# SECURE KEY UPDATES FOR DYNAMICALLY RECONFIGURABLE LOGIC LOCKED DESIGNS

by

Gregory R Williams

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Engineering

Charlotte

2022

Approved by:

_____
Dr. Fareena Saqib

_____
Dr. Arun Ravindran

_____
Dr. Ronald Sass

# ABSTRACT

GREGORY R WILLIAMS. Secure key updates for dynamically reconfigurable logic locked designs. (Under the direction of DR. FAREENA SAQIB)

Due to globalization and the shift to the horizontal business model, there are emerging security concerns in the semiconductor industry including FPGAs. Modern FPGAs are system on a chip platforms that integrates a processing system with programmable logic. The horizontal design flow for FPGAs supports third party intellectual properties integration into a design. Untrustworthy entities within in the design flow can have several points of attack against the intellectual properties such as intellectual property piracy, reverse engineering, hardware Trojans, and bitstream cloning.

Logic locking is a mechanism to design trusted intellectual property, and its distribution. Logic locking inserts additional logic into a design with key inputs where the outputs are obfuscated and the design is functional only when a correct key combination is given. Current logic locking schemes hard code the key value into an IP and rely on the assumption that the key will be kept secure during the life cycle of the chip. This work proposes a key update mechanism for logic locked IPs that unlike current schemes, provides dynamically reconfigurable lock updates to the IP and key deployment in a trusted execution environment. We assess the security of the locked IPs and evaluate resource and timing overhead of the proposed scheme. This work demonstrates that the reconfigurable logic locked IP technique is feasible, with results supporting that the locking scheme meets all timing requirements.

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and chair of my committee, Dr. Fareena Saqib, for the continuous motivation and distribution of her immense knowledge towards my master's study and thesis research. Her guidance and expertise has assisted me throughout my studies and writing this thesis. I sincerely appreciate all of her efforts to ensure that my research and this thesis were presented at its best. In addition to my advisor, I would like to thank my thesis committee members Dr. Arun Ravindran and Dr. Ronald Sass for their valuable suggestions, time and support during my thesis. Finally, I would like to thank my friends and family for their constant support.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

3PIP Third Party IP

AXI Advanced eXtensible Interface

DIP Distinguishing Input Pattern

DMA Direct Memory Access

DPR Dynamic Partial Reconfiguration

FPGA Field Programmable Gate Array

ICAP Internal Configuration Access Port

IP Intellectual Property

LUT Look Up Table

PCAP Processor Configuration Access Port

PL Programmable Logic

PS Processing System

RTL Register Transfer Level

SOC System on Chip

TEE Trusted Execution Environment

# CHAPTER 1: INTRODUCTION

## 1.1 Motivation

With the rise of globalization and increasing demand, the semiconductor industry in recent years has shifted to the horizontal business model where the design house, foundry, and assembly are distributed across different companies [1]. Modern Field-Programmable Gate Arrays (FPGA)s are System on a Chip (SoC) platforms and are used in commercial and government systems. The FPGA design flow support Third Party Intellectual Properties (3PIP) integration that are sourced from many different vendors and are combined together in a FPGA design [2]. The design flow improves the design time and cost, but introduces trust and security issues for the Intellectual Property (IP) owner and system integrator through the distribution and integration of the IP [3].

The horizontal FPGA design flow, shown in Figure 1.1, starts with integration of 3PIP designs, that are integrated with in-house SoC design solutions. The system integrator combines the 3PIP designs along with other design components at the system level and generates the final bitstream. A bitstream is a binary file that programs the FPGA to run the intended application. The bitstream is provided to a system programmer who is responsible for loading the bitstream into the FPGA to be used in the field.

Untrustworthy entities within the design flow can have several points of attack against IPs such as IP piracy, reverse engineering, hardware Trojans, and bitstream cloning. A hardware Trojan is a malicious modification of an IP by an untrusted IP owner or system integrator to produce undesired behavior of an IP and to open side channels for eavesdropping [4]. IP piracy is a risk to 3PIP owners that license their

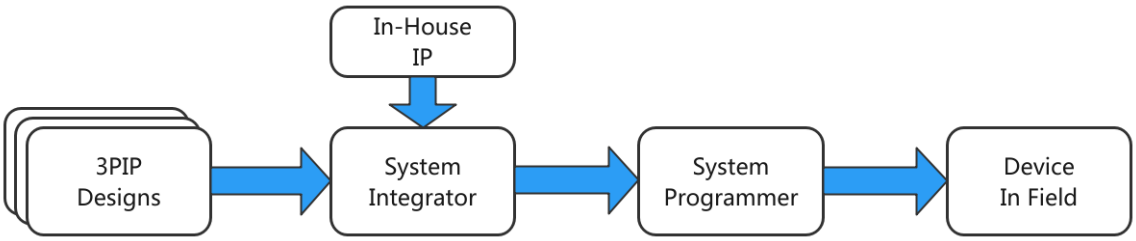



Figure 1.1: Modern FPGA design flow with the horizontal business model.

IP designs to untrustworthy system integrators who may over use their IP and sell more copies than agreed upon in the licensing agreements. 3PIP vendors additionally are at risk of system integrators using reverse engineering attacks to produce cloned IPs.

To provide IP protection, hardware obfuscation techniques such as logic locking have been proposed. Logic locking inserts additional logic into a design at the Register Transfer Level (RTL), gate, or layout level. Inputs to the inserted logic are designated as key inputs. The inserted logic is designed to produce incorrect outputs of the IP if an incorrect key is given. If the key inputs are correct the logic will leave the IP unaffected and produce the correct outputs. Users of the IP will not have a functional design unless they obtain the correct key from the IP vendor. A conceptual overview of a logic locked IP is shown in Figure 1.2.

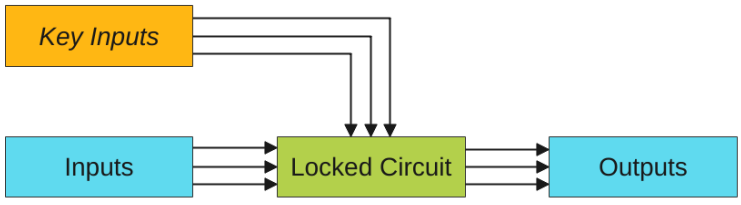



Figure 1.2: Overview of Logic Locking.

Logic locking is a promising defense against IP piracy, Trojan insertions, and counterfeiting as the original behavior of the IP is obfuscated to make the reverse engineering process difficult [5]. IPs are not usable unless the IP owner provides a correct key combination to unlock. Current logic locking techniques modify IPs by inserting

key logic at the RTL, gate, and layout level, which physically embeds a key value into the design. The current methods do not provide a mechanism to update the value of the key. To have IPs with different key values, design updates are needed and a new IP with different key logic must be generated. If the same locked IP is used on many devices, a leak of the key value on one device in the field means the security of all instances of the IP on other chips are also compromised.

Current logic locking techniques rely on an assumption that the secret key for the IP resides in tamper-proof memory on a SoC platform [6]. On a SoC platform the secret key is read from the tamper-proof memory and programmed to the IP by software running on a processor within the SoC. The memory containing the secret key may be tamper-proof from external probes, but an attacker may gain malicious access of host software executing on the processor. The malicious access can be used to access and leak the secret key of the IP.

In this work, we propose and demonstrate a novel logic locking scheme for combinational logic locking for FPGA architectures. Unlike existing logic locking schemes, our proposed scheme provides a key update mechanism for the locked IP through the use of partial reconfiguration. The partial reconfiguration changes the key logic of the IP, subsequently updating the correct key combination. Furthermore, we demonstrate a secure key-provisioning system enabled with a hardware based Trusted Execution Environment (TEE) to protect the key during deployment and reconfiguration. This work demonstrates that the reconfigurable logic locked IP technique is feasible, with results supporting that the locking scheme meets all timing requirements.

## 1.2 Contributions

This work makes the following contributions:

- Proposes an automated framework to allow the IP owner to obfuscate the IP design in the form of RTL files and integrates logic locking at the gate level. The IP designer can generate a locked IP without the prior knowledge on IP

security.

- Proposes and demonstrates a novel logic locking scheme that incorporates key logic into reconfigurable partitions of the FPGA to allow multiple configurations of the secret key for different clients and the ability to update the key during runtime.

- Proposes and demonstrates a secure system for key deployment and updates to the locked IP on a FPGA SoC Platform. The system is enabled with ARM TrustZone hardware extensions to establish a TEE for secure key application to unlock the IP.

## 1.3 Organization

The organization of the thesis is as follows. Chapter 2 describes the background information and existing works related to this work. Chapter 3 discusses the logic locking framework, proposed scheme for 3PIP integration, and the experimental setup necessary for implementation on a Xilinx ZYNQ platform. The automated framework, results, security and overhead analysis of the proposed technique is discussed in Chapter 4. Lastly, chapter 5 provides a conclusion and Chapter 6 suggests future work.

# CHAPTER 2: BACKGROUND

## 2.1 Logic Locking

To combat the attacks against 3PIP's in the modern day semiconductor supply chain, IP owners use hardware obfuscation techniques to hide the structure and logic of their IP. A hardware obfuscation technique that has recently come to prominence in the fight against IP piracy is logic locking [7]. Logic locking is a technique defined by inserting additional logic with key inputs into a netlist or gate level description of the circuit. The additional logic is designed to leave the circuit unchanged if the correct key is given but corrupt the outputs of the design if an incorrect key is given. Logic locking, sometimes called logic encryption, has implementations for sequential and combinational circuits, however this work focuses on combinational locking techniques.

### 2.1.1 Random Logic Locking

An elementary implementation of combinational locking logic is to insert XOR/XNOR gates, known as key gates, randomly throughout a netlist with one input connected to a key input. This technique is known as random insertion [8]. An incorrect input to a key gate causes the gate to flip its input, corrupting the functionality of the netlist. When a correct key is given, the key gate does not flip its input leaving the netlist to behave as intended. Table 2.1 shows the truth table for both XOR and XNOR gates. From the table, it can be observed that a key input of 0 for XOR and a key input of 1 for XNOR leaves the input unchanged at the output. This is considered the correct key input for the respective gates.

An example of random insertion using c17 from the ISCAS benchmark circuits is

Table 2.1: Truth table for XOR and XNOR key gates.

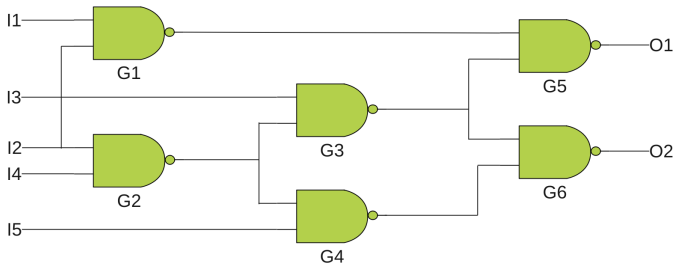
(a) XOR truth table

| K | In | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

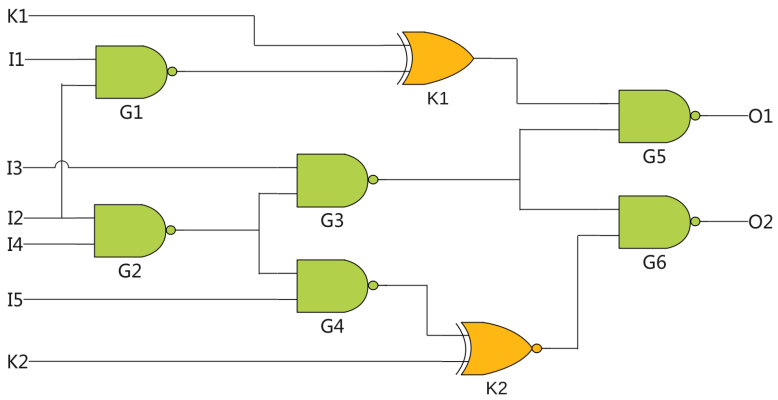
(b) XNOR truth table

| K | In | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

shown in Figure 2.1. The circuit in Figure 2.1b is modified in by inserting two key gates with inputs K1 and K2. An incorrect key such as 10 for K1 and K2 respectively causes both key gates to flip their inputs from the netlist which can produce incorrect outputs. If the correct key is given and key inputs K1 and K2 are 0 and 1 respectively, key gates K1 and K2 do not flip the input, which makes the circuit logically equivalent to the circuit if Figure 2.1a and produces the correct outputs.




