GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Hardware Trojans: implementation and testing

Autor: Luis Foncillas Gutiérrez

Director: Erdal Oruklu

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Hardware Trojans: implementation and testing

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2022/23 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Luis Foncillas Gutiérrez Fecha: …3…/ …7…/ 2023

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Dr. Erdal Oruklu Fecha: 03…/ 07…/ 2023

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Hardware Trojans: implementation and testing

Autor: Luis Foncillas Gutiérrez

Director: Dr. Erdal Oruklu

Madrid

TROYANOS HARDWARE: IMPLEMENTACIÓN Y VERIFICACIÓN

Autor: Foncillas Gutiérrez, Luis.
Director: Oruklu, Erdal.
Entidad Colaboradora: Illinois Institute of Technology

RESUMEN DEL PROYECTO

La producción de circuitos integrados es un proceso global, en cuyo diseño y producción intervienen varias casas de diseño y fundiciones de todo el mundo. Este proceso de producción global ha abierto la puerta a que agentes maliciosos introduzcan modificaciones, conocidas como troyanos de hardware, en el diseño del circuito con el fin de perturbar la funcionalidad diseñada. El proyecto diseña e implementa troyanos de hardware en un conjunto de circuitos de encriptado, utilizando VHDL y realizando pruebas para un dispositivo FPGA, ideando así posibles mecanismos de defensa contra tales intrusiones.

Palabras clave: Troyano de hardware, Circuitos integrados, *Design for Security*

1. Introducción

La cadena de suministro de circuitos integrados implica a varios sujetos repartidos por todo el mundo, la falta de confianza en fabricantes terceros o licenciatarios de propiedad intelectural supone un riesgo en la integridad y seguridad de dichos circuitos, abriendo la posibilidad a una modificación maliciosa del hardware conocida como Troyano de Hardware. Estos troyanos pretenden impedir la funcionalidad o dificultar el rendimiento de los circuitos atacados.

El proyecto tendrá como objetivo insertar un troyano de hardware en un circuito criptográfico, tratando de filtrar información/denegar servicio, esencialmente inutilizando su funcionalidad e idear y explorar potenciales métodos de defensa contra este tipo de ataques, y aplicarlos a los circuitos diseñados.

2. Definición del proyecto

La relativa falta de conocimientos sobre los troyanos de hardware y sus implementaciones hace que los métodos de defensa y prevención sean escasos o caros. Este proyecto pretende, primero actuando como agente malicioso en un conjunto de circuitos criptográficos, y luego estudiando técnicas de defensa contra troyanos de hardware, idear un nuevo método a través del cual defenderse adecuadamente contra la inserción de dichos troyanos. Estos esfuerzos conducirán a hallazgos relevantes para el desarrollo en FPGAs, utilizando código VHDL para primero insertar y posteriormente prevenir troyanos.

Se ha desarrollado un conjunto de dos circuitos criptográficos para el proyecto, cuyo conocimiento a fondo ha permitido un acercamiento a la metodología detrás del desarrollo de troyanos hardware. Una vez hecho esto, la siguiente parte del estudio se centró en la defensa contra troyanos. Con lo aprendido al atacar los circuitos, aprovechando sus debilidades y, basándonos en las técnicas de prevención de troyanos

ya establecidas, se aplicaron los esfuerzos de defensa a los circuitos criptográficos desarrollados.

3. Descripción del modelo/sistema/herramienta

En el proyecto se desarrollaron dos circuitos criptográficos, el primero, un prototipo sencillo, desarrollado con un simple xor y cifrado César. El segundo circuito es más complejo, utilizando el prototipo como base, incluye una máquina de estados para controlar el proceso de cifrado. A continuación, se estudió la posibilidad de insertar un troyano en ambos circuitos de cifrado. El primer troyano, con un mecanismo de activación externo, se insertó en el prototipo. Al insertar una entrada específica al circuito, el troyano se activa, filtrando la clave de cifrado a través de la salida habitual. El segundo diseño de troyano se desarrolló para, además de afectar a la función de cifrado, perturbar la máquina de estados del segundo diseño de circuito. Para explotar mejor la arquitectura del circuito, se eligió un mecanismo de activación interno. En función del número de cifrados, el troyano se activaría para realizar diferentes acciones, primero filtrando la clave de cifrado, después denegando el servicio mediante la salida cero y eliminando los esfuerzos de cifrado, y finalmente interrumpiendo el flujo de la máquina de estados, bloqueando el circuito hasta su reinicio.

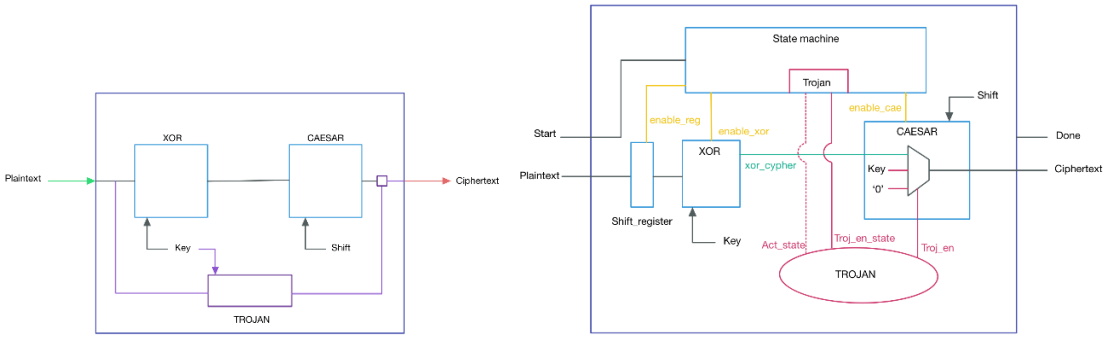

Figura 1: Circuitos de encriptado con troyanos

Para finalizar el estudio, y con los conocimientos obtenidos de la inserción del troyano, se implementaron mecanismos de defensa de los circuitos, implementando técnicas DFS (Design for Security) y métodos de verificación efectivos sobre los circuitos, modificando el diseño.

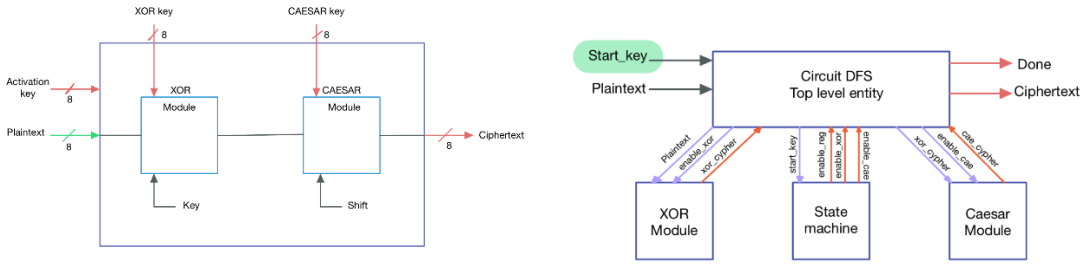

Figura 2: Circuitos reforzados con técnicas de defensa DFS

4. Resultados

Todas las simulaciones del troyano proporcionaron la funcionalidad correcta, demostrando una implementación efectiva, especialmente el segundo diseño, que ofreció una buena visión de la metodología detrás del diseño de troyanos de hardware. Tenga en cuenta que la clave de cifrado para ambos diseños es 01101110.

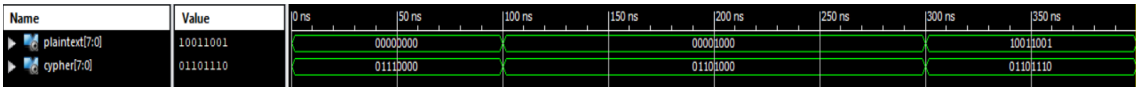

Figura 3: Simulación del primer troyano

input	output
00000000	01110000
00001000	01101000
10011001	01101110

Tabla 1 : Resultados de la simulación del primer troyano

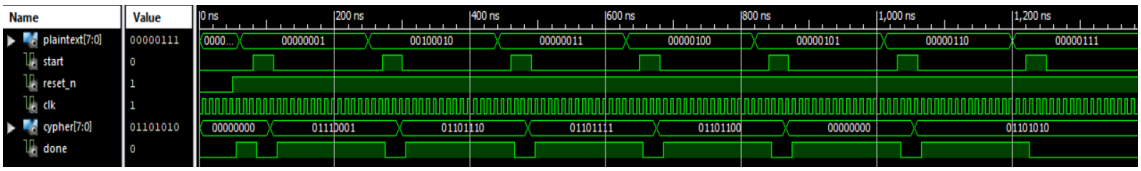

Figura 4: Simulación de troyano de la máquina de estados

input	output
00000001	01110001
00100010	01101110
00000011	01101111
00000100	01101100
00000101	00000000
00000110	01101010
00000111	01101010

Tabla 2 : Resultados de la simulación del segundo troyano

Los métodos de defensa también resultaron eficaces, dificultando la inserción de un troyano o, si este estaba dentro, ofreciendo buenas posibilidades de detección, especialmente en el primer circuito. Los problemas surgieron en el segundo diseño, ya que el mecanismo de activación, al ser interno, es difícil de detectar con los métodos de verificación tradicionales presentes en FPGA.

5. Conclusiones

El principal objetivo del proyecto, comprender los troyanos de hardware, fue un éxito. Se obtuvo una visión útil de cómo se diseñan e implementan los troyanos en el hardware del host, lo que proporcionó una primera visión de cómo defenderse adecuadamente contra este tipo de ataques. Desde el punto de vista de la defensa, la parte de ataque del estudio mostró cómo un cambio de perspectiva puede ofrecer una estructura más robusta y resistente a la inserción de troyanos. La exploración de los retos de implementar únicamente una defensa VHDL demostró que ningún método o vía es infalible en la defensa contra troyanos, reforzando la necesidad de incorporar varias tácticas defensivas.

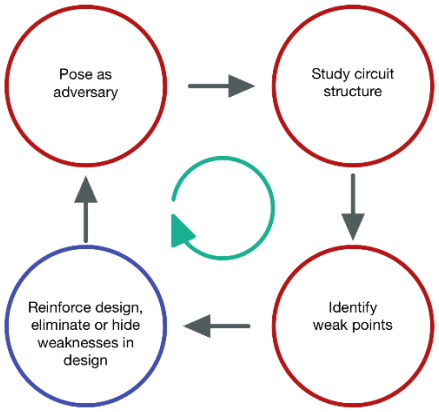

Figura 5: estrategia de defensa basada en bucle ataque-defensa

El hecho de haber actuado como un adversario, intentando insertar un troyano de hardware en los circuitos, ha demostrado que es una técnica muy efectiva en preparar mejor la defensa de los circuitos, pues se conocen de antemano sus debilidades. El bucle de ataque-defensa realizado en el proyecto ha demostrado su valor como técnica a la hora de defender de intrusiones de troyanos.

HARDWARE TROJANS: IMPLEMENTATION AND TESTING

Author: Foncillas Gutiérrez, Luis.
Supervisor: Oruklu, Erdal.
Collaborating Entity: Illinois Institute of Technology

ABSTRACT

Modern integrated circuit production is a global process, various design houses and manufacturing foundries from over the world intervene in the design and production of a single circuit. This global production process has opened the door for malicious agents to insert modifications, known as hardware trojans, in the circuit design aiming to disrupt the intended functionality. The project designs and implements hardware trojans on a set of encryption circuits, using VHDL and testing for an FPGA device, devising then potential defense mechanisms for such intrusions.

Keywords: Hardware Trojans, Integrated circuits, Design for Security.

1. Introduction

Modern Integrated circuit supply chain involves various actors spread across the globe, the lack of trust in third party manufacturers or IP licensers suppose a risk in the integrity and security of said circuits, opening the chance for a malicious hardware modification known as a Hardware Trojan. These trojans aim to impede functionality or hamper performance of the target circuits.

The project will aim to insert a hardware Trojan into a cryptographic circuit, trying to leak information/deny service, essentially rendering the cryptographic effort useless. Once these efforts have been completed, potential defense methods will be explored and applied to the designed circuits.

2. Project definition

The relative lack of knowledge about hardware trojans and their implementations mean that defense and prevention methods are scarce or expensive. This project pretends, first by acting as the malicious agent on a set of cryptographic circuits, and then studying modern hardware trojan defense techniques, to devise a new method trough which to properly defend against hardware trojan insertion. These efforts will lead to findings relevant to FPGA development, using VHDL code to first insert and then prevent hardware trojans.

A set of two cryptographic circuits have been developed for the project, with their intimate knowledge and study allowing for a proper study of the methodology behind hardware trojan development. Once this had been done, the next part of the study focused on trojan defense. Seeing what posing as the adversary shows regarding exploiting circuit weaknesses, basing our effort on established trojan prevention techniques, defense efforts were applied to the cryptographic circuits developed.

3. Implementation

To start with two cryptographic circuits were developed, the first, a simple prototype, as developed with a simple xor and Caesar cipher encryption. The second circuit is more complex, using the prototype as a base, it includes a state machine to control the encryption process. Both encryption circuits were then studied to have a trojan inserted in them. The first trojan, with an external triggering mechanism, was inserted in the prototype. If a specific input plaintext is sent to the circuit, the trojan activates, leaking the encryption key through the usual encryption output. The second trojan design was developed to, apart from affecting the encryption function, to target the state machine of the second circuit design. To better exploit the circuit architecture, an internally triggering mechanism was chosen. Based on the number of encryptions, the trojan would activate to do different actions, first leaking the encryption key, then denying service by outputting zero and eliminating encryption efforts, and finally disrupting the state machine flow, essentially locking the circuit until reset.

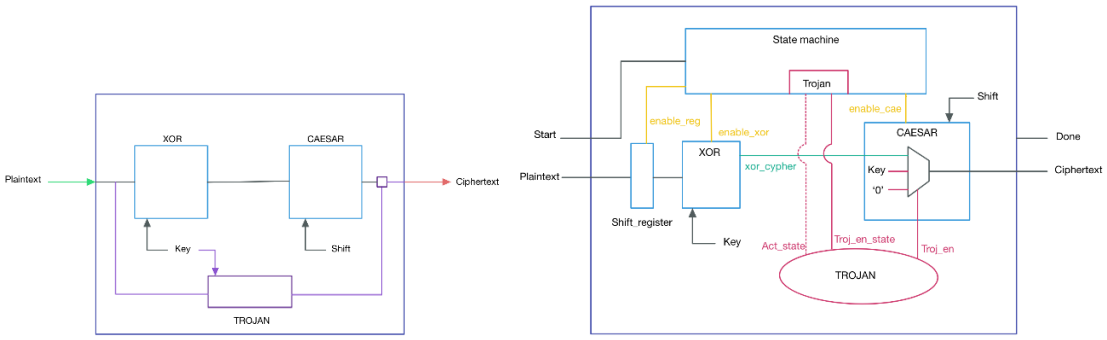

Figure 1: Encryption circuits schematics with trojan present

To end the study, and with the knowledge gained from the trojan insertion, defense mechanisms for the circuits, implementing DFS techniques and effective verification methods were implemented on the circuits, modifying the design.

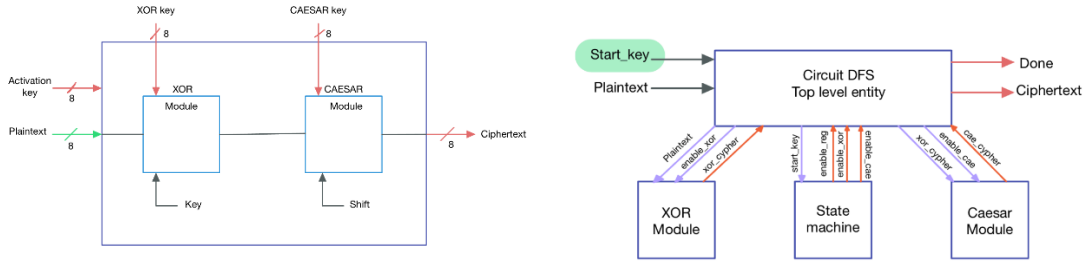

Figure 2: Reinforced designs with defense techniques

4. Results

All trojan simulations gave the correct functionality, demonstrating an effective implementation, specially the second design, which offered good insight in the methodology behind hardware trojan design. Please note that the encryption key for both designs is 01101110.

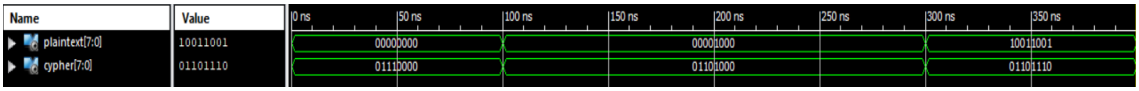

Figure 3: Prototype circuit trojan simulation

input	output
00000000	01110000
00001000	01101000
10011001	01101110

Table 1: Prototype circuit trojan simulation outputs

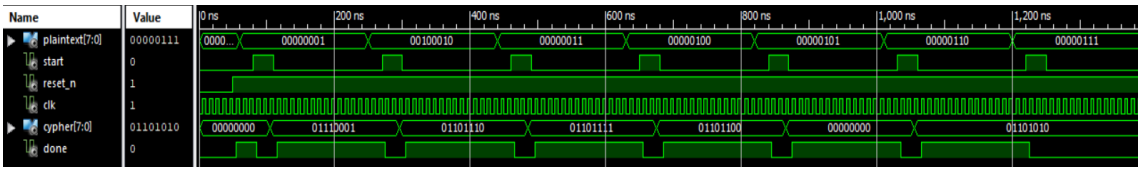

Figure 4: Second circuit trojan simulation

input	output
00000001	01110001
00100010	01101110
00000011	01101111
00000100	01101100
00000101	00000000
00000110	01101010
00000111	01101010

Table 2: Second circuit trojan simulation outputs

The defense methods also proved effective, hampering trojan insertion efforts, or if inserted, offering a good chance for detection, specially in the first prototype circuit. Problems arose in the second design, seeing as the triggering mechanism, it being internal, is difficult to detect using traditional verification methods present for FPGA development.

