection Attacks are classified as non-invasive.

Furthermore, hardware attacks can be categorized into active and passive attacks, based on the direction of physical information flow:

- **Active Attacks.** In active attacks, the attacker actively disrupting the device's operations to alter intermediate values. Fault injection attacks are examples of active attacks.

- **Passive Attacks.** In passive attacks, the attacker focuses on collecting data from the physical device without altering its operation. A common example is side-channel attacks, where attackers analyze power consumption patterns to reveal sensitive information, such as encryption keys.

5.2 Fault Injections Attacks

Fault Injection Attack (FIA) is a common attack vector used in both hardware and software attacks on computer systems. In hardware attacks, fault injections involve creating **physical disturbances**, which can cause incorrect instructions to be executed or faulty values to be stored in memory. In software attacks, **unexpected input values** are introduced to expose possible vulnerabilities.

A successful Fault Injection Attack can **bypass security mechanisms**, including secure boot processes. The attacker induces a fault during algorithm execution and subsequently uses various analysis techniques to exploit information from the altered execution. When combined with cryptanalysis, this can lead to cryptographic attacks, potentially enabling the recovery of encryption keys. Certain fault injection techniques are preferred for their ability to precisely control the injection process, either in terms of timing (temporal) or location (spatial). Some attacks are able to accurately target a specific moment (temporal precision) or a certain memory register (spatial precision) [25].

Clock Fault Injection is considered a global attack because a clock glitch can affect multiple components simultaneously. Despite its broad impact, it is technically simple as noninvasive attack and it allows precise control over timing. This technique operates by adding extra rising edges to a device's input clock, thereby disrupting the target's timing constraints. This is achieved by inserting clock edges that are either too narrow or too wide. However, the primary limitation of Clock Fault Injection is its ineffectiveness against devices that utilize internal oscillators or a Phase-Locked Loop (PLL) to generate a new clock from the external signal, as these mechanisms can reduce the impact of the injected glitches.

Electromagnetic (EM) Fault Injection utilizes a powerful electromagnetic pulse to induce faults, providing more localized effects necessitating advanced equipment. It is categorized as semi-invasive attack as it might pose risks to the IC. A common method for inducing an electromagnetic (EM) glitch involves generating a strong electromagnetic pulse by creating a changing magnetic field through a wire loop. According to Faraday's law, a changing magnetic field induces voltage spikes in the wires of a chip, potentially causing signal levels to flip from 1 to 0 or vice versa. This process becomes more difficult with Package-on-Package (PoP) technology, where the memory die is stacked on top of the processor die, complicating access. Despite its complexity, EM glitching allows precise control over specific registers, making it a potent method for targeted attacks.

Optical Fault Injection is as a technique known for its unique spatial and temporal precision. It involves using laser beams or other intense light sources to induce faults into target device. Transistors can change states when exposed to an optical pulse, as semiconductors are light-sensitive. However, the complexity and cost of this attack rise considerably if the target IC chip needs to be decapsulated, a process that typically involves the use of acid and specialized equipment.

Voltage Fault Injection, compared to Clock Fault Injection, offers greater temporal precision and practicality, while also requiring significantly less complex equipment than Electromagnetic (EM) or Optical Fault Injection methods. This technique involves intentionally modifying a chip's power supply by briefly lowering the voltage or introducing a positive or negative power spike. This manipulation can disrupt the chip's normal operation during critical tasks, potentially causing faults that attackers can exploit for various attacks.

5.2.1 Simulation-Based Fault Injection

As mentioned earlier, fault injection can occur at either the hardware or software level. Our method of choice is **simulation-based fault injection**, a widely used method for emulating hardware errors within software. In this technique, the system under test is simulated on a separate. Specifically, fault simulation technique uses software tools to simulate faults in a virtual environment, enabling us **studying potential impact without damaging at all the actual hardware**. Fault simulation allows engineers to test the robustness and the reliability of hardware designs, ensuring they can withstand various fault conditions. Also, fault simulation enhances security evaluation by identifying potential vulnerabilities that might be exploited by attackers.

In conducting these fault injections experiment, I was provided with the essential scripts and codes required to perform these fault simulations effectively. This foundation allowed me to focus on configuring and executing fault injection scenarios, ensuring that the time provided for this Thesis was enough. Having these pre-built tools expedited the setup process, making it easier to perform fault injections and analyze the effects of both single-bit and multiple-bit fault models on system resilience and reliability.

A key challenge is selecting a fault model that is both **easy to implement** and **accurately represents real hardware faults**. The single bit-flip model has been extensively utilized as an engineering approximation to simulate particle-induced soft errors in both combinational logic and memory elements. However, previous studies have shown that many soft errors occurring in the processor, are presented as multiple-bit errors at the application level [26], [27], [28]. This observation has prompted researchers to scrutinize the reliability of the single bit-flip model in accurately exhibiting transient faults caused by soft errors. Consequently, the fault model should **account for both single-bit and multiple-bit errors** when assessing metrics such as **error coverage** [29], [30] and **error resilience** [31], [32].

Single-Bit Fault Model. Single-Bit Fault Model has been the traditional approach to estimate the impact of soft errors on programs. It involves flipping a single bit in the program's data or control flow and assessing the effect of this change on the program's output.

Multiple-Bit Fault Model. With advancements in technology and continued scaling, the likelihood of multiple-bit errors has increased. The multiple-bit fault model introduces errors by flipping two or more bits simultaneously, providing a more accurate representation of modern hardware faults.

Error models are usually implemented at the algorithm, source code, or command level. Typically, cryptanalysis techniques are based on error models in which errors are introduced during the data flow of the target device and are intended to affect a single bit or more (byte or word) of a variable that is a key part of system security. The ways to affect the target are by bit-flip, bit-set, bit-reset or giving a random value.

5.2.2 Statistical Fault Injection

Fault Injection can be implemented by **different methods** but with the same **goal of finding circuit vulnerabilities**. The identification of these vulnerabilities is the starting point of the evaluation. Then, the results are taken into account in relation to the required specifications for safety and reliability. In the event that the necessary specifications that have been set are not met, it is determined where necessary the integration of countermeasures and protection techniques to improve the durability of the circuit. If, despite the identification of vulnerabilities in the circuit, it is chosen by design not to integrate countermeasures, this is done by **accepting any risk that has arisen**. The earlier the study is done the more design resources will be saved. So, fault tolerance tests should be implemented as early as possible. For this reason, this dissertation focuses on injecting errors through simulation in HLS-generated RTLs, enabling the study of circuit robustness early in the design flow.

As mentioned earlier, to facilitate these fault injection experiments, I was provided with the essential scripts and codes needed for efficient statistical fault injections. This groundwork allowed me to concentrate on debugging and adjusting fault injection scenarios. With these pre-developed resources, I could efficiently perform analysis and generate representative data, even with the limitations of time and complexity.

The aim is always to achieve the desired result in a **reasonable period of time**, which, however, allows for **safe and representative conclusions**. As the complexity of circuits increases, their analysis through fault models becomes more difficult and time-consuming. The possible number of faults to be injected to cover all possibilities can reach unimaginable numbers, which leads to the adoption of more practical solutions. For this reason it is customary to select a random subset of possible errors which is a representative sample of the set of possible errors.

Through statistical methods, it is possible to calculate a **minimum required number of faults**, given a certain margin of error on the measured results. This method offers some kind of quantification regarding the error margins around the results and to what degree of confidence they correspond.

Since a circuit has finite number of elements, the N memory elements that exist depend on the circuit in question. These are the ones that can be exposed to an attack which can change their value during the operation of the circuit in a given clock cycles [33]. In particular, the number of random errors being introduced can be calculated through a proposed mathematical equation, presented below [33]:

$$n = \frac{N}{1 + e^2 * \frac{N-1}{t^2 * p * (1-p)}}$$

• where N is the number of all memory elements of the circuit
• where p is defined as the standard error
• where e is the margin of error which must be within the interval [Peval – e ; Peval + e], or [Peval – e∗100 ; Peval + e∗100]
• where t corresponds to the confidence level and indicates the probability the exact value will be actually within the error range. Usually, confidence level 95% is chosen.

Chapter 6

Ciphers Overview

Before we dive into cipher's detailed overview, we need to clarify the origins of the implementations that were used. The implementation of "GIFT-64-128" is based on [34] to ensure accuracy and alignment with standard specifications. The implementations for the other three(3) ciphers ("LED-64", "KATAN32" & "SIMON64/128") were provided to me, serving as a foundation for my work. From these implementations I adjusted and debugged the codes as needed. Also, I produced the necessary Header and Testbench files for Vivado HLS, adapting the codes to ensure compatibility with Vivado HLS for successful compilation and synthesis.

6.1 Cipher "GIFT-64-128"

GIFT is a family of lightweight block ciphers, very versatile and performs also very well on software. Designed for low-resource environments like RFID tags and sensor networks, improving over PRESENT [35] in both security and efficiency [36].

According to [36], compared to PRESENT, GIFT presents the following advantages:
✓ Smaller area due to smaller S-box.
✓ Fewer subkey additions.
✓ Better resistance against LC thanks to good choice of S-box and bit permutation.
✓ Fewer rounds.
✓ Faster & simpler key schedule.

There are two(2) versions of GIFT:

- *GIFT-64-128* is a 28-round SPN cipher with 64-bit size state.
- *GIFT-128* is a 40-round SPN cipher with 128-bit size state.

Both versions have a 128-bit length key.

6.1.1 Specifications

GIFT-64-128 can be represented in three(3) different representations. In this Thesis, we adopt the classical 1D representation of GIFT-64-128, GIFT-64 for short.

GIFT-64 is a **28-round** SPN cipher with **128-bit length key**. As described in [36], during initialization, the cipher takes an n-bit plaintext b_{n-1}, b_{n-2},..., b_0 as the cipher state S, where $n = 64$ and b_0 is the least significant bit. The cipher state can also be represented as s 4-bit nibbles $S = w_{s-1}||w_{s-2}||...||w_0$, where $s = 16$. Additionally, the cipher receives a 128-bit key $K = k_7||k_6||...||k_0$ as the key state, with k_i being a 16-bit word.

