7-2023

FSMLock: Sequential Logic Locking through Encryption

Matthew Krebs
mlk6450@rit.edu

FSMLock: Sequential Logic Locking through Encryption

Matthew Krebs

FSMLock: Sequential Logic Locking through Encryption

Matthew Krebs

July 2023

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

FSMLock: Sequential Logic Locking through Encryption

Matthew Krebs

Committee Approval:

Dr. Marcin Łukowiak *Advisor* Date
Department of Computer Engineering

Dr. Stanisław Radziszowski Date
Department of Computer Science

Dr. Michael Zuzak Date
Department of Computer Engineering

Dr. Michael Kurdziel Date
Principal Fellow, L3Harris Technologies, Inc.

Acknowledgments

Foremost, I would like to thank my mother, Arlene Krebs, girlfriend, Nicole Zhe, and sister, Laura Krebs, who supported me and listened to endless computer engineering monologues during my time at RIT. Without their encouragement and love in times of struggle, I would not have made it as far as I have today.

I also appreciate the guidance and financial support provided by L3Harris Technologies on this project. The representatives thereof, Dr. Michael Kurdziel and Steve Farris, have played an essential role in the scoping and direction of the project and its related deliverables.

Finally, I would like to thank my advisor Dr. Marcin Łukowiak, committee members, and other research group members, including Dr. Stanisław Radziszowski, Dr. Alan Kaminsky, Dr. Peter Bajorski, and Dr. Michael Zuzak. Our weekly meetings: guided my research, supported my continual progress on the project, and allowed me to voice my questions/ideas and receive immediate feedback. Besides Dr. Marcin Łukowiak's contribution to this project as my advisor, I want to credit him as the primary researcher responsible for the conception of this logic locking methodology and thank him for selecting me to be the graduate researcher responsible for its materialization.

Dedicated to my late father, James Krebs, whose unwavering support of my education will remain with me forever. While he couldn't witness this significant milestone, I am confident he never doubted I would get here.

Abstract

As the technology node size for integrated circuit (IC) designs decreases, the cost of building and maintaining an IC foundry rapidly increases. Companies unable to afford local manufacturing have become reliant on outsourcing the physical manufacturing process. This introduces confidentiality, integrity, and authenticity security vulnerabilities into the IC design lifecycle. Even companies that manufacture in-house and use field-programmable gate array (FPGA) chips may require third-party system integrators to assemble the final product. When said product is sent to a third-party foundry or system integrator, the embodied IC/FPGA circuitry is susceptible to IP theft, Trojan insertion, and reverse engineering (RE) attacks. To address this, we realize a novel approach to sequential logic locking, FSMLock, that conceals a finite state machine's (FSM) output and next-state logic through classical encryption. The FSM is abstracted as the configuration data for a lookup table (LUT), encrypted with a chip-specific (individual) internal key, and stored in the newly mandated non-volatile memory (NVM). The configuration data is then decrypted in blocks and loaded into the in-scope random access memory (RAM) when required. Doing so locks the sequential FSM logic and conceals its functionality from third-party foundries and system integrators, system design engineers with access to the post-locked hardware description language (HDL) files, and end-users with production units.

FSMLock has applications in reconfigurable hardware, such as FPGAs, even when no third-party access is initially required. In older and low-cost FPGA devices with externally stored bitstream configurations, the absence of trusted bitstream encryption/authentication means that if the bitstream is recovered from the external memory device, an adversary can reconstruct and modify the original design functionality. FSMLock can improve the security of such FPGA chips by storing targeted FSM logic in encrypted NVM. Therefore, a breach of the bitstream contents and the NVM's individual internal key would be required to compromise the security of the targeted

sequential circuitry. Further, if a key preprocessor utilizing a physically unclonable function (PUF) is included to discriminate the boundary level (chip) key from the internal key, the confidentiality of the locked circuit is assured, even considering the disclosure of a chip key with its paired encrypted NVM configuration.

For the scope of this thesis, we sought to develop an automated software toolset capable of translating pre-partitioned FSMs into encrypted memory configurations. When the configuration is combined with the provided HDL entity responsible for run-time decryption and scope control, a locked HDL model of the FSM, i.e., the FSMLock primitive, is formed.

Contents

List of Figures

List of Tables

Acronyms

BMEM Block Memory

EDA Electric Design Automation

FPGA Field-Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

IC Integrated Circuit

IP Intellectual Property

LUT Lookup Table

NVM Non-volatile Memory

RAM Random Access Memory

RE Reverse Engineering

RTL Register Transfer Level

SEI State Effective Inputs

SET State Entry Table

STG State Transition Graph

STT State Transition Table

Chapter 1

Introduction

1.1 Motivation

The digital integrated circuit (IC) is fundamental to all complex digital computing systems. They are present in general-purpose computers, microcontrollers, and other application-specific devices. However, as the performance and complexity of such devices are steadily increasing, it has become less economical for small IC companies to perform in-house fabrication [10, 11, 12]. This industry change is mainly due to the excessive capital investment required to build and maintain a modern fabrication plant. All but the largest companies have no feasible way to manufacture their chips and, as such, are forced to outsource the physical IC fabrication. This workflow is referred to as "fabless manufacturing". Although this process may be sufficient for low-importance products, it introduces major confidentiality, integrity, and availability vulnerabilities deemed un-trustworthy by several government programs [12]. In addition, health and safety concerns exist for other mission-critical applications in fields such as medical, aerospace, defense, automotive, banking, and energy [13].[1]

Likewise, security vulnerabilities exist in reconfigurable hardware such as field-programmable gate arrays (FPGAs). FPGAs are a popular alternative to custom IC designs because they allow system designers to realize their hardware without

[1] To make matters worse, due to the ongoing COVID-19 epidemic, it is reasonable to forecast, as G. Mura et al. does [14], that the market of electronic parts may soon become increasingly overrun by counterfeit components due to the lack of availability of the original product.

requiring physical manufacturing. This makes them more desirable for low-sales volume products, where the cost of mask generation can not be economically offset, and in versatile devices that may require hardware updates after initial deployment. Since FPGA designs can be loaded onto the chip without the intervention of a third-party manufacturing plant, foundry-related threats are less applicable. That does not mean that FPGAs are without security vulnerabilities; instead, the primary attack surface of a reprogrammable FPGA is the configuration bitstream. Since many reprogrammable FPGAs, such as static random access memory (SRAM) FPGAs, do not retain their configuration between power cycles, an external memory device must store the configuration bitstream. Therefore, anyone with physical access to the FPGA system can load the configuration bitstream off the memory device and examine its contents. In older and low-cost devices, the absence of trusted bitstream encryption/authentication means an adversary can recover, reconstruct, and modify the original design functionality. An example of which is the unpatchable vulnerability found in the Xilinx 7-Series (and Virtex-6) bitstream encryption [15]. FPGA vulnerabilities are not isolated to reprogrammable FPGAs; one-time programable FPGAs using anti-fuse technology are susceptible to RE attacks similar to ICs through optical analysis of the delayered chip, as discussed in Subsection 2.2.2.

In the case of both IC and FPGA designs, there may also exist the requirement for a third-party system integrator who is responsible for assembling the pre-manufactured ICs and/or programmed FPGAs into the final board or system. Like when the IC is sent to a third-party foundry, there exist security vulnerabilities when the third-party system integrator is in possession of the sensitive digital logic. A system integrator may not have the breadth of assets available to a malicious foundry, as discussed in Subsection 3.1.1, but they surely possess a subset of their ability and can therefore perform damaging attacks on the IC/FPGA system.

Preventative measures to mitigate the security vulnerabilities introduced by fab-

less IC manufacturing, third-party integrators, and FPGA systems exist, including IP encryption, hardware/IP authentication, split manufacturing, logic obfuscation, and logic locking. However, these techniques' implementations vary in effectiveness and are designed with different security goals in mind. Logic locking, in particular, aims to prevent unauthorized use and duplication of ICs through the requirement of a key to unlock the functionality of the IC. Although this is a noble goal, we note that many modern methods of logic locking make trade-offs regarding security and output corruption.

1.2 Objective

In short, this thesis aims to develop a methodology for creating instances of the sequential logic locking primitive shown in Figure 1.1. Also, an output product of the research is an automated toolset capable of generating said primitives with minimal system designer input. The idea behind this logic-locking technique comes from the work of [1], and the high-level diagram used to describe the primitive in said work is illustrated in Figure 1.1. The name "FSMLock" was chosen by the author of this thesis to represent this structure, thereafter the initial conception of the idea in [1], to summarize best what the technique can be used for.

Along with creating an automated toolset for generating FSMLock primitive, this thesis will model its theoretical resource utilization, performance characteristics, and security properties. Using this information, we aim to provide use-case recommendations for the primitive, such that a security-focused system designer could enforce the FSMLock constraints early into the design process to minimize the hardware inflation and performance impact on the locked circuitry.