(a) Original c17 ISCAS Benchmark Circuit




(b) Logic Locked c17 with Key Inputs K1 & K2

Figure 2.1: Logic locking example for ISCAS benchmark c17.

### 2.1.2 Fault-Analysis Based Logic Locking

The insertion of gates into the netlist does not guarantee that the outputs change when an incorrect key is applied. An incorrect key can be blocked from propagating to the output for certain input patterns. For example, take the logic locked circuit shown in Figure 2.1b. If gate G3 evaluates to a value of 0, gate G5 evaluates to 1 regardless of the input coming from the key gate, and even an incorrect key produces correct results. Key gates can also be masked by a second key gate on the same path. If the second key gate also has an incorrect key input, it can reverse what is done by the first incorrect key resulting in correct outputs.

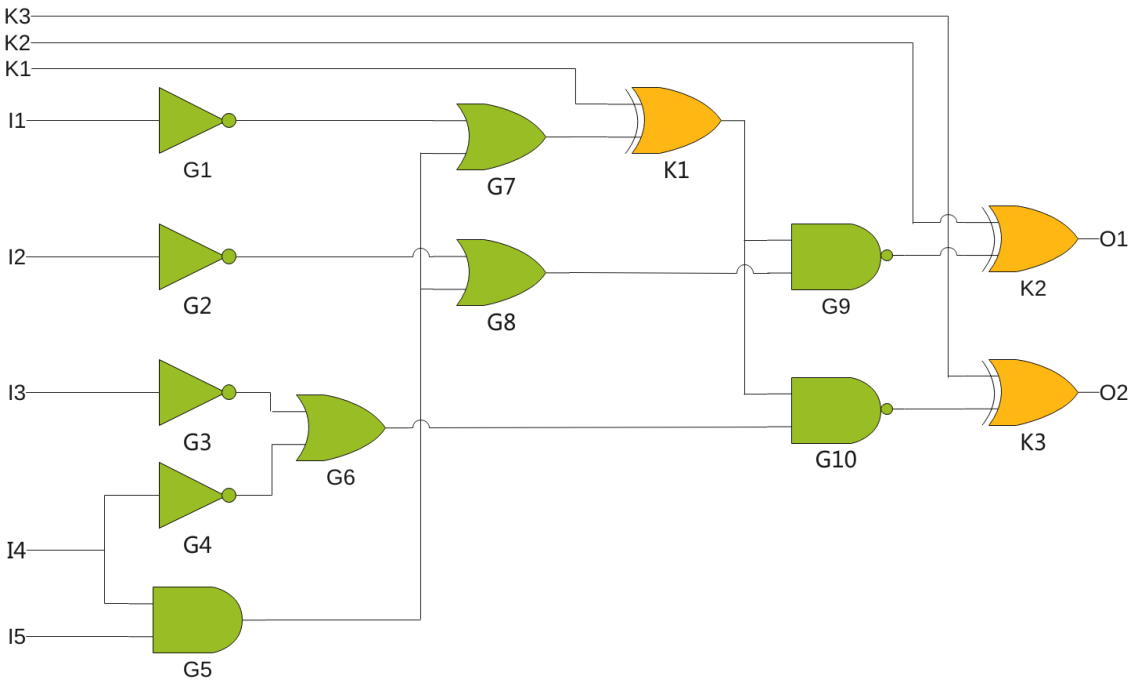



Figure 2.2: Improper placement of key gates resulting in a masked incorrect key.

The circuit in Figure 2.2 has been locked with the insertion of 3 key gates. The gates are configured in a way that produces the masking effect for a certain range of inputs. For an input pattern of 00000 and correct key input of 000, the expected outputs O1 and O2 should be 00. For an incorrect key input of 111, the effect introduced by gate K1 is reversed by the effects created by gates K2 and K3. Consequently, the correct

outputs 00 are produced with incorrect keys.

The fault analysis based insertion algorithm demonstrated in [9] was proposed in 2015 to help mitigate against the insertion issues discussed with random insertion. The insertion algorithm uses fault propagation analysis to determine where key gates are most likely to change outputs if an incorrect key is applied. This probability known as the fault impact and is determined by the number of faults detected at the output of a gate for a set of test input patterns. The insertion algorithm then inserts key gates into the locations with the highest fault impact, choosing randomly whether it is an XOR or XNOR gate.

Fault analysis based insertion has been proven to be more effective than random insertion in changing outputs if a wrong key is applied. The algorithm targets a 50% Hamming distance between correct outputs and incorrect outputs when a wrong key is applied, providing the maximum ambiguity to an attacker. The insertion algorithm was tested on the ISCAS benchmark suite. Random insertion was not able to obtain a 50% Hamming distance on any of the benchmark circuits. Fault analysis based insertion was able to obtain a 50% Hamming distance on all benchmarks except for C5315 and C7552. It was also determined that fault analysis based insertion requires less gates to obtain the 50% metric leading to less overhead in the locked design.

### 2.1.3 Boolean Satisfiability (SAT) Attack

Traditional insertion techniques for combinational logic locking such as random insertion and fault analysis based insertion are shown in [10] [11] to be vulnerable to Boolean Satisfiability or SAT attacks. The SAT attack extracts the secret key by iteratively ruling out incorrect key values using distinguishing input patterns (DIP)s. A DIP is defined by a input value for which at least two different key values produce differing outputs. Since the outputs are different for the key values, at least one or both of key values are incorrect. The SAT attack can rule out multiple incorrect key values per iteration, reducing the key search space even more for the next iteration.

The DIPs for a locked netlist are discovered by using two copies of the locked circuit, one being a functional IC with the correct key embedded inside. The primary inputs of the two circuits are kept the same, while the key value for the non-functioning circuit is left independent. The outputs of the two circuits are XORed then ORed together, creating a diff signal that evaluates to 1 if any of the outputs of the two circuits are different. The combined circuit is then passed to a SAT solver that finds a DIP for which the diff signal is on. This DIP is then applied to the functional IC to obtain the correct output which is then used to identify incorrect keys. The process of finding a DIP and ruling out incorrect keys is then repeated and continues until no more DIPs are found, meaning that all incorrect key values have been found.

### 2.1.4 SAT Attack Resistant Logic Locking

The worst case scenario for a SAT attack is when a DIP can only rule out one incorrect key per iteration, effectively making the SAT attack use brute force to find the secret key. Depending on the key size for the locked netlist, this can make the SAT attack computationally unfeasible. The truth table in Table 2.2 represents a locked netlist that ensures the worst case for the SAT attack. The netlist has 3 primary inputs and 3 key inputs resulting in 8 different key combinations. The output Y for the netlist is shown for every input pattern and key combination. For each input pattern there is at most one incorrect key combination that produces an incorrect output. With this configuration, the SAT attack can only rule out one incorrect key combination per iteration and the number of DIPs for the algorithm to find increase exponentially with key size.

Motivated by the example in Table 2.2, SAT Attack Resistant Locking or SARLock proposed in [12] is a logic locking technique that ensures that the SAT attack can rule out only one incorrect key combination per iteration. The proposed architecture shown in Figure 2.3 uses a comparator circuit that asserts a flip signal for a specific input and key value. The flip signal is then XORed with an output of the circuit and

Table 2.2: Truth table representing worst case scenario for SAT attack.

| | | | | | Output Y for different key values | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | A | B | C | Y | k0 | k1 | k2 | k3 | k4 | k5 | k6 | k7 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

corrupts the output if asserted. The mask logic is inserted to prevent a correct key from asserting the flip signal.

The comparator circuit logic has primary inputs $IN$ and key inputs $K$, and a single bit output $flip$. The comparator logic block is represented by the Boolean function $flip = F(IN, K)$. The Boolean function is a one-point function and for each incorrect key guess to the comparator logic, the flip signal is different at only one input with respect to the correct key. The resulting architecture achieves the desired SAT resistance and overhead grows linearly with key size, while the number of DIPs grows exponentially.

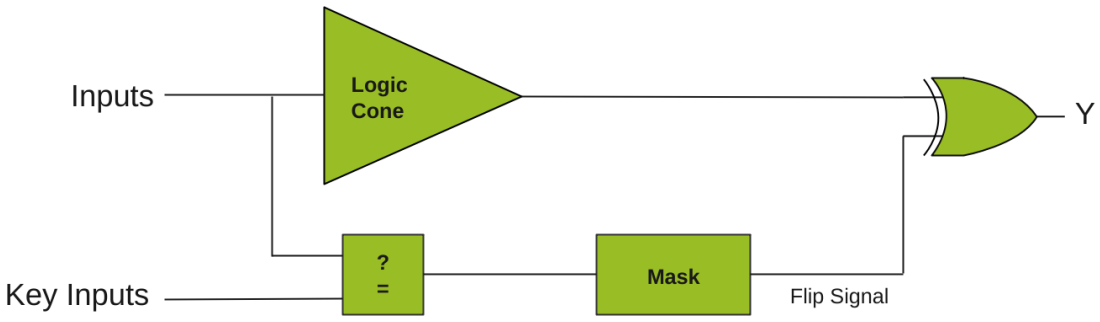



Figure 2.3: SAT attack resistant circuit diagram.

SARLock insertion successfully provides SAT resistance but does not protect against other attacks on logic locking for example, a removal attack. A removal attack is where an attacker identifies the logic that corresponds to the locking circuitry and removes

it from the netlist [13]. Since SARLock is isolated outside of the original netlist, it can be easier for the attacker to isolate and remove it. SARLock's defense against the SAT attack involves selectively flipping output bits resulting in a small Hamming distance for incorrect keys. To provide maximum security, the authors propose using two layer logic locking which combines SARLock with a traditional insertion technique such as fault analysis based insertion. Combining these two techniques offers maximum security with resistance against the SAT attack and provides a 50% Hamming distance across outputs with an incorrect key.

## 2.2 Dynamic Partial Reconfiguration

Dynamic Partial Reconfiguration (DPR) is a technology offered by many FPGA architectures to modify sections of implemented logic during the runtime of the FPGA while the remaining logic continues to operate without interruption. A dynamic partial reconfigurable design consists of non-reconfigurable logic known as static logic, a reconfigurable area in the FPGA hardware known as a reconfigurable partition, and reconfigurable modules to be placed into the reconfigurable partitions [14]. Figure 2.4 illustrates a top-level diagram of a DPR design consisting of three reconfigurable partitions each with their own set of reconfigurable modules.

The dynamic partial reconfiguration technology provides more system flexibility for application developers to take advantage of. An example is a system acting as a communication hub may support many different functions to operate on the data. Each time a new function is needed, the communication link must be temporarily closed to reconfigure the FPGA fabric for the new function. The same system enabled with DPR, can keep the links of communication active and partially reconfigure the PL fabric to support the new function. Another example is an application might need to support a variety of functions that cannot fit all on the FPGA at once. DPR enables a system where time-multiplexing can be used to cycle through different functions, reducing the overall size of the logic. An application developer can use

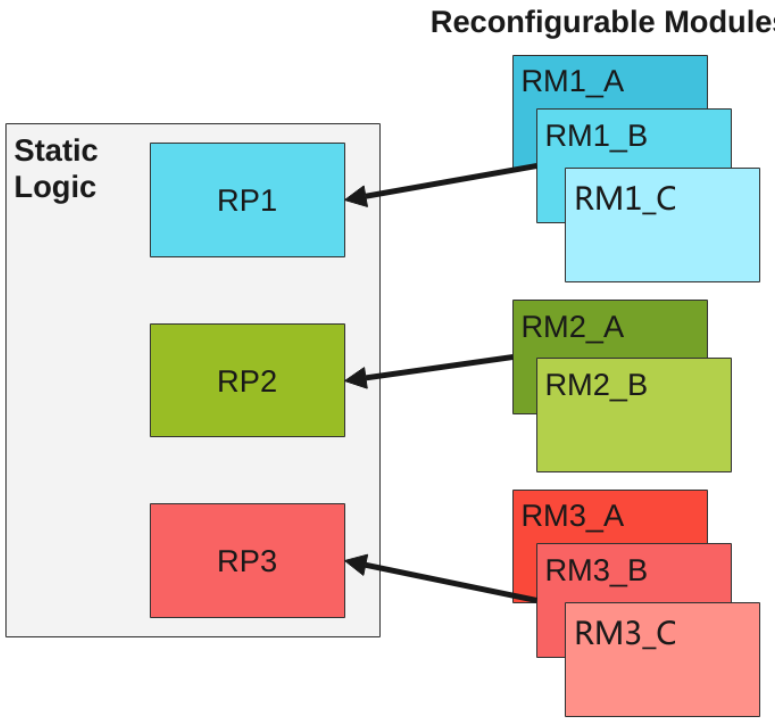



Figure 2.4: Basic DPR architecture with static logic, reconfigurable partitions, and reconfigurable modules for each partition.