Round Function. Each round of GIFT-64 has 3 steps:

$\Rightarrow$ SubCells
$\Rightarrow$ Permutation Bits
$\Rightarrow$ Add Round Key

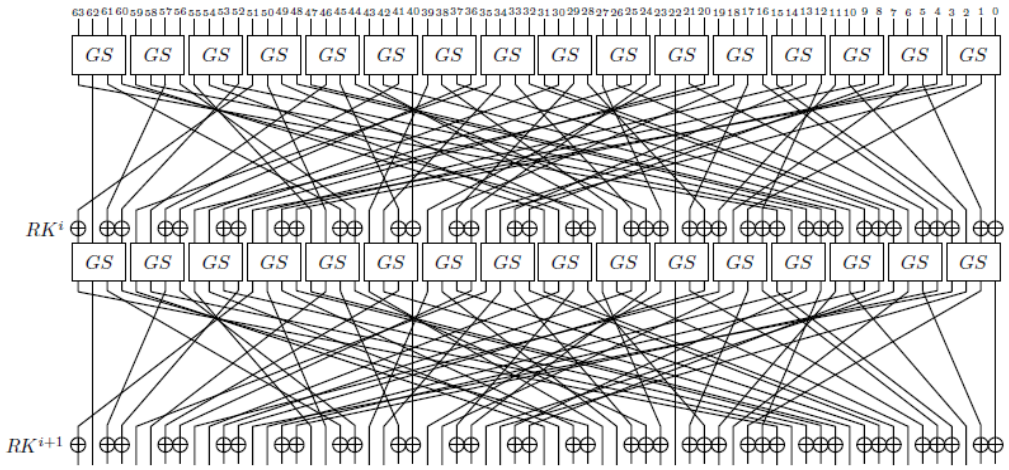

Figure 6.1: "GIFT-64-128" Rounds [36]

SubCells. GIFT-64 uses an invertible 4-bit S-box, GS. The Sbox is applied to every nibble of the cipher state.

Permutation Bits. The bit permutation used in GIFT-64 are given by [36]. It maps bits from bit position i of the cipher state to bit position $P(i)$.

Add Round Key. This step involves adding the round key and round constants. An $n/2$-bit round key RK is derived from the key state and further partitioned into two(2) s-bit words $RK = U||V = u_{s-1}...u_0||v_{s-1}...v_0$, where $s = 16$ for GIFT-64. A single bit "1" and 6-bit round constant $RC = c_5c_4c_3c_2c_1c_0$ are XORed into the cipher state at bit position n-1, 23, 19, 15,11, 7 and 3 respectively.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{64}(i)$	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{64}(i)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{64}(i)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{64}(i)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

Figure 6.2: "GIFT-64-128" Bit Permutation [36]

As for the **key schedule**, a round key is initially derived from the state prior to the key state update. In the case of, two(2) 16-bit words from the key state are extracted as the round key $RK = U||V$. The round constant's update function is defined in [36] as:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1)$$

The six(6) bits are initialized to "0" and updated before being used in a given round. The constants for each round are listed in the table below, encoded as byte values, with c_0 representing the least significant bit.

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

Figure 6.3: "GIFT-64-128" Round Constants [36]

6.1.2 Security Analysis

Differential Cryptanalysis (DC) [37] and **Linear Cryptanalysis** (LC) [38] are among the most powerful techniques available for block ciphers. Assessing a block cipher's resistance to Differential and Linear Cryptanalysis is one of the most common and essential forms of security analysis.

GIFT-64 is designed to minimize the probability of differential characteristics, making it difficult to predict how small changes in the input will affect the output after several rounds. It has a 9-round differential probability of $2^{-44.415}$. When taking the average per round and propagating forward, the differential probability is anticipated to drop below 2^{-63} by the time we reach the 14th round, given that the differential probability begins from the second round due to the absence of a whitening key at the outset [36]. It is worth noting that the differential probability is nearly equal to that of the optimal differential characteristic.

As for the LC, GIFT family is designed to prevent strong linear approximations, thus making it resistant to linear cryptanalysis. It exhibits a 9-round linear hull effect of $2^{-49.997}$, and it is expected to take 13 rounds to achieve a correlation potential lower than 2^{-64}.

Regarding **Integral Attacks**, GIFT family employs strong diffusion mechanisms to ensure that each bit of the output depends on every bit of the input after several rounds, which helps in resisting integral attacks.

Impossible Differential cryptanalysis [39] [40] exploits a pair of differences, Δ_1 and Δ_2, where the state difference Δ_1 never transitions to Δ_2 after a certain number of rounds. Such pairs, Δ_1 and Δ_2, are referred to as **Impossible Differential Attacks**. Subkeys leading to impossible differentials were detected to be wrong [36]. GIFT-64 achieves full diffusion after just three(3) rounds, and no 6-round truncated Impossible Differentials have been found.

The **Meet-in-the-Middle (MITM) Attack** is a classical approach that divides the encryption algorithm into two independent functions [41]. In this type of attack the attacker tries to find a matching state in the middle of the encryption process by partially encrypting the plaintext and decrypting the ciphertext. The security analysis shows that GIFT's key schedule and structure make it infeasible to find such a match efficiently.

Invariant Subspace Attack leverages a linear subspace A and a constant u that remain invariant under the round transformation. In its generalized form, it exploits the property that the subspace $A \oplus u$ is mapped to another subspace $A' \oplus v$ after the round transformation. For dimension 1 of GIFT-64, there are five transitions. In any of these cases, XORing the constant to Most Significant Bit (MSB), breaks the invariant subspace, thus GIFT-64 resists the invariant subspace attacks [36].

Nonlinear Invariant Attacks [42] are weak-key attacks that can be executed when the round constant is XORed only to specific bits of the nibbles. For the SPN structure, the attacks are mounted when:

· S-box has the quadratic nonlinear invariant
· Linear layer is represented by the multiplication with an orthogonal binary matrix

In GIFT's S-box there is no quadratic nonlinear invariant, making it strong against the nonlinear invariant attacks [36].

6.2 Cipher "LED-64"

Light Encryption Device (LED) block cipher is based on AES-like design principles [43]. This enables the establishment of straightforward bounds on the number of active S-boxes during block cipher encryption. Notably, while AES-based methods are well-suited for software implementations, they do not always yield the most efficient designs in hardware. The version presented here includes a slight modification to the original description from [44].

6.2.1 Specifications

In contemporary symmetric cryptography, an AES-like design is considered the optimal foundation for a clean and secure architecture. The design of LED shares many similarities with this established approach, incorporating elements such as *S-boxes*, *ShiftRows*, and a variant of *MixColumns* in their conventional roles.

LED is a **64-bit block cipher** that has two(2) primary instances, which utilize 64-bit and 128-bit keys [45]. In our case we choose the instance with 64-bit length key and **32 rounds**. The cipher state is conceptually organized in a (4×4) grid where each nibble corresponds to an element from $GF(2^4)$, with the polynomial $X^4 + X + 1$ serving as the basis for field multiplication [43].

For a 64-bit plaintext m, the sixteen(16) 4-bit nibbles $m_0||m_1||...||m_{14}||m_{15}$ are organized into a (4×4) square array, serving as the initial value of the cipher $STATE$. It is important to note that the **state is loaded row-wise** rather than in the column-wiseapproach typically seen in AES, which is considered more hardware-friendly choice [46].

The key is viewed nibble-wise and is denoted $k_0, k_1, ..., k_l$ the l nibbles of the key. Then the i-th subkey SK^i, also arranged in a (4×4) square array, is simply obtained by setting $sk^i_j = k_{(j+i*16 \ mod \ l)}$. For LED-64 all subkeys are equal to the 64-bit key K.

Add Round Key. This operation combines nibbles of subkey SK^i with the $STATE$, respecting array positioning, using XOR.

Operation "Step". It consists of four(4) rounds of encryption applied to the cipher state, with each of these rounds utilizing the following sequence:

$$\Rightarrow \text{AddConstants}$$
$$\Rightarrow \text{SubCells}$$
$$\Rightarrow \text{ShiftRows}$$
$$\Rightarrow \text{MixColumnsSerial}$$

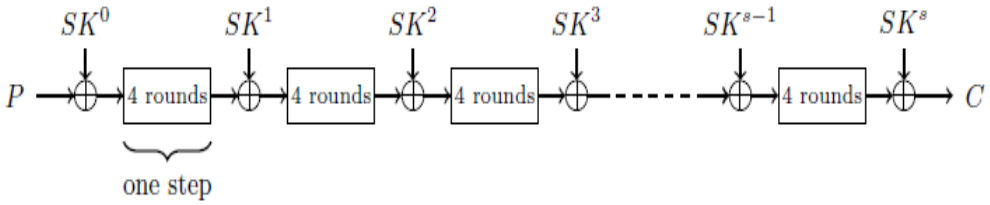

Figure 6.4: Use of subkeys SK^i during the s Steps [43]

AddConstants. In each round, six(6) bits (rc_5, rc_4, rc_3, rc_2, rc_1, rc_0) are shifted one position to the left with the new value for rc_0 calculated as $rc_5 \oplus rc_4 \oplus 1$. These six(6) bits are initialized to "0" and are updated **before** being utilized in the round.

SubCells. Each nibble in the $STATE$ array is replaced by the nibble generated after using the S-box of PRESENT cipher [35].

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 6.5: PRESENT cipher S-box [35]

ShiftRows. ShiftRows transformation in LED-64 is specified to **cyclically rotate** the rows of the $STATE$ array by different offsets. This is intended to provide diffusion by spreading the influence of individual bits across multiple columns. Here is how the shifts are defined for the 1D state matrix of 16 elements:

· $0 - 3$: No shift (0 positions)
· $4 - 7$: Shift left by 1 position
· $8 - 11$: Shift left by 2 positions
· $12 - 15$: Shift left by 3 positions

MixColumnsSerial. This involves mixing the columns of the $STATE$ array using a predefined (4×4) *MixColMatrix* Maximum Distance Separable (MDS) matrix. LED reuses the tactic adopted in [47] to define a MDS matrix for linear diffusion that is suitable for compact serial implementation. Each element of the $STATE$ is multiplied by this MixColMatrix. The multiplication is carried out in the finite field $GF(2^4)$, and the results are combined using XOR operations. The MixColumnsSerial function ensures each element of the $STATE$ is mixed according to the MixColMatrix, providing diffusion by spreading the influence of each nibble across multiple positions [43]. Combined with the ShiftRows and SubCell transformations, strengthens the cipher against various cryptographic attacks.