5. Findings

The main goal of the project, understanding hardware trojans was successful. Useful insight was gained in how the trojans are designed and implemented on host hardware, providing an early look in how to properly defend against these types of attacks. From the defense point of view, the attack portion of the study showed how a change of perspective can offer a more robust structure resilient to trojan insertion. Exploring the challenges in implementing only a VHDL defense proved that no method or avenue is foolproof in the defense against trojans, reinforcing the need to incorporate various defensive tactics.

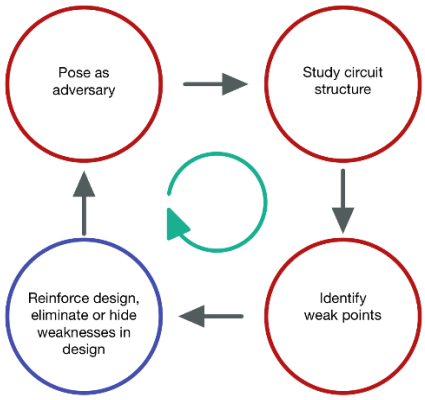

Figure 5: Defense method proposal based on feedback loop.

Posing as an attacker however has shown that it is a very effective technique in the fight against hardware trojans. The weak points of a circuit can be better known and reinforced if one tries attacking it first. A feedback loop of attack-defense is an effective method of reinforcing the design and structure of a circuit in the face of trojan attacks.

CONTENTS

FIGURES

TABLES

CHAPTER 1: INTRODUCTION

As the hardware industry has advanced over the last couple of decades, design houses and foundries from across the world intervene in the design and fabrication of a single circuit. The global economy has allowed for the offloading of many processes in remote locations [1], meaning that design and fabrication are usually separated into two distinct and separate processes. The outsourcing of production, nowadays relegated to external foundries, and the fact that many smaller components that build modern integrated circuits (IC) are sourced from third party design houses, raises questions about the integrity of the circuits, seeing as there are many open avenues for a malicious agent, an *adversary*, to introduce changes in the design without the original design houses noticing.

These malicious changes, known as Hardware Trojans [2], have emerged as a major security threat for most modern IC. The rise of System-on-Chip (SoC) designs, which integrate most or all components of a computer on a single IC, and embedded computing has allowed the public to have access to highly capable computing devices, such as smartphones or laptops, and low powered machines, such as wearable medical devices or electronic car keys. This physical access to technology has contributed to the rise of hardware security modules on the computing elements themselves, in the form of specialized circuitry. With so many people accessing sensitive information on their devices, such as banking information or private legal documents, security has been shown to be imperative, as even the slightest mistake can lead to millions of affected users, with no real easy way to fix a Hardware problem other than mass recalls.

Figure 1 : Apple M1 Pro SoC [4]

It is in this context that hardware trojans show their potential danger. Hardware Trojans relate to a malicious modification of an IC during its design or fabrication process, especially when handled by external agents, such as an untrusted design house or foundry, or even when designed with the use of third-party tools or components over which the original design house has no control. These modifications aim to modify the functionality of an IC, reducing performance, changing behavior, or even neutralizing the computer in its totality, acting as a "kill-switch". These changes are introduced by the *adversary* in a manner such that standard verification tests will not detect the intrusion, with the Trojans presence only being revealed after prolonged operation in the hands of consumers.

The name comes from the Trojan war in mythical ancient Greece, where a wooden horse was gifted to the Trojan army, who brough it inside their city, thought impenetrable, without the knowledge that in the inside of the horse the enemy's forces were hiding. The horse turned into a powerful weapon that came to be one of the major causes for the fall of Troy [5]. The name then gives the Hardware Trojans their nature: it is intended to be a weapon with malicious intent, and should try to evade all detection, acting stealthily when under standard circuit verification processes.

As stated before, the global economy has been one of the main contributing factors to the rise in Hardware Trojan attacks. Modern IC design techniques usually involve using components and intellectual property cores derived from third parties, which design firms have to trust, but have no real guarantee over the integrity and security measures implemented in said components. Even the tools used to design IC may be vulnerable to tampering, leaving design firms oblivious to the actual end result of their product. The problem grows when considering the manufacturers themselves, as economically the most logical solution for a lot of teams is to outsource production, due to the high cost of having an in-house manufacturing solution, even if the manufacturing partner operates in an insecure facility. This globalization has forced the design firms to let go of the control they once had over the security measures they require, having to trust their partners to enforce them fully. The many steps involved in modern IC design and manufacturing all pose an opportunity for attack, seeing as each different partner involved in the overall process can be a possible adversary.

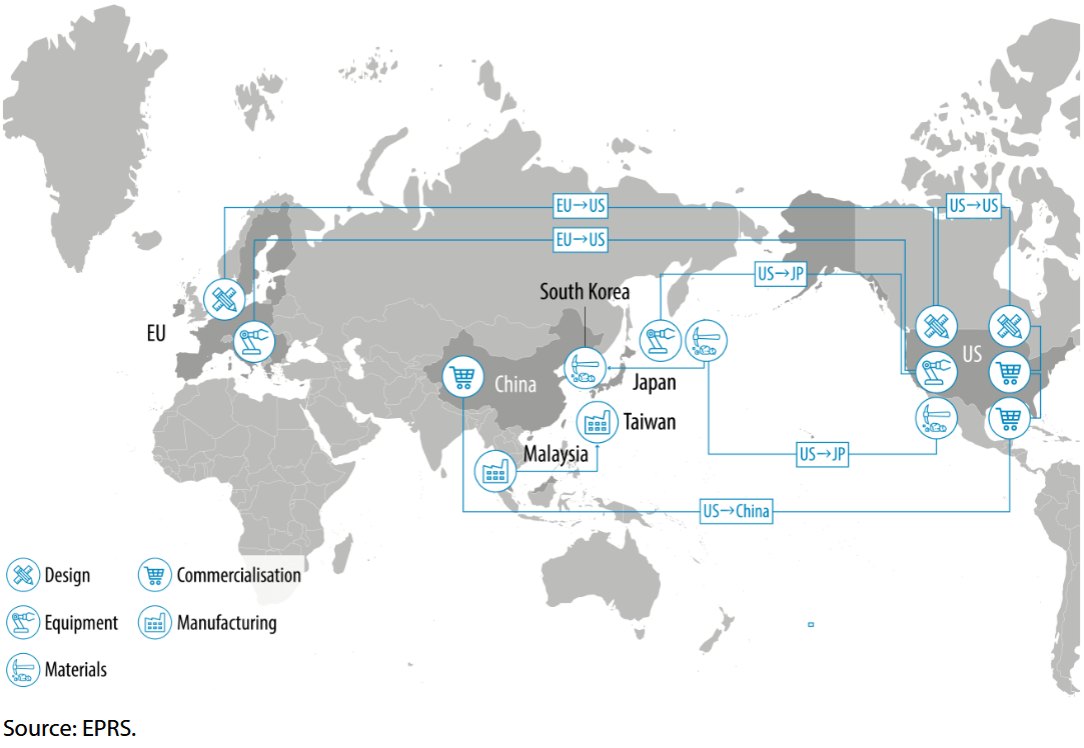

Figure 2 : Global semiconductor supply chain [6]

In recent times Hardware Trojan attacks have been discovered, with a wide range of impacts. In 2012 a hidden back-door was discovered in military systems and aircraft such as the Boeing 787 [7] , allowing the adversary to take control of the flight remotely through the internet. Or in 2007, Israeli forces managed to destroy a nuclear reactor, due to the Syrian air defense system not responding in time, this was speculated to have been caused by a built-in kill-switch on the system [3]. No system is completely secure for hardware tampering, meaning that potentially every circuit in the wild today is susceptible to a Hardware trojan attack.

Modern verification techniques should be capable of detecting these unwanted and hazardous modifications. But the modern nature of IC design has complicated the issue[8], mainly due to the lack of a golden model [9], a reference model of the entire built circuit. Licensed IP cores from third parties may not be tested and simulated properly, leading to not being able to recreate a golden model. Exhaustive verification would be the way to go, but on most modern IC it is not feasible to test, as the possibilities far exceed any reasonable time limits for simulation [10]. Once the IC leave the production run, they could be tested, either via reverse engineering, tested against a reference model, or via side-channel analysis, but these options prove to be very costly, either due to the tools necessary for said tests, or may be ineffective due to the fact that not all of the IC manufactured could have been affected by the Hardware Trojan attack, and if the sample size tested is not sufficiently large, due to cost or time constraints, the attack may go unnoticed.

1.1 MOTIVATION

The very present danger Hardware Trojans pose to our everyday lives highlight the pressing need to gain a deeper understanding of the entire lifecycle of hardware trojans, ranging from their design to their insertion. The more we comprehend how these trojans operate and how they manage to remain undetected, the better equipped we become to mitigate the risks they pose. It is crucial to acknowledge that hardware trojans extend beyond targeting large, centralized computer systems. Their true potential for damage lies

in compromising consumer devices, where they can illicitly acquire sensitive information for malicious purposes or disrupt interconnected networks on a significant scale.

The primary objective of this project is to foster comprehension. By thoroughly studying the intricacies involved in designing and implementing a hardware trojan, we can effectively develop defensive strategies and explore detection methods. Essentially, by adopting an adversarial mindset and analyzing the various stages of an attack, we can enhance our ability to protect against such threats.

1.2 OBJECTIVES

The project will aim to develop the following:

1) Design a simple cryptographic circuit: this circuit will be a very simple VHDL implementation of a cryptographic circuit, using XOR encryption and Caesar cipher as a base, as cryptography is not the focus of the project, and it needs to be in scale of the FPGA board available.
2) Design and implement a Hardware trojan: based on the circuit previously developed, a set of hardware trojans will be designed and inserted into the circuit, trying to explore different avenues for attacks and exploit the structural weaknesses of the circuits. It is in this point where the methodology behind Hardware Trojan attacks will be studied and applied, hopefully leading to a better understanding of the process.
3) Test the hardware trojan: simulation and verification of the Trojans inserted in the cryptographic circuits.
4) Exploring possible methods of defense for the trojan implementations: for the previously designed trojans approaches for defense or detection of the trojans will be explored and developed, and their effectiveness will be studied.

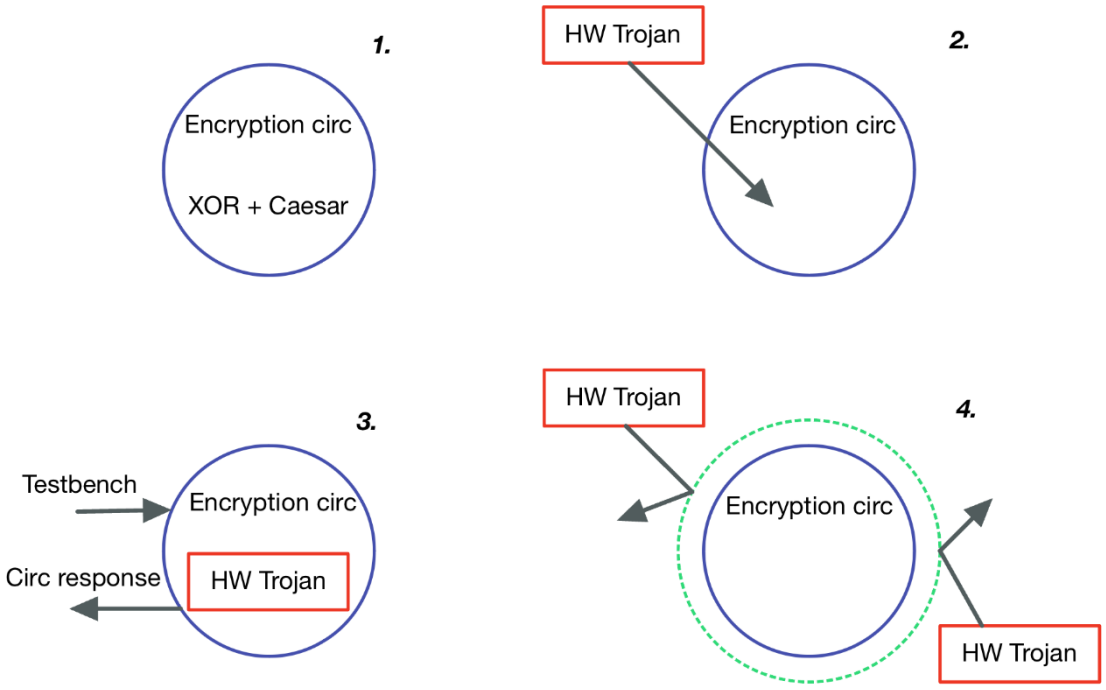

Figure 3 : Project objectives

1.3 METHODOLOGY

The project will be based on a feedback loop of progress. To start with, a simple prototype cryptographic circuit will be designed. This circuit will then be subjected to a Hardware Trojan attack, since we have participated in both the design and the attack, the inner workings of the circuit will be well-known to us, with its weak points being known from the get-go. This intimate knowledge of the circuit will allow us to tailor the hardware trojan to the circuit, trying to affect its main functionality, and do so in a stealthy manner. From then, a more comprehensive study will be conducted on how the Trojan has affected the circuit, allowing us to devise a coherent defense structure for the existing design, finalizing in a new circuit. This process will then be repeated on a more complex circuit, with different characteristics.

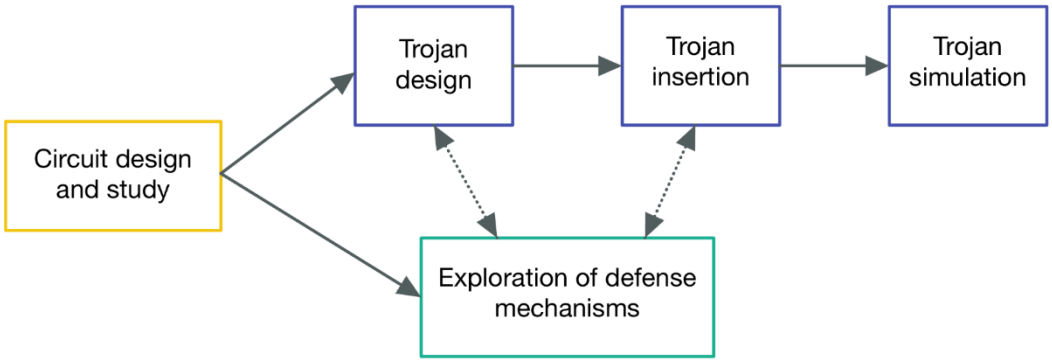

Figure 4 : project methodology

With this process, the nature of Hardware Trojan attacks will be better understood, and with the knowledge gained from posing as the attacker, the exploration of defense mechanisms will be more effective, and lead to a more structured process to better defend against hardware Trojans. Doing this process twice will create a feedback loop of knowledge, when designing the second more complex circuit, the lessons learned from the prototype can be applied, allowing for a comprehensive study on the nature of these attacks. During each step simulations to verify and test both the circuits and the Trojans will be done.

CHAPTER 2: DESCRIPTION OF TECHNOLOGIES

The project will have a focus on the study of Hardware Trojans on an FPGA environment; therefore, the choice of tools and platforms must be adequate for FPGA development and implementation.

2.1 CHOICE OF PLATFORM: FPGA

A Field-Programmable Gate Array chip is a reprogrammable integrated circuit that allows users to design and implement digital logic circuits [11] - [12]. FPGA boards provide a platform for designing, prototyping, and deploying digital circuits and systems. They offer flexibility and configurability, allowing users to define the desired functionality of the FPGA chip by programming it using hardware description languages like VHDL or Verilog. With various I/O interfaces, clock management resources, and on-board components, FPGA boards enable the development of custom digital circuits tailored to specific applications and can be used in diverse fields such as embedded systems, digital signal processing, communication systems, and more.

Figure 5 : Xilinx Spartan 6 FPGA chip [13]

The specific FPGA used for the project is a Xilinx Spartan 6 board. It is a low-cost, high-capacity FPGA, balancing power consumption, performance and cost [13]. The Spartan

6 series uses dual-register, 6-input LUTs, and one Series of built-in system-level modules, which include an SDRAM memory interface, PCIe interface, 18Kb Block Ram, and a robust hybrid clock management module.

2.2 VHDL

The project will be based on an FPGA platform; therefore, a hardware description language will be used. In our case, the main hardware description language of choice will be VHDL, as it offers intuitive and easy solutions for both the design of the cryptographic circuits and their respective trojans, and the later simulation and verification of the implementations developed via a VHDL testbench.

VHDL (Very High-Speed Integrated Circuit Hardware Description Language) is a hardware description language used to model and simulate digital systems. It is a standardized language that enables designers to describe the structure and functionality of electronic systems, such as integrated circuits, programmable logic devices, and system-on-chip designs [14] - [15].