DPR to also reduce power requirements by partially swap out power-hungry tasks when not needed.

### 2.2.1 Reconfigurable Partitions

Many FPGA families today have support for dynamic partial reconfiguration, although this work focuses on Xilinx's ZYNQ-7000 SoC architecture. The ZYNQ SoC integrates a dual-core Arm Cortex-A9 processor based processing system (PS) and an FPGA known as the programmable logic (PL). The architecture of the PL can be conceptually broken down into two layers, the configuration memory layer and hardware layer [15]. The hardware layer consists of the computational hardware resources such as lookup tables (LUTs), flip-flops, digital signal processors, and memory resources. The configuration memory layer is composed of SRAM and stores all of the information that determines initial states and configuration of the hardware resources

as well as the routing information. The data programmed into the configuration is known as a bitstream.

The ZYNQ architecture provides the ability of partial reconfiguration by allowing applications to download partial bitstreams to certain areas of configuration memory. The areas of configuration memory directly corresponds to the areas of the PL defined as reconfigurable partitions. Reconfigural partitions are defined with Partition Pins which are interfaces between static and reconfigurable logic. The pins are available within the reconfigurable partition but are implemented as part of the static logic, meaning that they must stay the same between configurations.

### 2.2.2 Reconfigurable Modules

The partial bitstreams downloaded to the reconfigurable partitions contain the information necessary to implement the logic of reconfigurable modules. A reconfigurable module is the logic that occupies the reconfigurable partition of configuration memory. There can be many reconfigurable modules per reconfigurable partition. I/O interfaces must be consistent throughout the reconfigurable modules to support the static partition pin interface.

### 2.2.3 ICAP and PCAP Interface

The ZYNQ architecture allows for modification of PL configuration memory through mainly two configuration paths [16]. The Internal Configuration Access Port (ICAP) provides an interface for reconfiguration by the PL. Xilinx provides the AXI HWICAP IP that is instantiated in PL designs for reconfiguration. The IP uses an AXI slave interface to take data from an application in the PL to write to the ICAP interface, which then program configuration memory the PL. The interface is usually handled by a MicroBlaze processor instantiated in the PL.

The PS includes an interface for PL device control called the Device Configuration Interface (DevC). The DevC interface includes three modules, one to access the PLs

analog to digital converters (XADC), manage PL security, and access to the AXI-PCAP interface. The Processor Configuration Access Port (PCAP) interface is used to download full or partial bitstreams to the PLs configuration memory. The PCAP interface contains an AXI DMA engine to move bitstreams from memory, mainly on chip memory, DDR, or addressable flash devices, to the PL configuration module. The DevC has a slave AXI interface connected to the PS AXI interconnect, making it accessible to an AXI master in the PS.

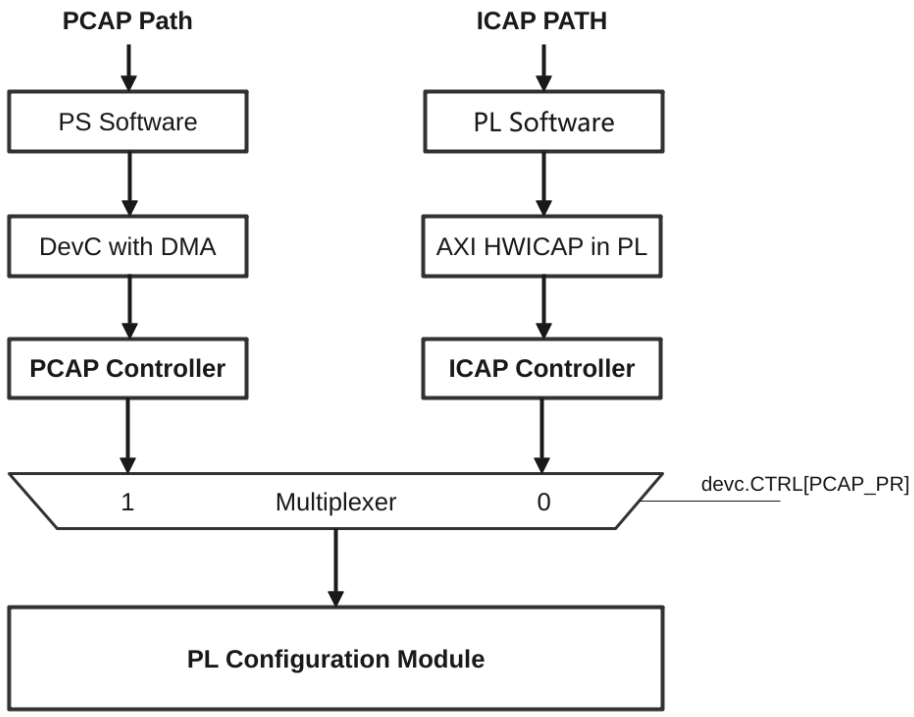



Figure 2.5: Data paths diagram for reconfiguration of PL configuration memory.

ICAP and PCAP interfaces can never be simultaneously active. Switching between them is possible through the programming of a register in the DevC. This register sets the select bit for a multiplexer to choose the interface to the PL configuration module. The data path diagram for the two paths is shown in Figure 2.5.

### 2.2.4 Partial Bistream Generation

Xilinx supports generation of full and partial bitstreams for partial configuration applications through the Vivado Design Suite using a mix of the Graphical User Interface (GUI) and TCL command line flow. The flow uses "bottom-up synthesis" meaning the static logic and reconfigurable modules are synthesized separately [17]. The reconfigurable logic modules must be instantiated in the top level for I/O purposes but are treated as black boxes for the synthesis. Synthesis results for each module, static and reconfigurable, are then written to a checkpoint file for later use. Physical areas of the device are then defined as reconfigurable partitions using the constraints file. A complete configuration with static logic and one reconfigurable module for each reconfigurable partition must go through implementation to generate a checkpoint for a fully routed design. A checkpoint is also generated for the routing of the static only logic. The static logic checkpoint is then used to generate routing the rest of the possible configurations of the design. Once all configurations are implemented and routed, a full bitstream with an initial configuration and partial bitstreams for the reconfigurable modules can be generated.

## 2.3 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a processing environment that guarantees secure process isolation and the authenticity of the executed code. Isolated execution for a TEE is commonly ensured through the implementation of a separation kernel [18]. The separation kernel enables the co-execution of two systems that have different security requirements on the same platform. The separation kernel does this by dividing system resources into several partitions, providing strict isolation between them. The separation kernel only allows interaction between partitions through a carefully controlled interface. The main security criteria of the isolation is that it should ensure that data within one partition should not be able to be

read or modified by other partitions, partitions cannot leak sensitive information to shared resources, and partitions cannot communicate unless permitted. To support the implementation of TEEs and separation kernels, processor architectures have introduced hardware extensions to provide isolation at the hardware level. Examples of commercial hardware based isolation schemes are ARM TrustZone and Intel SGX.

### 2.3.1 Intel SGX

Intel Software Guard Extensions (SGX) is a set of security extensions added to the Intel architecture that guarantees integrity and confidentiality for security-sensitive applications even if privileged software is potentially malicious. The fundamental idea of SGX is a protected environment containing security-sensitive code and data known as an enclave [19]. SGX security extensions provide the ability to have multiple enclaves and ensures isolation of each enclave from untrusted or malicious software outside of the enclave. The security extensions also allows software running inside enclaves to use an attestation scheme to allow local or remote parties to verify its authenticity.

Enclaves store their code and data in Processor Reserved Memory (PRM), a subset of RAM in the system that cannot be directly access by other software including system software. The CPUs memory controllers also block DMA transfers to the PRM further protecting the region from peripherals. If any non-enclave software attempts to access a virtual address that translates to a physical address in the PRM, the processor returns an abort page that is filled with all ones. SGX also provides isolation between enclaves, so if code running inside an enclave tries to access another enclaves memory in the PRM, a page fault exception is raised.

System software such as an OS kernel is responsible for the creation and teardown of enclaves through the use of the SGX instructions. Enclaves are created with the ECREATE instruction which finds free pages in the PRM and puts the enclave into an uninitialized state. The EADD instruction is used to load data into the enclave

but validates its inputs before modifying the enclave. EEXTEND is then used to measure the data loaded into the enclave if needed. The EINIT instruction is used to mark the enclave as initialized but is still not in the running state. EENTER and ERESUME are instructions that start or restart the execution of an initialized enclave. To teardown the enclave the EEXIT instruction must be used to move the enclave back to a non-running state. Once the enclave is not running the EREMOVE instruction is then used to completely free the enclave.

### 2.3.2 ARM TrustZone

ARM TrustZone is a hardware enforced isolation architecture supported by ARM Cortex-A application processors and Cortex-M micro-controllers. TrustZone divides the SoC into two domains, the secure world and the normal or non-secure world [20]. It gives the designer the ability to isolate subsets of hardware like peripherals, memory regions, areas of L2 cache, or even complete processors from the full system on chip (SoC) for the normal world. Software running on the SoC can be executed in either the secure or normal states.

To accomplish this separation, ARM introduced the Secure Configuration Register (SCR) that specifies the security state of the processor, what mode the processor is in (interrupt request, exception handling), and whether the normal world can disable interrupts and asynchronous aborts in the Current Program Status Register (CPSR). ARM SoCs provided TrustZone configuration registers that designates hardware items as secure or non-secure. Software running in the secure state defined in the SCR has full access to the entire SoC but software running in the non-secure state is isolated to the hardware defined as non-secure hardware. If non-secure software attempts to access a secure area of hardware, execution is halted and an exception is raised.

The software stack for a Cortex-A TrustZone enabled system is shown in Figure 2.6. The normal world consists of a rich operating system (OS) like the Linux kernel which is assumed to be flawed from a security perspective with its non-secure applications

running on top of it. Software running in the secure state should be a small and minimal OS to avoid the possibility of errors and exploits. Software running at the highest privilege level known as the secure monitor should provide the ability to handle exceptions from the normal world and provides mechanisms for secure context switching between the two worlds. Context switching for ARM TrustZone is very lightweight and efficient, only taking a few clock cycles to expose the full set of hardware to trusted software or limiting the hardware to non-trusted software.

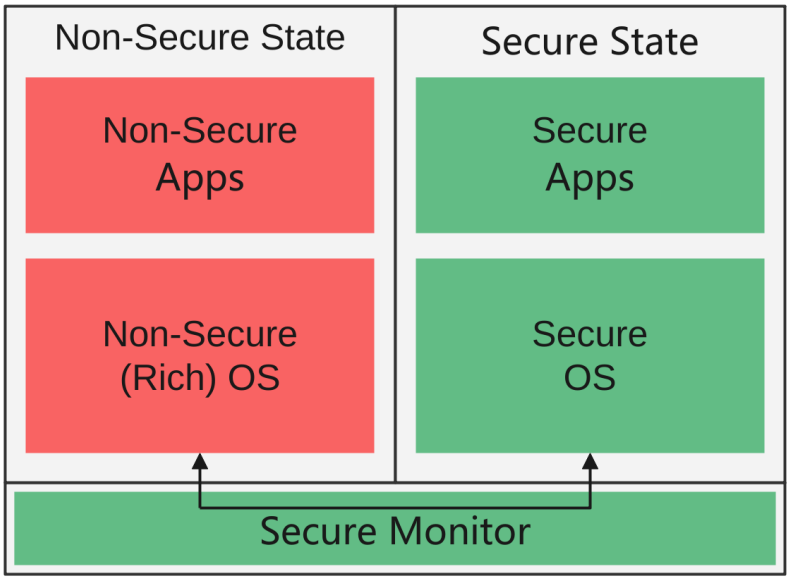



Figure 2.6: TrustZone software architecture for ARM Cortex-A processors.