The output product of this research is a toolset capable of generating the register transfer level (RTL) netlist for locked FSMs using the novel concept of encrypted next-state and output logic. This toolset is intended for use in the IC/FPGA

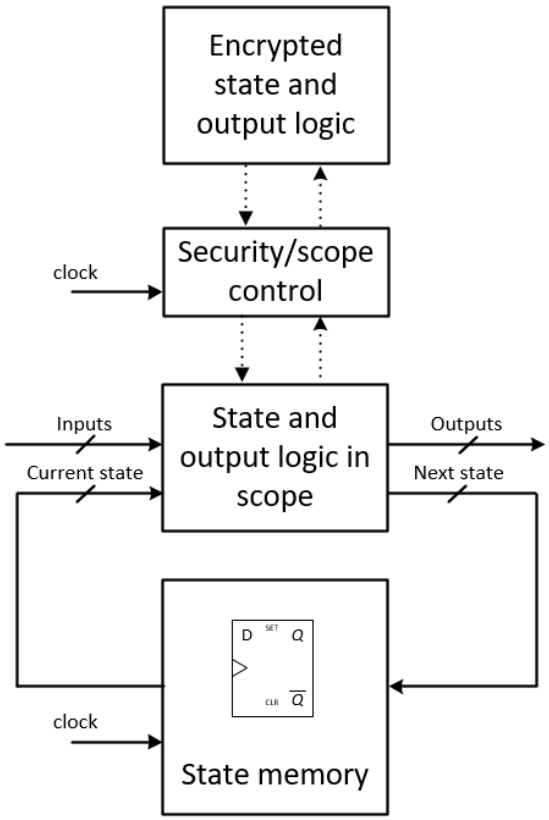

Figure 1.1: The model of the proposed FSM locking method from [1].

lifecycle (shown in Figure 2.1) system design stage; that is, on the design's behavioral/structural hardware description language (HDL) model before layout or mask generation. In the case of FPGA development, logic locking is performed before synthesis. Considering potential design constraints, the FSMLock automation toolset includes configuration options allowing customization of the final state machine RTL netlist output.

1.3 Agenda

First, we aim to provide a detailed and thorough overview of the field of hardware security. This is included to educate the reader on the field and set a stage for the place the novel logic locking primtive discussed in this thesis, FSMLock, fits into. Chapter 2 provides this background and covers the terminology, threats, and preventative

measures needed to grasp the extent of the hardware security field of research.

Next, considering the trade-offs present in existing methods of hardware security and logic locking–described in Subsection 2.3.5 and 2.3.6–in Chapter 3, we seek to define the methodology for the novel FSMLock sequential logic locking primitive. The FSMLock primitive maintains high output corruptibility and effective security while concealing the functionality of and preventing unauthorized access to the targeted locked sequential circuit. To do so, we instruct that a traditional cryptographic block cipher be used to encrypt the configuration data of a memory lookup table (LUT). Specifically, in FSMLock, this approach is applied to an obfuscated finite state machine (FSM) model, which uses a memory-based LUT structure to store next-state and output logic. Therefore, the next-state and output logic is concealed and inaccessible without the proper cryptographic internal key used to decrypt the encrypted memory-based LUT. Also, in Chapter 3, theoretical resource utilization models, performance characteristics, and claims regarding the security properties of the primitive are provided. The security characteristics are based on the assumed assets and goals defined within the threat model discussed in Section 3.1.

Then in Chapter 4, case studies utilizing the FSMLock primitive are reviewed. The theoretical, predicted, and experimental resource utilization values are compared. Differences between the theoretical values possible given the methodology and the predicted values using our implementation of the automation toolset are described. Further, discrepancies between the predicted resource utilization values and the experimental results are uncovered.

Lastly, in Chapter 5, we provide potential future improvements to the FSMLock primitive and work to be done on the automation toolset to ensure the experimental synthesis results align with what is predicted. Advancements in utilization and performance are modeled for each improvement. With these improved models and the review of the case studies provided in Chapter 4, use-case recommendations are

provided for the FSMLock primitive in Section 5.2. Considering the detailed background, methodology, and use-case recommendations provided, we hope this thesis enables and promotes the future use and improvement of the FSMLock sequential logic locking primitive.

Chapter 2

Background

2.1 Notation and Terminology

Before explaining the threats and preventative measures in the IC/FPGA design lifecycle, it is important to acknowledge the complex and often misused terminology existing in general data and information security as well as hardware security literature.

2.1.1 General Data and Information Protection

In the context of digital cryptography, "data" is an abstract term used to describe a collection of facts. An example of data is a generic file on a computer. When the type of a file is known, the data is put into context and the information encoded within the file's binary code is revealed.

A model used to evaluate the security of data and information of a system is the CIA triad (Confidentiality, Integrity, and Availability).[2] Confidentiality is the quality of preserving secrecy. For information to remain confidential, there must not exist leakage to unauthorized parties. Integrity is the quality of being whole and unadulterated. If data is tampered with, the information encoded by it could

[2]The exact origins of the "CIA triad" expression are unknown. The concepts which it encapsulates are the fundamentals of data and information protection and are believed by members of the cybersecurity community such as Ben Miller [16], coordinator of the CharmSec security conference, to trace back to early cyber security reports [17] and [18]

be maliciously altered, resulting in loss of integrity. Availability is the quality of providing access when required.

A vulnerability exists in a system if an adversary has the potential to compromise one or more of the CIA pillars. For a vulnerability to rank as a threat, two factors are considered: the adversary's capabilities and the value of the system asset, which is exposed via the vulnerability. If an adversary does not possess the skills necessary to take advantage of a vulnerability, there is less of a threat. Likewise, if the asset is of low value to the adversary, they are less likely to commence an attack against it, which reduces the threat.

2.1.2 Hardware Protection

When protecting a digital data/information asset, the term encryption is frequently used. This is an acceptable use of the term as there exists an agreement on the meaning: the lossless cryptographic process of converting data into a hidden and unintelligible format that is computationally infeasible to reverse without the use of a valid key [19, 20].

In the field of hardware protection, the terminology is not as well defined. For example, the terms "hardware/logic hiding", "hardware/logic encryption", "hardware/logic obfuscation", and "hardware/logic locking" are often interchanged with each other. As reinforced by S. Engels et al. in [21], the interchanging of these terms is ill-advised and misleading; encryption, obfuscation, and locking have different definitions and can imply different constructs. To reduce confusion, this thesis will use the following definitions derived from the work of [21].

Logic Obfuscation The transformation of sequential or combinational logic into an alternative but functionally equivalent representation

Logic Locking The transformation of sequential or combinational logic into a restricted alternative representation which requires the intervention of a key to access the device's functionality

Note the exclusion of the terms "hardware/logic hiding" and "hardware/logic encryption" in the above definitions. These terms were intentionally omitted and will not be used in this thesis because "locking" is already generic enough to encompass any scheme that aims to hide a circuit's functionality or uses encryption to restrict access–such as the novel technique, FSMLock, discussed in this work.

In summary, the fundamental difference between logic obfuscation and logic locking is whether the process changes the circuit's behavior. Even if a logic locking scheme does serve to transform a circuit into an alternative representation–potentially through the addition of locking circuitry–it should not be considered an obfuscation countermeasure because, by design, it changes the original circuit's behavior.

Additionally, logic locking literature does not have a consistent naming scheme for the keys involved. A non-comprehensive list of terms witnessed includes "chip key", "common key", "external key", "input key", "internal key", "master key", "secret key", and "unlock key". The intermittent and sometimes contradictory use of these terms has the potential to cause confusion. As such, when discussing forms of logic locking, the terminology "internal key" and "chip key" will be used in this thesis. The following definitions of these terms are derived from the work of S. Engels et al. [21].

Internal Key The key directly used by the locking circuitry known only to the IP-rights holder

Chip Key The (external) input to the IC/FPGA during unlocking known to the IP-rights holder and distributed to legitimate chip owners

The distinction between the chip key and the internal key is significant when considering key pre-processors such as in the work of [22], [23], [2], and [24]. A key pre-processor takes advantage of a cryptographic process or the inherent unclonable variability in modern manufacturing–such as in a physically unclonable function (PUF) [23]–to transform the chip key into the internal key. A key pre-processor can serve multiple purposes. An example of one is in the works [2] and [24], the cryptographic key pre-processor acts as a one-way random function used to counter satisfiability (SAT) attacks, described later in Subsection 2.3.5. Another purpose for key pre-processors is illustrated in the works [22] and [23], where through the use of a PUF, it enables locked IC/FPGAs with a global internal key to use an individual chip key: see the following definition of the terms "individual" and "global" in this context.

Individual ____ Key The respective key is different for each IC/FPGA

Global ____ Key The respective key is identical for each IC/FPGA

As will be discussed further in Subsection 3.1.2, the explicit distinction between individual and global keys plays an important role in classifying the outcome of an attack.

2.2 Threats Throughout the IC/FPGA Design Lifecycle

The lifecycle of an IC design is a staged process. For the scope of this research, this process is generalized into four stages: IP development, system design, physical manufacturing, and end-user deployment. These stages are based on those presented in [25] and are illustrated in Figure 2.1 alongside the FPGA lifecycle. The IC design lifecycle was partitioned into these four stages because they represent all potential handoffs in the fabless manufacturing workflow. At each handoff, there exists a shared trust between the stakeholders involved. The stakeholders for a given stage in the lifecycle providing the data entrust the stakeholders receiving it to use it only as intended. Likewise, the stakeholders receiving the data trust that the stakeholders providing it presented its functionality honestly. Note that the handoff between the system design and physical manufacturing stages is the only new transition introduced via the fabless manufacturing workflow. All others exist in an in-house design lifecycle as well, and, as such, the related security vulnerabilities are also pertinent to the in-house fabrication lifecycle. The additional configuration implementation stage is FPGA-specific, the stakeholders of which would be the Electronic Design Automation (EDA) vendors such as Intel (previously Altera) [26] and AMD (previously Xilinx) [27]. In each stage, threats exist that pose risks to the data and information encapsulated by the IC/FPGA design.

Consider an IC design where the system design team requires external IP blocks; the IP author(s) must first share their blocks with the system design team.[3] Once the system design team finishes, assuming they are fabless, they must provide the foundry responsible for physical manufacturing with the layout. Finally, once manufacturing is complete, the product will be handed off once more to the end-user. This scenario

[3]The shared use of existing IP is not a contrived assumption. It is a common practice for many IC and FPGA system design teams since it significantly reduces time, cost, and the complexity of an overall design.