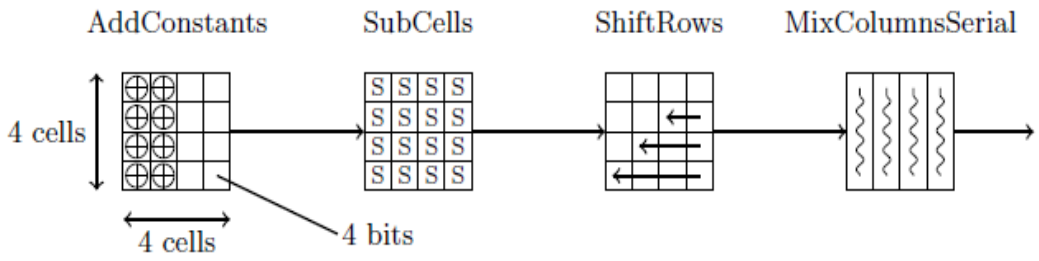

Figure 6.6: LED Single Round Overview [43]

6.2.2 Security Analysis

The LED block cipher is easy to analyze, enabling a precise assessment of the required number of rounds for adequate security. The **key schedule** was selected for its simplicity and security features. Its straightforward nature facilitates direct derivation of bounds on the minimum number of active S-boxes, even in the context of related-key attacks. When excluding related-key attacks, it can be directly concluded that any differential path for LED will have at least $s \cdot 25$ active S-boxes [43].

In the context of **Related-Key Attacks**, it is essential to differentiate between the various key-size versions. For our 64-bit key version in particular, if we assume that differences are introduced at the key input, every subkey SK^i in the 64-bit variant of LED will be active. Consequently, it is evident that it is impossible to enforce two(2) consecutive non-active big steps, as at least one(1) of the two(2) big steps will always be active [43].

Regarding **Differential Cryptanalysis**, LED's structure ensures minimum number of active S-boxes per round, reducing the probability of successful differential attacks. The analysis showed that any differential path will encounter a significant number of active S-boxes, making it highly unlikely for an attacker to find useful differential characteristics. As for the **Linear Cryptanalysis**, LED's design also ensures high number of active S-boxes in linear paths, making it difficult to exploit linear approximations. Over four rounds, upper bound for the best linear approximation probability is 2^{-50}, while the best linear hull probability is bounded by 2^{-32} [43].

The **Slide Attack** is a technique for block cipher cryptanalysis [48] that takes advantage of the self-similarity present in a permutation. In the case of LED, the addition of round-dependent constants ensures that each round is made unique. This feature renders the slide attack infeasible.

Generally, the number of rounds in LED is deliberately set to ensure a **wide margin of security**. Even if an attack technique could theoretically reduce the rounds required for a breach, the chosen number of rounds provides sufficient buffer to **prevent practical exploitation**.

In conclusion, security analysis confirms that LED is a **well-designed block cipher** that balances hardware efficiency with strong cryptographic security. Its resistance to various forms of cryptanalysis, particularly related-key and differential attacks, combined with conservative design choices, ensure that LED **remains robust** even as cryptanalytic techniques evolve.

6.3 Cipher "KATAN32"

The **KATAN family of block ciphers** was designed to meet the needs of resource-constrained environments, such as RFID tags and embedded systems [49]. The family consists of three(3) main variants based on block sizes: KATAN32, KATAN48, KATAN64 (n-bit for KATANn). All three(3) "version" accept same length key, are highly compact, and attain the minimal size while ensuring sufficient security. We have chosen **KATAN32**.

6.3.1 Specifications

In general, the KATAN family of block ciphers features very simple structure. The plaintext is loaded into two(2) registers, with their lengths determined by the block size. During each round, multiple bits are extracted from the registers and fed into two(2) nonlinear Boolean functions. The outputs of these functions are then stored in the least significant bits of the registers after they have been shifted. The ciphers have 254 rounds in order to ensure sufficient mixing.

Specifically, KATAN32 is the smallest of this family. It uses a **80-bit length key** and has **32-bit block size**, which means its plaintext and ciphertext has size of 32 bits. The plaintext is loaded into two(2) registers, $L1$ with a length of 13 bits and $L2$ with a length of 19 bits [49]. The Least Significant Bit (LSB) of the plaintext is loaded into bit 0 of $L2$, while the Most Significant Bit (MSB) is placed in bit 12 of $L1$. In each round, $L1$ and $L2$ are shifted to the left, and the newly computed bits are stored in the LSBs of both registers. After **254 rounds**, the contents of the registers are output as the ciphertext.

KATAN32 uses two(2) **nonlinear functions** $f_a(\cdot)$ and $f_b(\cdot)$ in each round. The nonlinear functions f_a and f_b are presented below, as defined in [49]:

$$f_a(L_1) = L_1[x_1] \oplus L_1[x_2] \oplus (L_1[x_3] \cdot L_1[x_4]) \oplus (L_1[x_5] \cdot IR) \oplus k_a$$

$$f_b(L_2) = L_2[y_1] \oplus L_2[y_2] \oplus (L_2[y_3] \cdot L_2[y_4]) \oplus (L_2[y_5] \cdot L_2[y_6]) \oplus k_b$$

where IR is irregular update rule (i.e., $L_1[x_5]$ is XORed in the rounds where the irregular update is used), and k_a and k_b are the two(2) subkey bits [49].

After the nonlinear functions are computed, L_1 and L_2 registers are shifted, causing the Most Significant Bit (MSB) to drop off, while the Least Significant Bit (LSB) is filled with the output from the second nonlinear function. Thus, after the round, the LSB of L_1 contains the output of f_b, and the LSB of L_2 holds the output of f_a. The **key schedule** for KATAN32 involves loading the 80-bit key into a Linear Feedback Shift Register (LFSR), with the LSB of the key placed in position "0". In each round, positions "0" and "1" of the LFSR serve as the round's subkeys k_{2i} and k_{2i+1}, and the LFSR is clocked twice.

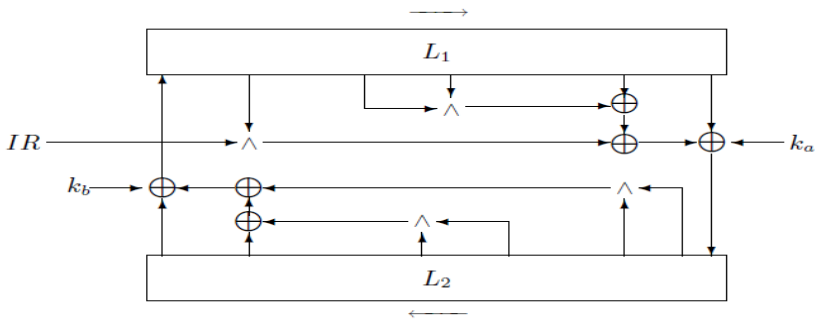

Figure 6.7: "KATAN" Round Overview [49]

6.3.2 Security Analysis

The design philosophy was based on offering a very high level of security, having very large security margins.

Differential and Linear Cryptanalysis were analyzed under the assumption that the intermediate encryption values are independent [50]. This simplifies the analysis and is not expected to change the results too much [49]. Using computer-aided search for Differential Cryptanalysis (DC), the results indicated that, depending on the number of rounds used, the best 42-round differential characteristic for KATAN32 has a probability of 2^{-11}, which **could be even lower with a different set of rounds**. Therefore, any 126-round differential characteristic cannot have a probability greater than $(2^{-11})^3 = 2^{-33}$ [49]. For LC, comparable results were found, with the best 42-round linear approximation exhibiting a bias of 2^{-6} [49].

Regarding **Combined Attacks**, KATAN32 is secure against combined attacks like differential-linear and boomerang attacks due its **quick diffusion** and **large security margins**. The probability of a successful attack drops exponentially with the number of rounds. Another combined attack is the impossible differential attack, which relies on identifying a differential with a probability of zero over as many rounds as possible. However, due to the rapid diffusion, altering even a single bit will inevitably affect all bits after at most 42 rounds, making it impossible to conduct an impossible differential attack over more than 168 rounds.

Slide Attacks exploit the ability to find two messages that share much of the encryption process. However, due to the **differences in the round functions**, this is feasible only for a **very limited number of rounds**. In fact, no slide property with a probability of 2^{-32} exists starting from the first round of the cipher. The earliest round where such a property can be constructed is round 19. If an attacker obtains the same intermediate encryption value at round 19 and round 118, they may find a 'slid' pair, maintaining this equality with a probability 2^{-31} until the end of the cipher. This demonstrates that there are **no advantageous slid properties** in the KATAN cipher family overall [49].

As for the **Related-Key Attacks**, the attacker seeks two(2) intermediate encryption values and keys that evolve in the same manner for as many rounds as possible. As previously mentioned, there are **no "good" relations over different rounds**, meaning the two(2) intermediate encryption values must be in the same round. After at most 80 rounds, the difference in the subkeys prevents any effective related-key differential from propagating [49].

In [51], authors combined the **Cube Attack** [52] and its extended variant with fault analysis to form successful **Hybrid Attacks** against the full-round versions of the KATAN family. The attacks utilize Cube Attacks and their extensions to identify the effective fault injection rounds, derive differential characteristics, and generate linear and quadratic equations. The complexity of the attack on KATAN32 is 2^{59} computations and about 115 fault injections [51].

In conclusion, while KATAN32 provides significant **resistance to basic attacks**. Advanced fault and algebraic attacks can **reduce the complexity of key recovery**, making it a **potential target** for attackers with access to **side-channel or fault injection** capabilities [51].