There have been open source TEEs that have been developed to use ARM TrustZone and execute in the secure state. Open Portable Trusted Execution Environment (OP-TEE) is a TEE that follows the GlobalPlatform API standard for TEEs and is ported to many ARM boards today. The GlobalPlatform API describes and defines how a rich OS should communicate with a TEE. Another TEE is Nagoya (Japan) University's TOPPERS-SafeG which is a secure dual OS monitor. The purpose of the monitor is to establish a trusted and non-trusted state for two separate operating systems to be executed in. The monitor also facilitates secure communication between the two different operating systems through system calls to the monitor. The

trusted state executes a small RTOS or baremetal OS where the non-trusted state executes a larger general purpose OS like Linux or Android. Example commercial implementations of ARM TrustZone today are applications such as Samsung KNOX and Android's Keystore.

# CHAPTER 3: PROPOSED SCHEME AND EXPERIMENTAL SETUP

We propose a novel logic locking scheme enabled with Dynamic Partial Reconfiguration (DPR) to provide updates to the IPs key value during the runtime of the IP. We present an automated framework for logic locked IP creation and a design for secure integration of the locked IP on a FPGA SoC platforms. The FPGA SoC platform consists of an ARM based Processing System (PS) integrated with FPGA resources, the Programmable Logic (PL). The PS is connected to the PL through the Advanced eXtensible Interface (AXI) interconnect for communication. The ARM processor of the SoC is enabled with ARM TrustZone security extensions to facilitate the secure deployment, management, and updates of the secret key for the locked IP.

## 3.1 Logic Locking Framework

The proposed framework provides an automated tool for state of the art logic locking insertion schemes [21]. The framework uses a Python script to automate the use of low level tools and provides a simple Graphical User Interface (GUI) shown in Figure 3.1 to control the flow. The flow takes inputs of RTL Verilog files and produces logic locked Verilog files with logic inserted at the gate level. A step-by-step procedure of the framework flow for fault analysis insertion is given below.

The framework is given an input Verilog file containing a description of the IPs logic at the RTL or at the gate level. The key insertion scheme requires insertion of key logic at the gate level, so the RTL is synthesized using the Synopsis Design Compiler. The compiler optimizes the RTL logic of the IP and converts the Verilog to the gate level netlist. The new gate level Verilog file is analyzed to determine key insertions following SARLock, fault analysis insertion, and random key insertions. The netlist

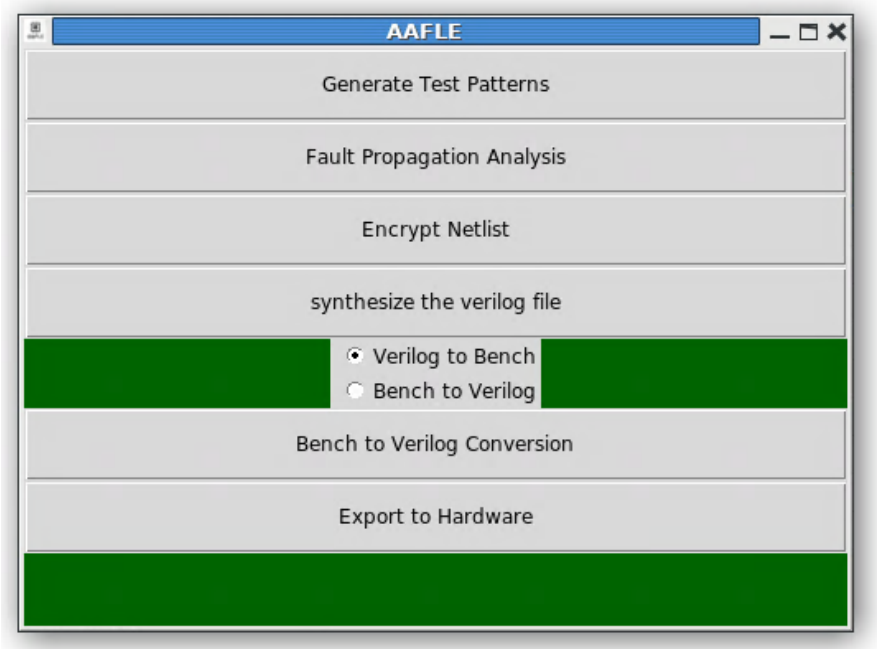



Figure 3.1: Logic locking framework GUI.

is analyzed for testable faults of the given netlist for fault analysis based insertion using ATPG generated input patterns and fault impact of each node is recorded.

The Verilog file is converted to the Berkeley bench format using the ABC tool. The bench format is a gate level description used for gate insertion. The output of the fault simulator is used to systematically insert key gates into netlist with the highest fault impact, randomly chooses whether the gate is XOR or XNOR. The netlist bench file with the key gates inserted is then converted back to Verilog and given to the user.

The framework supports automatic integration of the locked IP with the PS and bitstream generation with Vivado. The locked IP is integrated with a slave AXI port that provides key inputs. Vivado TCL command line is used to connect the AXI interface with the PS, place inputs and outputs of the IP in the PL, and generate a bitstream of the resulting hardware design. Once the generated bitstream is programmed to the PL, secure interface of the PS writes the key value to the memory-mapped AXI nterface to unlock the IP.

## 3.2 Proposed Scheme

### 3.2.1 Partially Reconfigurable Logic Locking

Obfuscation of the IP can be done using combinational logic locking techniques such as random insertion, fault analysis based insertion, or SARLock. The proposed scheme places the additional key logic into reconfigurable partitions of the FPGA. The logic locking technique allows for changes to the key value through partial reconfiguration of key logic, while the rest of the primary logic is unaffected. Figure 3.2 shows the proposed logic locking scheme with the c17 ISCAS benchmark circuit. Reconfigurable partitions, highlighted with the red boxes, have a key input and hold reconfigurable modules that act as the locking logic.

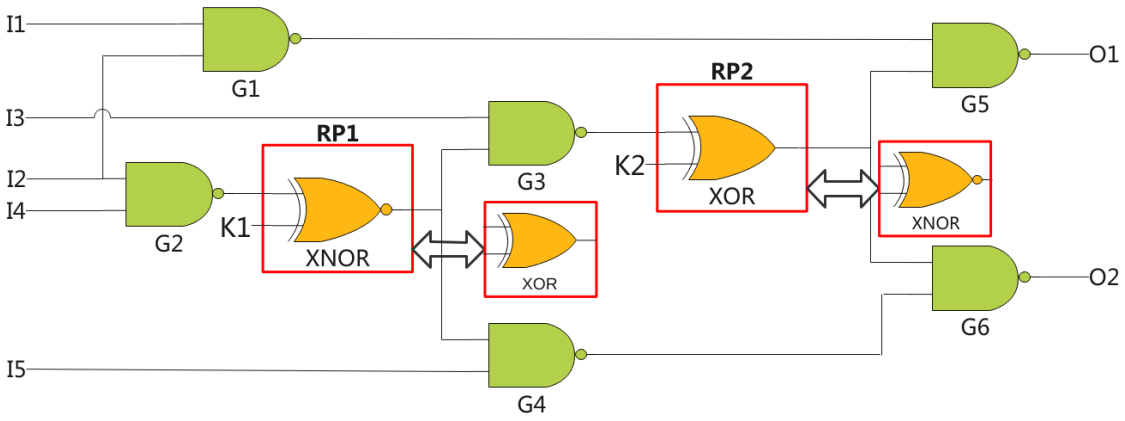



Figure 3.2: Example ISCAS benchmark c17 logic locked with key gates placed in reconfigurable partitions.

To insert gates, the chosen nets are rerouted to an input of a reconfigurable partition. The reconfigurable partition has one output that is routed to the nets original connection. If the key input to the reconfigurable partition is correct, the net is unaffected and the netlist behaves as intended. If the key is wrong, the reconfigurable partition flips the nets value, potentially corrupting the output of netlist. The example in Figure 3.2 uses XOR/XNOR gates as the reconfigurable modules to provide different configurations of logic locked design with different key values. Partial bit-

streams for XOR and XNOR gates are generated for each reconfigurable partition in the design. Each partial bit stream is specific to the reconfigurable partition it was generated for. Key values are configured through downloading a partial bitstream to the reconfigurable partitions of the configuration memory.

### 3.2.2 IP Integration on FPGA SoC Platform

The following section discusses a system design to securely integrate the 3PIPs on an FPGA SoC platform. IPs are passed through the automated logic locking framework for key insertion at the gate level. Key logic is extracted from the locked netlist and inserted into reconfigurable partitions of the PL. An AXI slave port with multiple data registers is instantiated along with the IP. The registers from the AXI interfaces are used to store primary inputs and outputs of the locked IP and to provide key inputs to all reconfigurable partitions. The resulting block diagram of the integrated IP is shown in Figure 3.3.

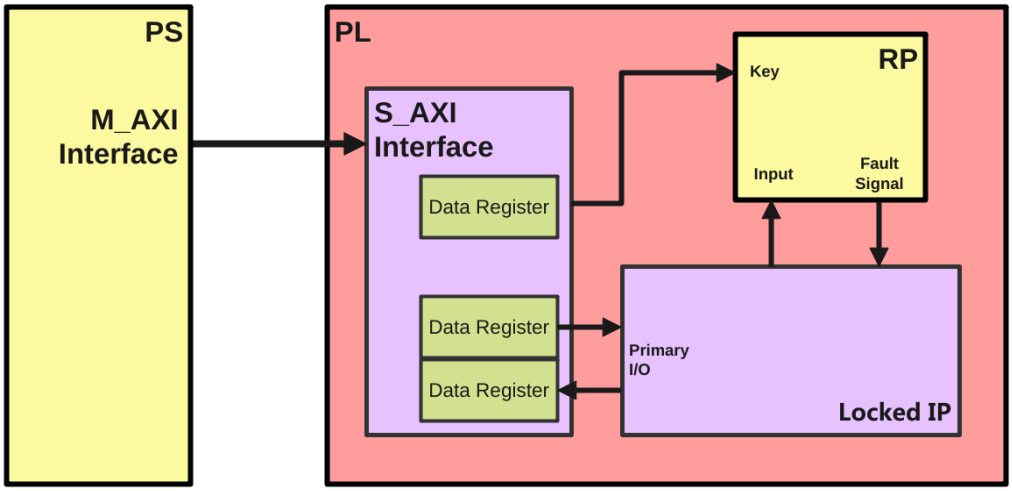



Figure 3.3: Block level architecture of secure 3PIP placed in the PL.

The AXI interface of the IP is connected to the AXI interconnect between the PS and PL, allowing the IP to communicate with an AXI master in the PS. The connection allows PS read and write access to the registers located inside the AXI interface. The PS is responsible for setting primary and key inputs and monitoring outputs.

The AXI HWICAP IP is not included as part of the bitstream, so reconfiguration of the reconfigurable partitions must come from the PCAP.

To generate random configurations of the reconfigurable design, a Fibonacci Linear Feedback Shift Register (LFSR) is implemented in the PL. The Fibonacci LFSR uses a shift register that is initially set with a seed value. Some of the bits in the register, the "taps", are XORed together to produce a sequence of output bits. The output is determined by the current state of the LFSR, and because there are finite states, the output sequence is on a cycle. A well chosen feedback function can produce a sequence that is seemingly random and has a very long cycle. The LFSR is built using the register and a few XOR gates, so it provides a lightweight means of psuedo-random number generation.

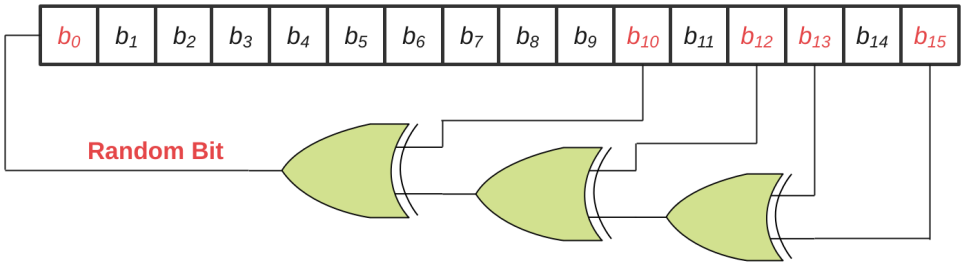



Figure 3.4: LFSR with 4 "taps" resulting in the addition of 3 XOR gates.