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           Y : out  STD_LOGIC_VECTOR (7 downto 0));
end AND;
architecture Behavioral of AND is

begin
    Y = A and B;

end Behavioral;
```

VHDL was created in a United States Department of Defense program, Very High-Speed Integrated Circuits Program (VHSIC) [15]. The program aimed to develop a new HDL for use in integrated circuit development, which resulted in VHDL version 7,2, released in 1985, with IEEE standardization efforts beginning in the following year.

VHDL allows designers to specify the functionality of a digital system using a combination of concurrent and sequential statements. It supports the representation of complex digital circuits and systems by providing a hierarchical structure for modular design, based on the behavioral model design, which is normally used to describe the functionality of an inner module of the circuit, with the modules that group many of these smaller modules being the structural model design. This allows designers to create reusable components and easily integrate them into larger designs, translating to a more structured way to describe and define their circuits, separating different functional blocks which allow easier handling of highly complex structures.

After the circuit is designed and compiled, its functionality can be tested via what is known as a testbench. A VHDL testbench is a separate VHDL script in which the designer generates a set of stimuli to feed into the circuits inputs and sets controls over the outputs in order to check the designed circuit is working as intended.

Once the VHDL design is simulated and verified, it can be synthesized into a target hardware technology, such as an FPGA (Field-Programmable Gate Array) or an ASIC (Application-Specific Integrated Circuit). Synthesis tools map the VHDL code to the specific gates and flip-flops available in the target technology, optimizing for factors like area, power, and performance.

VHDL serves as a useful tool for describing and simulating digital systems. Its modular nature allows for efficient design and development, making it the perfect choice for this project.

2.3 DEVELOPMENT PLATFORM

Figure 6 : Xilinx ISE logo [16]

The choice of board of the project is a Xilinx Spartan 6 FPGA board, specifically model xc6slx45, with speed setting -3. The VHDL development environment must then be compatible with the board of choice. This leads us to Xilinx *ISE*, a native Xilinx platform software which supports the Spartan 6 board. It is not as modern or as efficient as the new Xilinx software, *Vivado*, but seeing as that software does not support the choice of board there is no option other than *ISE*.

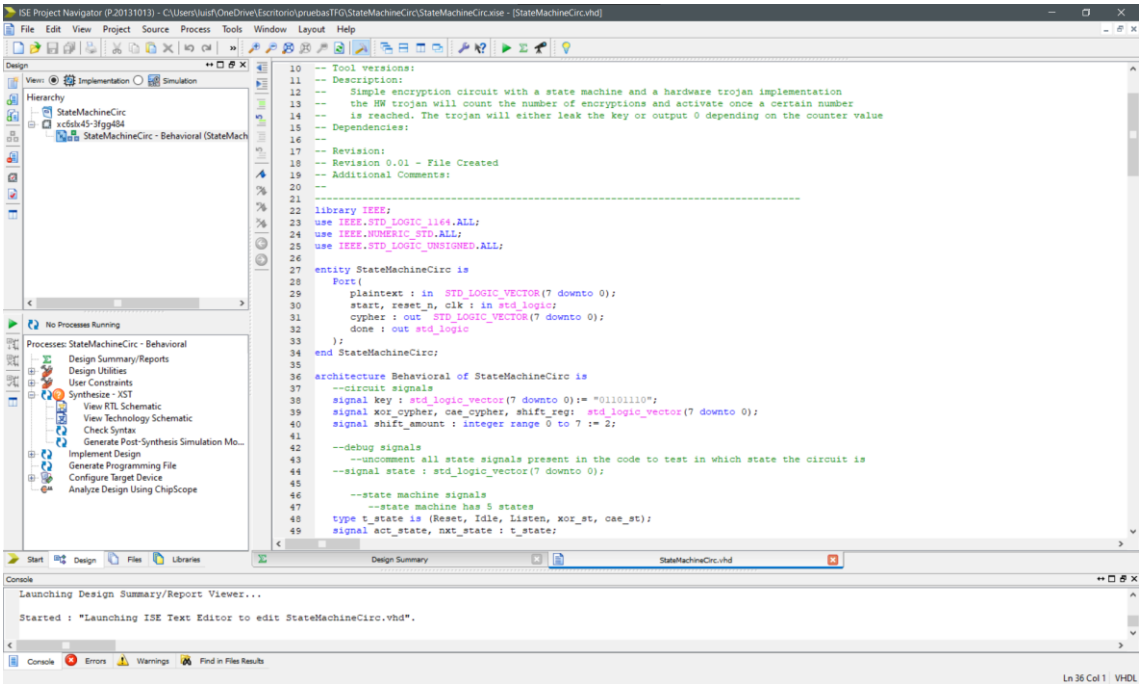

Figure 7 : Xilinx ISE main screen

ISE will be used to code and compile the VHDL implementation code and testbench script for verification. For the actual simulation of the circuit an integrated simulation environment in *ISE*, *ISim*, will be used.

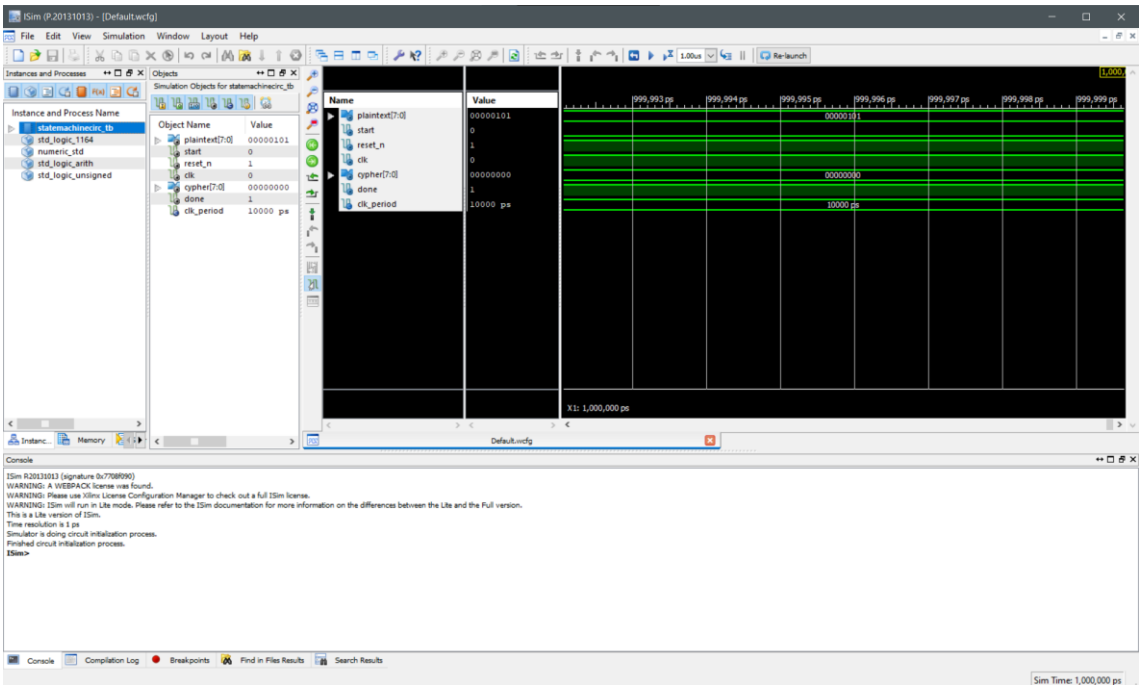

Figure 8 : Modelsim simulation screen

2.4 CRYPTOGRAPHIC CIRCUIT ELEMENTS

As the focus of the project will be on Hardware trojan development on FPGA boards, the encryption algorithms must not be too complex, as the FPGA boards may not have sufficient overhead to support the algorithms themselves, let alone additional structures. This is the reason why popular encryption algorithms such as AES and RSA are not present in the circuits developed, the FPGA board of choice, the Spartan 6, does not have sufficient hardware overhead to support the VHDL implementations of said algorithms.

Even if the encryption algorithms are not very complex, the project can still be successful, the methodology behind hardware trojan design and integration depends on the whole architecture of the circuits designed, not only on the encryption base.

2.4.1 Caesar Cipher:

The Caesar cipher is a simple and widely known encryption technique. Originally used by Julius Caesar in his private correspondence, it is a substitution cipher where each letter in the plaintext is shifted a certain number of positions down the alphabet [17] – [18]. It requires two elements, the plaintext, and the shift amount. To decrypt a Caesar cipher the process is very simple, with the same shift amount, shift each letter of the ciphertext a up the alphabet, restoring the original text.

The encryption and decryption processes are represented by the following equations [18]:

$$E_n(x) = (x + n)\ mod\ 26$$

$$D_n(x) = (x - n)\ mod\ 26$$

Where *n* is the shift amount, and *mod* is the modulo operation.

A graphical representation would be as follows, two aligned alphabets, in this case, a shift amount of 3 is represented, A is shifter three letters down the alphabet to turn into D.

Plaintext	A	B	C	D	E	F
Cipher	D	E	F	G	H	I

Table 1 : Caesar Cipher example

2.4.2 XOR cipher:

The XOR cipher is a simple symmetric encryption algorithm that works on binary data. It is based on the exclusive or (XOR) operation, which outputs binary 1 if both of the elements being compared are different.

Y = A XOR B		
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 2 : XOR truth table

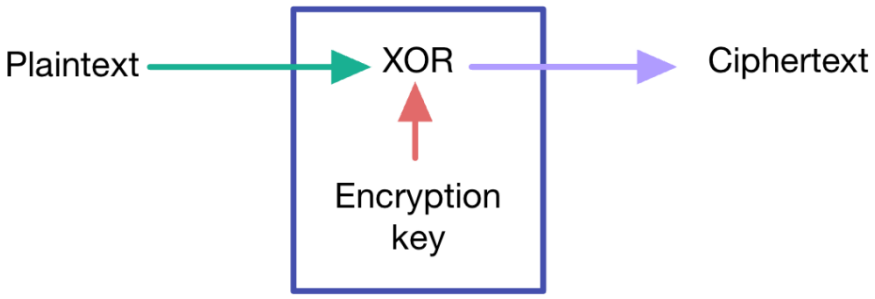

Figure 9 : XOR encryption block schematic

The XOR cipher requires a key to function, which will be a binary sequence of digits. This key should be equal or greater in length compared to the plaintext to encrypt. The XOR operation is then applied between the plaintext and the key, resulting in the ciphertext. For decryption purposes the same key is applied, and the same XOR operation is applied to the ciphertext.

It is important to recognize that both encryption methods are not very strong, and are very easy to break, but that is not the focus of the project. These algorithms have been chosen for ease of implementation in VHDL, and for the little overhead they need in order to be implemented, seeing as more complex encryption algorithms such as RSA or AES need greater hardware requirements than what are available in the board used as a basis for the project. However, from these simple examples more complex trojans for heavier algorithms could be devised, as the rationality behind the process would be the same.

CHAPTER 3: STATE OF AFFAIRS

Hardware trojans might seem like a shadowy force of which little is known about, but that could not be farther from the truth. Several aspects of their nature have been established to form a scheme to classify them depending on when, where, and how the hardware trojans are implemented. These studies have also allowed for a rough outline behind the way these types of attacks are introduced in IC. This chapter will go over this public knowledge, as well as discuss the various defense methods that have been proposed to combat and properly prevent hardware trojan attacks. All the methods and techniques mentioned in this chapter will aid in the planification of an attack on two host circuits, as well as devising defense strategies for them to prevent said attacks.

3.1 Modern Integrated Circuit design and fabrication

The process behind modern IC design and fabrication involves many different steps. It typically starts with the design phase, where engineers use hardware description languages, such as VHDL or Verilog to define the behavior of a circuit, also establishing in the process the architecture and interconnections present in the circuit design. This design is then verified and validated through simulations using testbenches, scripts of code that simulate the behavior of the circuit under certain inputs chosen by the designers.

Once the design has been finalized and properly tested, it undergoes photolithography, a process where the design is transferred onto a silicon wafer. Photolithography involves depositing and etching layers of materials onto the wafer to create the structural base for the circuit. Specialized software is usually involved in this process, transforming the HDL description into a physical circuit, whose properties are then usually refined by the engineer in order to meet the optimal specifications for the design.

This physical structure is then subjected to various processes such as doping, oxidation and deposition to reach the desired electrical properties, creating transistors, interconnects, and other components necessary for the design.

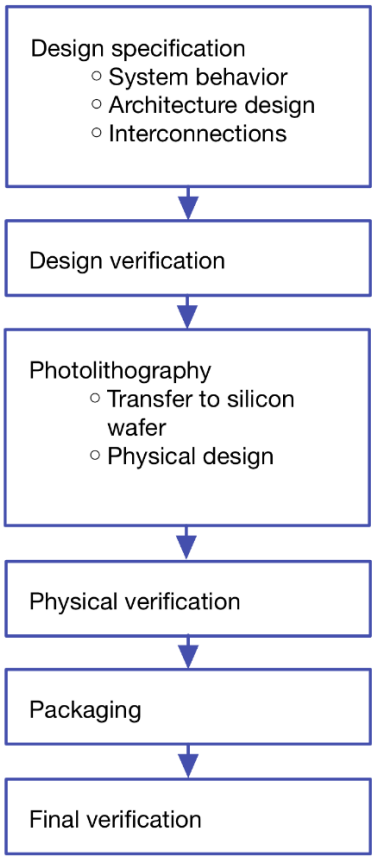

Figure 10 : Modern IC production process

After the fabrication process, the wafers are inspected and tested to identify defective properties or deviations from the original design, discarding the ones that deviate from the desired specifications. The wafers are then sent to packaging, which involve encapsulating the individual chips present in the wafers in protective casings, which provide electrical connections and protect them from environmental factors. Another round of testing is conducted on the packaged chips to ensure correct functionality. Once they pass the last round of verification, the chips are ready for whatever purpose they were designed for, integrating them in smartphones, computers, or other electronic devices [19] – [21].

3.2 Trojan Taxonomy

Hardware Trojans can be separated and classified based on a variety of different factors, such as insertion phase, (when the Trojan will be implemented), the abstraction level of the trojan design, its trigger mechanism, the effect it has on the host circuit, the location where it is inserted and its physical characteristics.

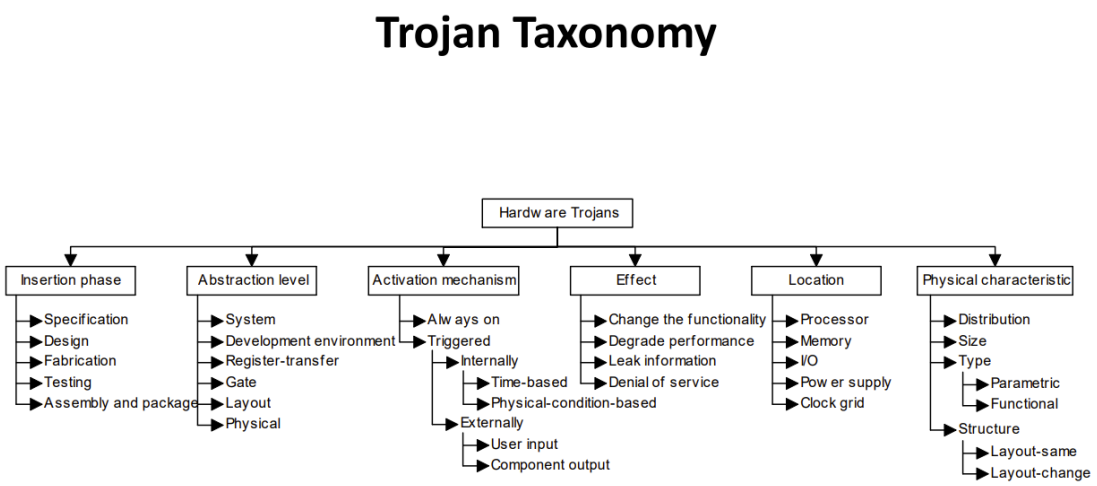

Figure 11 : Trojan Taxonomy [22]

During the insertion phase, a trojan can be inserted by modifying the design specification, like the operating temperature of an IC, to degrade its performance and dependability. During its design and fabrication stages, as mentioned before, it can be subjected to tampering by undesirable parties. When a trojan reaches the testing stage, the adversary can maneuver around the usual testing methods, ensuring it cannot be traced.

The abstraction level of a Trojan design determines how involved the trojan is in the circuit's characteristics. The higher the abstraction level the less control the designed has over the Trojans implementation. At the system level the adversary can only define the Trojan based on the modules that conform the host circuit, only being able to tamper with the interfaces and their interconnections. The lower the level goes, the more options open for the adversary, from being able to take advantage of hardware description languages or other software tools, to even modifying the physical characteristics of the transistors that form the circuit.

The activation mechanism of the Trojan is very important, as it is one of the main factors to take into account when considering detectability of the implementation. A trojan may always be active, or it may be conditionally activated due to external or internal factors. An *adversary* may decide to set the trojan to activate with a specific input sequence or let the trojan activate itself when certain conditions are met, such as a timer or when a certain temperature is reached.

A Trojans effect depends on the target circuits characteristics. If a cryptographic circuit is the subject of an attack, the trojan may leak the encryption key, or the original plaintext. In the case of a modern processor, from performance degradation to functionality changes are possible. The extent of the effect is up to the imagination of the attacker, and the options available to it offered by the original circuit's design.

The last two methods to differentiate Trojans are the most logical, based on the Trojans location, such as a processor's memory controller, or the physical characteristics of the Trojan implementation itself.

Even though there are many ways to distinguish between Hardware Trojans, this study will focus on two categories, activation mechanisms and effect. The Activation mechanism of trojans greatly affects the implementation and design, as too easy a trigger will be cause for early detection, and therefore not a good implementation of a trojan.

When studying the trojans in terms of their effect on the circuit, the most generic distinctions are changing functionality or degrading performance. In the case of cryptographic circuits, this can be concreted into two cases: denial of service or information leakage, which will render any cryptographic circuit useless.

3.3 Hardware Trojan methodology

Trojans are divided into many different groups and are usually categorized according to a functionality or behavioral pattern. Trojan taxonomy shows the many possibilities trojans have for approaching an attack, from the abstraction level to the way the trojan is activated. Although these infinite possibilities would infer that there is not a standard

method for Hardware Trojan design and Implementation, that is not the case, there is a pattern in the approach [2], [23]:

1. Design: In this phase, the attacker identifies the target system and analyzes its architecture and design. The attacker then identifies the potential insertion points for the Trojan and selects a suitable Trojan design that will meet their objectives while remaining undetected.

2. Implementation: The attacker modifies the design of the hardware component by inserting the Trojan circuitry. The attacker typically uses a hardware description language such as VHDL or Verilog to make the modifications. The Trojan circuitry may be inserted directly into the original design, or it may be added as a separate module that is connected to the original design.

3. Testing: The hardware component is then tested to ensure that the Trojan circuitry is functioning as intended and that it is not causing any unintended effects. The attacker may also perform testing to ensure that the Trojan is difficult to detect by security measures such as side-channel analysis or functional testing.

4. Deployment: The Trojan-infected hardware component is deployed to the target system.

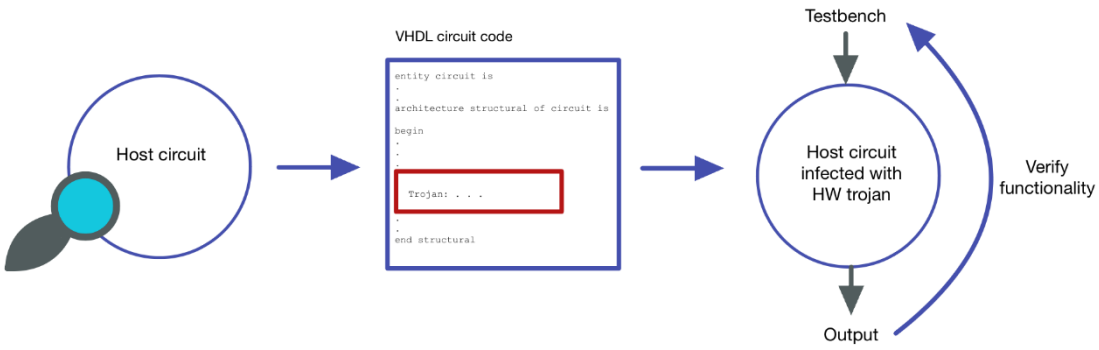

Figure 12 : Hardware trojan insertion process

The methodology behind a hardware trojan is very simple, and in a similar manner to other type of hardware attacks, it is all based on an intimate knowledge of the architecture of the target circuit. Once the adversary is familiar with the circuit, the hosts' weaknesses are then exploited and the hardware trojan is successfully inserted into the host circuit.

3.4 Trojan defense and detection

Perhaps the most damaging aspect of the hardware trojan is its secretive nature. The adversary will design and try to insert the trojan in a way that evades all detection. A host may be suffering an attack by a trojan, and not even be aware of it, as the process of detecting a hardware trojan is highly complicated. This is mainly due to the small overhead a Hardware Trojan has on the overall circuit [23], not considering that Trojans can be designed with complicated triggers in mind, passing any functional tests the circuits may be subjected to. Not only that, but due to the iterative nature of circuit manufacturing and design nowadays, leftover unused blocks may be kept from previous designs, acting as shelters for the Trojans.

There exist some general common techniques for Hardware Trojan defense, which can be done on the prevention stage:

1. Design for security (DFS): aims to prevent hardware Trojan prevention by making it difficult for attackers to understand the function and behavior of a hardware component, this can be done with encryption, circuit obfuscation or authentication.
2. Trustworthy manufacturing: to prevent Trojan insertion during the manufacturing process it is important to use trusted foundries, suppliers, and distributors. This is attained by implementing supply chain security measures, such as tamper-evident packaging, or by directly manufacturing in house, not outsourcing the process.
3. Testing and verification: comprehensive testing and verification of the circuits can detect and prevent less complex Hardware Trojans.
4. Side-Channel analysis: Side-channel attacks involve exploiting information leaked by a device during its operation, such as power consumption,

electromagnetic radiation, or thermal emissions. By analyzing these side-channel signals, attackers can gain information about the internal workings of the device, including the presence of hardware Trojans. To prevent such attacks, designers can implement countermeasures such as differential power analysis (DPA) and electromagnetic interference (EMI) shielding.

5. Reverse engineering: by breaking down the manufactured circuits and comparing to the golden or reference model of the circuit, any Trojans present could be detected.
6. Trusted Execution Environments (TEE): TEEs are isolated environments that provide secure storage and execution of sensitive code and data. By running critical functions within a TEE, designers can prevent hardware Trojans from compromising sensitive information or critical operations.

A lot of these techniques can be combined and implemented in a comprehensive security strategy to ensure the highest level of security available.

However, not all these techniques are effective in a general manner or cheap to implement. A lot of them are usually not feasible on actual production circuits, as they require what is known as the golden model, a reference design used to compare characteristics, which, due to the outsourcing nature of modern production schemes, is not realistically available.

Reverse engineering approaches are not very effective as they are expensive, having to break down the circuit, and the fact that the Hardware Trojan may only be inserted into a small selection of circuits in the whole production batch, proving the samples tested useless.

Logic Testing is not usually viable due to the large number of test patterns [2], especially in more complex and intricate circuits, and the specific and complex nature of Trojan trigger design. Side Channel Analysis is highly effective, but the great cost of the measuring tools and large noise signals received during the process and measurement render it less effective than desired.

To make matters worse, a lot of the techniques previously outlined are not effective on FPGA design, as they require a specific circuit board, not a multipurpose circuit board like an FPGA. Specifically for FPGA design, without taking into consideration Side Channel Analysis, two important avenues for study of defense have risen: Design for Security (DFS), and MERO testbench design.

3.4.1 Design for security

Design for security, specifically Circuit obfuscation, aims to prevent hardware Trojan prevention by making it difficult for attackers to understand the function and behavior of a hardware component. Circuit obfuscation will increase complexity and randomness of the hardware design, by various methods:

1. Logic locking: adding an extra layer of security by encrypting/locking the design using a secret key. It functions like a traditional lock, hiding functionality behind a key. It can be applied to the circuit as a whole, in a similar manner to how a computer password works, or it can be applied in a way that modifies the output of the design if the incorrect key is inserted.

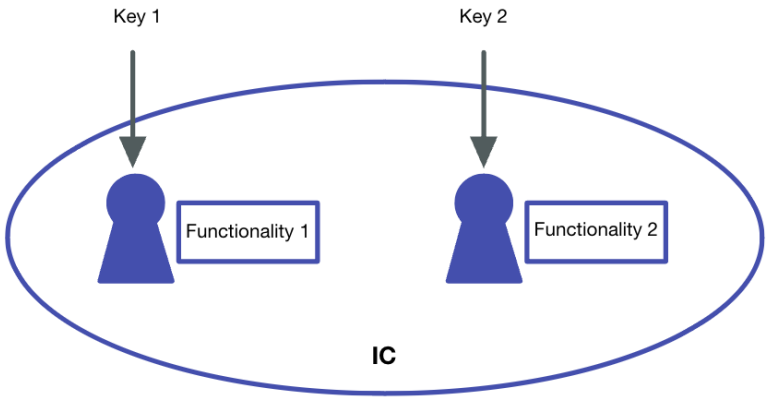

Figure 13 : Logic locking

2. Randomization: involves implementing random elements to hardware design, such as interconnects or random gate placements.

3. Polymorphism: using multiple versions of the same circuit that have different functions, making the attacker have a harder time identifying the true circuit functionality. This can have a high hardware cost due to requirements needed to properly hide the functionality, as the other implementations need to have a similar complexity for the effort to be effective. This additional hardware cost can be softened by implementing other desired designs into the circuit, not wasting the resources used.

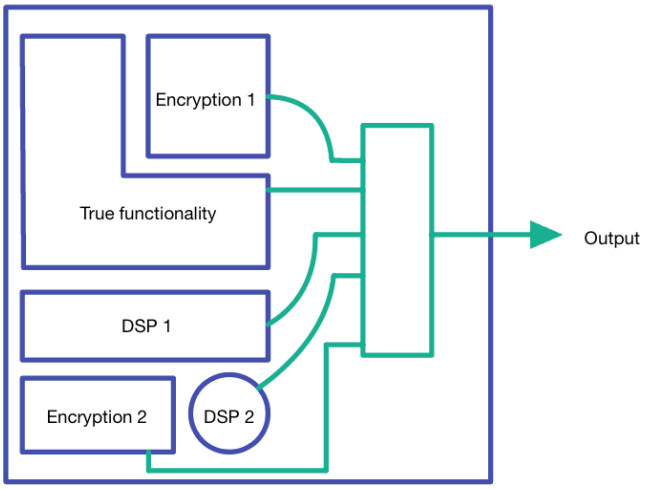

Figure 14 : Polymorphism

4. Obfuscation: involves hiding the function and behavior of the circuit behind code obfuscation or encryption.

These techniques can be combined to offer a more robust defense, however, it is important to note that these techniques can also greatly increase the cost of the hardware design, as well as add unexpected new vulnerabilities due to the higher complexity of the design.

3.4.2 MERO testbench

The other main pillar to take into consideration is MERO testbench design. Although it is important to design an effective testbench in general for hardware verification, MERO differs from standard testbench design by testing for unexpected circuit behavior, instead of testing for the usual functionality [24].

MERO starts with establishing normal circuit behavior, and from then enters a recursive loop, generating excitation patterns, analyzing the circuit response, and in the case of a rare pattern or unexpected output, the testbench is refined to explore the root causes of the rare pattern, by exploring similar inputs. In depth explanations of the steps are as follows:

1. Establish normal behavior: establish normal circuit behavior by simulating its response to a variety of input conditions. This is done to determine the expected output under normal conditions.
2. Generate excitation patterns: once the normal behavior is set, a special set of input patters are generated, these patterns are designed to trigger any hardware Trojans present in the circuit.
3. Apply patterns: The patterns are inserted into the circuit and the response is measured.
4. Analyze response: after measuring the circuit response to the pattern, it is compared to the normal behavior to see if any unexpected or rare patterns occur.
5. Identify potential Trojans: If any rare patterns are detected, it may indicate the presence of a Hardware Trojan, thus the pattern that triggered the response is analyzed, and other, similar patterns are generated, repeating the process.

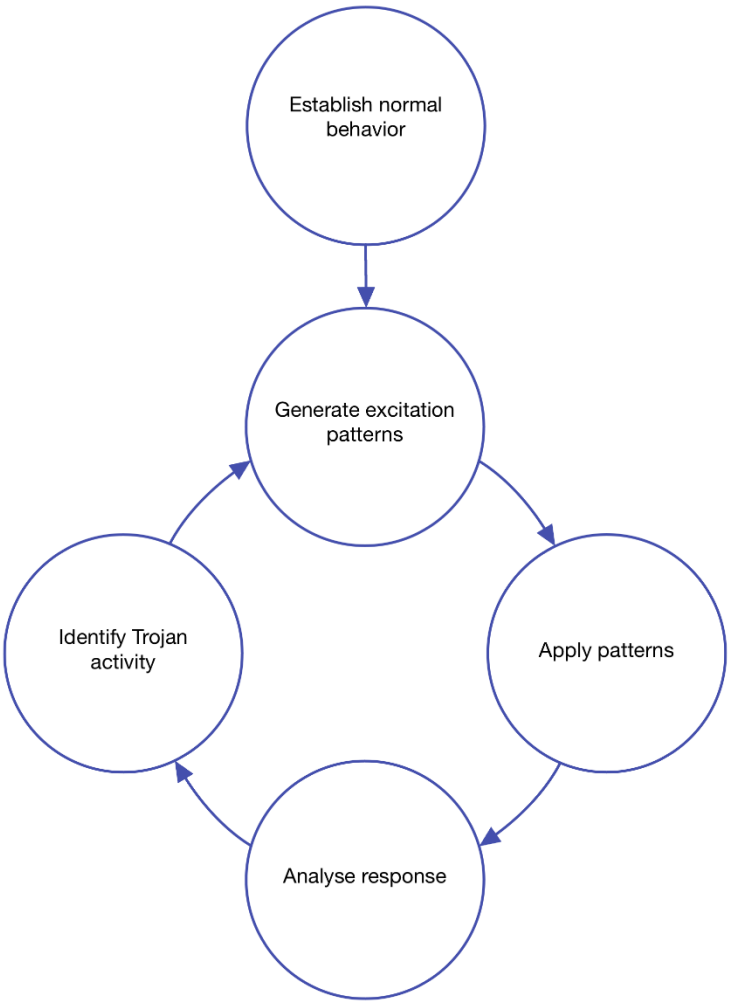

Figure 15 : MERO testbench design

Once sufficient tests are made, potential trojans are identified, if the detection of rare patterns is successful. Overall, MERO has proven to be highly effective for detecting hardware trojans that may be missed with traditional testing methods.

3.4.3 Potential strategies for defense

With the previous methods and techniques an initial defense strategy could be developed. The options displayed show that in the prevention stage, for FPGA development, a combination of robust verification, that is via exhaustive verification if feasible, or the development of a MERO testbench, and a strong, resilient design that applies the core

ideas of DFS, be it circuit obfuscation or logic locking, will prove to be a good combination for a starting point in hardware trojan defense.

An expansive analysis of the host circuits will be conducted, and from then on their potential weaknesses will be remedied with DFS techniques if applicable, with other avenues of attack being covered with the testbench design.

CHAPTER 4: HARDWARE TROJAN IMPLEMENTATION

With all the strategies and methods laid out, they will now be exercised to instigate a hardware trojan attack. A set of host circuits will be developed, with each of them being subjected to an attack that exploits their internal structure and functionality.

4.1 Implementation details

The Hardware trojan design and implementation has been done in two main stages, one with a prototype circuit and another with a more complex circuit built on top of the original, expanding the Trojan insertion possibilities. The cryptographic circuits are based on a combination of a Caesar cipher and an XOR cipher, which are described in chapter 2. The Trojan design will follow the standard procedure outlined in chapter 3: an in-depth study of the circuit's functionality and its characteristics will grant avenues for attack, these will then be exploited, and a Trojan will be inserted into the existing design, aiming for a stealth approach, and changing the original circuit's purpose.

4.2 Prototype circuit

The original prototype circuit will consist of a direct VHDL implementation of an XOR cipher followed by a Caesar cipher.

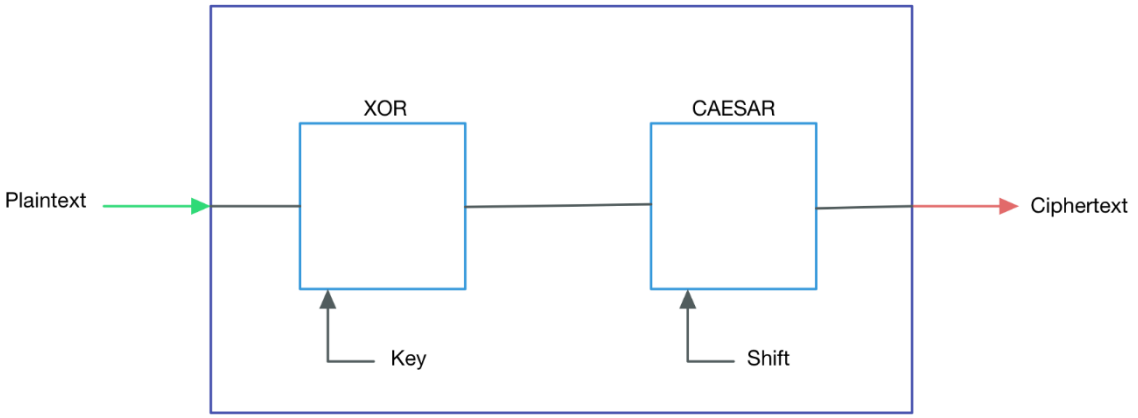

Figure 16 : Initial encryption circuit