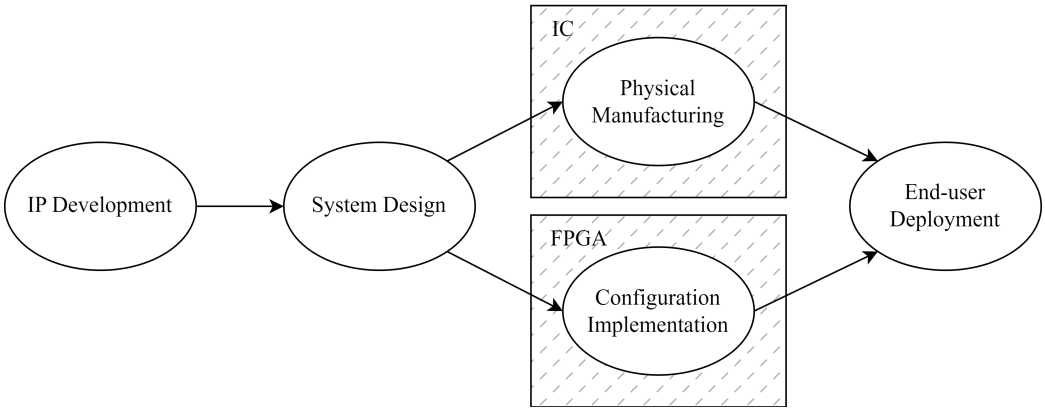

Figure 2.1: Generalized IC/FPGA design lifecycle. The nodes represent stages in the lifecycle, each with its corresponding stakeholders. The edges between stages illustrate the handoffs between stakeholders.

depicts an IC being passed between four independent parties with three handoffs, as illustrated in Figure 2.1.

Similarly, an FPGA design may begin with the creation of third-party IP blocks. Next, the system design team integrates said blocks into their design and utilizes the chip vendor's closed-source EDA tool for implementation. Finally, the resultant configuration bitstream is loaded into non-volatile memory and delivered to the end-user. This scenario also depicts four independent parties with three trusted handoffs.

In the following subsections, a collection of threats will be addressed. The effects of these threats have been classified by the stage of the IC/FPGA lifecycle in which they are present and by what type of security vulnerability is exposed in the CIA triad.

2.2.1 Trojan Insertion

For an IC/FPGA design, a primary threat that risks the integrity and availability of the circuit is a hardware Trojan. A hardware Trojan is a malicious circuit that alters the design [25] and evades detection under conventional post-manufacturing test/validation processes [28]. To evade detection, Trojans are often inserted with triggers attached to signals with low controllability [25, 29]. Research has been done

to minimize the existence of low controllability signals through the insertion of logic locking like gates [29] and accentuate the danger of traditionally unreachable sequential states (which align themselves well as Trojan triggers since system designs may not think to or know how to reach them) [25]. On the other hand, the preventative measures and design best practices are not foolproof and, as such, Trojan insertion is still possible in any IC/FPGA lifecycle stage leading up to end-user deployment.

In the first shared stage of the IC/FPGA design lifecycle (IP development), the IC design is susceptible to Trojan insertion via untrusted third-party IP authors. Because IP designs are often encrypted to prevent piracy, as described later in Subsection 2.3.1, there is the potential for a malicious IP author to conceal undocumented circuit elements in their IP without the system design team knowing.

In the system design stage, Trojan insertion is possible in a similar fashion. A rogue designer can include unwanted circuitry in the HDL and the corresponding mask layout. Even a trusted designer can inadvertently insert a Trojan using an untrusted or compromised third-party EDA tool.

During the physical manufacturing of an IC, Trojans are inserted amidst the semiconductor wafer production via changes to the lithography mask. As explained in [30], one method of doing so is utilizing an engineering change order which is described as an "effortless exercise" for the attacker. The Trojan inserted in [30] is a side-channel Trojan (see Subsection 2.2.3 for discussion of the term side-channel) used to determine encryption engine keys via power signature readings. It was determined that by adding the Trojan to the spare and filler cells and using the least congested metal layers, the Trojan could be added to the design without compromising the initial timing requirements, hence increasing the difficulty of detecting it in compromised devices.

Lastly, during FPGA End-user Deployment, Trojan insertion is possible in devices without–or with untrusted–bitstream encryption and authentication mecha-

nisms through manipulations of the bitstream configuration. An example of which is the non-cryptographic CRC checksum used in many older devices, such as those in the Xilinx Virtex-5 lineup [31]. When an FPGA device uses CRC to authenticate the configuration, an attacker can alter the bitstream configuration, calculate the new CRC, and modify the stored value accordingly. This results in an altered circuit configuration without compromising the CRC authentication process [32]. An example of this is shown in [31], where through disabling a block memory device, an AES engine could be compromised to leak the 10th round key of the AES cipher and consequently recover the input key using a reverse key scheduling algorithm. This attack can even be performed on encrypted bitstreams from the Xilinx Virtex-5 lineup due to a security flaw in the Cipher Block Chaining (CBC) mode of operation used for bitstream encryption. Because the cipher text from the previous block is XORed against the unencrypted cipher text from the current block during the CBC decryption process, a single bit can be precisely flipped in the current block through modification to the previous blocks cipher text [20].

2.2.2 Reverse Engineering

Reverse engineering (RE) refers to the process of information retrieval from a product to understand its composition and inner workings [33]. Although the term's connotation is often negative, RE can also be performed in non-malicious scenarios. Consider counterfeit product and hardware Trojan detection [33]. Both tasks require advanced RE tools to ensure the product received is genuine and behaves as predicted. Furthermore, RE is legal in many countries for competitive product analyses, education, and research [33]. In the United States, the Semiconductor Chip Protection Act of 1984 preserves the right to RE a semiconductor for the previously mentioned purposes as long as no copyrights or patents are infringed [34].[4]

[4]The "substantial similarity" test of copyright law is used to arbitrate the tipping point between competitive emulation and infringement on the exclusive rights of the owner's mask work [34, 35].

When aiming to reproduce a product through RE, the objective is to create a clone or a surrogate. A clone seeks to exhibit the original's exact form, fit, function, and mechanism of operation. A surrogate only aims to fulfill the same purpose, potentially using a different mechanism of operation [36]. In the context of IC/FPGA vulnerabilities, RE refers to the malicious intent to gain information on an IC or FPGA configuration and breach the design's confidentiality. Once collected, this information can be used to aid other attacks such as Trojan insertion, as described in Subsection 2.2.1, or used to reproduce any segment of the design without the authorization of the design owner(s).

The practice of reproducing a design or a substantial portion of it without authorization is referred to as "piracy". A few examples of piracy include a foundry manufacturing and reselling counterfeit clones of an IC derived through RE of the original design, a foundry overproducing the *original* design and selling the excess to a third party, or even a member of the system design team reusing the netlist/layout of a temporarily licensed IP block in other unauthorized designs. In an FPGA system, piracy also includes distributing the configuration data.

In the IC Trojan insertion attack described in [30], four inputs are required to initiate the necessary RE: the technology library, cell library, gate-level netlist, and the timing constraints. Since this potential attack is performed during the physical manufacturing process, the malicious foundry is assumed to have access to the first two. The ownership of the technology and cell libraries is a safe assumption considering that the technology library and cell library are node dependent and are distributed initially via the foundry. Therefore, only the gate-level netlist and timing constraints must be derived through RE. Unless novel timing camouflage methods such as wave-pipelining proposed in [37] are used, RE minimum timing constraints can be trivially estimated by observing the combinational logic block with the longest propagation delay. Consequently, this Trojan attack is foremost dependent on a RE attack of the

gate-level netlist. As will be discussed, various methods exist to generate gate-level netlists.

The optical imaging RE attack was popularized over 20 years ago when ICs commonly had one layer of metal and were designed using a 1-2 µm process. An IC optical imaging RE attack proceeds as follows: package removal, delayering, imaging, annotation, schematic read-back, and analysis [38]. Intricate and detailed processes are required to perform this attack because of the extreme accuracy necessary to delayer the IC without disturbing the subsequent layers. Delayering and imaging are the most resource-intensive stages of this attack, but additional work is still required to generate the final gate-level netlist. Mainly, the gates in the images must be identified, and the nets must be labeled so that they can be entered into a schematic and analyzed. This task, which would be near impossible by hand, has been automated by software applications such as Chipworks' ICWorks [38] and Degate [39]. Although the limited resolution of the optical imaging in the visual spectrum makes it insufficient for RE attacks on smaller modern technology nodes, alternative imaging methods such as scanning electron microscope (SEM) imaging and focused ion beam (FIB) imaging exist, which continue to enable this type of attack today [33, 38].

Another approach to IC RE is to retrieve the gate-level netlist before manufacturing. A Reverse Engineering Framework from GDSII to Gate-level Netlist (ReGDS) is an example of such an attack [40]. Unlike imaging attacks, the ReGDS attack is performed on the GDSII IC layout files presented to the foundry. In the ReGDS attack, two steps are required to generate the gate-level netlist from the GDSII layout. First, a Layout vs. Schematic (LVS) verification tool, provided by EDA vendors such as Synopsys [41], Cadence [42], and Siemens [43], is used to extract the transistor level netlist from the layout. Second, the newly proposed graph matching algorithm, Digital Connectivity Index, uniquely identifies the logic gates within the transistor level netlist via the transistors drain/source/gate connectivity [40]. The result of these

steps is an RTL netlist. Such an attack has inherent advantages over imaging-based attacks; the most obvious of which is that the attack does not require a preexisting fabricated IC to tear down. A byproduct of this advantage is that the attack does not rely on physical processes and is thus more economical. The ability to RE before fabrication is particularly intriguing because it permits inserting a Trojan into the, presumably, golden example batch (presumed perfect production batch, against which all others are verified) of an IC.