A 16 bit Fibonacci LFSR design with generator polynomial $x^{16}+x^{14}+x^{13}+x^{11}+1$, is show in Figure 3.4. For connection to the PS, the LFSR is implemented with its own AXI slave interface. The AXI interface logic contains data registers accessible from the PS. The data registers are used to enable the LFSR to start its execution and to store the current state of the LFSR.

### 3.2.3 Secure Key Reconfiguration and Provisioning

The system design uses the ARM based PS of the SoC to control the deployment and reconfiguration of the secret key to the IP. The PS is enabled with ARM TrustZone security extensions to facilitate a hardware based TEE and to isolate hardware

for PL control. The key is only ever handled by software executing in the secure domain. The TrustZone features are also used to isolate the PS connection to the PL from any software executing in the non-secure domain.

Deployment of the secret key to the IP in the PL is done through a master AXI port in the PS connected to the AXI interconnect. The locked IP has a slave AXI port connected to the system AXI Interconnect, in order to receive data from the PS. The interface for access the PCAP controller also has a slave connection to the AXI Interconnect able to accept request from the PS master. The master AXI port is accessible to the PS through a memory mapped interface. The PS accesses the data registers in the slave through read or writes to the slaves AXI address space.

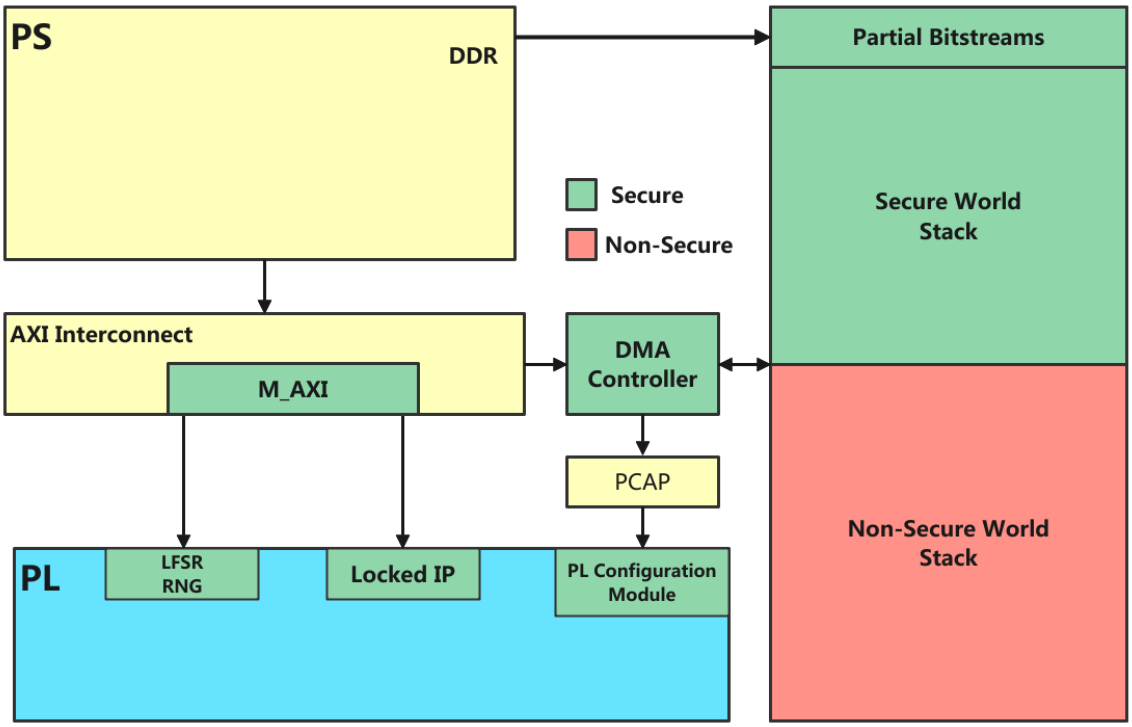



Figure 3.5: TrustZone configuration of hardware resources on SoC for 3PIP Integration.

TrustZone is used to configure areas of DDR memory and PS master AXI interfaces as secure hardware shown in Figure 3.5. Any request made to the master AXI port by software running in non-secure world is guaranteed by hardware to be ignored and not

propagated to the AXI interconnect. This prohibits non-secure software from having any communication with the PCAP interface as well as the IPs in the PL. Partial bitstreams for the XOR/XNOR reconfigurable modules are stored in secure areas of DDR memory. Any non-secure software attempting to access these areas of DDR sets off an exception to be handled by the secure monitor, where the access is denied. The DMA engine response from moving partial bitstreams from DDR to the PCAP is capable of securing DMA channels under TrustZone. If a secure master makes a request, a secure channel is established and the destination must be a secure area of the device. If it is not, the DMA channel creates an external abort to be handled by the secure monitor. The system architecture allows for continuous execution of a rich OS that may be flawed from a security perspective. The hardware backed TrustZone extensions ensure protection of the IP hardware from exploits in the rich OS.

## 3.3 Experimental Setup

The secure system design for integration of the locked IP on a FPGA SoC platform is implemented on a Avnet Zedboard FPGA development board with a Xilinx ZYNQ SoC. The SoC has a PS consisting of dual core ARM Cortex-A9 processors, on-chip memory, a set of I/O peripherals and Xilinx FPGA fabric in the PL.

### 3.3.1 Gate Insertion

The automated framework is used to create an obfuscated RTL IP for integration of the FPGA platform. The framework is tested on the ISCAS benchmark circuits c17, c432, and c499 using fault analysis based key insertion at the gate level. Verilog test benches were created for each benchmark to compare correct outputs to outputs with an incorrect given for Hamming distance analysis. Figure 3.6 shows the original RTL description and the RTL with key gates inserted.

```
module c17 (
    G1, G2, G3, G6, G7,
    G22, G23
    );

    input G1, G2, G3, G6, G7;
    output G22, G23;

    wire G10, G11, G16, G19;

    assign G10 =  ~G1 | ~G3;
    assign G11 =  ~G3 | ~G6;
    assign G16 =  ~G2 | ~G11;
    assign G19 =  ~G11 | ~G7;
    assign G22 =  ~G10 | ~G16;
    assign G23 =  ~G16 | ~G19;

endmodule
```

(a) Original c17 RTL

```
module c17_3keys  (
    K1, K2, K3,
    G1, G2, G3, G6, G7,
    G22, G23
    );

  input  K1, K2, K3, G1, G2, G3, G6, G7;
  output G22, G23;

  wire key0, key1, key2, G10, G11, G16, G19;

  assign key0 = G3 ^ K1;
  assign key1 = G16 ^ K2;
  assign key2 = G11 ^ K3;

  assign G10 = ~G1 | ~key0;
  assign G11 = ~key0 | ~G6;
  assign G16 = ~G2 | ~key2;
  assign G19 = ~key2 | ~G7;
  assign G22 = ~G10 | ~key1;
  assign G23 = ~key1 | ~G19;

endmodule
```

(b) Logic Locked c17 RTL

Figure 3.6: Logic locking framework tested on ISCAS C17 Verilog RTL files using fault analysis based insertion with 3 keys.

### 3.3.2 Hardware Design

The c17 benchmark locked with 3 keys was chosen for implementation on the platform. For bottom-up synthesis of the design, the reconfigurable modules, in this case XOR/XNOR key gates, are first synthesized separately. The Vivado TCL flow is used to synthesize Verilog descriptions of the XOR and XNOR gates and design checkpoints are saved for later use. The synthesis is executed in out of context mode meaning I/O insertion is prevented. The synthesizer also is passed an option to rebuild the hierarchy of the RTL design after synthesis.

```
## Synthesis of XOR gate ##
read_verilog xor.vhd
synth_design -mode out_of_context -flatten_hierarchy rebuilt -top xor
write_checkpoint -force xor.dcp
close_design
## Synthesis of XNOR gate ##
```

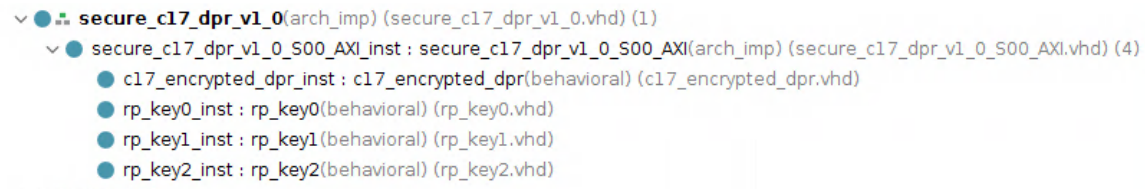
```
read_verilog xnor.vhd
synth_design -mode out_of_context -flatten_hierarchy rebuilt -top xnor
write_checkpoint -force xnor.dcp
close_design
```

For creation of the static logic of the design, the locked IP is instantiated in a top level RTL file that includes an instantiation of a slave AXI interface. Key logic is extracted from the netlist and instead routed to three reconfigurable modules, one for each key gate. The data registers of the AXI interface are tied to inputs and outputs of the IP and key inputs to the reconfigurable modules. For synthesis of the static logic, reconfigurable modules of the design are defined as "black boxes" with only input and output ports defined. The resulting VHDL description of the reconfigurable modules and file hierarchy of the locked IP is shown in Figure 3.7. The black box RTL has port definitions of 2 inputs and 1 output with no behavioral description so it can instead be filled with either an XOR or XNOR gate. The file hierarchy shows both the locked c17 IP and the 3 reconfigurable modules are instantiated alongside of an AXI slave interface.

```
entity rp_key0 is
    port (
    y : out std_logic;
    a : in  std_logic;
    b : in std_logic
   );
end rp_key0;

--empty black box for static checkpoint generation
architecture behavioral of rp_key0 is
begin
end behavioral;
```

(a) Black Box RTL for RMs




(b) File hierarchy of locked IP

Figure 3.7: Creation of static logic of IP for synthesis.

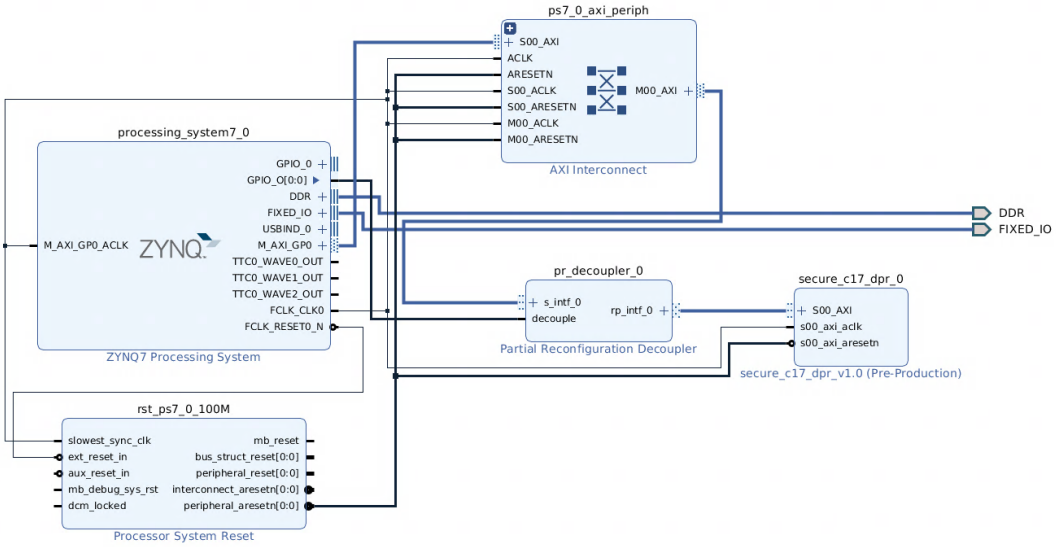



Figure 3.8: Block design of secure IP connected to ZYNQ PS.

The resulting locked IP is instantiated in a Vivado block design and connected to the AXI Interconnect and the ZYNQ PS in the block design editor shown in Figure 3.8. The Xilinx Partial Reconfiguration Decoupler IP is placed between the AXI Interconnect and locked IP to ensure the safety of the interface while partial reconfiguration is occurring. Vivado is used to create an HDL wrapper for the block design. The HDL wrapper containing all of the static logic is then synthesized and a design checkpoint is saved.