```
xor_cypher <= plaintext XOR key;
cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);
```

The circuit will have a plaintext input and a ciphertext output (in this case being the *cae_cypher* signal). The key and shift amount are predetermined in the circuit as internal signals defined in VHDL. The circuit will then encrypt the plaintext with the XOR encryption, and that ciphertext will be encrypted again with the binary Caesar encryption algorithm, this last ciphertext being the output of the main circuit.

4.2.1 Hardware Trojan implementation: externally triggered trojan

A quick study of the circuit shows that it does not offer many possibilities for creative Trojan design. Internally, the functionality consists of two simple lines of VHDL code. If one where to modify said lines, the trojan would be always activated, and the functionality always modified, thus being easily detectable, and not resulting in an effective approach. The next logical step would be to explore other avenues of activation: external and internal triggers. Seeing as the internal structure is not very complex, an external trigger looks to be the best option for this circuit design.

The next step is to decide what the actual trojan will be, and how that relates to the trigger mechanism. The circuit under attack has cryptographic purposes, so the main goal will be

to deny said encryption efforts, and, by the nature of the circuit, one of the better options would be to leak the encryption key used in the XOR cipher. Leaking the shift amount could also be done, but seeing as the Caesar cipher is easily breakable (there are very few options for the shift amount), the option to leak the XOR key seemed best. The XOR key is then leaked through the regular ciphertext output.

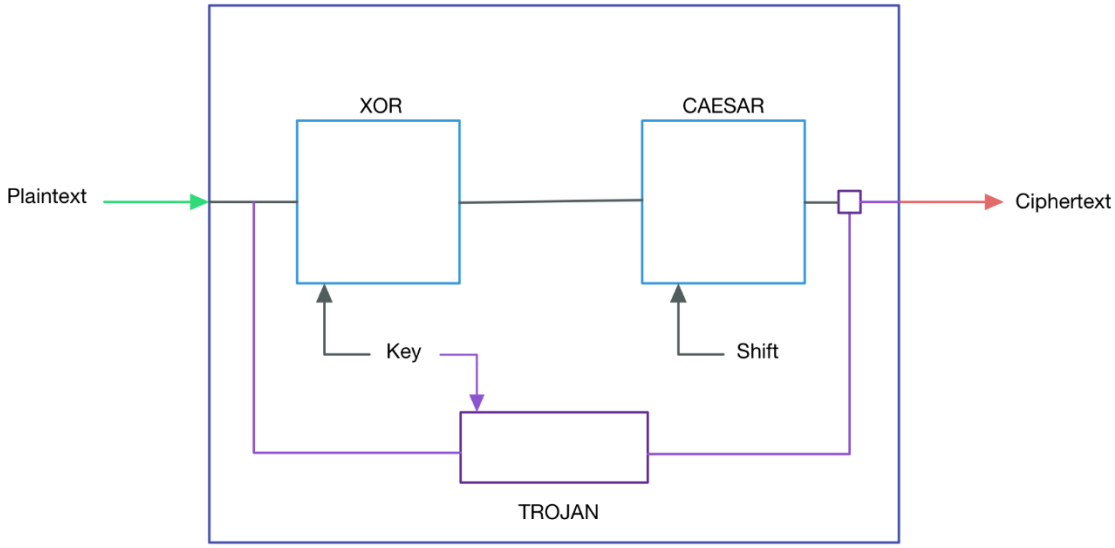

Figure 17 : Prototype circuit with HW trojan

To activate the trojan an arbitrary plaintext binary sequence is chosen, and whenever that sequence is used as input to the circuit, the trojan will activate and leak the encryption key through the circuit output.