6.4 Cipher "SIMON64/128"

The **SIMON family** is a group of **lightweight block ciphers** developed by the U.S. National Security Agency (NSA) in 2013 [53], to provide cryptographic security with minimal computational resources. These ciphers are primarily used in resource-constrained environments like RFID tags, IoT devices, and embedded systems. SIMON ciphers use a Feistel structure and their round functions rely heavily on basic bitwise operations such as circular shifts, bitwise AND, and bitwise XOR [54]. SIMON family includes variants with block sizes ranging from 32 to 128 bits and key lengths ranging from 64 to 256 bits. A SIMON block cipher with an n-bit word and m-bit key is refereed as SIMON2n/m.

6.4.1 Specifications

Our choice is a configuration with **32-bit words** and a 128-bit key, which becomes **SIMON64/128**. The SIMON64 version operates on a **64-bit block size** with a key size of either 96 or 128 bits [54]. As noted, we select a **128-bit length key**. The number of rounds varies depending on the key size, with SIMON64/128 performing **44 rounds**.

Round Function. The SIMON round function used for encryption is presented below, as defined in [55]:

$$R(l, r, k) = ((S^1(l) \& S^8(l)) \oplus S^2(l) \oplus r \oplus k, l)$$

Where l is the left-most word of a given block, r the right-most word and k the appropriate round key. The structure follows the general Feistel scheme, where the block is split into two equal halves, and in each round, the operations include the following [54]:

⇒ **Left Circular Rotation**: The left half of the input is circularly shifted by 1, 8, and 2 bits.

⇒ **Bitwise AND**: The result of these shifts is combined using the bitwise AND operation.

⇒ **Bitwise XOR**: The output from the AND operation is XORed with the other half of the input and the round key.

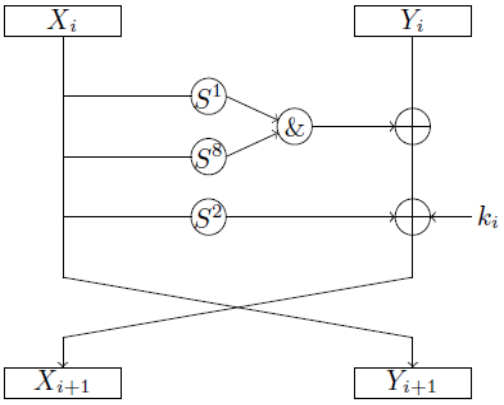

Figure 6.8: "SIMON" Feistel Structure of Round Function [54]

Key Schedule. SIMON's key schedule offers **key expansion capabilities** by generating all round keys from the master key in succession. In SIMON64/128, the key schedule produces forty-four(44) round keys, each 32-bit sized, from the initial 128-bit master key, using bitwise XOR and right circular rotations. This setup enhances the security by ensuring that every round key is distinct and not easily predictable from previous keys.

6.4.2 Security Analysis

Since its introduction back in 2013, SIMON64/128 has undergone extensive cryptanalysis. It generally shows **resilience against common cryptanalytic attacks** like Differential and Linear Cryptanalysis. Also, SIMON has no look-up tables, making it naturally immune to **Cache-Timing Attacks** [56].

Cryptanalysts have successfully attacked reduced-round versions of SIMON family. The full 44-round version of SIMON64/128 remains secure against **Reduced-Round Attacks**. Furthermore, like many lightweight ciphers, SIMON64/128 may be susceptible to **Side-Channel Attacks**, as in low-resource environments, security posture depends on the hardware implementation itself.

Finally, SIMON's family structure relies heavily on basic bitwise operations, making it susceptible to **Rotational Attacks**, where patterns in data rotations are exploited. Again, although these attacks can break reduced-round versions, they are generally impractical against the full 44-round version of SIMON64/128.

Chapter 7

Ciphers Implementation

For the needs of the experiments, all the four(4) ciphers followed the same implementation process. First of all, after each cipher was studied in terms of its structure and specifications, all cipher were implemented in C/C++ code. For each cipher were created three(3) files. A Source file, a Header file and a Testbench file. All these three(3) files were the input files into Vivado HLS in order to start the actual HLS implementation process. The device which was used is the *"xc7a12ticsg325-1L"*.

During HLS implementation phase in Vivado HLS tool, the important part is the selections of the directives. For the three(3) basic variables used in every implementation we applied the same directives.

• for "**input**" variable:
⇒ set_directive_array_partition -type complete -dim 1 "top_function" input

• for "**masterkey**" variable:
⇒ set_directive_array_partition -type complete -dim 1 "top_function" masterkey

• for "**output**" variable:
⇒ set_directive_array_partition -type complete -dim 1 "top_function" output
⇒ set_directive_interface -mode ap_vld -register "top_function" output

Variable "input" stores the cleartext data which are going to be encrypted. Variable "masterkey" stores the main key used in the encryption process. Variable "output" stores the produced ciphertext.

After the HLS implementation process has finished, the output VHDL files are extracted from Vivado HLS tool so they can be used for our fault injections. Also, for the needs of statistical method, a Python script was created with the sole purpose of generating and formatting the random numbers which were used in the Statistical Fault Injection test cases.Before proceeding to the implementation's results, it is worth noting that the final LUT count, after physical optimizations and full implementation, is typically lower.

In brief, *"GIFT-64-128"* and *"LED-64"* recorded similar logic and power usages. Their only notable difference is in memory usage. We could expect that due to their alike design principles which are based on. *"KATAN32"* has by far the highest usage in term of logic, memory and power usage. On the other hand, *"SIMON64/128"* displays the lowest utilization in logic, memory and power.

7.1 Implementation of "GIFT-64-128"

"GIFT-64-128" logic usage is significantly low as none percentage is bigger than 8.00%. It uses thirty-six(36) LUT slices as memory and zero(0) register as latch. Regarding memory usage, the site type is "RAMB18". Its static and total power consumption is 0.186 W and 16.953 W respectively.

Site Type	Used	Available	Utilization
Slice LUTs	618	8000	7.73%
LUT as Logic	582	8000	7.28%
LUT as Memory	36	5000	0.72%
Slice Registers	548	16000	3.43%
Register as Flip Flop	548	16000	3.43%
Register as Latch	0	16000	0.00%
F7 Muxes	51	7300	0.70%
F8 Muxes	8	3650	0.722%

Table 7.1: Slice Logic of "GIFT-64-128"

Site Type	Used	Available	Utilization
Block RAM Tile	1	20	5.00%
RAMB18	2	40	5.00%
RAMB36	0	20	0.00%

Table 7.2: Memory Usage of "GIFT-64-128"

On-Chip	Power	Used	Available	Utilization
Slice Logic	2.466 W	1246	—	—
LUT as Logic	2.148 W	582	8000	7.28%
LUT as Memory	0.170 W	36	5000	0.72%
Register	0.103 W	548	16000	3.43%
F7/F8 Muxes	0.041 W	59	14600	0.40%
BUFG	0.005 W	1	32	3.13%
Signals	3.796 W	1477	—	—
Block RAM	0.196 W	1	20	5.00%
I/O	10.308 W	534	150	356.00%
Static Power	0.186 W	—	—	—
Total	16.953 W	—	—	—

Table 7.3: Power Usage of "GIFT-64-128"

7.2 Implementation of "LED-64"

"LED-64" logic usage is also incredible low as none percentage is bigger than 7.00%. It uses sixteen(16) LUT slices as memory and zero(0) register as latch. Regarding memory usage, the site type is "RAMB18". Its static and total power consumption is 0.188 W and 17.041 W respectively.

Site Type	Used	Available	Utilization
Slice LUTs	487	8000	6.09%
LUT as Logic	471	8000	5.89%
LUT as Memory	16	5000	0.32%
Slice Registers	471	16000	2.94%
Register as Flip Flop	471	16000	2.94%
Register as Latch	0	16000	0.00%
F7 Muxes	48	7300	0.66%
F8 Muxes	24	3650	0.66%

Table 7.4: Slice Logic of "LED-64"

Site Type	Used	Available	Utilization
Block RAM Tile	0.5	20	2.50%
RAMB18	1	40	2.50%
RAMB36	0	20	0.00%

Table 7.5: Memory Usage of "LED-64"

On-Chip	Power	Used	Available	Utilization
Slice Logic	1.769 W	1047	—	—
LUT as Logic	1.597 W	471	8000	5.89%
LUT as Memory	0.020 W	16	5000	0.32%
Register	0.095 W	471	16000	2.94%
F7/F8 Muxes	0.052 W	72	14600	0.49%
BUFG	0.005 W	1	32	3.13%
Signals	3.709 W	1122	—	—
Block RAM	0.164 W	0.5	20	2.50%
I/O	11.211	406	150	270.67%
Static Power	0.188 W	—	—	—
Total	17.041 W	—	—	—

Table 7.6: Power Usage of "LED-64"

7.3 Implementation of "KATAN32"

"KATAN32" logic usage is significantly higher as LUT slices utilization percentage is almost 40.00%. It does not use LUT slices as memory, neither register as latch. Regarding memory usage, the site type is "RAMB36". Its static and total power consumption is 0.239 W and 170.363 W respectively.

Site Type	Used	Available	Utilization
Slice LUTs	3198	8000	39.98%
LUT as Logic	3198	8000	39.98%
LUT as Memory	0	5000	0.00%
Slice Registers	3032	16000	18.95%
Register as Flip Flop	3032	16000	18.95%
Register as Latch	0	16000	0.00%
F7 Muxes	643	7300	8.81%
F8 Muxes	64	3650	1.75%

Table 7.7: Slice Logic of "KATAN32"

Site Type	Used	Available	Utilization
Block RAM Tile	6	20	30.0%
RAMB18	0	40	0.00%
RAMB36	6	20	30.0%

Table 7.8: Memory Usage of "KATAN32"

On-Chip	Power	Used	Available	Utilization
Slice Logic	15.759 W	6952	—	—
LUT as Logic	14.828 W	3198	8000	39.98%
CARRY4	0.013 W	3	3650	0.08%
Register	0.312 W	3032	16000	18.95%
F7/F8 Muxes	0.601 W	707	14600	4.84%
BUFG	0.005 W	1	32	3.13%
Signals	25.372 W	13083	—	—
Block RAM	1.296 W	6	20	30.00%
I/O	127.697 W	9254	150	6169.33%
Static Power	0.239 W	—	—	—
Total	170.363 W	—	—	—

Table 7.9: Power Usage of "KATAN32"

7.4 Implementation of "SIMON64/128"

"SIMON64/128" logic usage is the lowest recorded as none percentage is bigger than 3.00%. It does not use LUT slices as memory, neither register as latch or F7/F8 muxes.. Regarding memory usage, the site type is "RAMB36". Its static and total power consumption is 0.080 W and 8.214 W respectively.