The static logic synthesis checkpoint is loaded and the black boxes in the netlist of the static design are selected. Each black box is loaded with the design checkpoints of the synthesized XOR/XNOR reconfigurable modules. The following TCL commands create a configuration of the locked IP with key 010 and marks the key modules as reconfigurable.

```
read_checkpoint -cell rp_key0_inst xor2.dcp
read_checkpoint -cell rp_key1_inst xnor2.dcp
read_checkpoint -cell rp_key2_inst xor2.dcp
set_property HD.RECONFIGURABLE 1 [get_cells rp_key0_inst]
```

```
set_property HD.RECONFIGURABLE 1 [get_cells rp_key1_inst]
set_property HD.RECONFIGURABLE 1 [get_cells rp_key2_inst]
```

The nets marked as reconfigurable must be placed into reconfigurable partitions before implementation. The Vivado GUI is used to draw partially reconfigurable blocks (pblocks) of FPGA resources. The reconfigurable XOR and XNOR gates only use 1 LUT of FPGA resources, so no DSPS or block RAM need to be highlighted in the pblock. Pblocks containing just FPGA slices are drawn for all 3 key gates and linked with the key logic nets. Partial reconfiguration DRC checks are ran on the created floorplan to check for violations. Once validated, the floorplan is saved to the constraints file for the implementation phase.

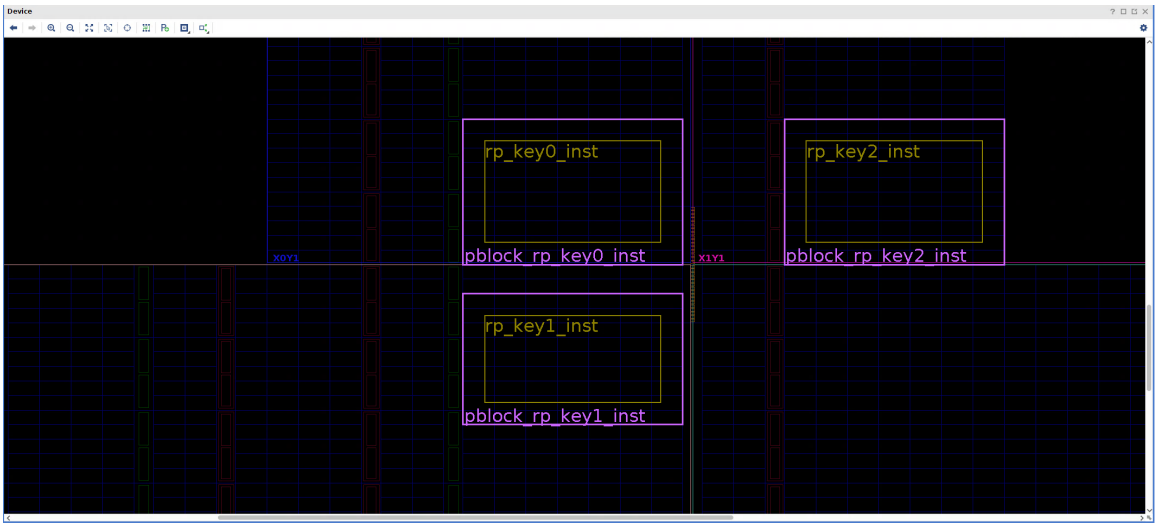



Figure 3.9: Defining pblocks for each key gate.

The implementation phase is responsible for optimizing the logic, placing it in the FPGA fabric, and the routing of the signals. During implementation the constraints file created from pblock floor planning is read to know where to place pblocks in the fabric and what logic is associated with them. After the implementation, the placement and routing for the first configuration of the locked c17 is complete. Design checkpoints are saved for the full routed design and for each reconfigurable partition. The routed static logic design checkpoint is now generated to be used in other config-

urations. The reconfigurable partition nets are updated to behave as black box logic, leaving only the routed static logic for the writing of the checkpoint.

```
#Save full and partial routed designs
write_checkpoint -force key_010_route.dcp
write_checkpoint -force -cell rp_key0_inst key_0_xor_route.dcp
write_checkpoint -force -cell rp_key1_inst key_1_xnor_route.dcp
write_checkpoint -force -cell rp_key2_inst key_2_xor_route.dcp
#Save static routing
update_design -cell rp_key0_inst -black_box
update_design -cell rp_key1_inst -black_box
update_design -cell rp_key2_inst -black_box
lock_design -level routing
write_checkpoint -force static_route_design.dcp
```

The static design checkpoint is loaded to create new configurations of the IP with different key values. A new configuration is made by reading in the reconfigurable module synthesis design checkpoints into the nets assigned to reconfigurable partitions. The new configuration is ran through the implementation phase and checkpoints are saved for the full configuration and reconfigurable modules in the same manner as the first configuration. The process of loading the static routed design checkpoint and reading in reconfigurable synthesis checkpoints is repeated for every configuration needed.

```
# Configuration of key 010
open_checkpoint static_route_design.dcp
read_checkpoint -cell rp_key0_inst xnor2.dcp
read_checkpoint -cell rp_key1_inst xor2.dcp
read_checkpoint -cell rp_key2_inst xnor2.dcp
```

The pr_verify TCL command is given every routed design checkpoint from every configuration to ensure that static implementation and interfaces are consistent across all configurations. Bitstreams for the full configurations and their partial bitstreams for the reconfigurable modules are generated by reading in the configurations routed design checkpoint and executing write_bitstream. Both partial bitstreams for each reconfigurable partition are converted to BIN format to be programmed by the PS over PCAP at runtime.

### 3.3.3 Software Architecture

The ZYNQ platform provides an area of configuration registers to configure hardware as either secure or non-secure [22]. Table 3.1 provides a summary of the registers configured for the IPs secure integration on the SoC. These registers are only programmable by software in the secure world context.

Table 3.1: Summary of TrustZone configuration registers for ZYNQ SoC platform.

| **Register Name** | **Address** | **Width** | **Description** |
|---|---|---|---|
| TZ_DDR_RAM | 0xF8000430 | 32 | DDR RAM TrustZone Config |
| TZ_DMA_NS | 0xF8000440 | 32 | DMAC TrustZone Config |
| security_fssw_s0 | 0xF890001C | 1 | M_AXI_GP0 security setting |
| security_fssw_s1 | 0xF8900020 | 1 | M_AXI_GP1 security setting |
| security_apb_slaves | 0xE0200018 | 15 | APB slave security setting |

The TZ_DDR_RAM register configures incremental 64 MB sections of memory as secure or non-secure memory regions. Setting a particular bit to 0 indicates a secure memory region for that 64 MB segment. A 1 indicates a non-secure memory segment accessible by secure or non-secure software. The security state of the DMA controller is set by the last bit of the TZ_DMA_NS register. If the bit is 0 the DMA controller operates in the secure state meaning the controller has the capability of establish both secure and non-secure channels. The DMA controller operating in the non-secure state can only create non-secure channels and access non-secure addresses. When the DMA controller is in the secure state, security of the channel is determined

by the security state of the processor making the request. If the DMA controller is accessed by a non-secure processor, a non-secure channel is established and any access to secure regions is blocked.

The security_fssw_s0 and security_fssw_s1 registers control the security state of the two PS master AXI ports to the PL. Setting the registers to 0 means that any AXI request from a non-secure processor is not propagated to the PL logic. The AXI protocol provides the ability to transmit security status, allowing for the designation of secure IPs in the PL. Non-secure software using the AXI master port receives AXI errors when trying to access a secure slave in the PL. These secure AXI transactions have been shown to be vulnerable to malicious non-secure IPs connected to the same AXI interconnect [23]. The proposed mitigation is to separate the PL into secure and non-secure IPs, connecting secure IPs to an AXI master designated as secure hardware in the PS. The other AXI master is designated as non-secure hardware to allow access from both secure and non-secure software. The security_apb_slaves register is set to set the UART hardware as non-secure to allow access from both worlds.

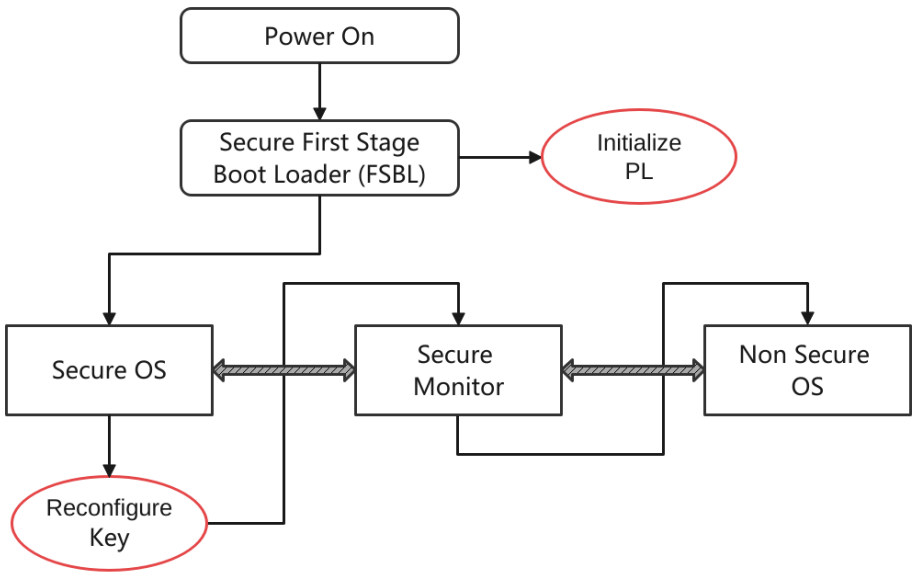



Figure 3.10: Boot flow diagram of TrustZone application.

The boot flow of the integrated system, shown in Figure 3.10 is as follows. Upon boot, the First Stage Boot Loader (FSBL) is copied from a boot device into on chip memory to begin execution. The purpose of the FSBL is to initialize the memory and hardware of the SoC as well as program the PL with the PL bitstream located on the boot device. The initial bitstream ignores key inputs and places NOT gates into reconfigurable partitions to ensure the IP is dysfunctional until it is configured. The FSBL finishes execution and releases control to start the execution of the secure world application. The secure application reconfigures IP and programs the key value over the secure AXI interface. The TrustZone configuration is then initialized and the secure monitor is established.

The secure world application relinquishes its control by executing a Secure Monitor Call (SMC) instruction. This instruction raises an exception to be handled by the monitor. The monitor handles the exception by executing a series of steps. The monitor determines which world the call has come from and saves its current register context. The monitor restores the other worlds context, switches the processor security state, and returns from the exception. The non-secure application continues to operate with the fully functional IP placed in the PL with no ability to access the IP or any other sensitive areas of the SoC.

### 3.3.4 Reconfigurable IP Locking

Partial reconfiguration of the IP takes place within the secure world for trusted execution and access to the IP hardware. Partial bitstreams for each reconfigurable module are stored in areas of DDR marked for the secure world. Because the memory is marked as secure, DMA transactions to move the bitstream to the PL are transferred using a secure channel and the execution must be started by a processor in the secure state. The master AXI connection to locked IP is designated as secure hardware. AXI transactions to read and write from the IP are only propagated to the PL if a processor running in the secure state makes the request.

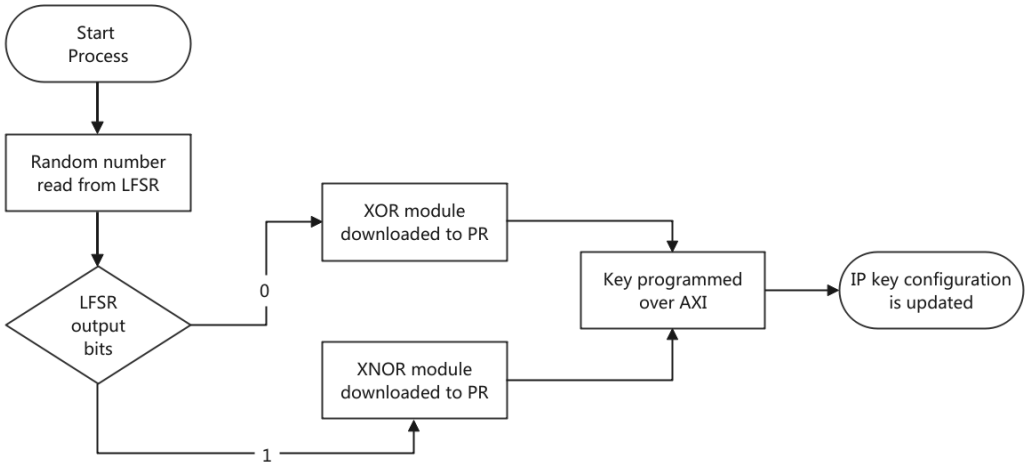



Figure 3.11: Reconfiguration flow of partially reconfigurable IP.