```
xor_cypher <= plaintext XOR key;
cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);

    --TROJAN
    trojan: process(plaintext, key, cae_cypher)
    begin
        if plaintext = "10011001" then
            cypher<=key;
        else
            cypher<=cae_cypher;
        end if;
    end process trojan;
```

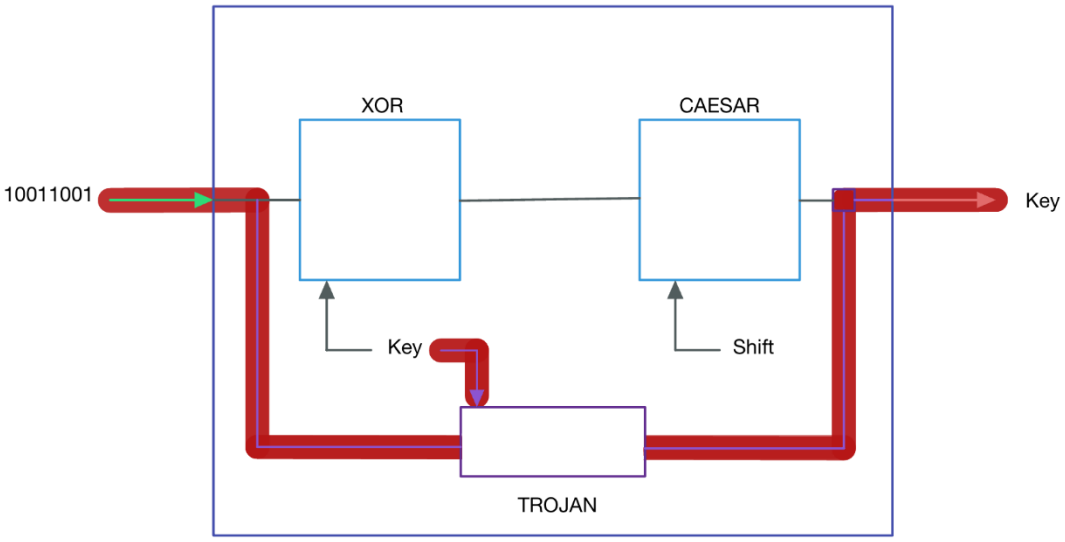

Figure 18 : prototype circuit trojan functionality

This original circuit, although it offered an initial look at the process behind Trojan design and implementation, it proved to be too simple to offer any real insight into the development process behind Trojans. For this reason, a more complex encryption circuit needs to be developed, leading to the second design of this study.

4.3 Second circuit design: state machine circuit

The second circuit design parted from the original prototype circuit of a simple XOR and Caesar cipher implementations. The need for greater complexity to allow for a more comprehensive trojan implementation led to the decision to add a state-machine to the circuit. The state machine will control the encryption process and introduce a more complex internal structure to the original circuit design. The design then has four different states: Idle, Listen XOR and CAESAR. The following states do as follow:

- Idle: the circuit waits for an activation signal *start* in order to start the encryption process.
- Listen: in this second state the circuit will receive the input plaintext and stores it in a bus, which will then be used in the next encryption process.

- XOR: applies the XOR encryption and feeds it to the next stage.
- CAESAR: applies the Caesar cipher to the XOR ciphertext and outputs the ciphertext directly to the circuit output.

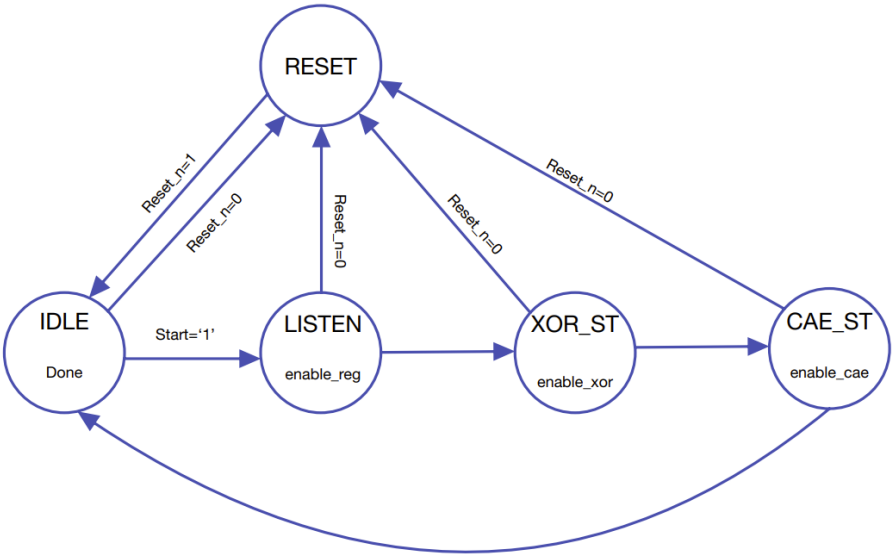

Figure 19 : Second design state machine

After CAESAR the circuit returns to the idle state, where the output from the previous cycle stays until the next encryption process is activated by the *start* signal. The state transitions are dictated by a rising clock edge, meaning that each state will last for one clock cycle, and all actions will be synchronized to a clock cycle.

```
--state machine transition
StateTransition : process (start, act_state, reset_n)
begin
    nxt_state <= act_state;
    case act_state is
        when Reset =>
            if reset_n = '1' then
                nxt_state <= Idle;
            end if;
        when Idle =>
            if start='1' then
                nxt_state <= Listen;
            end if;
        when Listen =>
                nxt_state <= xor_st;
        when xor_st =>
                nxt_state <= cae_st;
        when cae_st =>
                nxt_state <= Idle;
        when others=>
```

```
                nxt_state <= Idle;
        end case;
    end process StateTransition;
```

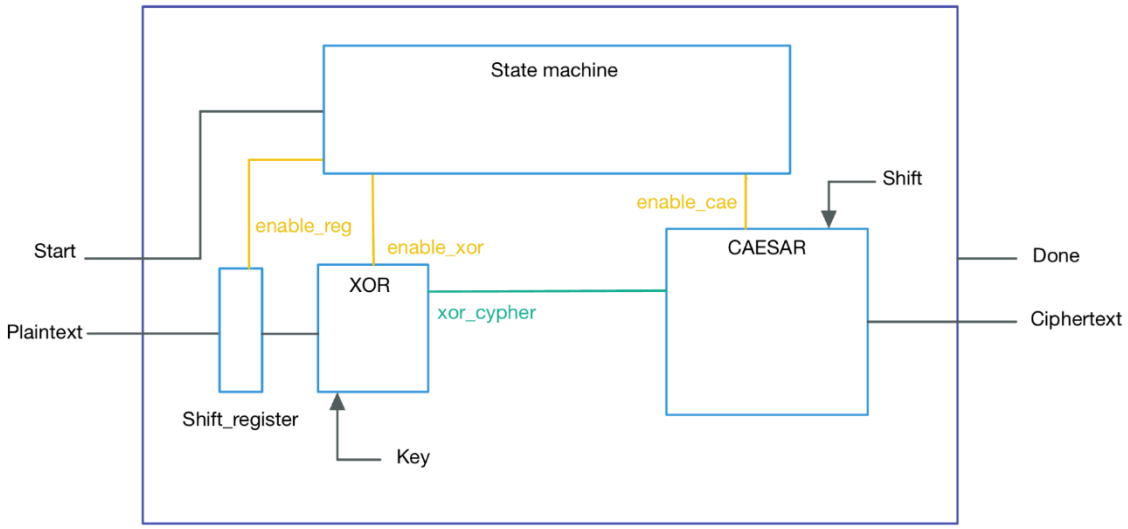

Figure 20 : Second circuit design

4.3.1 Second design Trojan implementation: internally triggered trojan

After verifying and testing the second circuit design, the Hardware Trojan implementation process begins. A similar approach to the previous example was done. First, a study of the circuit is done. There are two major components in the circuit structure, the state machine, and the encryption process. For this second Trojan design, both characteristics of the circuit will be attacked and manipulated, for a more complex Trojan implementation.

4.3.2 Triggering mechanism

Seeing as the internal structure of the circuit is governed by the state machine, it would make most sense to take advantage of said internal structure to trigger the trojan mechanism, and so an internally triggered trojan design was chosen as a basis. One option would be to count clock cycles and activate the trojan after an arbitrary number of clock

cycles has passed. Another option, and the approach that was chosen, was to count encryption iterations. A counter would be set up, and, depending on the number of encryptions, a different Trojan payload would be delivered. This approach makes sense, as it utilizes the internal structure of the circuit to its advantage, and the more integrated the trojan is to the original structure of the circuit, the less overhead it presents and thus much less noticeable.

```
trojan : process(reset_n, act_state)
    begin
        if reset_n = '0' then
            troj_counter <= (others=>'0');
            troj_counter_state <= (others=>'0');
            troj_en <= "00";
            troj_en_state <= '0';
        elsif act_state = Listen then
            troj_counter <= troj_counter + 1;
            troj_counter_state <= troj_counter_state + 1;

                --counter for key leak/ DoS
            if troj_counter = "001" then
                troj_en <= "01";
            elsif troj_counter = "100" then
                troj_counter <= (others=>'0');
                troj_en <= "10";
            else
                troj_en <= "00";
            end if;

                --counter for state machine freeze
            if troj_counter_state = "110" then
                troj_en_state <= '1';
                troj_counter_state <= (others=>'0');
            end if;
        end if;
    end process trojan;
```

4.3.3 Payloads

The Hardware Trojan design aimed to target the two main characteristics of the circuit, the state machine, and the encryption process, and so the payloads were chosen to affect those areas and disrupt their functionality.

For the encryption efforts of the circuit, the original concept trojan idea of key leakage was reused, but this time internally triggered. However, an additional payload was added

to the design. This second payload was intended to deny service to any encryption effort, so the Trojan would erase all encryption processes done before and output binary zero on the circuit output, eliminating the ciphertext.

When looking to affect the state machine, many options were considered, such as skipping a state, but seeing as they also touched on the encryption process, they seemed a bit redundant, and having too many payloads would result in easier detectability. The third and final payload of the Trojan consists of the trojan freezing the state machine indefinitely, halting all processes and forcing a restart on the system. This option also showed an aspect of trojan payloads that were not considered on the other payloads: stealth. The payloads that affected the encryption process are easier to detect, seeing as if they happen too frequently or are too obvious the designer will look to the internal structure and revise it, maybe finding the trojan if it was inserted in the original design process by the malicious agent. However, the state machine freeze could be attributed to a simple hardware malfunction, and a reset would fix the problem. If the trigger is set to activate once a very high amount of encryption cycles have passed, there is a minuscule chance of discovery, with the trojan never being addressed.

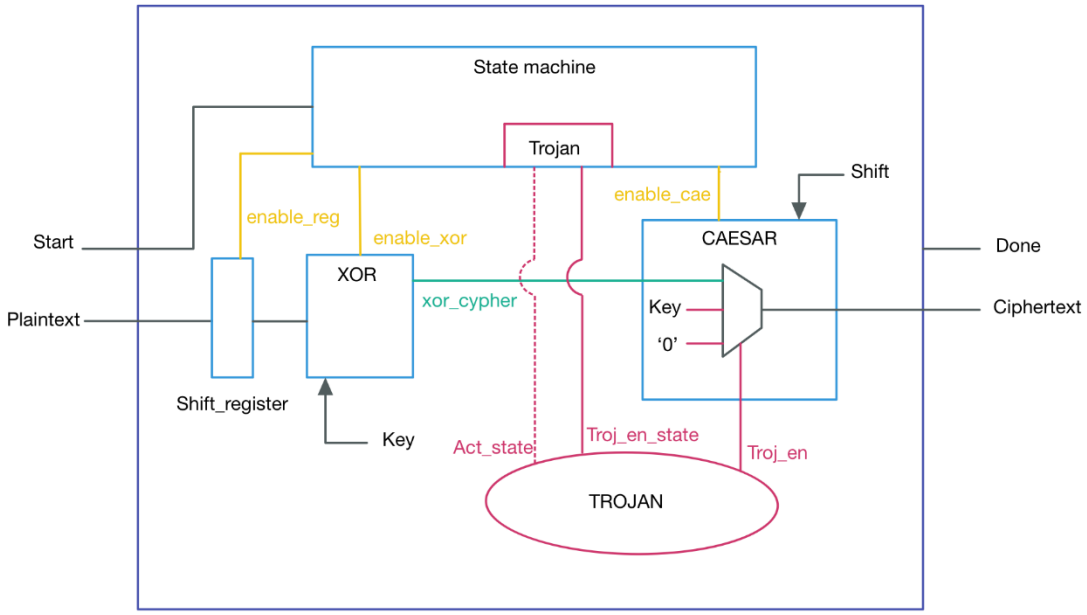

Figure 21 : Second design with HW trojan

```
--STATE MACHINE
        --state machine variation
    StateVar : process (clk,reset_n)
    begin
        if reset_n='0' then
            act_state<= Reset;
        elsif clk'event and clk='1' then
            if troj_en_state = '0' then
                act_state <= nxt_state;
            else
                act_state <= act_state;
            end if;
        end if;
    end process StateVar;
```

```
caesar: process (enable_cae, reset_n, xor_cypher)
    begin
        if reset_n='0' then
            cae_cypher <= (others=>'0');
        elsif enable_cae='1' then
            if troj_en <= "00" then
                cae_cypher <= std_logic_vector(unsigned(xor_cypher) +
shift_amount);
            elsif troj_en <= "01" then
                cae_cypher <= key;
            elsif troj_en <= "10" then
                cae_cypher <= (others=>'0');
            else
                cae_cypher <= std_logic_vector(unsigned(xor_cypher) +
shift_amount);
            end if;
        end if;
    end process caesar;
```

4.3.4 Integration in the circuit

These payloads are activated after an arbitrary number of encryption sequences have passed, for easier simulation purposes a very low number of cycles was chosen for the payloads to trigger, in an actual trojan design a higher number would be preferred, as it hinders detectability of the trojans.

cycle trigger	payload	location
2nd	key leakage	caesar module
5th	output zero (deny service)	caesar module
8th	freeze state machine	state machine

Table 3 : Second circuit HW trojan implementation

4.4 Simulation and results:

Once the circuits were designed and tested, and the respective Trojans inserted, a VHDL testbench was designed to evaluate and verify the functionality of the Trojan design. The testbenches simulated the functionality of the circuit according to different inputs, with enough encryption cycles being accounted for in the testbench for the second design in order to go through the necessary cycles to get the Trojans properly activated. The screenshots below show the circuit ouput *cypher* which displays the encryption result, or, in the case of certain encryptions, the Trojan output.

4.4.1 Prototype circuit

The first design was simulated with a very simple testbench, which only tested a regular input, followed by the trigger input, to assess whether the Trojan reacted properly to the trigger and leaked the key. The testbench was designed to notify the user via console if the key had been successfully leaked.

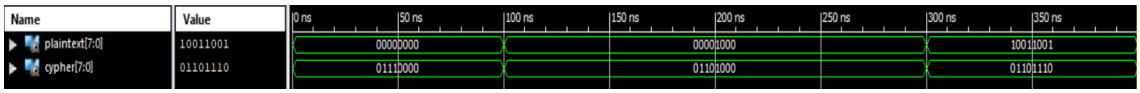

Figure 22 : Prototype design trojan simulation