Site Type	Used	Available	Utilization
Slice LUTs	214	8000	2.68%
LUT as Logic	214	8000	2.68%
LUT as Memory	0	5000	0.00%
Slice Registers	192	16000	1.20%
Register as Flip Flop	192	16000	1.20%
Register as Latch	0	16000	0.00%
F7 Muxes	0	7300	0.00%
F8 Muxes	0	3650	0.00%

Table 7.10: Slice Logic of "SIMON64/128"

Site Type	Used	Available	Utilization
Block RAM Tile	1	20	5.00%
RAMB18	0	40	0.00%
RAMB36	1	20	5.00%

Table 7.11: Memory Usage of "SIMON64/128"

On-Chip	Power	Used	Available	Utilization
Slice Logic	2.245 W	413	—	—
LUT as Logic	2.134 W	214	8000	2.68%
Register	0.105 W	192	16000	1.20%
F7/F8 Muxes	—	—	—	—
BUFG	0.005 W	1	32	3.13%
Signals	3.172 W	666	—	—
Block RAM	0.309 W	1	20	5.00%
I/O	2.407 W	264	150	176.00%
Static Power	0.080 W	—	—	—
Total	8.214 W	—	—	—

Table 7.12: Power Usage of "SIMON64/128"

Chapter 8

Results Analysis

For the purposes of the Thesis, ten(10) Statistical Fault Injection (SFI) test cases were created, from $M1$ to $M10$. For each case, the multiplicity of Fault Injection is increased by 1. We symbolize test cases by 'Mi' where *'M'* stands for 'Multiplicity' and *'i'* for the number of bit-flips, e.g. $M1$ is single bit-flip.

We categorized faults into three(3) categories as **Critical Errors**, **Hang Errors** and **Silent Errors**.

- **Critical Errors.** These are severe errors that cause the system to crash, halt, or become unresponsive. They typically result in a complete failure of the device or application and may require a reset or power cycle to recover.
- **Hang Errors.** These are observable errors that affect the system but do not cause it to crash. They may manifest as incorrect outputs or glitches in the system, but the device remains operational and running.
- **Silent Errors.** These are errors that occur without being immediately detected. The system appears to function normally, but the internal state or output is incorrect. Silent errors can be particularly dangerous as they may go unnoticed and lead to larger issues later on.

The tables bellow show percentage values. Positive percentages corresponds to increment, while negative percentages represent decrement. In *'Transition Results'* tables, we keep percentage sign to distinguish rises from drops. Furthermore, in the following graphs we have illustrated Critical Errors with blue color, Hang Errors with orange, and Silent Errors with green.

It is important to note that the **0.5% error margin**, which is been set to this threshold due to the Thesis's time barrier. In our case study, due to the exponential increase in simulation time, the error margin could not be lower. This means the actual results may differ from the outputted percentages by $\pm 0.5\%$. For this reason, percentage differences of less than 0.5% in absolute value, are considered negligible.

8.1 Results of "GIFT-64-128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1	0.93%	12.30%	86.77%
M2	1.49%	22.36%	76.16%
M3	1.86%	31.56%	66.58%
M4	2.17%	39.18%	58.64%
M5	2.31%	46.60%	51.09%
M6	2.43%	52.96%	44.62%
M7	2.65%	58.00%	39.35%
M8	2.45%	62.97%	34.59%
M9	2.64%	66.98%	30.38%
M10	2.49%	71.00%	26.51%
Average	2.14%	46.39%	51.47%
Maximum	2.65%	71.00%	86.77%
Minimum	0.93%	12.30%	26.51%

Table 8.1: Results of "GIFT-64-128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M2	0.55%	10.06%	-10.62%
M2-M3	0.38%	9.20%	-9.58%
M3-M4	0.31%	7.63%	-7.94%
M4-M5	0.14%	7.42%	-7.56%
M5-M6	0.11%	6.36%	-6.47%
M6-M7	0.23%	5.04%	-5.27%
M7-M8	-0.21%	4.97%	-4.76%
M8-M9	0.19%	4.01%	-4.21%
M9-M10	-0.15%	4.02%	-3.87%
Average	0.17%	6.52%	-6.70%
Maximum	0.55%	10.06%	-10.62%
Minimum	0.11%	4.01%	-3.87%

Table 8.2: Transition Results of "GIFT-64-128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M5	1.38%	34.31%	-35.69%
M5-M10	0.18%	24.40%	-24.58%
M1-M10	1.56%	58.70%	-60.27%

Table 8.3: Aggregate Transition Results of "GIFT-64-128"

Critical Errors percentage has a slightly increasing trend, having its minimum value 0.93% at $M1$ and its maximum value 2.65% at $M7$. In total, from $M1$ to $M10$, Critical Errors are increased by 1.56%. The average increment percentage over consecutive cases is only 0.17%. From $M4$ and after, it remains broadly "stable" around its average value. The biggest increase appears from $M1$ to $M2$, as $M2$ is increased by 0.55%. Although this is the biggest difference between consecutive multiplicities, it is still pretty close to error margin, making the difference almost insensitive. Also, average and maximum values have small difference 0.51%, again almost equal to error margin. While for the multiplicities $M1$ to $M5$ the increase reaches 1.38%, between $M5 - M10$ the rise of Critical Errors percentage is only 0.18% which is much less than the error margin. That happened because between $M7 - M8$ and $M9 - M10$, Critical Errors percentage is decreased. With that been said, the difference between $M5 - M10$ is actually insensitive.

Hang Errors percentage is constantly increasing, having its minimum value 12.30% at $M1$, and reaching its maximum value 71.00% at $M10$. The average percentage is 46.39%, differing 24.61% from the maximum percentage value and 34.09% from the minimum. In total, from $M1$ to $M10$, Hang Errors are increased by 58.70%, having increased by 34.31% from $M1$ to $M5$ and 24.40% from $M5$ to $M10$. The biggest increase appears from $M1$ to $M2$, rising by 10.06%. Also, taking the error margin under consideration, $M5$'s percentage 46.60% may be considered "average" value, as they differ only by 0.21%. The percentage of the increase between consecutive cases decreases as the multiplicities proceed. The average increment percentage of Hang Errors is 6.52%.

Silent Errors percentage fluctuation is opposite to Hang Errors. It starts from its maximum value 86.77% at $M1$, dropping to minimum value 26.51% at $M10$. The average percentage is 51.47%, differing 35.30% from the maximum percentage value and 24.96% from the minimum. In total, Silent Errors are decreased by 60.27%, having decreased by 35.69% from $M1$ to $M5$ and 24.58% from $M5$ to $M10$. Similar to Hang Errors allocation, Silent Errors biggest drop take place from $M1$ to $M2$ where the percentage of Silent Errors drops by 10.62%. The percentage of the decrease between consecutive cases decreases as the multiplicities proceed. The average reduction percentage of Silent Errors is 6.70%.

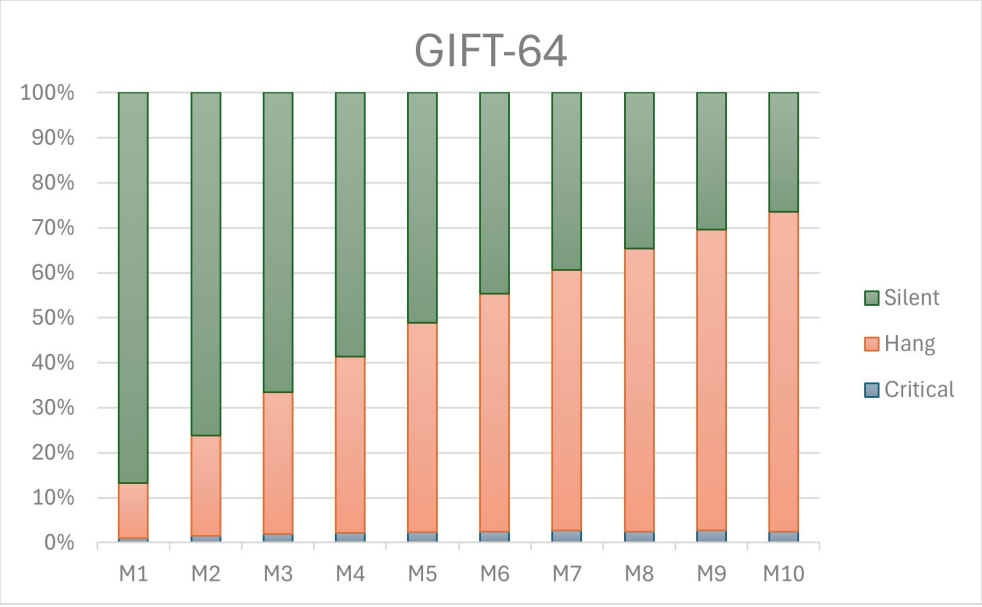

Figure 8.1: "GIFT-64-128" Graph

8.2 Results of "LED-64"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1	1.75%	13.19%	85.07%
M2	3.31%	23.68%	73.00%
M3	4.25%	33.25%	62.49%
M4	4.88%	41.21%	53.90%
M5	5.17%	48.56%	46.27%
M6	5.44%	54.80%	39.77%
M7	5.40%	60.18%	34.42%
M8	5.56%	65.31%	29.13%
M9	5.27%	69.48%	25.26%
M10	5.14%	72.91%	21.95%
Average	4.62%	48.26%	47.13%
Maximum	5.56%	72.91%	85.07%
Minimum	1.75%	13.19%	21.95%

Table 8.4: Results of "LED-64"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M2	1.57%	10.50%	-12.06%
M2-M3	0.94%	9.57%	-10.51%
M3-M4	0.63%	7.96%	-8.59%
M4-M5	0.29%	7.34%	-7.64%
M5-M6	0.26%	6.24%	-6.50%
M6-M7	-0.03%	5.38%	-5.34%
M7-M8	0.16%	5.13%	-5.29%
M8-M9	-0.29%	4.17%	-3.87%
M9-M10	-0.13%	3.44%	-3.31%
Average	0.38%	6.64%	-7.01%
Maximum	1.57%	10.50%	-12.06%
Minimum	-0.03%	3.44%	-3.31%

Table 8.5: Transition Results of "LED-64"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M5	3.43%	35.37%	-38.80%
M5-M10	-0.03%	24.35%	-24.32%
M1-M10	3.40%	59.72%	-63.12%

Table 8.6: Aggregate Transition Results of "LED-64"

Critical Errors percentage records a total increase of 3.40%, with the minimum value 1.75% at $M1$ and the maximum value 5.56% at $M8$. From $M1$ to $M5$, the percentage is increased by 3.43%. On the other hand, the difference between $M5 - M10$ is -0.03% which is much smaller than the error margin. This means that after $M4$, the Critical Error percentage remains stable. Another notable fact is the small difference between average and maximum value, which is less than 1%, specifically 0.94%. Also, the biggest increase appears from $M1$ to $M2$, rising by 1.57% which is nearly doubling $M1$'s percentage. Between $M6 - M7$, the percentage drops by 0.03% which is equal to the percentage drop between $M5 - M10$. The average increment percentage of Critical Errors is only 0.38%.