ICs are not unique in their vulnerability to RE attacks. SRAM-based FPGAs, which are the predominant FPGAs used today, are susceptible to RE attacks as well through their configuration bitstream. A configuration bitstream is a (typically propriety) file that is generated by the vendor-specific EDA tools during the Configuration Implementation state in the FPGA lifecycle. Common EDAs include Vivado, by AMD [27], and Quartus, by Intel [26]. The configuration bitstream stores all of the information needed to program the FPGA interconnects and reconfigurable logic blocks, consisting of LUTs, FFs, and other device primitives. Because SRAM FPGAs must be programmed on each power-up, an external memory device often stores the configuration bitstream. A RE attack against an SRAM FPGA occurs in the End-user Deployment stage of the FPGA lifecycle through a malicious party with physical access to the FPGA system. Such an attacker could effortlessly load the configuration information off of the external memory device and RE the circuit RTL information, as explained in the work of J. Note et al. [44].

To protect against FPGA bitstream RE, many modern FPGAs include bitstream encryption. In doing so, the configuration confidentiality is cryptographically secured. However, not all commercially available bitstream encryption techniques effectively thwart the security threats related to FPGA bitstream RE. Specifically, there have been several vulnerabilities found, such as those exploited in the attack on the AES engine [31], discussed previously in Subsection 2.2.1, and those brought up in [15].

The paper [15], aptly named "The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs", describes how, through manipulations of the encrypted bitstream (possible during the End-user Deployment stage), the unencrypted configuration content can be read out. This vulnerability is unpatchable and affects all Xilinx 7-series FPGAs and Virtex-6 devices, the latter being with limitations [15]. RE attacks of SRAM FPGA configuration files, even when encrypted on specific devices, are a legitimate threat to the confidentiality of the RTL information and IP encompassed by said configuration.

The final RE attack that will be discussed is particularly relevant to the topic of this thesis, which is gate-level FSM RE. This is a particularly challenging proposition because it requires one of the aforementioned forms of RE to identify the RTL netlist of a design and a computational method for register categorization. As discussed later in Section 3.2, and shown in Figure 3.1, an FSM is traditionally composed of combinational logic gates and memory registers such as flip flops. It is beneficial for an attacker to isolate the FSM control path registers from data path registers because explicit knowledge of a circuit's control path can provide additional insight into the circuit's purpose and workings, which sole knowledge of the RTL netlist does not provide. That being said, there are also legitimate reasons for FSM RE, the foremost of these being failure analysis and the detection of counterfeit products and hardware Trojans. Hardware Trojans detection is notable among this list because, as discussed in [25], the existence of unreachable FSM states lends themselves well to Trojan insertion since the system design is not expecting a potential transition to them and, as such, any additional logic added to reach them may evade detection under conventional post-manufacturing test/validation processes.

In consideration of the legitimate reasons for FSM RE, and in pursuit of bringing light on the illegitimate attacks possible on thought to be secure FSM designs, successful research has been performed to aid in the automation of FSM extraction from

gate-level netlists [45, 46, 47]. These works culminate in the work of M. Fyrbiak et al. [8], in which a summary of the FSM identification process, the algorithmic steps required, and case studies of the RE technology are provided. More so, [8] illustrates the potential of this RE tool against sequentially logic locked designs and points out vulnerabilities in their assumed security characteristics when the prospect of gate-level FSM RE is considered. Further discussion of these sequential logic locking techniques and their related flaws are discussed in Subsection 2.3.6.

2.2.3 System Analysis

The last stage of the IC/FPGA lifecycle, end-user deployment, contains vulnerabilities relating to confidentiality. In this context, it is assumed the end-user does not have access to the layout, fabrication byproducts, or the tools necessary to perform invasive post-fabrication attacks such as optical imaging discussed in Subsection 2.2.2. Therefore, all attacks against the IC/FPGA performed via an end-user are non-invasive and, instead, reliant on observations of the device in operation. We will categorize this style of attack as hardware system analysis attacks.

A key aspect of hardware system analysis attacks is one's ability to monitor some aspect of the IC while it is in operation. Despite the rising popularity of system-on-a-chip designs, which envelop a wide variety of processing and memory components onto a single chip, there is still the need to interface with the IC/FPGA at some boundary level. Therefore, one could record the data present at this boundary level (on the package pins of the IC/FPGA or the surrounding traces of the printed circuit board) with probes or other measurement equipment to obtain a functional understanding of the chip's purpose [38].

A system analysis attack, although revealing to the data passing through the boundary of an IC/FPGA and hence potentially its purpose, is not a substantial threat to the IC/FPGA's design information on its own. Very little information on

the inner construction of a modern IC with billions of transistors is attainable via only observation at the package boundary. Nonetheless, it is still worth considering because system analysis attacks are fundamental to the operation of some hardware Trojans referred to as "side-channel hardware Trojans". Instead of system analysis on the package boundary signals, observations of the side-channel, i.e., non-functional properties, is possible. Examples of non-functional properties include power consumption, execution time, leakage current, temperature, electromagnetic emanations, and backscattering [48]. Measurements of the above properties with, or even without, the inclusion of a hardware Trojan designed to selectively amplify them have the potential to leak information beyond what is readily available at the package pins. This is because said measurements record the non-functional byproducts directly influenced by the signals and architectural components internal to the IC/FPGA that are assumed confidential by the system designer.

2.3 Preventative Measures against IC/FPGA Threats

The vulnerabilities present in the system design and physical manufacturing stages of the IC lifecycle lend themselves to computationally easy attacks for a foundry-level adversary that can circumvent the security of potentially high-value IC designs. As described in Subsection 2.1.1, those factors directly and affirmatively rank a high-level threat. Likewise, the easily accessible configuration data in the end-user deployment state of the FPGA lifecycle poses a threat to the confidentiality of the circuit information. Therefore, it is advantageous for engineers designing security-sensitive IC or FPGA systems to include some collection of preventative measures to mitigate these risks.

The concept of preventative measures against the threats present in the IC/FPGA lifecycle is not new. Ideas such as IP encryption, hardware authentication, logic obfuscation, logic locking, etc., have existed for many years. What makes these

preventative measures challenging to achieve are two fundamental opposing facets: 1) successive parties in the lifecycle must have sufficient knowledge of the design to perform their required duties; 2) but it is in the best interest of the preceding party to withhold the sensitive intellectual property contained within the design. This duality leads to imperfect solutions, as described in the following subsections.

2.3.1 IP Encryption

IP encryption is a tool used between the IP development and system design stages. It protects the IP author from unauthorized use of their IP via the involvement of a third-party tool vendor. Note, the use of the terms "IP author", "IP user", and "tool vendor" in this section originate and derive meaning from [11].

As previously mentioned, it is common for system design teams to reuse preexisting IP blocks. Unfortunately, this convenience introduces a confidentiality risk concerning the fair use of the IP from the IP author's perspective. Ex: What stops an authorized IP user from illegally redistributing the design to non-authorized users? IP encryption is used to mitigate this risk. An IP author may choose to encrypt the entirety of their IP library and rely on a trusted tool vendor to selectively distribute it to the IP users without exposing the raw HDL. This is an example of a trust model [11]. The IP author and users must trust that the tool vendor will maintain the IP's confidentiality, integrity, and availability. Assuming the tool vendor is credible and has sufficient resources to protect and distribute the IP entrusted to them, this trust model does a fine job of preserving the IP development from the system design stage.

However, IP encryption does not remedy the threats present in the IC/FPGA lifecycle after system design. This is because IP encryption is a software solution aimed only at preventing logical IP theft via the IP user, i.e., the system design team. Once the encrypted IP is brought into the IC/FPGA design by an IP user—a process aided via the intervention of the trusted third-party tool vendor—the final

layout or configuration bitstream must still be generated and provided to the fab for physical manufacturing or loaded onto non-volatile memory. Therefore, the IP encryption does not affect the IC layout or FPGA bitstream. As such, physical Trojan insertion and reverse engineering attacks discussed in Subsections 2.2.1 and 2.2.2 are unaffected.

2.3.2 Hardware/IP Authentication

Unlike other preventative measures, hardware/IP authentication does not aim to pre-emptively stop malicious attacks. As the name suggests, hardware/IP authentication aims to authenticate, i.e., verify, the owner of a given facet or the entirety of the design. Hardware/IP authentication can be used in IP development or the system design stages.

A digital signature, such as a hardware watermark, must exist to authenticate a hardware design. A hardware watermark is a "mechanism for identification that is: 1) nearly invisible to human and machine inspection; 2) difficult to remove; and 3) permanently embedded as an integral part of the design" [49]. These characteristics are important because they ensure that if the design is copied, the pirate is unlikely to have noticed the watermark. Furthermore, even if the watermark is identified, removing it would be impractical. Therefore, the watermark will likely remain in the cloned design allowing the rightful author to legally RE the clone (as described in Subsection 2.2.2), confirm the presence of the watermark, and pursue appropriate legal actions against the pirate. The presence of the watermark in the non-licensed design is legally significant because it implies a higher probabilistic proof of authorship [50]. For a watermark to be considered a strong digital signature, it must be statistically improbable for other like designs to exhibit it.