```
** Failure:Key has been leaked
User(VHDL) Code Called Simulation Stop
In process XOR_tb.vhd:stim_proc
```

Figure 23 : Prototype design testbench console output

input	output
00000000	01110000
00001000	01101000
10011001	01101110

Table 4 : prototype circuit simulation results

As seen in the screenshots above, and the following table displaying the results, the Trojan was successful in leaking the key, which is the binary vector 01101110, responding properly to the trigger (input 10011001).

4.4.2 Second circuit

In the second circuit, the Trojan is triggered internally, depending on the number of encryption cycles, as mentioned before when discussing the design of the Trojan, the number of encryption cycles needed to activate the triggers are 2, 5 and 8. The testbench designed needs to accommodate enough encryption cycles to activate all the triggers, and have each Trojan deliver its payload.

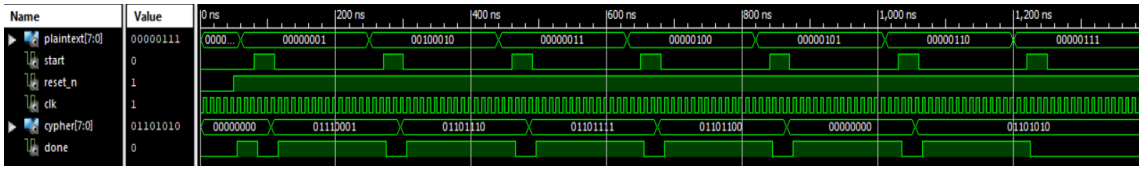

Figure 24 : Second circuit simulation screenshot

input	output
00000001	01110001
00100010	01101110
00000011	01101111
00000100	01101100
00000101	00000000
00000110	01101010
00000111	01101010

Table 5 : Second circuit simulation results

As seen above, the Trojan functionality works as expected, with the necessary Trojan triggers activating properly, and the payloads being delivered accordingly. In the case of the second encryption, the Trojan will activate and leak the key. The second trojan trigger activates on the fifth encryption, in which the Trojan will deny the circuit of its cryptographic purposes, outputting zero, nullifying the efforts done for that cycle. The last Trojan cycle comes in the seventh encryption (input 00000111), where the state machine freezes in a constant loop, never really finishing the encryption. This can be seen in the simulation result thanks to the signal *done*, which is usually active during the idle state, notifying the user that the circuit output is the final encryption result, and in the case of the last cycle, it never activates, indicating that the encryption is not done.

Seeing as there are two distinct types of payloads, one regarding the encryption itself and another targeting the state machine, a slight modification of the circuit and testbench was done in order to better demonstrate the functionality of the state machine, and how it is affected by the Trojan. For this second simulation an internal state counter was connected to the circuit output, essentially showing the internal state of the circuit during each clock cycle, shown in the ouput as an 8-bit vector (001 being Idle, 010 Listen, 011 XOR and

100 CAE). The screenshot directly below shows the usual state machine flow, switching state on every rising clock edge.

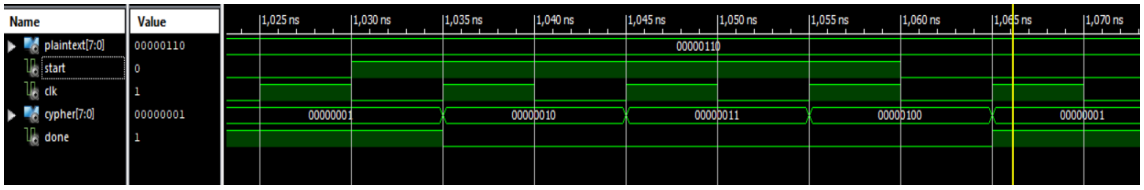

Figure 25 : State machine full cycle simulation

For the last Trojan payload, the one that freezes the state machine, the circuit must stay in the Listen state. Once the *start* signal activates, the circuit moves onto the Listen state from Idle, resetting the done signal denoting the start of another encryption cycle, but never manages to finish it.

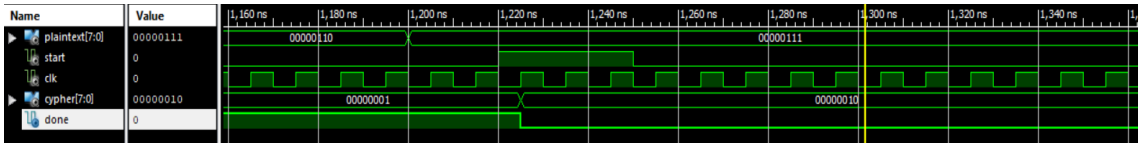

Figure 26 : Second circuit state machine trojan simulation

4.5 Trojan Insertion Results

The Trojan implementations have proved to be highly successful. Although the first circuit developed, the prototype design, did not allow for creative Trojan insertion, it permitted an initial approach for a simple trojan implementation and trigger. The second design, with a more comprehensive structure, permitted a more whole and complex Trojan implementation, allowing for the integration of different trojan trigger mechanisms and payloads. The second design, as it incorporates a state machine, also showed a glimpse into how a hardware trojan can affect a circuit beyond the cryptographic area. A state machine inhibitor could be applied to many different designs, and although simple, it helped establish an initial workflow and methodology behind trojan implementation and design, which could be useful for more complex circuits. Mainstream computer and handheld computing devices take advantage of a multistage

processor design and pipelined architectures, this initial study could serve as a basis for trojan insertion in a pipelined architecture, seeing as a processor pipeline could be abstracted into a state machine of sorts.

CHAPTER 5: HARDWARE TROJAN DEFENSE

For Hardware Trojan defense, similar to the insertion phase, intimate knowledge of the circuit is imperative, as knowing the weaknesses of one's design will allow for a more robust defense, reinforcing the internal structure. In this final part of the study, a close examination of the two developed circuits will be conducted, analyzing how the Trojans were implemented, devising possible defense methods against the intrusions and how those possible defense methods affect the overall complexity of the circuit.

As previously mentioned, since the focus of the study will be on FPGA development, only solutions applicable to VHDL will be discussed, these being Design for Security (DFS) and Verification via Testbench.

5.1 Prototype circuit

For the first design, as discussed in the previous chapter, there are not many possibilities for a Trojan. The main vulnerabilities the circuit offers, from a VHDL perspective, are the input and the encryption process itself. We know the Trojan attacks the input, activating the trojan externally.

5.1.1 DFS

The circuit being as simple as it is can be made more secure by simply adding a couple of elements. The need to secure the encryption process could be done in many ways, the first, and easiest one, making the circuit require keys to activate the encryption process. This could be further expanded by dividing the encryption process into blocks, taking advantage of a VHDL structural architecture, but seeing as the inner workings of each block would be too simple, and not really provide a challenge to the adversary, logic locking techniques prove to be more effective. Taking advantage of the key system already implemented, the decision was made to have the keys not only authorize the circuit functionality, but also output a wrong ciphertext when the key is not correct, hampering efforts from a malicious observer.

These techniques will make the circuit harder to access and study, especially if we require a separate key for each of the functional blocks resulting in three distinct keys, which each access a different part of the circuit. This might not completely secure the circuit from any attack, but it makes the process of inserting a Trojan more costly and arduous, as the circuit has been overcomplicated for what is essentially two lines of VHDL code. This overcomplication of the design, along with a robust verification via Testbench, would cover the weaknesses present in the circuit.

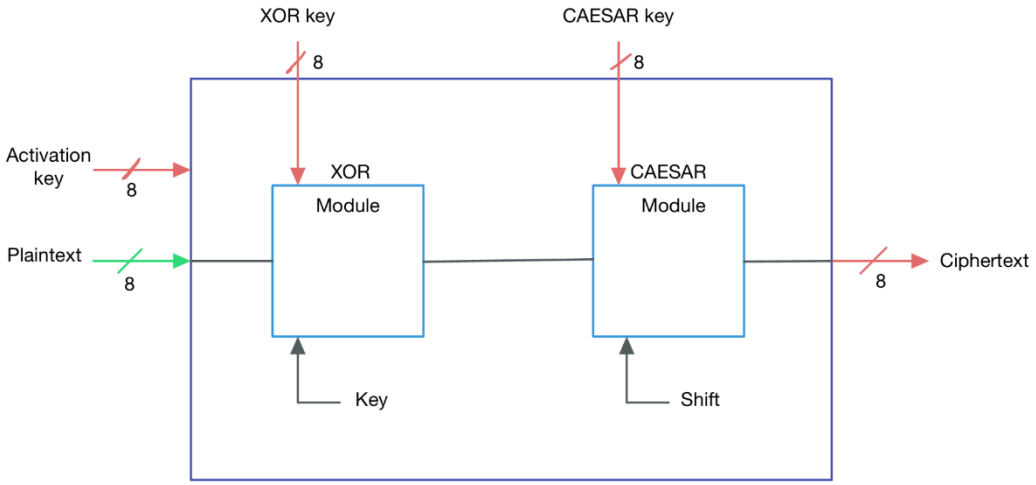

Figure 27 : protype circuit with DFS techniques