For reconfiguration of the locked IP, the secure world reads a random number from the LFSR within the PL over the secure master AXI port. Each bit of the random number generated by the LFSR represents a reconfigurable partition, shown in Figure 3.9, to be configured with either and XOR or XNOR reconfigurable modules. The PS uses the PCAP interface to initiate DMA transactions over a secure channel to download a partial bit file to every reconfigurable partition in the logic locked IP based on the LFSR result. The LFSR result is then programmed to the IP over the secure AXI connection which unlocks the IP for use. The flow of the reconfiguration is shown in Figure 3.11.

# CHAPTER 4: RESULTS

## 4.1 Automated Framework

The automated framework has been tested on the Verilog descriptions of ISCAS 85 benchmark circuits C17, C432, and C499 [24] to perform random, SARLock, fault-analysis based insertion. In the results we demonstrate the fault-based insertion on the benchmarks. The netlist level benchmark representations were used for fault analysis where the fault impact of each node is calculated using ATPG generated test patterns from Synopsys TetraMAX tool. An example of the fault analysis is shown in Figure 4.1. The figure shows a list of faults and the net that is causing them. For example in Figure 4.1, 3 faults are detected for test 4 for the given input pattern on nets new_n58, N108->new_n88, and N108. The framework uses the analysis to determine the net with the highest fault impact for key gate insertion. The nets with the highest fault impact are nets that cause the most faults for all test input patterns. Table 4.1 shows the number of faults detected by the framework and the total fault coverage for each benchmark.

Table 4.1: Fault impact summary for ISCAS 85 benchmark circuits.

| | C17 | C432 | C499 |
|---|---|---|---|
| # of Primary Inputs | 5 | 36 | 41 |
| # of Primary Outputs | 2 | 7 | 32 |
| # of Detected Faults | 22 | 472 | 1271 |
| # of Undetected Faults | 0 | 28 | 83 |
| Fault Coverage | 100% | 94.4% | 93.9% |

Key gates are systematically inserted into netlist where the fault impact is the highest. Gates inserted into nets with higher fault impacts are more likely to change output bits when an incorrect key is applied. Table 4.2 shows the range of key size needed for the fault analysis based insertion scheme to achieve the 50% Hammming

```
test    4: 101̄1101̄0101101̄0001̄1011110110100000 0111111    3 faults detected
           new_n58_ /1
           N108->new_n88_ /1
           N108 /1

test    5: 0000110100100101111000010110010000 10 1111111    2 faults detected
           N11 /1
           N11->new_n269_ /1

test    6: 0000001010110010101111100111000011 10 1011010    9 faults detected
           new_n96_ /1
           new_n95_ /1
           N30 /1
           new_n63_ /1
           new_n101_ /1
           N86 /0
           N329->new_n247_ /1
           new_n261_ /1
           N30->new_n261_ /1
test    8: 1010000101001000011100100110000111 01 1011100   26 faults detected
```

Figure 4.1: Fault log output during the frameworks fault analysis.

distance metric. The tables demonstrate that the automated framework reaches the 50% Hamming distance criteria to provide the most ambiguity of the locked IPs design. The table also includes the percentage overhead in terms of gates for locking the benchmark circuit. The smaller circuit c17 has a large overhead of 40% but needs only 2 key gates. The table shows that for the larger circuits c432 and c499, overhead stays under 20%.

Table 4.2: Range of key sizes to reach 50% Hamming distance.

| Range of Key Size | Benchmark | # of Gates | % of Total Gates |
|---|---|---|---|
| 2-5 | C17 | 5 | 40% |
| 17-20 | C432 | 160 | 11% |
| 39-42 | C499 | 202 | 19% |

Table 4.3 shows the outputs for the logic locked C17 benchmark using the fault analysis based key insertion. The c17 benchmark is passed to the automated logic locking framework with 3 keys that has a correct key combination of 000. The locked benchmark is programmed and verified on the FPGA platform and interfaced using Vivado Virtual Input Output (VIO) debug interface. The VIO interface is used to test the locked benchmark for every input pattern and key combination.

The table shows the IPs correct functionality when a correct key of 000 is applied to the key gates. The number of bit flips between correct outputs and outputs with

Table 4.3: Outputs for c17 locked with 3 keys with correct key is 000.

| | | | | | | Output Y for different key values | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | Y | k0 | k1 | k2 | k3 | k4 | k5 | k6 | k7 |
| 0 | 0 | 0 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 0 | 0 | 0 | 1 | 00 | 00 | 01 | 11 | 11 | 00 | 01 | 11 | 11 |
| 0 | 0 | 0 | 1 | 0 | 11 | 11 | 11 | 00 | 00 | 00 | 00 | 11 | 11 |
| 0 | 0 | 0 | 1 | 1 | 11 | 11 | 11 | 00 | 01 | 00 | 01 | 11 | 11 |
| 0 | 0 | 1 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 0 | 1 | 0 | 1 | 01 | 01 | 00 | 11 | 11 | 01 | 00 | 11 | 11 |
| 0 | 0 | 1 | 1 | 0 | 11 | 11 | 11 | 00 | 00 | 00 | 00 | 11 | 11 |
| 0 | 0 | 1 | 1 | 1 | 11 | 11 | 11 | 01 | 00 | 01 | 00 | 11 | 11 |
| 0 | 1 | 0 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 1 | 0 | 0 | 1 | 00 | 00 | 01 | 11 | 11 | 00 | 01 | 11 | 11 |
| 0 | 1 | 0 | 1 | 0 | 11 | 11 | 00 | 00 | 11 | 00 | 11 | 11 | 00 |
| 0 | 1 | 0 | 1 | 1 | 11 | 11 | 01 | 00 | 11 | 00 | 11 | 11 | 01 |
| 0 | 1 | 1 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 1 | 1 | 0 | 1 | 01 | 01 | 00 | 11 | 11 | 01 | 00 | 11 | 11 |
| 0 | 1 | 1 | 1 | 0 | 00 | 00 | 11 | 11 | 00 | 11 | 00 | 00 | 11 |
| 0 | 1 | 1 | 1 | 1 | 01 | 01 | 11 | 11 | 00 | 11 | 00 | 01 | 11 |
| 1 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 11 | 11 | 00 | 00 | 11 | 11 |
| 1 | 0 | 0 | 0 | 1 | 10 | 10 | 11 | 11 | 11 | 00 | 01 | 11 | 11 |
| 1 | 0 | 0 | 1 | 0 | 11 | 11 | 11 | 10 | 10 | 00 | 00 | 11 | 11 |
| 1 | 0 | 0 | 1 | 1 | 11 | 11 | 11 | 10 | 11 | 00 | 01 | 11 | 11 |
| 1 | 0 | 1 | 0 | 0 | 10 | 10 | 10 | 11 | 11 | 00 | 00 | 11 | 11 |
| 1 | 0 | 1 | 0 | 1 | 11 | 11 | 10 | 11 | 11 | 01 | 00 | 11 | 11 |
| 1 | 0 | 1 | 1 | 0 | 11 | 11 | 11 | 10 | 10 | 00 | 00 | 11 | 11 |
| 1 | 0 | 1 | 1 | 1 | 11 | 11 | 11 | 11 | 10 | 01 | 00 | 11 | 11 |
| 1 | 1 | 0 | 0 | 0 | 10 | 10 | 00 | 11 | 11 | 00 | 10 | 11 | 11 |
| 1 | 1 | 0 | 0 | 1 | 10 | 10 | 01 | 11 | 11 | 00 | 11 | 11 | 11 |
| 1 | 1 | 0 | 1 | 0 | 11 | 11 | 00 | 10 | 11 | 00 | 11 | 11 | 10 |
| 1 | 1 | 0 | 1 | 1 | 11 | 11 | 01 | 10 | 11 | 00 | 11 | 11 | 11 |
| 1 | 1 | 1 | 0 | 0 | 00 | 00 | 10 | 11 | 11 | 10 | 00 | 11 | 11 |
| 1 | 1 | 1 | 0 | 1 | 01 | 01 | 10 | 11 | 11 | 11 | 00 | 11 | 11 |
| 1 | 1 | 1 | 1 | 0 | 00 | 00 | 11 | 11 | 10 | 11 | 00 | 10 | 11 |
| 1 | 1 | 1 | 1 | 1 | 01 | 01 | 11 | 11 | 10 | 11 | 00 | 11 | 11 |

an incorrect key total to 222 for incorrect keys k1-k7 with a total of 448 output bits tested. This gives an average Hamming distance across the outputs for incorrect keys k1-k7 for all input patterns of 49.6%. The table demonstrates that the locked IP meets the Hamming distance criteria that provides the most ambiguity of the design, making sensitization and removal attacks more difficult.

The locked IP is reconfigured to have a correct key combination of 101 and the VIO interface is used to test outputs for the new configuration. The ouputs for each input and key pattern are shown in Table 4.4. The table demonstrates that k5 is the

correct key and produces correct outputs. The reconfigured key produces the same Hamming distance across outputs of 49.6% when incorrect keys are applied.

Table 4.4: Outputs for c17 locked with 3 keys with correct key is 101.

| | | | | | | Output Y for different key values | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | Y | k0 | k1 | k2 | k3 | k4 | k5 | k6 | k7 |
| 0 | 0 | 0 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 0 | 0 | 0 | 1 | 00 | 01 | 00 | 11 | 11 | 01 | 00 | 11 | 11 |
| 0 | 0 | 0 | 1 | 0 | 11 | 00 | 00 | 11 | 11 | 11 | 11 | 00 | 00 |
| 0 | 0 | 0 | 1 | 1 | 11 | 01 | 00 | 11 | 11 | 11 | 11 | 01 | 00 |
| 0 | 0 | 1 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 0 | 1 | 0 | 1 | 01 | 00 | 01 | 11 | 11 | 00 | 01 | 11 | 11 |
| 0 | 0 | 1 | 1 | 0 | 11 | 00 | 00 | 11 | 11 | 11 | 11 | 00 | 00 |
| 0 | 0 | 1 | 1 | 1 | 11 | 00 | 01 | 11 | 11 | 11 | 11 | 00 | 01 |
| 0 | 1 | 0 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 1 | 0 | 0 | 1 | 00 | 01 | 00 | 11 | 11 | 01 | 00 | 11 | 11 |
| 0 | 1 | 0 | 1 | 0 | 11 | 11 | 00 | 00 | 11 | 00 | 11 | 11 | 00 |
| 0 | 1 | 0 | 1 | 1 | 11 | 11 | 00 | 01 | 11 | 01 | 11 | 11 | 00 |
| 0 | 1 | 1 | 0 | 0 | 00 | 00 | 00 | 11 | 11 | 00 | 00 | 11 | 11 |
| 0 | 1 | 1 | 0 | 1 | 01 | 00 | 01 | 11 | 11 | 00 | 01 | 11 | 11 |
| 0 | 1 | 1 | 1 | 0 | 00 | 00 | 11 | 11 | 00 | 11 | 00 | 00 | 11 |
| 0 | 1 | 1 | 1 | 1 | 01 | 00 | 11 | 11 | 01 | 11 | 01 | 00 | 11 |
| 1 | 0 | 0 | 0 | 0 | 10 | 00 | 00 | 11 | 11 | 10 | 10 | 11 | 11 |
| 1 | 0 | 0 | 0 | 1 | 10 | 01 | 00 | 11 | 11 | 11 | 10 | 11 | 11 |
| 1 | 0 | 0 | 1 | 0 | 11 | 00 | 00 | 11 | 11 | 11 | 11 | 10 | 10 |
| 1 | 0 | 0 | 1 | 1 | 11 | 01 | 00 | 11 | 11 | 11 | 11 | 11 | 10 |
| 1 | 0 | 1 | 0 | 0 | 10 | 00 | 00 | 11 | 11 | 10 | 10 | 11 | 11 |
| 1 | 0 | 1 | 0 | 1 | 11 | 00 | 01 | 11 | 11 | 10 | 11 | 11 | 11 |
| 1 | 0 | 1 | 1 | 0 | 11 | 00 | 00 | 11 | 11 | 11 | 11 | 10 | 10 |
| 1 | 0 | 1 | 1 | 1 | 11 | 00 | 01 | 11 | 11 | 11 | 11 | 10 | 11 |
| 1 | 1 | 0 | 0 | 0 | 10 | 10 | 00 | 11 | 11 | 00 | 10 | 11 | 11 |
| 1 | 1 | 0 | 0 | 1 | 10 | 11 | 00 | 11 | 11 | 01 | 10 | 11 | 11 |
| 1 | 1 | 0 | 1 | 0 | 11 | 11 | 00 | 10 | 11 | 00 | 11 | 11 | 10 |
| 1 | 1 | 0 | 1 | 1 | 11 | 11 | 00 | 11 | 11 | 01 | 11 | 11 | 10 |
| 1 | 1 | 1 | 0 | 0 | 00 | 00 | 10 | 11 | 11 | 10 | 00 | 11 | 11 |
| 1 | 1 | 1 | 0 | 1 | 01 | 00 | 11 | 11 | 11 | 10 | 01 | 11 | 11 |
| 1 | 1 | 1 | 1 | 0 | 00 | 00 | 11 | 11 | 10 | 11 | 00 | 10 | 11 |
| 1 | 1 | 1 | 1 | 1 | 01 | 00 | 11 | 11 | 11 | 11 | 01 | 10 | 11 |