Hang Errors percentage is constantly increasing, having its minimum value 13.19% at $M1$, and reaching its maximum value 72.91% at $M10$. The average percentage is 48.26%, differing 24.65% from the maximum percentage value and 35.07% from the minimum. In total, from $M1$ to $M10$, Hang Errors are increased by 59.72%, having increased by 35.37% from $M1$ to $M5$ and 24.35% from $M5$ to $M10$. The biggest increase appears from $M1$ to $M2$, rising by 10.50%. Also, taking the error margin under consideration, $M5$'s percentage 48.56% may be considered the average value, as they differ only by 0.30%. The percentage of the increase between consecutive cases decreases constantly, as the multiplicities proceed. The average increment percentage of Hang Errors is 6.64%.

Silent Errors percentage fluctuation is opposite to Hang Errors. It starts from its maximum value 85.07% at $M1$, dropping to minimum value 21.95% at $M10$. In total, Silent Errors decrease by 63.12%. Similar to Hang Errors allocation, Silent Errors biggest drop take place from $M1$ to $M2$ where the percentage of Silent Errors drops by 12.06%. What is also interesting is that from $M1$ to $M3$ the percentage drops by 22.57%, while between $M5 - M10$ the decrease records 24.32%, differing only by 1.74%. Finally, the percentage of the decrease between consecutive cases decreases constantly, as the multiplicities proceed. The average reduction percentage of Silent Errors is 7.01%.

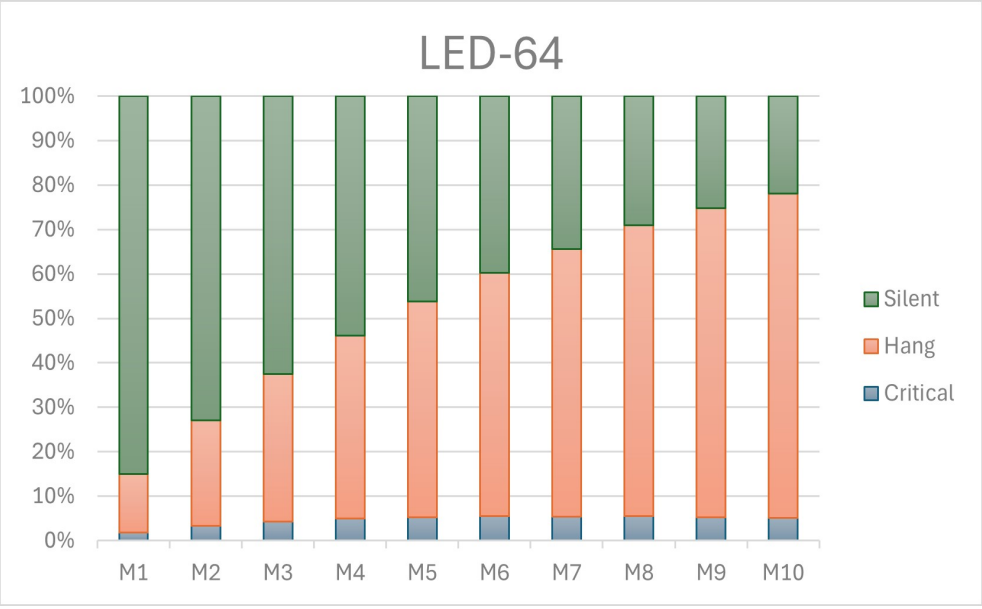

Figure 8.2: "LED-64" Graph

8.3 Results of "KATAN32"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1	1.30%	3.17%	95.53%
M2	2.08%	5.56%	92.37%
M3	2.76%	8.46%	88.78%
M4	3.27%	10.94%	85.79%
M5	3.75%	13.62%	82.63%
M6	3.97%	16.09%	79.94%
M7	4.21%	18.60%	77.19%
M8	4.63%	20.35%	75.02%
M9	4.71%	23.14%	72.15%
M10	4.97%	25.37%	69.66%
Average	3.57%	14.53%	81.90%
Maximum	4.97%	25.37%	95.53%
Minimum	1.30%	3.17%	69.66%

Table 8.7: Results of "KATAN32"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M2	0.78%	2.38%	-3.17%
M2-M3	0.69%	2.90%	-3.59%
M3-M4	0.51%	2.48%	-2.99%
M4-M5	0.47%	2.68%	-3.16%
M5-M6	0.23%	2.47%	-2.70%
M6-M7	0.24%	2.51%	-2.75%
M7-M8	0.42%	1.75%	-2.17%
M8-M9	0.08%	2.79%	-2.87%
M9-M10	0.25%	2.24%	-2.49%
Average	0.41%	2.47%	-2.88%
Maximum	0.78%	2.90%	-3.59%
Minimum	0.08%	1.75%	-2.17%

Table 8.8: Transition Results of "KATAN32"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M5	2.45%	10.45%	-12.90%
M5-M10	1.22%	11.75%	-12.98%
M1-M10	3.67%	22.20%	-25.88%

Table 8.9: Aggregate Transition Results of "KATAN32"

Critical Errors percentage records a total increase of 3.67%, with the minimum value 1.30% at $M1$ and the maximum value 4.97% at $M10$. From $M1$ to $M5$, the percentage is increased by 2.45% which more than double of the rise between $M5 - M10$ (1.22%). Another notable fact is the small difference between average and maximum value, which is 1.4%, compared to the difference between average and minimum value, which is 2.27%. Something also notable is a little spike of 0.42% from $M7$ to $M8$, considering the increase between $M6 - M7$ and $M8 - M9$ which are 0.24% and 0.08% respectively. Of course, taking under consideration the error margin, those changes from $M6$ to $M9$ may be insensitive. The biggest increase appears from $M1$ to $M2$, rising by 0.78%. On the contrary, the smallest increase is shown between $M8 - M9$, having raised only by 0.08% remaining essentially constant. Finally, the average increment percentage of Critical Errors is 0.41%, which is just under the error margin.

Hang Errors percentage is constantly increasing, having its minimum value 3.17% at $M1$, and reaching its maximum value 25.37% at $M10$. The average percentage is 14.53%, differing 10.84% from the maximum percentage value and 11.36% from the minimum. In total, from $M1$ to $M10$, Hang Errors are increased by 22.20%, having increased by 10.45% from $M1$ to $M5$ and 11.75% from $M5$ to $M10$. The biggest increase appears from $M2$ to $M3$, rising by 2.90%, while between $M7 - M8$ the least increase by 1.75%. The average increment rate of Hang Errors is 2.47%.

Silent Errors percentage fluctuation is opposite to Hang Errors. It starts from its maximum value 95.53% at $M1$, dropping to minimum value 69.66% at $M10$. The average percentage is 81.90%, differing 13.63% from the maximum percentage value and 12.25% from the minimum. In total, Silent Errors decrease by 25.88%, having decreased by 12.90% from $M1$ to $M5$ and 12.98% from $M5$ to $M10$. These percentage drop is approximately the same between the two half of the test cases. Silent Errors biggest drop take place from $M2$ to $M3$ where their percentage drops by 3.59%. The average reduction rate of Silent Errors is 2.88%.

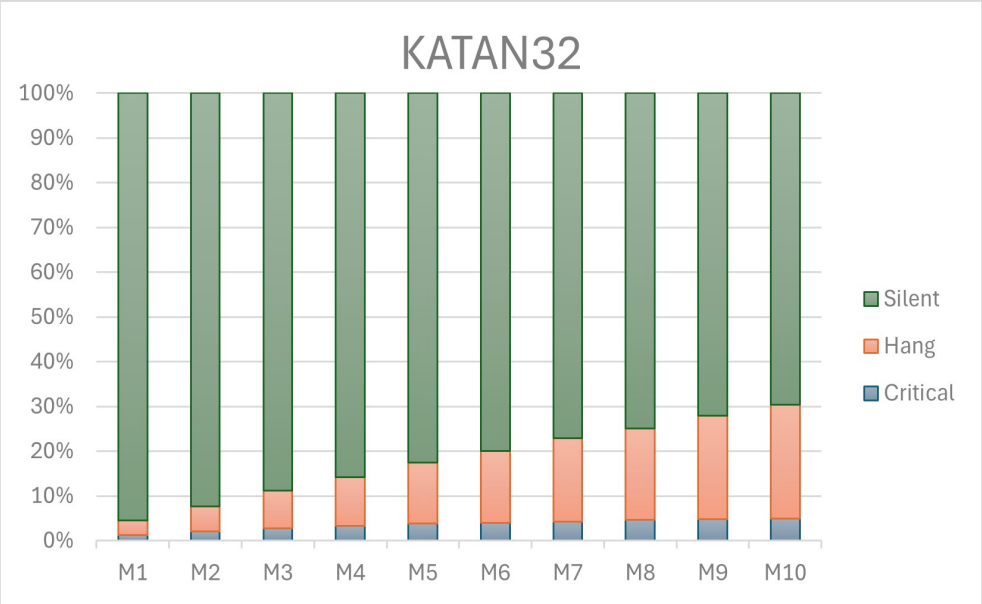

Figure 8.3: "KATAN32" Graph

8.4 Results of "SIMON64/128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1	17.06%	9.64%	73.29%
M2	24.42%	15.86%	59.72%
M3	27.86%	22.47%	49.67%
M4	29.12%	28.67%	42.22%
M5	28.89%	34.20%	36.91%
M6	28.16%	39.44%	32.39%
M7	27.24%	44.00%	28.75%
M8	26.26%	47.97%	25.77%
M9	25.30%	51.50%	23.20%
M10	24.19%	55.34%	20.47%
Average	25.85%	34.91%	39.24%
Maximum	29.12%	55.34%	73.29%
Minimum	17.06%	9.64%	20.47%