```
--encryption circuit using logic locking techniques
    encryption : process(plaintext, key, key1, key2, key3, shift_amount)
    begin
        if auth_key = key1 then
            if xor_key = key2 then
                xor_cypher <= plaintext XOR key;
                if cae_key = key3 then
                    cae_cypher <= std_logic_vector(unsigned(xor_cypher) +
shift_amount);
                else
                    cae_cypher <= not xor_cypher;
                end if;
            else
                xor_cypher <= plaintext and "00010111";
            end if;
        else
            cae_cypher <= not plaintext;
        end if;
    end process encryption;
```

5.1.2 Verification

Seeing as the circuit only has a plaintext input of 8 bits, there are only 256 possible inputs, which is a feasible size to exhaustively test. This would erase the need for MERO testbench design, as all possible inputs are covered. In addition, the testbench could place alerts for when critical elements of the circuit are leaked, such as the XOR encryption key, or the Caesar cipher shift amount, covering any vulnerabilities that may escape normal testing.

Were we to follow the DFS methods stated before and incorporated a set of keys to control the encryption process, the testbench design would become more complicated as well, needing to cover all the possible inputs on the plaintext, and all possible key combinations inserted into the circuit, as the adversary may have tampered with the authentication system of the circuit.

If we chose to have 3 authentication keys in the circuit, each authentication key consisting of an 8-bit sequence, then the total number of inputs to consider in the testbench would rise to 2^32, which would rule exhaustive verification out of the question. MERO could then be applied to test this hypothetical circuit, subdividing the exhaustive verification into various smaller testbenches, and applying the MERO methodology to refine and develop a more precise testbench, were unexpected behavior be found in the circuit. The problem with a MERO approach is that it requires a statistical analysis of the input frequency, identifying rare combinations, which, on this circuit, is the same for all. In the end the testbench was designed with an exhaustive approach, which would take long to simulate but still cover all possible avenues, and seeing the simplicity in the design, can still be completed, although it may take a lot longer.

It is important to decide whether this strategy acts in benefit of the design. The extra security awarded by the keys comes at a cost to simulation complexity, and the feasibility of the original design's exhaustive verification is a factor not to be taken lightly, seeing as it is a robust defense. If authentication in the field were a necessity, then maybe a

compromise with the key sizes could be considered, reaching a point where exhaustive verification was still possible, with the added security that the extra set of keys bring.

5.2 Second Circuit

For the second circuit, the case is much different, as the circuit is already more complex. There are two main aspects of the circuit to analyze and try to defend, the state machine, which governs the internal flow of the circuit, and the cryptographic purposes of the circuit.

5.2.1 DFS

With design for security, one of the possible solutions would be to, similar to one of the solutions applied to the prototype, apply a circuit obfuscation technique: divide the circuit into different functional blocks taking advantage of the VHDL structural model, separating them and making the overall structure more complex, leaving less wiggle room for the *adversary* to insert a Trojan stealthily.

This seems like the most robust option, as adding a layer of keys would disrupt the state machine behavior: instead of having the circuit move from state to state automatically after every clock cycle, having keys in a similar way to the prototype, one per stage, would disrupt the flow of the circuit, requiring the user to constantly input the keys, and essentially acting himself as the state machine, and thus the circuit would be changed too drastically and would end up as a similar circuit to the prototype developed before. If one was insistent on implementing a key authentication system, it would be best to require it as a single external input to activate the circuit, instead of the *start* signal already present in the circuit.

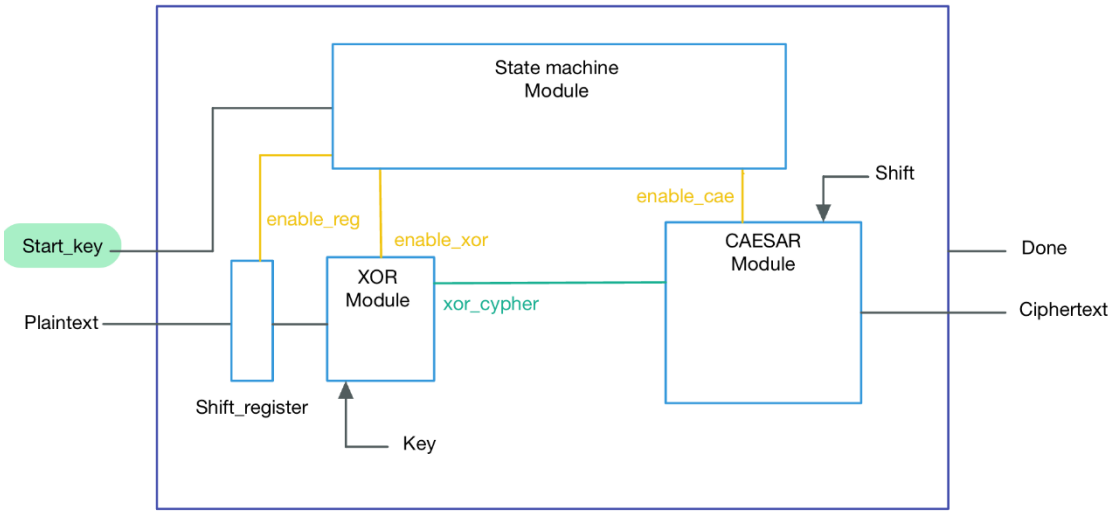

Figure 28 : Second circuit with DFS techniques

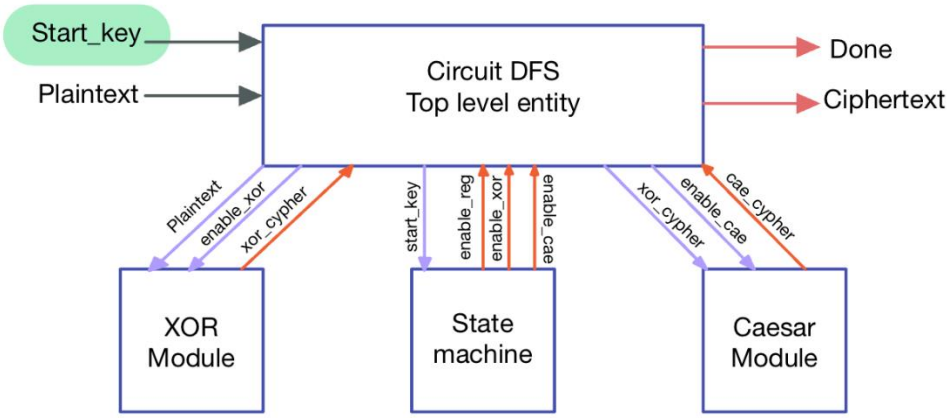

Figure 29: Second circuit new block design

The final, more robust design after implementing DFS techniques would change the original design in the following way: it would separate the circuit into smaller functional blocks and would require an input key to activate the encryption process and state machine functionality.

5.2.2 Verification

Similar to the prototype, the final circuit has an 8-bit sequence as the plaintext input, so a testbench verification for the encryption from the input side would be feasible. If one were to add a single 8-bit key to the design for authentication purposes, the simulation would need to consider many more options (2^16). This increment in simulation requirements means a MERO approach would benefit the circuit, recursively testing for unusual patterns.

But, as we know, the Trojan inserted in the circuit activates internally, not from the inputs of the circuit. This poses a problem, as formal verification cannot accurately test for this type of trigger. If the trigger is based on a timer or counter, which is our case, for how many cycles should one simulate? In the case of there being a very high number of cycles required for the Trojan to activate, there exists a very strong possibility that it may go undetected during testing, and only manifest its effects on the actual production circuit, after a long time has passed. Seeing as it is not realistic to simulate indefinitely, a compromise would need to be reached, by conducting a study on how long the circuit would usually be operated for between resets, and redoing the simulations based on that. And here we finally meet the true difficulties of dealing with Hardware Trojans, there are too many possibilities for an attack, and our verification methods and defense and prevention techniques might not cover all possible avenues.

5.3 Results

The exploration of defense methods has also been successful, although the similarities in design in both encryption circuits has given a more limited scope in what is possible. In the case of the first encryption circuit a robust implementation was obtained, locking all functional modules behind a key, however this increase in input options would allow a potential adversary to tamper with the authentication, apart from increasing the computational overhead of the simulation. This first defense strategy is a good example of how a simple change can increase the complexity of the process, as the extra keys suppose extra simulation requirements, and we go from one simple exhaustive testbench

present in the original design, to the need to develop a recursive system for testing, implementing MERO, without a true guarantee that the system is trojan free.

The defense strategy for the second circuit illustrates the process of implementing DFS in a more general manner, similar to how it also offered a more general take on trojan implementation. The defense strategy depends greatly on the design's characteristics. The initial strategy devised would compromise the circuit's functionality, being more of a problem than a solution. The encryption process was addressed with an initial authentication key, which, in combination with the input plaintext size, would be an acceptable number for exhaustive verification, or would take less computation if a recursive approach were taken. The issue rises however on the internal trigger mechanism and serves to illustrate the potential danger that these types of attacks have. Simulating and testing infinitely will never be an option, and a lot of these types of attacks take advantage of that, leaving the manufacturers to constantly be on the lookout for reports of their products malfunctioning, and highlighting the need for extra security hardware present on the chip, such as a security module and performance monitor.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

Hardware Trojans pose a grave threat to the security and integrity of integrated circuits. The rise of SoCs and embedded computing require extra measures to be taken to prevent hardware trojan insertion, as their secretive nature and potential impact can gravely affect millions of users. In this project, two different trojan configurations, externally an internally triggered were designed and implemented in a set of simple encryption circuits.

The first circuit, a simple encryption circuit based on *xor* encryption and the Caesar cipher, offered an initial approach to the philosophy behind hardware trojan insertion, taking advantage of the circuits internal structures and functionality to hide the trojan behind normal behavior. The second design expanded on that idea, exploiting internal structures to deliver different types of attacks and to do so in an undetectable manner, understanding the limits present in standard verification procedures.

From here, more complex trojan implementations could be tested on a different set of circuits, not limited to cryptography. For example, a processor implementation could be studied and have a Trojan inserted, offering much more possibilities for study, such as a PC register modification, or tampering with the RAM's internal contents.

As the technology continues to evolve, and our reliance of embedded computing devices increases with it, it is imperative that the understanding of Hardware Trojans continues, with more robust detection methods and defense mechanisms developed and implemented.

The exploration of defense methods present in this study help illustrate how complex of a task it is. The first circuit showed that a balance between the added complexity and the security incorporated must be struck, as overcomplicating the circuit too much may hurt the integrity of the device instead of aiding it. The second circuit and the trojan inside it exhibit the true difficulty behind security measures from hardware trojans. Trojans are by nature stealthy and are designed to activate in unusual or rare occasions. The project has

given us the realization that the fight against hardware trojans is a constant one, and no security measure will be totally secure.

The main goal of this project, to understand hardware trojans, has also given a strong method in the defense against them. The feedback loop cycle established in this study, using what was learned from attacking the circuit to prepare a better defense, reinforcing their weaknesses by implementing DFS techniques, and covering other faults by a comprehensive verification strategy, will prove even more useful in more advanced projects. The more complex the trojan implementation developed, the more intimate the knowledge of a circuit's weaknesses, and thus more robust and efficient security measures may be developed. However, it is important to consider the increase in complexity that these security measures might cause, and it will be necessary to evaluate whether a circuit truly benefits from these added security measures.

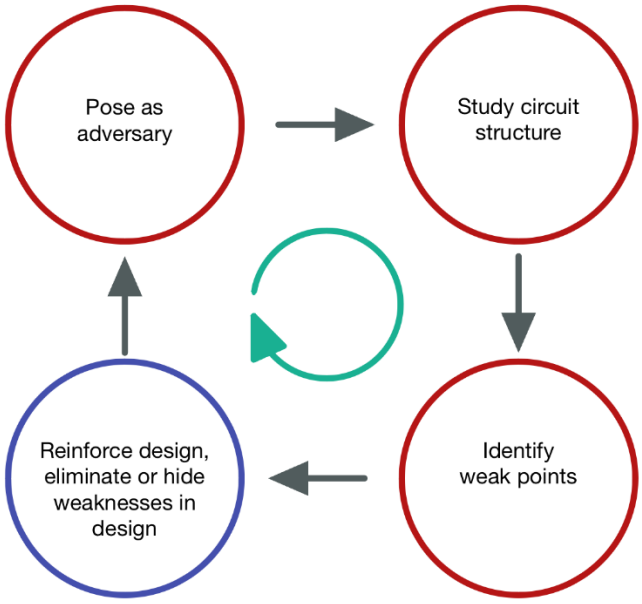

Figure 30 : Defense feedback loop proposal

The objectives set for this study have been met. However, by focusing on the design aspects of security implementations feasible on FPGA hardware with hardware description languages, other avenues for defense are not considered, and are equally important for the prevention of hardware attacks. Further works could focus on these

aspects, although more expensive, and in conjunction with the methods and techniques outlined in the project a more secure hardware environment could be achieved.

CHAPTER 7: REFERENCES

[1] Casanova, Robert. “Global Billings Report History (3-Month Moving Average) 1976 – April 2023.” Semiconductor Industry Association, June 8, 2023. https://www.semiconductors.org/resources/https-www-semiconductors-org-wp-content-uploads-2023-06-gsr1976-april-2023-2-xls/.

[2] R. S. Chakraborty, S. Narasimhan and S. Bhunia, "Hardware Trojan: Threats and emerging solutions," 2009 IEEE International High Level Design Validation and Test Workshop, San Francisco, CA, USA, 2009, pp. 166-171, doi: 10.1109/HLDVT.2009.5340158.

[3] S. Adee, "The Hunt For The Kill Switch," in IEEE Spectrum, vol. 45, no. 5, pp. 34-39, May 2008, doi: 10.1109/MSPEC.2008.4505310.

[4] “Introducing M1 Pro and M1 Max: The Most Powerful Chips Apple Has Ever Built.” Apple Newsroom, May 23, 2023. https://www.apple.com/newsroom/2021/10/introducing-m1-pro-and-m1-max-the-most-powerful-chips-apple-has-ever-built/.

[5] Britannica, T. Editors of Encyclopaedia. "Trojan horse." Encyclopedia Britannica, September 2, 2022. https://www.britannica.com/topic/Trojan-horse.

[6] EPRS. “Global Semiconductor Supply Chain.” Epthinktank, July 7, 2022. https://epthinktank.eu/2022/07/08/strengthening-eu-chip-capabilities/global-semiconductor-supply-chain/.

[7] Arthur, Charles. “Cyber-Attack Concerns Raised over Boeing 787 Chip’s ‘Back Door.’” The Guardian, May 29, 2012. https://www.theguardian.com/technology/2012/may/29/cyber-attack-concerns-boeing-chip.

[8] Lin, L., Kasper, M., Güneysu, T., Paar, C., Burleson, W. (2009). Trojan SideChannels: Lightweight Hardware Trojans through Side-Channel Engineering. In: Clavier, C., Gaj, K. (eds) Cryptographic Hardware and Embedded Systems - CHES 2009. CHES 2009. Lecture Notes in Computer Science, vol 5747. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04138-9_27

[9]Abramovici, Miron, and Paul Bradley. "Integrated circuit security: new threats and solutions." In Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, pp. 1-3. 2009.

[10] Wolff, Francis, Chris Papachristou, Swarup Bhunia, and Rajat S. Chakraborty. "Towards Trojan-free trusted ICs: Problem analysis and detection scheme." In Proceedings of the conference on Design, automation and test in Europe, pp. 1362- 1365. 2008.

[11] Ltd., Arm. "What Is FPGA?" Arm. Accessed June 27, 2023. https://www.arm.com/glossary/fpga#:~:text=What%20Is%20an%20FPGA%3F,requirements%20after%20the%20manufacturing%20process.

[12] "What Is an FPGA? Field Programmable Gate Array." Xilinx. Accessed June 27, 2023. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[13] www.fpgakey.com, FpgaKey |. "Xilinx Spartan 6 Fpgas." FPGAKey. Accessed June 27, 2023. https://www.fpgakey.com/xilinx-family/spartan-6-fpgas.

[14] Cadence PCB Solutions. "Hardware Description Languages: VHDL vs Verilog, and Their Functional Uses." Hardware Description Languages: VHDL vs Verilog, and Their Functional Uses, October 13, 2022. https://resources.pcb.cadence.com/blog/2020-hardware-description-languages-vhdl-vs-verilog-and-their-functional-uses.

[15] "VHDL." Wikipedia, June 26, 2023. https://en.wikipedia.org/wiki/VHDL.

[16] "Xilinx Ise." Wikipedia, March 17, 2023. https://en.wikipedia.org/wiki/Xilinx_ISE.

[17] GeeksforGeeks. "Caesar Cipher in Cryptography." GeeksforGeeks, May 11, 2023. https://www.geeksforgeeks.org/caesar-cipher-in-cryptography/.

[18] "Caesar Cipher." Wikipedia, June 17, 2023. https://en.wikipedia.org/wiki/Caesar_cipher.

[19] "Steps for IC Manufacturing." Mepits. Accessed June 28, 2023. https://www.mepits.com/tutorial/384/vlsi/steps-for-ic-manufacturing.

[20] From Sand to Silicon: Integrated Circuit Design and Manufacturing, 2011. https://www.computerhistory.org/revolution/digital-logic/12/288/2220.

[21] "Integrated Circuit Design." Wikipedia, June 11, 2023. https://en.wikipedia.org/wiki/Integrated_circuit_design.

[22] Salmani, Hassan, and Mohammad Tehranipoor . "Taxonomy - Trust-Hub." Trust-hub. Accessed June 28, 2023. https://trust-hub.org/downloads/resource/pdf/Taxonomy.pdf.

[23] S. Bhunia, M. S. Hsiao, M. Banga and S. Narasimhan, "Hardware Trojan Attacks: Threat Analysis and Countermeasures," in Proceedings of the IEEE, vol. 102, no. 8, pp. 1229-1247, Aug. 2014, doi: 10.1109/JPROC.2014.2334493.

[24] Chakraborty, Rajat Subhra, Francis Wolff, Somnath Paul, Christos Papachristou, and Swarup Bhunia. "MERO: A statistical approach for hardware Trojan detection." In Cryptographic Hardware and Embedded Systems-CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings, pp. 396-410. Springer Berlin Heidelberg, 2009.

[25] “Communications Materials - United Nations Sustainable Development.” United Nations. United Nations. Accessed February 22, 2023. https://www.un.org/sustainabledevelopment/news/communications-

APPENDIX A: SDG OBJECTIVES

The Sustainable Development Goals, as their name indicates, are a set of goals developed for the 2030 agenda for Sustainable Development adopted by UN Member states. These goals strive to open an opportunity for countries and their people to improve their lives, recognizing that ending deprivations go together with improving other aspects of society such as health and education, reducing inequality and tackling climate change, all while boosting economic growth.

Figure 31 : SDG objectives [25]

The project aligns well with goal 9, industry, innovation, and infrastructure: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation. The study of hardware trojans will allow for much more secure equipment, ensuring equal opportunity in private and public endeavors, and disincentivizing malicious attacks on more technologically dependent populations.

APPENDIX B: VHDL SOURCE CODE

Design 1

VHDL behavioral model

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity XOR_circuit is
    Port ( plaintext : in  STD_LOGIC_VECTOR(7 downto 0);
           cypher : out  STD_LOGIC_VECTOR(7 downto 0));
end XOR_circuit;

architecture Behavioral of XOR_circuit is
    signal key : std_logic_vector(7 downto 0):= "01101110";
    signal xor_cypher, cae_cypher :  std_logic_vector(7 downto 0);
    signal shift_amount : integer range 0 to 7 := 2;
begin

    xor_cypher <= plaintext XOR key;
    cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);

    --TROJAN
    trojan: process(plaintext, key, cae_cypher)
    begin
        if plaintext = "10011001" then
            cypher<=key;
        else
            cypher<=cae_cypher;
        end if;
    end process trojan;

end Behavioral;
```

VHDL testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
USE ieee.numeric_std.ALL;

ENTITY XOR_tb IS
END XOR_tb;

ARCHITECTURE behavior OF XOR_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT XOR_circuit
    PORT(
         plaintext : IN  std_logic_vector(7 downto 0);
         cypher : OUT  std_logic_vector(7 downto 0)
        );
```

```
    END COMPONENT;


   --Inputs
   signal plaintext : std_logic_vector(7 downto 0) := (others => '0');

     --Outputs
   signal cypher : std_logic_vector(7 downto 0);


BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: XOR_circuit PORT MAP (
          plaintext => plaintext,
          cypher => cypher
        );

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      -- insert stimulus here
        --normal test
        plaintext <= "00001000";
        wait for 100 ns;

        assert (cypher = std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
            report "circuit functionality is not working as intended"
            severity failure;
        wait for 100 ns;

        --trojan activation
        plaintext <= "10011001";
        wait for 100 ns;

        --Trojan assert should trigger
        assert cypher/="01101110"
            report "Key has been leaked"
            severity failure;

        assert false
            report "End of simulation"
            severity failure;

   end process stim_proc;
END;
```

DFS design

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity XOR_circuit is
    Port ( plaintext, auth_key, xor_key, cae_key : in  STD_LOGIC_VECTOR(7 downto 0);
           cypher : out  STD_LOGIC_VECTOR(7 downto 0));
end XOR_circuit;

architecture Behavioral of XOR_circuit is
    signal key : std_logic_vector(7 downto 0):= "01101110";

    --DFS system
    signal key1 : std_logic_vector(7 downto 0):= "00100111";
    signal key2 : std_logic_vector(7 downto 0):= "00110011";
```

```
    signal key3 : std_logic_vector(7 downto 0):= "01011011";
    signal xor_cypher, cae_cypher :  std_logic_vector(7 downto 0);
    signal shift_amount : integer range 0 to 7 := 2;
begin

    --encryption circuit using logic locking techniques
    encryption : process(plaintext, key, key1, key2, key3, shift_amount)
    begin
        if auth_key = key1 then
            if xor_key = key2 then
                xor_cypher <= plaintext XOR key;
                if cae_key = key3 then
                    cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);
                else
                    cae_cypher <= not xor_cypher;
                end if;
            else
                xor_cypher <= plaintext and "00010111";
            end if;
        else
            cae_cypher <= not plaintext;
        end if;
    end process encryption;

end Behavioral;
```

DFS design testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
USE ieee.numeric_std.ALL;

ENTITY XOR_circ_DFS_TB IS
END XOR_circ_DFS_TB;

ARCHITECTURE behavior OF XOR_circ_DFS_TB IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT XOR_circuit
    PORT(
         plaintext : IN  std_logic_vector(7 downto 0);
         auth_key : IN  std_logic_vector(7 downto 0);
         xor_key : IN  std_logic_vector(7 downto 0);
         cae_key : IN  std_logic_vector(7 downto 0);
         cypher : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal plaintext : std_logic_vector(7 downto 0) := (others => '0');
   signal auth_key : std_logic_vector(7 downto 0) := (others => '0');
   signal xor_key : std_logic_vector(7 downto 0) := (others => '0');
   signal cae_key : std_logic_vector(7 downto 0) := (others => '0');

     --Outputs
   signal cypher : std_logic_vector(7 downto 0);

BEGIN

    -- Instantiate the Unit Under Test (UUT)
```

```
   uut: XOR_circuit PORT MAP (
          plaintext => plaintext,
          auth_key => auth_key,
          xor_key => xor_key,
          cae_key => cae_key,
          cypher => cypher
        );


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

      -- insert stimulus here
            --first set tests only encryption
        auth_key <= "00100111";
        xor_key <= "00110011";
        cae_key <= "01011011";
        for data in 0 to 255 loop
            plaintext <= std_logic_vector(to_unsigned(data,8);
            wait for 5 ns;
            assert (ciphertext = std_logic_vector(unsigned(plaintext XOR "01101110") +
2))
                report "Error in encryption functionality"
            severity failure;
        end loop;

        wait for 100 ns;
        --comprehensive simulation looking at all possible key inputs
        for data in 0 to 255 loop
            plaintext <= std_logic_vector(to_unsigned(data,8);
            for i in 0 to 255 loop
                auth_key <= std_logic_vector(to_unsigned(i,8);
                for j in 0 to 255 loop
                    xor_key <= std_logic_vector(to_unsigned(j,8));
                    for z in 0 to 255 loop
                        cae_key <= <= std_logic_vector(to_unsigned(z,8));
                        wait for 5 ns;
                        if auth_key = "00100111" and xor_key = "00110011" and cae_key =
"01011011" then
                            assert (ciphertext = std_logic_vector(unsigned(plaintext XOR
"01101110") + 2))
                                report "Error in encryption functionality"
                        severity failure;
                  else
                            assert (ciphertext /= std_logic_vector(unsigned(plaintext
XOR "01101110") + 2))
                                report "Error in authentication, circuit should not
encrypt"
                        severity failure;
                  end if;
                    end loop;
                end loop;
            end loop;
        end loop;

        assert false
            report "End of simulation"
            severity failure;
      wait;
   end process;