An example of a hardware watermark with high proof of authorship is the approach introduced by A. B. Kahng et al. in [49]. This technique comprises the inclusion

of non-addressable FPGA configurable logic block outputs such that the original functionality of the system is not altered. Various improvements to this approach are discussed in [50], [51], and [52]. This watermarking method provides proof of authorship because, with the configuration of an increasing number of non-functional LUT outputs, it becomes increasingly unlikely that any other implementation would make the same set of random choices. Another popular watermarking technique is "constraint-based", as discussed in [50] and [49].

2.3.3 Split Manufacturing

Split manufacturing only pertains to the fabless IC lifecycle as it is related to the fabrication of ICs at potentially untrusted foundries. In summary, split manufacturing uses two foundries to manufacture an IC. The front end of line (FEOL) layers (the transistors and lower metal layers) are manufactured at the first untrusted foundry. Then, the back end of line (BEOL) layers (top metal layers) are aligned, integrated, and tested on the partially fabricated waver by a second trusted foundry [10].

Since the BEOL metal layers are thicker than the FEOL layers, the BEOL trusted foundry can use an older/larger processing node compared to the FEOL foundry. Hence, a lower upfront investment is required for the trusted foundry. Split manufacturing increases security, without compromising on the smaller processing node, compared to if the entire design was manufactured at the untrusted FEOL foundry.

While split manufacturing has the potential to increase security, the process does increase fabrication complexity, requires additional attention from the system designer, and potentially decreases performance. This is because to prevent proximity attacks, i.e., when an attacker at the FEOL foundry can guess the higher metal connections based on the surrounding pins in the lower layers, system designers must "[rearrange] the pins such that they are no longer closest to the pins that they are supposed to connect [to]" [10]. Therefore if appropriately done, split manufacturing

results in a trade-off between security and simplicity/performance.

2.3.4 Logic Obfuscation

Logic obfuscation is such a popular term that it is often improperly used to describe several *different* types of hardware protection techniques. With that in mind, we agree with S. Engels et al. [21] in that using the term to describe fundamentally different categories of hardware countermeasures is confusing and misleading. To remedy this, all references to the term logic obfuscation will follow the definition given in Subsection 2.1.2–"the transformation of sequential or combinational logic into an alternative but functionally equivalent representation". The phrase "functionally equivalent" distinguishes our definition of the term. Another interpretation of this distinction is provided in work [10] by J. Rajendran et al.; obfuscation is defined as "not prevent[ing] black-box usage".

In accordance with the aforementioned definition of the term logic obfuscation, this thesis will not discuss documented "hardware/logic obfuscation" techniques that alter the functionality of the original design in this section. Instead, these will be classified as logic *locking* techniques and accordingly included in Subsections 2.3.5 and 2.3.6.

Logic obfuscation provides security by transforming logic from one representation to another. The goal of logic obfuscation is to increase the complexity of RE attacks by making the inference of physical hardware structures more challenging.

One example of logic obfuscation is IC gate camouflaging. To perform gate camouflaging, system design teams layout standard cells that look alike irrespective of their functionality [10]. This hinders imaging-based RTL netlist RE because the camouflaged gates are not uniquely distinguishable, resulting in netlist errors. As the number of errors increases, the likelihood of the IC working as intended decreases. Another example of logic obfuscation is the timing camouflage proposed

in [37] that aims to thwart counterfeiting by invalidating the conventional timing model. Pipelined registers are removed to facilitate multiple propagating "waves" of data through a combinational logic block. Therefore, if an attacker obtains the post-obfuscated netlist and uses conventional timing analysis to determine the usable range of clock periods, the resulting system would not function. This is because the netlist was designed to operate at a higher specific frequency which violates conventional setup timing, allowing the proper number of waves to propagate through each block of combinational logic [37].

2.3.5 Combinational Logic Locking

Because the foundry must always possess knowledge of the layout to manufacture the IC, and several ways exist to RE an IC using that information, passing an IC design from the system design team to physical manufacturing without exposing the final circuit is an unattainable goal. Therefore, this transition in the IC lifecycle presents unavoidable security vulnerabilities, particularly confidentiality-related. In response, logic locking aims to limit the amount of intelligible information a foundry has at its disposal while assuming it possesses the locked netlist (attained through RE of the GDSII layout or imaging-based RE attacks on a post-fabricated IC).

Although the previous scenario depicts an attack on the IC lifecycle, logic locking is not limited to ICs. On the contrary, in the FPGA lifecycle, a similar threat exists considering the RE of configuration bitstream, as discussed in Subsection 2.2.2.

Logic locking is defined in Subsection 2.1.2 as the transformation of sequential or combinational logic into a restricted alternative representation which requires the intervention of a key to access the device's functionality. Including a key hinders a potential attacker (such as a malicious foundry) because, depending on the type of logic locking used, the absence of it can prevent black box usage of overproduced ICs and unlicensed FPGA configurations and make it computationally or economically

infeasible to nullify the locking scheme on a netlist level [21].

Note, the terms internal key and chip key, along with their corresponding descriptors individual and global, are also defined in Subsection 2.1.2. This terminology plays an important role in logic locking literature because it defines how the key is handled and, as previously mentioned, what the outcome of an attack that discloses said key will compromise. E.g., a leak of an individual chip key will result in only a single ICs functionality being unlocked, achieving none of the attack goals later listed in Subsection 3.1.2. Whereas a leak of a global internal key will enable the attacker to achieve goals two and three, and a leak of a global chip key will enable all three! As such, the use of these terms will be carefully included in this thesis to best characterize the explored logic locking methodologies.

2.3.5.1 Random Locking

A well-known form of combinational logic locking is that introduced by Jarrod A. Royt et al. in the work "EPIC: Ending Piracy of Integrated Circuits" [23]. The methodology of which involves adding several XOR and XNOR key gates between the original netlist's preexisting gates. One input to these key gates is the original wire (P), the other input is a wire connected to the internal key register (K), and the output is the post-locked wire connected to the original downstream gate (P'). Figure 2.2 illustrates XOR/XNOR key gates. When an invalid key is used, the circuit's behavior is altered, as if stray inverters were placed on the selected wires.

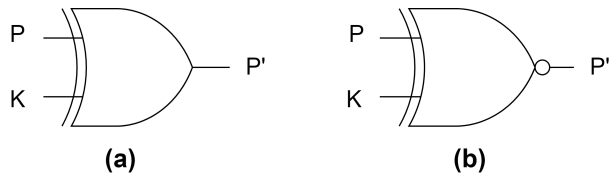

Figure 2.2: XOR (a) and XNOR (b) key gates with original input (P), key input (K), and locked output (P').

The EPIC methodology utilizes key gates placed randomly through the netlist

and defines a complex key distribution network utilizing public key cryptography in combination with a PUF, to generate individual chip keys despite the global internal key [23].

2.3.5.2 Fault Analysis-Based Logic Locking

Although EPIC [23] was influential in sparking the field of logic locking, research progressed, resulting in the discovery of security concerns with the random placement of key gates. One of the first attempts to improve this was using fault analysis-based key gate placement algorithms, introduced by J. Rajendran et al. in [53] and then later revised in [54].

Although not necessarily a security flaw, J. Rajendran et al. was concerned about the low output corruptibility cause by the random placement of gates. "Specifically, to maximize the ambiguity for an attacker, [J. Rajendran et al.'s Fault Analysis] technique targets 50% Hamming distance between the correct and wrong outputs (ideal case) when a wrong key is applied" [54]. To achieve this, the logic locking technique proposed in [53] utilizes fault impact to iteratively place key gates at nodes with the highest potential to propagate a fault to the primary outputs. Fault impact is calculated using Equation 2.1, where NoP_0 is the number of patterns that detect a stuck-at-0 (s-a-0) fault at a particular node and NoO_0 is the number of output bits that are effected given the s-a-0 fault. Similarly, the NoP_1 and NoO_1 terms correspond to the stuck-at-1 (s-a-1) faults [53, 54].

$$\text{Fault Impact} = (NoP_0 \times NoO_0 + NoP_1 \times NoO_1) \tag{2.1}$$

Through iteratively calculating the next node with the highest Fault Impact and placing a lock gate there, this approach is able to near the 50% Hamming distance metric, given a sufficient number of locking gates. However, as will be discussed in Subsubsections 2.3.5.3 and 2.3.5.4, the resultant high output corruptibility of this ap-

proach proved to be undesirable due to other, more pressing, security vulnerabilities.

2.3.5.3 Strong Logic Locking

Future research in the field [55] and [4] uncovered security flaws in the key gate placement strategies of random and Fault Analysis-Based logic locking schemes. In summary, neither random nor Fault Analysis-Based placement of key gates results in a secure locking scheme because when gates are placed without thorough consideration of the interference between each other, there is the potential for runs of key gates, isolated key gates, and mutable key gates [4, 2]. Each compromises the system's security by reducing the number of effective key bits when hill-climbing [55, 3] or key propagation attacks are performed [3].

Runs of key gates reduce the number of effective key bits by introducing additional valid keys. A run exists when a set of key gates are connected without additional logic between them. For every N key gates in a run, the number of valid keys increases by $2^{(N-1)}$ [4, 2]. An isolated key gate occurs when there is no path from a key gate to all other key gates and vice-versa. Isolated key gates are inherently susceptible to key propagation attacks because, after identifying a pattern that uniquely propagates the effect of an isolated key gate's key to the chip's output, an attacker can then apply that pattern to the functional IC, compare the outputs, and deduce the correct key value. Key gate mutability is when the effect of one key gate is prevented from reaching the circuit's output and is permissive to key propagation attacks because it makes it easier to isolate and sensitize a singular key gate at the chip's output as if it was an isolated key gate. Mutability can significantly reduce effective key space and occurs in three configurations: dominating key gates, concurrently mutable convergent key gates, and sequentially mutable gates. A dominating key gate arises when it is in every sensitization path between another key gate and the output. Concurrently mutable convergent key gates are those in which sensitization paths converge at the

same gate, such that the key bits of both key gates can be determined by muting the other. Sequentially mutable convergent key gates are those in which only one of the convergent key gates can be muted due to the mutability caused via overlapping sensitization paths.