## 4.2 Timing Analysis

The c17 benchmark is locked with a key size of 3 gates. The logic for the 3 key gates is mapped to the reconfigurable partitions. The routing of key gates to reconfigurable partitions comes with timing and routing costs. Figure 4.2.a shows the timing analysis for a locked design with only one key versus 4.2.b which shows

the overhead of the partially reconfigurable locked design. The figure shows that the reconfigurable design does meet timing requirements but as a significantly larger worst negative slack, meaning the partially reconfigurable design has a larger path delay.

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 4.213 ns | Worst Hold Slack (WHS): | 0.053 ns | Worst Pulse Width Slack (WPWS): | 4.020 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1472 | Total Number of Endpoints: | 1472 | Total Number of Endpoints: | 666 |

**All user specified timing constraints are met.**

(a) Locked c17 without partial reconfiguration

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.424 ns | Worst Hold Slack (WHS): | 0.043 ns | Worst Pulse Width Slack (WPWS): | 4.020 ns |
| Total Negative Slack (TNS): | 0.000 ns | Total Hold Slack (THS): | 0.000 ns | Total Pulse Width Negative Slack (TPWS): | 0.000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1472 | Total Number of Endpoints: | 1472 | Total Number of Endpoints: | 666 |

**All user specified timing constraints are met.**

(b) Locked c17 with partial reconfiguration

Figure 4.2: Comparison of timing results between normal and partially reconfigurable design.

Reconfigurable partitions that are placed on the FPGA physically farther from each other requires more sophisticated routing which can lead to higher delays. Depending on the design, this could lead to setup up violations. The setup violations can be mitigated at the floorplanning and placement step of the partially reconfigurable design. The key logic consists of only 1 LUT so reconfigurable partitions can be kept closer together during the floorplanning to decrease the routing delay. The reconfigurable design implementation meets all timing requirements and overall improves the capability of key updates for the logic locked IPs using the proposed partially reconfigurable based key update architecture for reconfigurable logic locking.

## 4.3 Bitstream Generation

Partial bitstreams for XOR and XNOR gates must be generated for each reconfigurable partition for the configuration of the key at runtime. The design is implemented with XOR gates placed in all 3 reconfigurable partitions creating a key value of 000. Figure 4.3 shows successful bitstream geneartion for the full configuration bit file as well as 3 partial bit files for every reconfigurable module. Another configuration of the design is implemented with XNOR gates in all of the reconfigurable partitions. Partial bitstreams were successfully generated for XNOR gates in the reconfigurable partitions. The partial bitstream files for each reconfigurable module are converted into the BIN format for programming of the PL through the PCAP interface.

```
Command: write_bitstream -force Bitstreams/key000.bit
Attempting to get a license for feature 'Implementation' and/or device 'xc7z020'
INFO: [Common 17-349] Got license for feature 'Implementation' and/or device 'xc7z020'
Running DRC as a precondition to command write_bitstream
INFO: [DRC 23-27] Running DRC with 8 threads
INFO: [Vivado 12-3199] DRC finished with 0 Errors
Partition "pblock_rp_key0_inst" Reconfigurable Module "design_1_i/secure_c17_dpr_0/U0/secure_c17_dpr_v1_0_S00_AXI_inst/rp_key0_inst"
Partition "pblock_rp_key2_inst" Reconfigurable Module "design_1_i/secure_c17_dpr_0/U0/secure_c17_dpr_v1_0_S00_AXI_inst/rp_key2_inst"
Partition "pblock_rp_key1_inst" Reconfigurable Module "design_1_i/secure_c17_dpr_0/U0/secure_c17_dpr_v1_0_S00_AXI_inst/rp_key1_inst"
Creating bitstream...
Writing bitstream Bitstreams/key000.bit...
Process Partition "pblock_rp_key0_inst"
Creating bitstream...
Partial bitstream contains 1404992 bits.
Writing bitstream Bitstreams/key000_pblock_rp_key0_inst_partial.bit...
Process Partition "pblock_rp_key2_inst"
Creating bitstream...
Partial bitstream contains 1404992 bits.
Writing bitstream Bitstreams/key000_pblock_rp_key2_inst_partial.bit...
Process Partition "pblock_rp_key1_inst"
Creating bitstream...
Partial bitstream contains 1404992 bits.
Writing bitstream Bitstreams/key000_pblock_rp_key1_inst_partial.bit...
write_bitstream completed successfully
write_bitstream: Time (s): cpu = 00:00:36 ; elapsed = 00:00:55 . Memory (MB): peak = 3024.426 ; gain = 228.117 ;
Bitstreams/key000.bit
```

Figure 4.3: Generation of full and partial bitstreams for locked c17.

The proposed secure system and logic locked IP is implemented on a Xilinx Zedboard FPGA development board witha ZYNQ-7000 SoC. The FSBL programs the PL with the initial bitstream and loads the secure application. The secure application initializes the TrustZone configuration registers, and transfers partial bitstreams from the boot device to secure areas of DDR. The PS uses the secure master AXI port to read a random sequence of bits from the LFSR. The sequence determines whether to download a XOR or XNOR partial bitstreams to the PL for that particular key index. The key is programmed to the PL using the secure master AXI port. The secure

app establishes the secure monitor, then makes a call to it to switch the processor's security state to non-secure.

## 4.4 Security Analysis

The IP is obfuscated with logic locking and produces incorrect outputs until a correct key is provided. Keys are only be provided through the IP owner which makes piracy or cloning of the IP unfeasible. The framework gate insertion meets the 50% Hamming distance criteria to provide the most ambiguity to the behavior of the locked IP. The secure application reads the key from the LFSR and stores the key in registers in the IP. After initialization the system continues to operate in the non-secure context and be isolated from the IPs resources. The partial bitstream files are placed into secure areas of memory, blocking the non-secure world from initiate DMA transfers to change the key logic. The master AXI port connected to the IP does not propagate any requests issued from a non-secure processor. The hardware backed mechanisms ensure that if the non-secure system software is exploited to access system resources, the IPs configuration and key is still protected.

# CHAPTER 5: CONCLUSIONS

In this work we propose a framework for obfuscation of IPs using state of the art logic locking techniques. The framework demonstrates an automated flow that takes RTL files of the design and supports different insertion algorithms. The results are demonstrated on ISCAS 85 benchmarks and shown to produce secure locked netlist with 50% Hamming Distance across outputs when incorrect keys are applied. This work additionally proposes and demonstrates a novel logic locking scheme for FPGA platforms where key logic is placed into partially reconfigurable areas of the FPGA. The logic locking scheme provides a mechanism that enables key updates for the locked IP during runtime and is not too costly as it meets all timing requirements. Furthermore, we propose and demonstrate a novel secure key provisioning system for the partially reconfigurable IP. The design ensures hardware based isolation of the IP and a TEE for secure deployment and configuration of the key.

# CHAPTER 6: FUTURE WORK

The logic locking framework proposed in this thesis intends to automate the process of obfuscating RTL files with support for many insertion algorithms. Currently the framework supports fault analysis based insertion and SARLock. Future work on the framework can support new insertion algorithms such a Strong Logic Locking (SLL or Stripped Functionality Logic Locking (SFLL). This work proposes using XOR and XNOR gates as the key logic to be placed in reconfigurable partitions. Potential future research can include the design of more complex key logic such as Boolean one-hit functions for placement in reconfigurable partitions. Tools for the synthesis of "relocatable" partial bitstreams where partial bitstreams can be programmed on many different reconfigurable partitions of the device have been integrated with Xilinx's Vivado development environment. Future work can also focus on using relocatable partial bitstreams to simplify the bitstream generation process since the same reconfigurable modules are shared between reconfigurable partitions.

# REFERENCES


[1] H. M. Kamali, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "Advances in logic locking: Past, present, and prospects," *Cryptology ePrint Archive*, 2022.

[2] A. Duncan, F. Rahman, A. Lukefahr, F. Farahmandi, and M. Tehranipoor, "Fpga bitstream security: A day in the life," in *2019 IEEE International Test Conference (ITC)*, pp. 1–10, 2019.

[3] S. Amir, B. Shakya, D. Forte, M. Tehranipoor, and S. Bhunia, "Comparative analysis of hardware obfuscation for ip protection," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pp. 363–368, 2017.

[4] M. Ender, P. Swierczynski, S. Wallat, M. Wilhelm, P. M. Knopp, and C. Paar, "Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 112–119, 2019.

[5] T. Hoque, R. S. Chakraborty, and S. Bhunia, "Hardware obfuscation and logic locking: A tutorial introduction," *IEEE Design Test*, vol. 37, no. 3, pp. 59–77, 2020.

[6] M. T. Rahman, S. Tajik, M. S. Rahman, M. Tehranipoor, and N. Asadizanjani, "The key is left under the mat: On the inappropriate security assumption of logic locking schemes," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 262–272, 2020.

[7] J. Mellor, A. Shelton, M. Yue, and F. Tehranipoor, "Attacks on logic locking obfuscation techniques," in *2021 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–6, 2021.

[8] M. Yasin and O. Sinanoglu, "Evolution of logic locking," in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6, 2017.

[9] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 410–424, 2015.

[10] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 137–143, 2015.

[11] S. Engels, M. Hoffmann, and C. Paar, "The end of logic locking? a critical view on the security of logic locking," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 796, 2019.

[12] M. Yasin, B. Mazumdar, J. J. V. Rajendran, and O. Sinanoglu, "Sarlock: Sat attack resistant logic locking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 236–241, 2016.

[13] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 471–476, 2019.

[14] R. Zamacola, A. G. MartÃnez, J. Mora, A. Otero, and E. d. L. Torre, "Impress: Automated tool for the implementation of highly flexible partial reconfigurable systems with xilinx vivado," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–8, Dec 2018.

[15] K. Vipin and S. A. Fahmy, "Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–39, 2018.

[16] Xilinx, "Zynq-7000 soc technical reference manual," 2021. Accessed February 2022.

[17] Xilinx, "Vivado design suite user guide: Partial reconfiguration," 2017. Accessed February 2022.

[18] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, 2015.

[19] V. Costan and S. Devadas, "Intel sgx explained." Cryptology ePrint Archive, Report 2016/086, 2016. `https://ia.cr/2016/086`.

[20] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[21] G. Williams, J. Aizprua, M. Alhaddad, D. Yang, N. BouSaba, and F. Saqib, "A soc design of trustzone based key provisioning for fpga ip protection," in *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 874–877, IEEE, 2021.

[22] Xilinx, "Programming arm trustzone architecture on the xilinx zynq-7000 all programmable soc," 2014. Accessed February 2022.

[23] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019.

[24] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the iscas-85 benchmarks: a case study in reverse engineering," *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.