Table 8.10: Results of "SIMON64/128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M2	7.36%	6.22%	-13.57%
M2-M3	3.44%	6.61%	-10.06%
M3-M4	1.26%	6.20%	-7.45%
M4-M5	-0.23%	5.53%	-5.30%
M5-M6	-0.73%	5.25%	-4.52%
M6-M7	-0.92%	4.56%	-3.64%
M7-M8	-0.98%	3.97%	-2.99%
M8-M9	-0.96%	3.53%	-2.57%
M9-M10	-1.11%	3.84%	-2.73%
Average	0.79%	5.08%	-5.87%
Maximum	7.36%	6.61%	-13.57%
Minimum	-0.23%	3.53%	-2.57%

Table 8.11: Transition Results of "SIMON64/128"

Multiplicity	Critical Errors	Hang Errors	Silent Errors
M1-M5	11.83%	24.56%	-36.38%
M5-M10	-4.70%	21.14%	-16.45%
M1-M10	7.13%	45.70%	-52.83%

Table 8.12: Aggregate Transition Results of "SIMON64/128"

Critical Errors percentage records a total increase of 7.13%, with the minimum percentage value 17.06% at $M1$ and the maximum 29.12% at $M4$. The average percentage is 25.85%, differing 3.27% from the maximum percentage value and 8.79% from the minimum. From $M1$ to $M5$, the percentage is increased by 11.83%, however, between $M5 - M10$ the percentage of Critical Errors drops by 4.70%. The biggest increase appears from $M1$ to $M2$, rising by 7.36%. On the contrary, between $M9 - M10$ it records its greatest decrease by 1.11%. The percentage is continuously rising until $M4$, after which, it is constantly dropping. The average increment percentage of Critical Errors is 0.79%

Hang Errors percentage is constantly increasing, having its minimum value 9.64% at $M1$, and reaching its maximum value 55.34% at $M10$. The average percentage is 34.91%, differing 20.43% from the maximum percentage value and 25.27% from the minimum. In total, from $M1$ to $M10$, Hang Errors are increased by 45.70%, having increased by 24.56% from $M1$ to $M5$ and 21.14% from $M5$ to $M10$. The biggest increase appears from $M2$ to $M3$, rising by 6.61%, while between $M8 - M9$ the least increase by 3.53%. The average increment percentage of Hang Errors is 5.08%.

Silent Errors percentage fluctuation is opposite to Hang Errors. It starts from its maximum value 73.29% at $M1$, dropping to minimum value 20.47% at $M10$. The average percentage is 39.24%, differing 34.06% from the maximum percentage value and 18.77% from the minimum. In total, Silent Errors decrease by 52.83%, having decreased by 36.38% from $M1$ to $M5$ and 16.45% from $M5$ to $M10$. Silent Errors biggest drop take place from $M1$ to $M2$ where their percentage drops by 13.57%. It is interesting that from $M1$ to $M3$ the percentage drops by 23.63%, which is 7.18% greater than the decrease between $M5 - M10$. The percentage of the decrease between consecutive cases decreases as the multiplicities proceed. The average reduction percentage of Silent Errors is 5.87%.

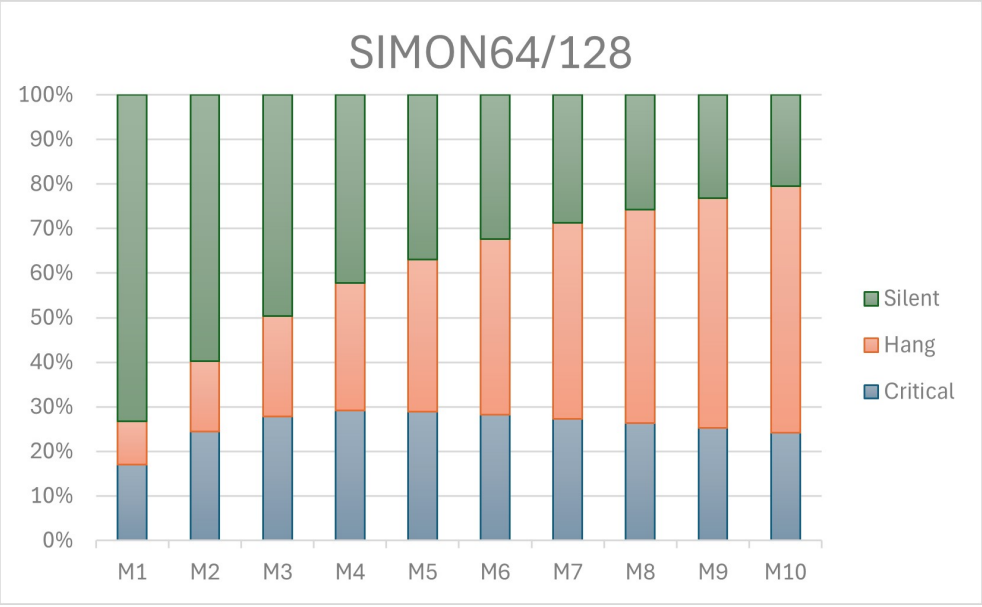

Figure 8.4: "SIMON64/128" Graph

Chapter 9

Conclusion

In this final chapter, we located key findings exported from our previous analysis. We compared each of the four(4) lightweight block ciphers against each other, taking into account their results from all their test cases $M1-M10$. In conclusion, we suggest possible further research on this topic.

9.1 Key Findings

The main findings of the study are presented below. The analysis focuses on trends in critical errors, hang errors, and silent errors across different multiplicities.

In brief, **GIFT-64-128** & **LED-64** share similar design principles, as they are based on **SPN structure and S-boxes**, being the most **stable regarding Critical Errors** among all tested ciphers. **KATAN32** uses **many rounds, nonlinear functions and Linear Feedback Shift Register (LFSR)** in order to ensure sufficient mixing. Overall, it shows more **stability regarding percentage changes between consecutive multiplicities**. Finally, **SIMON64/128** appears more **sensitive**, as by design relies heavily on **bitwise operations**.

9.1.1 Critical Errors

• **GIFT-64-128** shows a slow and stable increase in critical errors, with a significant difference between the first two(2) multiplicities, but the overall change remains minor beyond $M5$. Compared to the others ciphers it has the lower average of Critical Errors.

• **LED-64** also displays a stable increase in Critical Errors, peaking at 5.56% by $M8$. After $M5$, the percentage's differences are under the error margin and practically imperceptible.

• **KATAN32** shows a moderate increase, having a notable rise from $M1$ to $M5$. After "GIFT-64-128", it has the second lower average percentage of Critical Errors.

• **SIMON64/128** follows a pattern similar to "GIFT-64-128", with relatively low increments especially after $M4$, indicating stability in Critical Errors. However, the average percentage is by far the greatest, being bigger by twelve(12), five(5) and seven(7) times than "GIFT-64-128", "LED-64" and "KATAN32" respectively.

In general, regarding Critical Errors, the average percentage changes between consecutive multiplicities in "GIFT-64-128", "LED-64" and "KATAN32" are below the error margin, indicating imperceptible differences. This result may be related to their design primitives, as "GIFT-64-128" and "LED-64" uses SPN structure and S-boxes, and "KATAN32" has 254 rounds and two(2) nonlinear functions in order to ensure sufficient mixing. In contrast, "SIMON64/128" uses a Feistel structure and its round functions rely heavily on basic bitwise operations such as circular shifts, bitwise AND, and bitwise XOR, producing more notable percentage differences.

9.1.2 Hang Errors

• **GIFT-64-128** & **LED-64** show consistent and significant increases in Hang Errors, with "LED-64" having slightly higher average percentage.

• **KATAN32** has the lowest initial and average Hang Errors percentage. In addition, it shows the least overall increase compared to the others ciphers.

• **SIMON64/128** demonstrates moderate increases, lower than "GIFT-64-128" and "LED-64" but significantly higher than "KATAN32".

Regarding Hang Errors, all four(4) ciphers exhibit rising trend as the multiplicities increase. The similarities in the percentages between "GIFT-64-128" & "LED-64" may also be related to their similar design approach, indicating potential stability issues as multiplicity increases. Regarding "KATAN32", as its design ensures sufficient mixing, shows more stable increases by its average transition percentage.

9.1.3 Silent Errors

• **GIFT-64-128** & **LED-64** present the opposite trend from Hand Errors. They both show substantial decreases in Silent Errors, reflecting overall reduction close to 60%.

• **KATAN32** has the highest initial Silent Error percentage, and while it has a decreasing trend like all the others ciphers, it records the smallest decrease.

• **SIMON64/128** exhibits a similar downward trend in Silent Errors as the rest of the ciphers but it has the lowest initial percentage. It also has the smallest average percentage.

Silent Errors demonstrate reversed trend from the Hang Errors, as all four(4) ciphers show decreasing trend as the multiplicities increase. "GIFT-64-128" & "LED-64" percentages decreases significantly, suggesting improvements in these implementations' error handling mechanisms as multiplicities progress. "KATAN32", despite starting high, shows less improvement as it only drops by 25.88%. "SIMON64/128" follows similar downward trend with "GIFT-64-128" & "LED-64".