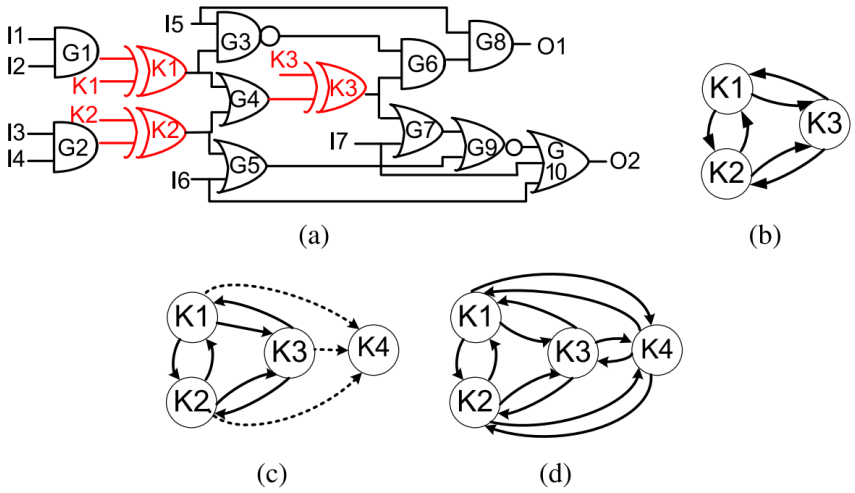

Figure 2.3: "(a) Example circuit with three key gates. (b) Interference graph of the key gates. Nonmutable keys are connected by solid edges. If the new key gate is inserted at the output (c) G10, it creates mutable edges (dotted lines) with the other key gates and (d) G5, it creates nonmutable edges (solid lines) with the other key gates." [2]

One response to the security vulnerability introduced via random and fault analysis-based key gate insertion is that of J. Rajendranj et al., who proposed a new logic locking method called "Strong Logic Obfuscation"[5] in [4, 2]. Strong Logic Obfuscation is a heuristic key gate placement algorithm that minimizes the aforementioned runs of key gates, isolated key gates, and various forms of mutable key gates. This algorithm generates a weighted mutability interference graph of the already placed key gates at each iteration to select the next best location for a key gate. An example of this, sourced from [2], is illustrated in Figure 2.3. Between the two potential placement locations G10 and G5, G5 is a better option because it does not cause mutable edges like that of G10. See Figure 2.3c and Figure 2.3d for the interference

[5]Although this usage of the term obfuscation goes against the definition provided in this thesis, this name was provided unmodified as to not retcon the official title of the methodology decided upon by Rajendranj et al. in [4, 2].

graphs after placement of the next key gate at G10 and G5 respectively. The placement algorithm continues placing gates until no more potential locations exist that prevent mutability and the desired number of cliques have been generated. A clique is a subgroup of nonmutable key gates, i.e., a group of key gates which are resilient to key propagation attacks. Only considering key propagation attacks, the number of key gates in a clique increases the effective security exponentially, and the number of similarly sized cliques in a locked circuit increases security linearly.

Although cliques of key gates were considered resilient to key propagation attacks and therefore require brute force to solve each clique's key bits by J. Rajendranj et al. [4, 2], future work by that of Yu-Wei Lee et al. shows that is not the case [56]. Using logic cone analysis, the effective key size of these cliques is reduced significantly by iteratively considering the least secure logic cone within each clique, therefore reducing the assumed single exponential brute force attack to a sum of smaller exponential attacks. To remedy this, Yu-Wei Lee at al. suggests an amendment to Strong Logic Locking by including a percentage of MUX key gates, as described in [56], therefore increasing the logic cones within each clique.

2.3.5.4 SARLock, Anti-SAT, and LUT-Lock

The SARLock [5], Anti-SAT [6], and LUT-Lock [57] logic locking methodologies are unique techniques introduced by independent research groups. Despite that, they will be grouped together for the scope of this thesis. The reason for this is that they all aim to minimize the effectiveness of satisfiability (SAT) attacks by pursuing the same end goal–albeit through different implementations.

Before describing the fundamental goal behind these logic locking methodologies that increases their resistance to SAT attacks and how they achieve that through their specific implementations, a brief overview of the SAT attack will be provided. The advent of an SAT attack vector on logic locking, popularized by P. Subramanyan

et al. [58], was revolutionary and devastating to the security of previously proposed logic locking techniques. Of the 441 encrypted circuits examined in [58], which utilize various combinational logic locking methodologies [23, 53, 56, 59], 418 (95%) were successfully unlocked within 10 hours of compute time. Furthermore, considering only the 21 circuits with a reasonable $\leq$5% area overhead allocated to locking logic, all attacks were successfully executed in fewer than 8 minutes [58]. The strength of SAT solvers comes from their Conflict-Driven Clause Learning ability. In each recursive iteration of the SAT attack, a new SAT problem is defined using the Conjunctive Normal Function (CNF) representation of the miter-like circuit under test, as shown in Figure 2.4, in addition to the previously found literal conflicts. The goal of the SAT solver is to find a satisfying value for all its literals.

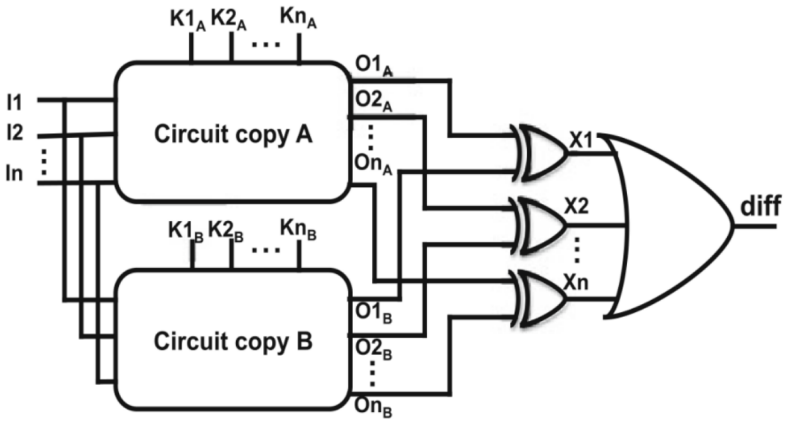

Figure 2.4: "Miter-like circuit used to determine distinguishing input patterns (DIPs)" [3] during satisfiability (SAT) attacks.

During each iteration of the SAT attack process, the Figure 2.4 miter-like circuit essentially compares the output of two copies of the encrypted circuit and finds a distinguishing input pattern (DIP) [3]. A DIP is defined as an input vector where for two different keys K_1 and K_2, the output of the locked circuit is different [58, 3]. If no DIP can be found, that signifies the equivalence class comprised of one or more keys that unlock the circuit has been reached. Each iteration invokes the Davis-Putnam-Logemann-Loveland (or derivatives thereof) algorithm, which is recursive in nature.

Through iteratively finding new DIPs and adding conflicts to the CNF during each iteration's recursive calls, keys in the key space are separated into equivalence classes and pruned in groups, drastically reducing the time required to obtain a functional key. A conflict is found when the output of the locked circuits differs from that of the unlocked copy (presumably obtained on the open market). For a more detailed description of the SAT attack process and algorithm, see the work of P. Subramanyan et al. [58].

In response to the onset of SAT attacks, SARLock [5], Anti-SAT [6], and LUT-Lock [57] aim to provide SAT resistance by increasing the number of iterations needed during the SAT attack. To do this, the size of each equivalence key class must be minimized–where the best case scenario is one key per class. This elicits the maximum number of DIPs, and hence rounds, required to solve for the key. As the size of each equivalence key class approaches one, the complexity of the SAT attack approaches $2^{|K|}$, where $|K|$ is the number of key bits, meaning that the complexity approaches that of a brute force attack.

SARLock [5] and Anti-SAT [6] are similar in construction. At their core, they both rely on structures that compare the primary inputs of the locked combinational circuit to judiciously flip one of the primary outputs. They reduce the size of each equivalence key class by flipping the output bit for only one DIP given each incorrect key.

SARLock specifies using a generic comparator circuit to flip the output bit only when the selected input bits are equal to that of the key and a mask to prevent flipping when the valid key is provided. To prevent removal attacks, it is also recommended that Strong Logic Locking [4, 2] is used alongside SARLock and that a scrambler is used to mix the key bits and prevent the comparator and the mask circuits from leaking explicit information about the key. A single instance of the recommended two-layer SARLock circuit structure is shown in Figure 2.5 and is sourced from the

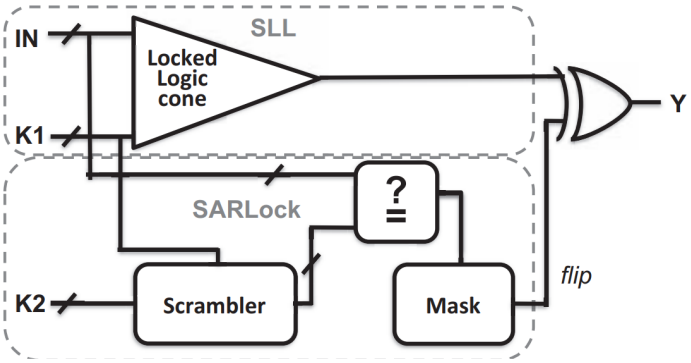

Figure 2.5: "SARLock+SLL: two-layer logic locking. $|K1|$ key bits are used for SLL [4, 2] and $|K2|$ key bits for SARLock." [5]

original SARLock research paper [5].