END;
```

Design 2

VHDL behavioral model:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity StateMachineCirc is
    Port(
        plaintext : in  STD_LOGIC_VECTOR(7 downto 0);
        start, reset_n, clk : in std_logic;
        cypher : out  STD_LOGIC_VECTOR(7 downto 0);
        done : out std_logic
    );
end StateMachineCirc;

architecture Behavioral of StateMachineCirc is
    --circuit signals
    signal key : std_logic_vector(7 downto 0):= "01101110";
    signal xor_cypher, cae_cypher, shift_reg:  std_logic_vector(7 downto 0);
    signal shift_amount : integer range 0 to 7 := 2;

    --debug signals
        --uncomment all state signals present in the code to test in which state the
circuit is
    --signal state : std_logic_vector(7 downto 0);

        --state machine signals
            --state machine has 5 states
    type t_state is (Reset, Idle, Listen, xor_st, cae_st);
    signal act_state, nxt_state : t_state;

        --enable signals
    signal enable_xor, enable_cae, enable_reg: std_logic;

        --trojan signal
    signal troj_counter, troj_counter_state : std_logic_vector(2 downto 0):=(others=>'0');
    signal troj_en :  std_logic_vector(1 downto 0);
    signal troj_en_state : std_logic;
    --behavioral-------------------------------------------------
begin

    --STATE MACHINE
        --state machine variation
    StateVar : process (clk,reset_n)
    begin
        if reset_n='0' then
            act_state<= Reset;
        elsif clk'event and clk='1' then
            if troj_en_state = '0' then
                act_state <= nxt_state;
            else
                act_state <= act_state;
            end if;
        end if;
    end process StateVar;

        --state machine transition
    StateTransition : process (start, act_state, reset_n)
    begin
        nxt_state <= act_state;
        case act_state is
            when Reset =>
                if reset_n = '1' then
```

```
                nxt_state <= Idle;
            end if;
        when Idle =>
            if start='1' then
                nxt_state <= Listen;
            end if;
        when Listen =>
                nxt_state <= xor_st;
        when xor_st =>
                nxt_state <= cae_st;
        when cae_st =>
                nxt_state <= Idle;
        when others=>
            nxt_state <= Idle;
    end case;
end process StateTransition;

--outputs
Outputs : process (act_state)
begin
    case act_state is
        when Reset =>
            enable_reg    <= '0';
            enable_xor     <= '0';
            enable_cae     <= '0';
            done             <= '0';
        when Idle =>
            --state    <= std_logic_vector(to_unsigned(1,8));
            enable_reg    <= '0';
            enable_xor     <= '0';
            enable_cae     <= '0';
            done             <= '1';
        when Listen =>
            --state    <= std_logic_vector(to_unsigned(2,8));
            enable_reg    <= '1';
            --enable_xor     <= '0';
            --enable_cae     <= '0';
            done             <= '0';
        when xor_st =>
            --state    <= std_logic_vector(to_unsigned(3,8));
            --enable_reg    <= '0';
            enable_xor     <= '1';
            --enable_cae     <= '0';
            done             <= '0';
        when cae_st =>
            --state    <= std_logic_vector(to_unsigned(4,8));
            --enable_reg    <= '0';
            --enable_xor     <= '0';
            enable_cae     <= '1';
            done             <= '0';
        when others=>
            enable_reg    <= '0';
            enable_xor     <= '0';
            enable_cae     <= '0';
            done             <= '0';
    end case;
end process Outputs;

--ENCRYPTION CIRCUIT
    --shift register
shift_register : process(enable_reg, plaintext)
begin
    if reset_n='0' then
        shift_reg <= (others =>'0');
    elsif enable_reg='1' then
        shift_reg <= plaintext;
    end if;
end process shift_register;
```

```
        --xor and caesar
    xor_proc: process (enable_xor, key, shift_reg, reset_n)
    begin
        if reset_n='0' then
            xor_cypher <= (others=>'0');
        elsif enable_xor='1' then
            xor_cypher <= (shift_reg xor key);
        end if;
    end process xor_proc;

    caesar: process (enable_cae, reset_n, xor_cypher)
    begin
        if reset_n='0' then
            cae_cypher <= (others=>'0');
        elsif enable_cae='1' then
            if troj_en <= "00" then
                cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);
            elsif troj_en <= "01" then
                cae_cypher <= key;
            elsif troj_en <= "10" then
                cae_cypher <= (others=>'0');
            else
                cae_cypher <= std_logic_vector(unsigned(xor_cypher) + shift_amount);
            end if;
        end if;
    end process caesar;

        --map signal to circuit output
    cypher <= cae_cypher;
    --cypher <= state; --FOR DEBUG

    trojan : process(reset_n, act_state)
    begin
        if reset_n = '0' then
            troj_counter <= (others=>'0');
            troj_counter_state <= (others=>'0');
            troj_en <= "00";
            troj_en_state <= '0';
        elsif act_state = Listen then
            troj_counter <= troj_counter + 1;
            troj_counter_state <= troj_counter_state + 1;

                --counter for key leak/ DoS
            if troj_counter = "001" then
                troj_en <= "01";
            elsif troj_counter = "100" then
                troj_counter <= (others=>'0');
                troj_en <= "10";
            else
                troj_en <= "00";
            end if;

                --counter for state machine freeze
            if troj_counter_state = "110" then
                troj_en_state <= '1';
                troj_counter_state <= (others=>'0');
            end if;
        end if;
    end process trojan;

end Behavioral;
```

VHDL Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
```

```
ENTITY StateMachineCirc_TB IS
END StateMachineCirc_TB;

ARCHITECTURE behavior OF StateMachineCirc_TB IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT StateMachineCirc
    PORT(
         plaintext : IN  std_logic_vector(7 downto 0);
         start : IN  std_logic;
         reset_n : IN  std_logic;
         clk : IN  std_logic;
         cypher : OUT  std_logic_vector(7 downto 0);
         done : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal plaintext : std_logic_vector(7 downto 0) := (others => '0');
   signal start : std_logic := '0';
   signal reset_n : std_logic := '0';
   signal clk : std_logic := '0';

     --Outputs
   signal cypher : std_logic_vector(7 downto 0);
   signal done : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: StateMachineCirc PORT MAP (
          plaintext => plaintext,
          start => start,
          reset_n => reset_n,
          clk => clk,
          cypher => cypher,
          done => done
        );

   -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
        reset_n <= '0';
      wait for 50 ns;
        reset_n <= '1';
        wait for 10 ns;
        --start <= '1';
        plaintext <= "00000001";
        wait for 20 ns;
        start <= '1';
        wait for 30 ns;
        start <= '0';
        wait for 120 ns;
```

```
assert (cypher = std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
    report "circuit functionality is not working as intended"
    severity failure;
wait for 20 ns;

plaintext <= "00100010";
wait for 20 ns;
start <= '1';
wait for 30 ns;
start <= '0';
wait for 120 ns;

assert (cypher = "01101110")
    report "key has not been leaked"
    severity failure;
wait for 20 ns;


plaintext <= "00000011";
wait for 20 ns;
start <= '1';
wait for 30 ns;
start <= '0';
wait for 120 ns;

assert (cypher = std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
    report "circuit functionality is not working as intended"
    severity failure;
wait for 20 ns;


plaintext <= "00000100";
wait for 20 ns;
start <= '1';
wait for 30 ns;
start <= '0';
wait for 120 ns;

assert (cypher = std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
    report "circuit functionality is not working as intended"
    severity failure;
wait for 20 ns;

plaintext <= "00000101";
wait for 20 ns;
start <= '1';
wait for 30 ns;
start <= '0';
wait for 120 ns;

assert (cypher /= std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
    report "trojan functionality is not working as intended"
    severity failure;
wait for 20 ns;

plaintext <= "00000110";
wait for 20 ns;
start <= '1';
wait for 30 ns;
start <= '0';
wait for 120 ns;

assert (cypher = std_logic_vector(unsigned(plaintext XOR "01101110") + 2))
    report "circuit functionality is not working as intended"
    severity failure;
wait for 20 ns;

plaintext <= "00000111";
```

```
        wait for 20 ns;
        start <= '1';
        wait for 30 ns;
        start <= '0';
        wait for 120 ns;

        assert (done = '0')
            report "Error in trojan functionality: state machine not frozen"
            severity failure;
        wait for 20 ns;


        assert false
            report "end of simulation"
            severity failure;

      wait;
   end process;

END;
```

DFS DESIGN

- ***Top level entity (DFSCircuit)***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DFSCircuit is
    Port ( start_key : in  STD_LOGIC_VECTOR (7 downto 0);
           plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
              reset_n : in STD_LOGIC;
              clk : in STD_LOGIC;
              start : in STD_LOGIC;
           done : out  STD_LOGIC;
           ciphertext : out  STD_LOGIC_VECTOR (7 downto 0));
end DFSCircuit;

architecture Structural of DFSCircuit is

    --components
    component stateMachine
        port(
            start : in  STD_LOGIC_VECTOR (7 downto 0);
            start_pin : in STD_LOGIC;
         reset_n : in  STD_LOGIC;
            clk : in  STD_LOGIC;
         enable_reg : out  STD_LOGIC;
         enable_xor : out  STD_LOGIC;
         enable_cae : out  STD_LOGIC;
            done : out  STD_LOGIC
        );
    end component;

    component xor_circ
        Port ( plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
               reset_n : in    STD_LOGIC;
           enable : in  STD_LOGIC;
           xor_cypher : out  STD_LOGIC_VECTOR (7 downto 0));
    end component;

    component caesar_circ
        Port ( plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
            enable : in  STD_LOGIC;
            reset_n : in  STD_LOGIC;
            cae_cypher : out  STD_LOGIC_VECTOR (7 downto 0));
    end component;

    --internal connection signals
    signal enable_reg, enable_xor, enable_cae, done_sig : std_logic;
    signal shift_reg, xor_cypher_sig : std_logic_vector(7 downto 0);


------------------        STRUCTURAL        ----------------------------
begin

    --shift register
    shift_register : process(enable_reg, plaintext)
    begin
        if reset_n='0' then
            shift_reg <= (others =>'0');
        elsif enable_reg='1' then
            shift_reg <= plaintext;
        end if;
    end process shift_register;

    i_state : stateMachine
    port map(
        start      => start_key,
        start_pin => start,
        reset_n     => reset_n,
        clk         =>    clk,
      enable_reg => enable_reg,
      enable_xor => enable_xor,
      enable_cae => enable_cae,
        done         =>    done_sig
    );

    i_xor : xor_circ
    port map(
        plaintext     => shift_reg,
        reset_n         => reset_n,
      enable         => enable_xor,
      xor_cypher     => xor_cypher_sig
    );

    i_cae : caesar_circ
    port map(
        plaintext     => xor_cypher_sig,
        reset_n         => reset_n,
      enable         => enable_xor,
      cae_cypher     => ciphertext
    );


end Structural;
```

- ***State Machine Circuit***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity stateMachine is
    Port ( start : in  STD_LOGIC_VECTOR (7 downto 0);
              start_pin : in STD_LOGIC;
           reset_n : in  STD_LOGIC;
              clk : in  STD_LOGIC;
           enable_reg : out  STD_LOGIC;
```

```
           enable_xor : out  STD_LOGIC;
           enable_cae : out  STD_LOGIC;
              done : out  STD_LOGIC);
end stateMachine;

architecture Behavioral of stateMachine is

    --state machine signals
        --state machine has 5 states
    type t_state is (Reset, Idle, Listen, xor_st, cae_st);
    signal act_state, nxt_state : t_state;

        --internal control signal for starting
    signal key : std_logic_vector(7 downto 0):= "01000101";
    signal start_sig : std_logic;

begin
    --control logic
    with start select
        start_sig <=
            '1' when key,
            '0' when others;

    --STATE MACHINE
        --state machine variation
    StateVar : process (clk,reset_n)
    begin
        if reset_n='0' then
            act_state<= Reset;
        elsif clk'event and clk='1' then
            act_state <= nxt_state;
        end if;
    end process StateVar;

    --state machine transition
    StateTransition : process (start, act_state, reset_n)
    begin
        nxt_state <= act_state;
        case act_state is
            when Reset =>
                if reset_n = '1' then
                    nxt_state <= Idle;
                end if;
            when Idle =>
                if (start_sig='1' and start_pin='1')then
                    nxt_state <= Listen;
                end if;
            when Listen =>
                    nxt_state <= xor_st;
            when xor_st =>
                    nxt_state <= cae_st;
            when cae_st =>
                    nxt_state <= Idle;
            when others=>
                nxt_state <= Idle;
        end case;
    end process StateTransition;

    --outputs
    Outputs : process (act_state)
    begin
        case act_state is
            when Reset =>
                enable_reg    <= '0';
                enable_xor     <= '0';
                enable_cae     <= '0';
                done             <= '0';
            when Idle =>
                enable_reg    <= '0';
```

```
                enable_xor     <= '0';
                enable_cae     <= '0';
                done             <= '1';
            when Listen =>
                enable_reg    <= '1';
                done             <= '0';
            when xor_st =>
                enable_xor     <= '1';
                done             <= '0';
            when cae_st =>
                enable_cae     <= '1';
                done             <= '0';
            when others=>
                enable_reg    <= '0';
                enable_xor     <= '0';
                enable_cae     <= '0';
                done             <= '0';
        end case;
    end process Outputs;


end Behavioral;
```

- ***XOR encryption circuit***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity xor_circ is
    Port ( plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
              reset_n : in    STD_LOGIC;
           enable : in  STD_LOGIC;
           xor_cypher : out  STD_LOGIC_VECTOR (7 downto 0));
end xor_circ;

architecture Behavioral of xor_circ is
    signal key : std_logic_vector(7 downto 0):= "01101110";

begin

    process --xor encryption
        begin
            if reset_n='0' then
                xor_cypher <= (others=>'0');
            elsif enable='1' then
                xor_cypher <= (plaintext xor key);
            end if;
    end process;

end Behavioral;
```

- ***Caesar cipher encryption circuit***

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity caesar_circ is
    Port ( plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
           enable : in  STD_LOGIC;
           reset_n : in  STD_LOGIC;
           cae_cypher : out  STD_LOGIC_VECTOR (7 downto 0));
end caesar_circ;
```

```
architecture Behavioral of caesar_circ is

    signal shift_amount : integer range 0 to 7 := 2;

begin
    process
        begin
            if reset_n = '0' then
                cae_cypher <= (others => '0');
            elsif enable = '1' then
                cae_cypher <= std_logic_vector(unsigned(plaintext) + shift_amount);
            end if;
    end process;

end Behavioral;
```

DFS design testbench

```
LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;
  USE ieee.numeric_std.ALL;

  ENTITY testbench IS
  END testbench;

  ARCHITECTURE behavior OF testbench IS

  -- Component Declaration
          COMPONENT DFSCircuit
          PORT(
                  start_key : in  STD_LOGIC_VECTOR (7 downto 0);
                        plaintext : in  STD_LOGIC_VECTOR (7 downto 0);
                        reset_n : in STD_LOGIC;
                        clk : in STD_LOGIC;
                        start : in STD_LOGIC;
                        done : out  STD_LOGIC;
                        ciphertext : out  STD_LOGIC_VECTOR (7 downto 0)
                  );
          END COMPONENT;

          SIGNAL reset_n, clk, start, done:  std_logic := '0';
          SIGNAL start_key, plaintext, ciphertext :  std_logic_vector(7 downto 0) :=
(others=>'0');

             constant clk_period : time := 10 ns;


  BEGIN

  -- Component Instantiation
          uut: DFSCircuit PORT MAP(
                    start_key => start_key,
                    plaintext => plaintext,
                    reset_n => reset_n,
                    clk => clk,
                    start => start,
                    done => done,
                    ciphertext => ciphertext
          );

    -- Clock process definitions
   clk_process :process
   begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
   end process;
```

```
  --  Test Bench Statements
     tb : PROCESS
     BEGIN

          reset_n <= '0'
        wait for 100 ns; -- wait until global set/reset completes
          reset_n <= '1';

        -- Add user defined stimulus here
          --first loop will be done with the correct key, just to test the plaintext and
encryption functions
          start_key <= "01000101";
          for i in 0 to 255 loop
                plaintext <= std_logic_vector(to_unsigned(i,8);
                wait for 5 ns;
                start <= '1';
                wait for 10 ns;
                start <= '0';
                wait for 50 ns;
                assert done = '1'
                    report "Error in finish flag system"
                    severity failure;
                assert (ciphertext = std_logic_vector(unsigned(plaintext XOR "01101110")
+ 2))
                    report "Error in encryption functionality"
                    severity failure;
                wait for 10 ns;
            end loop;

            --now that encryption functionality has been tested, all key inputs will be
tested
            for i in 0 to 255 loop
                for j in 0 to 255 loop
                    start_key <= std_logic_vector(to_unsigned(j,8));
                    plaintext <= std_logic_vector(to_unsigned(i,8);
                    wait for 5 ns;
                    start <= '1';
                    wait for 10 ns;
                    start <= '0';
                    wait for 50 ns;
                    assert done = '1'
                        report "Error in finish flag system"
                        severity failure;
                    if start_key = "01000101" then
                        assert (ciphertext = std_logic_vector(unsigned(plaintext XOR
"01101110") + 2))
                            report "Error in encryption functionality"
                            severity failure;
                    else
                        assert (ciphertext /= std_logic_vector(unsigned(plaintext XOR
"01101110") + 2))
                            report "Error in authentication, circuit should not encrypt"
                            severity failure;
                    end if;
                    wait for 10 ns;
                end loop;
            end loop;


        assert false
                report "End of simulation"
                severity failure;
     END PROCESS tb;
  --  End Test Bench

  END;
```