9.2 Further Research

First of all, further studies should focus on expanding Thesis's research without the restrictions of tight time schedule and resources limitation. Under those conditions, the error margin could be noticeably smaller, e.g. 0.1%. In this case, the exponentially increasing simulation time required would not be a problem.

Furthermore, future research could continue by examining the effects of more combinations of directives. Several papers such as [57] and [58], shape the direction of study over this topic, but the research can be expanded more.

Next steps could be related to fault mitigation techniques. Future studies could focus on developing and testing fault mitigation strategies specifically designed for HLS-generated cryptographic hardware. This could involve techniques such as redundancy, error-detection codes, and fault-tolerant synthesis configurations that can be adapted to lightweight ciphers without significantly impacting performance.

Under the umbrella of fault mitigation techniques, application of machine learning for fault prediction could be further investigated. Leveraging machine learning algorithms could offer a novel approach to predict and preempt fault-induced vulnerabilities in HLS implementations. Future research could focus on training models with fault injection data to predict susceptible areas within cryptographic designs, enabling proactive security optimizations during the HLS process.

Bibliography

[1] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges", *2011 9th IEEE International Conference on ASIC*, pp. 1102–1105, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:7787188.

[2] J. P. Elliott, "Understanding behavioral synthesis", in *Springer US*, 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:109554903.

[3] M. Forconesi, G. Sutter, S. Lopez-Buedo, J. E. Lopez de Vergara, and J. Aracil, "Bridging the gap between hardware and software open source network developments", *IEEE Network*, vol. 28, pp. 13–19, 2014. DOI: 10.1109/MNET.2014.6915434.

[4] E. Journal. "EDA (electronic design automation) - where electronics begins". (2013), [Online]. Available: https://embedjournal.com/eda-where-electronics-begins/.

[5] Wikipedia. "Electronic design automation". (2024), [Online]. Available: https://en.wikipedia.org/wiki/Electronic_design_automation.

[6] M. Lavagno and Scheffer, *Electronic Design Automation for Integrated Circuits Handbook*. Taylor and Francis, 2006.

[7] S. Tripathi, S. Saxena, and S. K. Mohapatra, *Advanced VLSI Design and Testability Issues*. CRC Press, 2020.

[8] V. A. Pedroni, *Circuit design with VHDL*. MIT press, 2020.

[9] Xilinx, *Introduction to FPGA design with vivado high-level synthesis*, version UG998 (v1.1), 2019. [Online]. Available: https://docs.amd.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls.

[10] S. A. V. Alfred J. Menezes Paul C. van Oorschot, *Handbook of Applied Cryptography*. CRC Press, 2018.

[11] "Principles of modern cryptography". (), [Online]. Available: http://www.queen.clara.net/pgp/art6.html.

[12] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[13] J. Kaltz and Y. Lindell, *Introduction to modern cryptography: Principles and protocols*, 2008.

[14] "Stream ciphers". (2015), [Online]. Available: https://cryptosmith.com/2007/06/07/stream-ciphers/.

[15] NIST. "Block cipher modes - block cipher techniques: CSRC". (), [Online]. Available: https://csrc.nist.gov/projects/block-cipher-techniques/BCM.

[16] V. R. Yunwen Liu and G. Leander, "Nonlinear diffusion layers", *Designs, Codes and Cryptography*, 2018.

[17] "Symmetric block ciphers". (2017), [Online]. Available: https://www.tenminutetutor.com/data-formats/cryptography/symmetric-block-ciphers/.

[18] NIST. "Cybersecurity". (2024), [Online]. Available: https://www.nist.gov/cybersecurity.

[19] NIST. "Cybersecurity framework". (2024), [Online]. Available: https://www.nist.gov/cyberframework.

[20] NIST. "Cryptography". (2024), [Online]. Available: https://www.nist.gov/cryptography.

[21] NIST. "Lightweight cryptography: CSRC". (2017), [Online]. Available: https://csrc.nist.gov/Projects/lightweight-cryptography.

[22] NIST. "Block cipher techniques: CSRC". (2017), [Online]. Available: https://csrc.nist.gov/projects/block-cipher-techniques.

[23] NIST. "FIPS general information". (2023), [Online]. Available: https://www.nist.gov/itl/fips-general-information.

[24] Y. L. Kazuo Sakiyama Yu Sasaki, *Security of block ciphers: From algorithm design to hardware implementation*. Wiley, 2015.

[25] C. O. Jasper van Woudenberg, *The hardware hacking handbook: breaking embedded security with hardware attacks*. No Starch Press, 2022.

[26] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design", in *Proceedings of the 50th Annual Design Automation Conference*, Association for Computing Machinery, 2013, pp. 1–10.

[27] G. Kanawati, N. Kanawati, and J. Abraham, "EMAX - an automatic extractor of high-level error models", in *9th Computing in Aerospace Conference*. 1993, pp. 1297–1306.

[28] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, *et al.*, "IBM experiments in soft fails in computer electronics (1978–1994)", *IBM Journal of Research and Development*, vol. 40, pp. 3–18, 1996.

[29] W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems", in *Proceedings of the 1969 24th National Conference*, Association for Computing Machinery, 1969, pp. 295–309.

[30] T. F. Arnold, "The concept of coverage and its effect on the reliability model of a repairable system", *IEEE Trans. Comput.*, pp. 251–254, 1973.

[31] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis", in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 168–179.

[32] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults", in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 11–16.

[33] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence", in *2009 Design, Automation Test in Europe Conference Exhibition*, 2009, pp. 502–506.

[34] "The GIFT block ciphers". (), [Online]. Available: https://giftcipher.github.io/gift/.

[35] A. Bogdanov, L. Knudsen, G. Leander, *et al.*, "PRESENT: An ultra-lightweight block cipher", in *Cryptographic Hardware and Embedded Systems - CHES 2007*, 2007.

[36] S. Banik, S. Pandey, T. Peyrin, Y. Sasaki, S. Sim, and Y. Todo, "GIFT: A small present towards reaching the limit of lightweight encryption", in *Cryptographic hardware and embedded systems - CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, Springer, 2017.

[37] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems", in *Advances in Cryptology-CRYPTO' 90*, Springer Berlin Heidelberg, 1991, pp. 2–21.

[38] M. Matsui, "Linear cryptanalysis method for DES cipher", in *Advances in Cryptology — EUROCRYPT '93*, Springer Berlin Heidelberg, 1994, pp. 386–397.

[39] E. Biham, A. Biryukov, and A. Shamir, *Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials*, Journal of Cryptology, 2005.

[40] J. Kelsey and B. Schneier, "Key-schedule cryptanalysis of DEAL", in *Selected Areas in Cryptography*, Springer Berlin Heidelberg, 2000, pp. 118–134.

[41] A. Bogdanov and C. Rechberger, "A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN", in *Selected Areas in Cryptography*, Springer Berlin Heidelberg, 2011, pp. 229–240.

[42] Y. Todo, G. Leander, and Y. Sasaki, *Nonlinear invariant attack –practical attack on full SCREAM, iSCREAM, and midori64*, 2016.

[43] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, *The LED block cipher*, Cryptology ePrint Archive, Paper 2012/600, This version describes a minor modiffcation to the original LED description, 2012.

[44] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw, "The LED block cipher", in *Cryptographic Hardware and Embedded Systems – CHES 2011*, Springer Berlin Heidelberg, 2011, pp. 326–341.

[45] W. Li, W. Zhang, D. Gu, *et al.*, "Impossible differential fault analysis on the LED lightweight cryptosystem in the vehicular ad-hoc networks", *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 1, pp. 84–92, 2016.

[46] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the limits: A very compact and a threshold implementation of AES", in *Advances in Cryptology – EUROCRYPT 2011*, Springer Berlin Heidelberg, 2011, pp. 69–88.

[47] T. P. Jian Guo and A. Poschmann, *The PHOTON family of lightweight hash functions*, 2011.

[48] A. Biryukov and D. Wagner, "Slide attacks", in *Fast Software Encryption*, Springer Berlin Heidelberg, 1999, pp. 245–259.

[49] C. De Cannière, O. Dunkelman, and M. Knežević, "KATAN and KTANTAN — a family of small and efficient hardware-oriented block ciphers", in *Cryptographic Hardware and Embedded Systems - CHES 2009*, Springer Berlin Heidelberg, 2009, pp. 272–288.

[50] C. Clavier and K. Gaj, *Cryptographic Hardware and Embedded Systems - CHES 2009*. Springer Berlin Heidelberg, 2009.

[51] S. F. Abdul-Latip, M. R. Reyhanitabar, W. Susilo, and J. Seberry, "Fault analysis of the KATAN family of block ciphers", in *Information Security Practice and Experience*, Springer Berlin Heidelberg, 2012, pp. 319–336.

[52] I. Dinur and A. Shamir, "Cube attacks on tweakable black box polynomials", in *Advances in Cryptology - EUROCRYPT 2009*, Springer Berlin Heidelberg, 2009, pp. 278–299.

[53] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, *The SIMON and SPECK families of lightweight block ciphers*, Cryptology ePrint Archive, Paper 2013/404, 2013.

[54] Z. Xiang, W. Zhang, and D. Lin, "On the division property of simon48 and simon64", in *Advances in Information and Computer Security*, Springer Berlin Heidelberg, 2016, pp. 147–163.

[55] J. Wetzels and W. Bokslag, *Simple SIMON: FPGA implementations of the SIMON 64/128 block cipher*, Cryptology ePrint Archive, Paper 2016/029, 2016.

[56] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES", in *Cryptographic Hardware and Embedded Systems - CHES 2006*, Springer Berlin Heidelberg, 2006, pp. 201–215.

[57] L. Zhang, D. Mu, W. Hu, Y. Tai, J. Blackstone, and R. Kastner, "Memory-based high-level synthesis optimizations security exploration on the power side-channel", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2124–2137, 2019.

[58] K. Georgopoulos, G. Chrysos, P. Malakonakis, *et al.*, "An evaluation of vivado HLS for efficient system design", in *International Symposium ELMAR*, IEEE, 2016, pp. 195–199.