Anti-SAT specifies the use of complementary functions g and $\bar{g}$ to compare the key and primary inputs. Removal attacks are addressed through the proposed inclusion of wire entanglement and circuit withholding [60].

Wire entanglement is the process of using an interconnected network to hide the mapping of N input signals onto a set of $< N$ output signals. The network's programming can be interpreted as the key which determines the mapping. With the linear inclusion of "noise", i.e., unnecessary inputs into an entangled circuit, the attack complexity on finding said key grows exponentially [60]. Circuit withholding is the process of replacing a cloud of logic gates with a reconfigurable logic block, akin to the LUT fabric within an FPGA, such that, without the programming of the reconfigurable logic block (held by the trusted IP rights holder), a foundry, or anyone with access to the RE netlist, has no knowledge of the required circuitry. Note circuit withholding has also been proposed as the sole source of logic locking in methodologies such as Reconfiguration Logic Barriers [59].

Two configurations of the Anti-SAT block are detailed in [59] and allow for the newly inserted key gates to act as inverters or buffers in their unlocked state. For brevity, Figure 2.6 illustrates the two Anti-SAT block configurations without wire entanglement and circuit withholding. For further information on the Anti-SAT logic

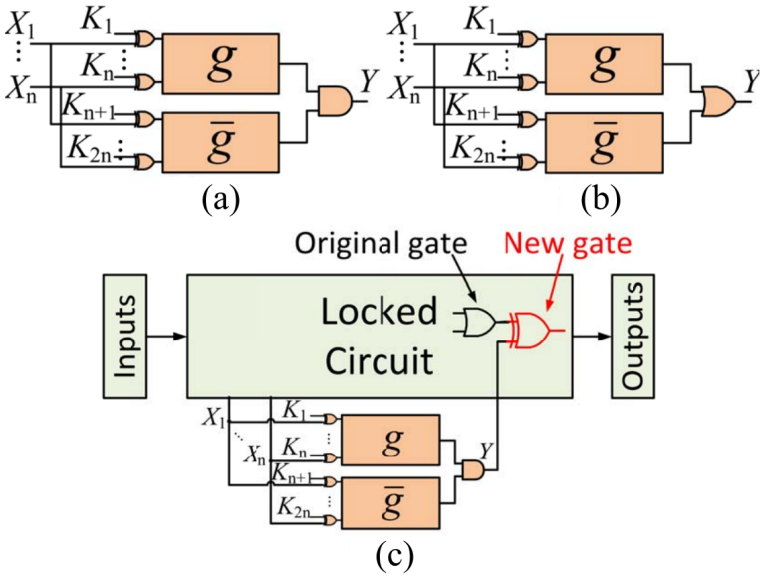

Figure 2.6: "Anti-SAT block configuration. (a) Type-0 Anti-SAT: always outputs 0 if key values are correct. (b) Type-1 Anti-SAT: always outputs 1 if key values are correct. (c) Integrating the Type-0 Anti-SAT block into a circuit." [6]

locking methodology, see [6], and likewise, for further discussion on specifically the wire entanglement and circuit withholding strategy, see [60].

LUT-Lock [57] similarly aims to achieve SAT resistance by increasing the number of iterations required but does so purely through a heuristic placement algorithm of LUTs within the combination logic block. The programming of these LUTs acts as the key, and the surrounding combinational logic is absorbed into the LUTs to integrate the LUTs into the design. An in-depth discussion of the heuristics placement strategy will not be provided in this thesis. Instead, a list of the five LUT-Lock heuristic algorithms which result in the SAT-resistant properties will be listed below:

1. FIC: Focusing on the Fan-In Cone of minimum number of primary output
2. HSC: Focusing on Higher Skew Gates in FIC
3. MFO-HSC: Focusing on gates with Minimum Fan-Out
4. MO-HSC: Focusing on Gates with least impact on POs

5. NB2-MO-HSC: Avoiding Back-to-Back insertion of LUTs

The [57] paper defining the LUT-Lock methodology also discusses additional security measures which can be performed on the LUT gates, including the connection of unused inputs to an internally implemented Non-Linear Feedback Shift Register (NLFSR) or a PUF. Although these measures were not directly considered when asserting the SAT resistance of the LUT-Lock methodology, they surely have their own benefits, including the backing for an individual internal key, and hence individual chip key.

A shared disadvantage to all logic-locking methodologies which aim for SAT resistance through the minimization of equivalence key class size is reduced output corruption. This dichotomy is unavoidable due to the fundamental relationship that output corruption has on the number of keys pruned at each SAT iteration [5, 6, 57]. As such, other forms of SAT resistance logic locking methodologies have been proposed, including those not translatable to SAT problems, such as SRCLock [61] and those which aim to increase the execution time of each SAT iteration, such as Full-Lock [7]. The latter of these, Full-Lock, will be discussed in the following paragraphs.

2.3.5.5 Full-Lock

The Full-Lock [7] logic locking methodology is a different interpretation of an SAT attack-resistant logic locking scheme. It is excluded from the previous paragraphs on SARLock [5], Anti-SAT [6], and LUT-Lock [57] because, unlike them, it does not aim to increase the number of SAT attack iterations. Instead, it relies on increasing the execution time of each SAT round. This is an important distinction because, as previously mentioned, a compromise must be made with output corruptibility to reduce the number of required SAT rounds. Therefore, by increasing the difficulty of each round, Full-Lock retains a higher output corruptibility without sacrificing SAT attack resistance.

Full-Lock is constructed using a set of small-size fully Programmable Logic and Routing blocks (PLR) networks. Each PLR relies on a key-configurable logarithmic-based network (CLN) to obfuscate routes and provide SAT resistance, as well as a group of LUTs to integrate the leading combinational logic into the PLR. This combination is very similar to the work "IC Piracy Prevention via Design Withholding and Entanglement" [60] because the CLN is simply a form of wire entanglement, and the group of LUTs is design withholding.

The CLNs comprise cascaded lightweight switch-boxes, which were carefully selected after extensive research and development to ensure the SAT resistance of the overall CLN. The chosen almost non-blocking CLN with 64 inputs allows only five iterations of SAT attack to be completed within 2×10^6 seconds [7]. When the PLR is inserted into a circuit design, it provides SAT-resistant locking characteristics due to the key controllable CLN and programmable LUTs. Figure 2.7, sourced from [7], illustrates the insertion of a Full-Lock PLR block. Figure 2.7c refers to a cyclic structure; this is in reference to another logic locking methodology [61] and is provided to show that Full-Lock may result in combinational cycles. Which, although initially believed to be SAT-resistant, have been defeated using cycSAT [62]. Nonetheless, the possible addition of cycles increases the specialty of the required SAT attack.

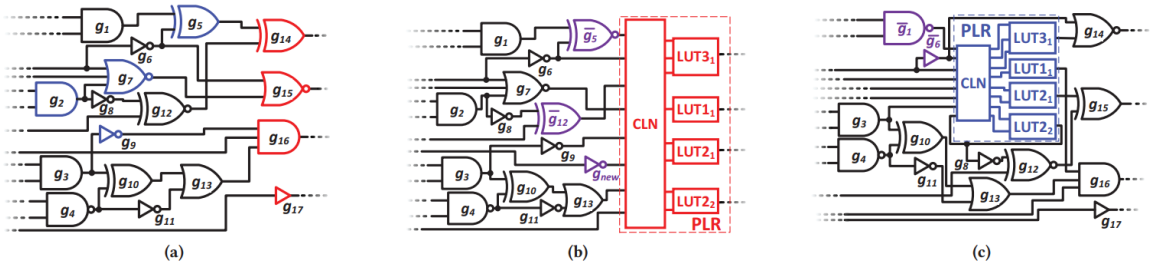

Figure 2.7: "PLR Insertion Example: (a) Gate-level of Original Circuit. (b) Adding PLR and Negating leading Gates with (b) Acyclic Structure, (c) Cyclic Structure." [7]

Lastly, it should be noted that the FSMLock methodology takes the same approach to SAT resistance as Full-Lock and other logic locking approaches that use the existence of SAT hard blocks to reduce the feasibility of SAT attacks. Although

SAT attacks such as [58] have not historically applied to sequential logic locking methodologies, such as FSMLock, it is still important to consider, given the recent advancements in circuit unrolling-based SAT attacks on sequential logic such as [63] and more recently improved in Fun-SAT [64] and RANE [65]. The former, Fun-SAT [64] solver tool demonstrates an on average 90x faster runtime than [63], and the latter RANE [65] is readily available to attackers given it is open source. Considering these advancements, a discussion on SAT resistance will be included in Section 3.2.3 alongside other FSMLock security-related claims.

2.3.6 Sequential Logic Locking

The previously mentioned forms of logic locking act on combination logic, i.e., logic without the inclusion of memory components such as flip-flops. Considering that the FSMLock methodology is a form of sequential logic locking, it is important to remark on the existence of previously proposed sequential logic locking techniques. Four methodologies will be covered: HARPOON [66, 67], Dynamic State Deflection [68], Hardware Nanomites [8], and ReTrustFSM [69].

2.3.6.1 HARPOON and Dynamic State Deflection

The HARPOON [66, 67] logic locking methodology is one of the earlier forms of sequential logic locking. HARPOON aims to prevent unauthorized use and RE of the sequential logic. To achieve this, the methodology species the addition of states to the state transition graph (STG), which must be navigated through at startup using a specific sequence of inputs (effectively the key) to enter the normal functionality of the sequential circuit.

In addition to locking the sequential logic, later revisions of the HARPOON methodology [67] specify the inclusion of additional authentication states. Similarly to the added "obfuscation" states which must be navigated through to unlock the

design, the authentication process requires a sequence of inputs to be provided to navigate through the set of authentication states. While doing so, a particular pattern appears at a subset of the primary outputs. This pattern acts as a digital watermark, as discussed in Subsection 2.3.2. An example FSM locked with HARPOON is illustrated in Figure 2.8, sourced from the work of [8].

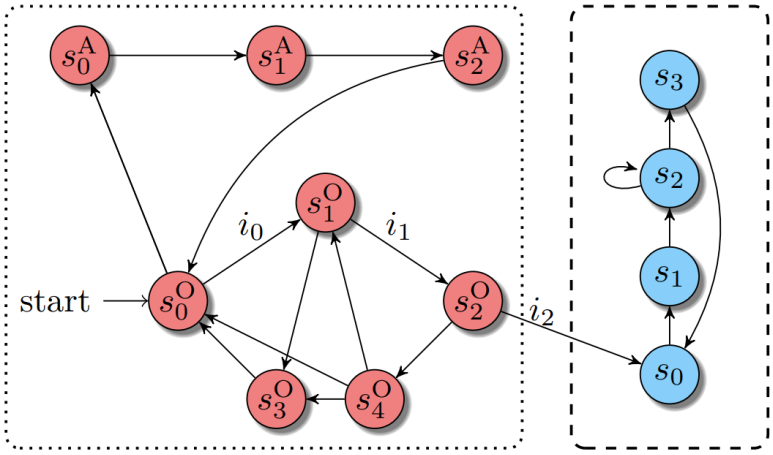

Figure 2.8: "HARPOON design methodology example. The original FSM (dashed blue part) is augmented by an obfuscation [sic] mode ${s_0}^O$, ${s_1}^O$, ${s_2}^O$, ${s_3}^O$, ${s_4}^O$ and an authentication mode ${s_0}^A$, ${s_1}^A$, ${s_2}^A$. The enabling key to reach the original initial state s_0 is (i_0, i_1, i_2)." [8]

Attacks on the HARPOON methodology have been presented. One such attack initial state patching [8] relies on the identification of the structural characteristics of the locked FSM. In such an attack, the original FSM can be structurally identified in the RTL netlist (presumably obtained via the RE attacks discussed in Subsection 2.2.2), then the RTL, mask, or configuration bitstream can be tampered with to set the initial state of the FSM to the unlocked state. Therefore circumventing the need to navigate through the locking state space.

Other research groups have amended the HARPOON methodology with the aim of preventing the aforementioned initial state patching attack. One such amendment is that of Dynamic State Deflection [68]. Dynamic State Deflection utilizes the same additional "obfuscation" and authentication states but introduces the use of isolation

black hole states. If an incorrect sequence of input values is entered, then the FSM transitions into an isolated state space that is impossible to escape. Furthermore, the correct key must be provided during all other transitions in the normal mode of operation, or else the locked FSM will transition into the black hole states. An example FSM locked with Dynamic State Deflection is illustrated in Figure 2.9, sourced from the work of [8].

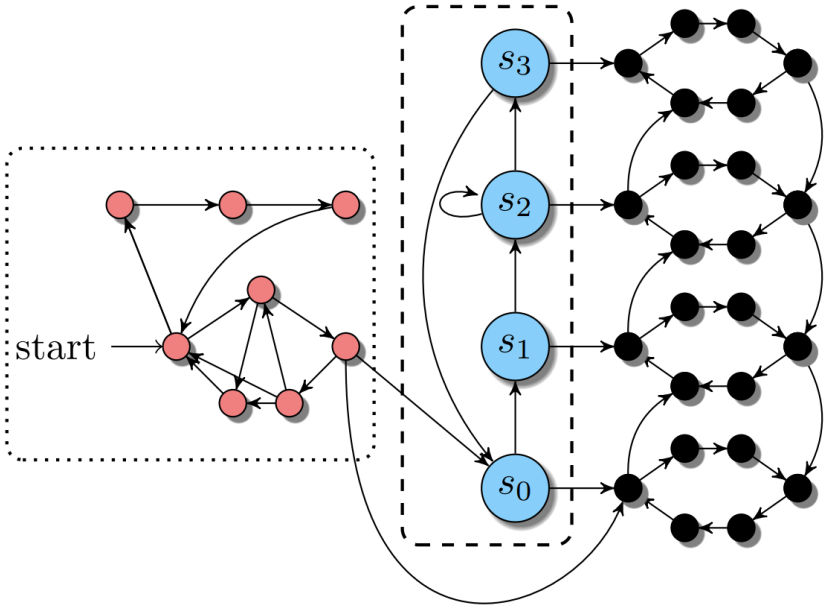

Figure 2.9: "Dynamic State Deflection design methodology example. The original FSM (dashed blue part) is augmented by an HARPOON obfuscation mode (dotted red part) and each original state is protected by a black hole (states marked in black)." [8]

Although the Dynamic State Deflection methodology aims to prevent initial state patching, as discussed in an overview of the existing FSM-based hardware locking methodologies provided by M. Fyrbiak et al. [8], it too is susceptible to attacks if the structural and functional characteristics of the RE FSM are considered. This is due to the advent of powerful FSM extraction tools such as [45, 46, 47], discussed previously in Subsection 2.2.2. Even other more recent HARPOON-based FSM locking methodologies, such as Active Hardware Metering [70] and Interlocking Obfuscation [71], which achieve higher resistance to initial state patching, also have flaws that can

make them susceptible to RE attacks given the current sophistication of RE toolsets available [8].

2.3.6.2 Hardware Nanomites

In response to the evaluated bleak security characteristics of previously introduced sequential logic locking techniques, M. Fyrbiak et al., alongside their review, proposed the novel methodology Hardware Nanomites [8]. The premise of Hardware Nanomites is simple, instead of relying on the difficulty of RE an FSM from a flattened netlist to secure a sequential circuit, circuit withholding, as defined previously in the review of the Anti-SAT methodology, can be performed. Specifically, Hardware Nanomites specifies the use of partial dynamic reconfiguration, a technology provided in several modern FPGA devices [26, 27], to selectively and dynamically load and overwrite partitions of the FSM into the FPGA fabric at run-time [8].

The security claims behind this design are that it is difficult to access, RE, and recombine the static partial bitstream configurations due to the missing mapping between them. Therefore static analysis techniques, such as the FSM extraction tools [45, 46, 47] used in the evaluation of other methodologies, would be exceedingly difficult for an attacker to perform. Furthermore, dynamic simulation-based attacks are hindered due to the "lack of an efficient gate-level simulation model for partially reconfigurable FPGA designs" [8].

Although we are hesitant to agree with the validity of these security claims since known RE attacks do exist against FPGA bitstream configurations [15], through RE-ing of the reconfiguration controller logic, a mapping between partial reconfiguration bitstreams can be derived, and research in the field of partial dynamic reconfiguration simulation is ongoing [72] (and supported by some mainstream EDA toolsets such as Quartus by Intel [73]), the premise behind Hardware Nanomites is sound and similar to that of FSMLock. Our interpretation of run-time variable circuit withholding for

the FSMLock methodology, which addresses these issues, will be provided in Chapter 3.

2.3.6.3 ReTrustFSM

The final sequential logic locking methodology which will be reviewed in this thesis is ReTrustFSM [69]. ReTrustFSM is another attempt to remedy the security concerns of the previously introduced methodologies. Mainly, RETrust aims to increase the resistance against combined structural and functional attacks and I/O query-based attacks, such as sequential SAT solvers, by including the output of a linear feedback shift register (LFSR) and a down counter in the next state transition logic. At run-time, after the FSM transitions through a subset of the original states (preFSM states), it reaches the newly inserted encFSM states, enabling the LFSR and down counter. When the down counter reaches 1, the LFSR halts shifting, and the current LFSR value is used as the FSM's next state. The state encoding represented by the final LFSR value is called the lockedFSM because it is the following state to the preFSM in the original FSM STG and is unreachable without the intervention of the inserted LFSR and counter logic. Finally, the FSM can freely transition through the remainder of the original states (postFSM). The LFSR seed and initial counter value are the explicit external secrecy (key) for the logic locking methodology. For a more detailed discussion on implementing the ReTrustFSM sequential logic locking methodology, see [69].

ReTrustFSM is included in this review of sequential logic locking methodologies because of its inclusion of, and convenient definition of, explicit external secrecy. As defined in [69], explicit external secrecy "requires an additional variable/input, referred to as the key". This differs from other sequential logic locking methodologies, such as HARPOON [67] and Dynamic State Deflection [68], which use implicit external secrecy, i.e., when the locked circuit is still only dependent on the primary inputs

but now is subject to an unlocking/activation sequence to reach the normal mode of operation.

The sole use of implicit external secrecy presents a concern due to the recent availability of powerful sequential SAT solvers such as FUN-SAT [64] and RANE [65]. These are capable of circuit unrolling and can therefore be used alongside model-checking to solve for an initialization key sequence at the primary inputs. To prevent this, ReTrustFSM relies on both explicit external secrecy (the LFSR seed and counter initiation value) and implicit external secrecy through the cycle-sensitive input sequence required to traverse the inserted encFSM states. Furthermore, by including the encFSM states after the preFSM states, ReTrustFSM increases the difficulty of the aforementioned sequential SAT attacks by increasing the required circuit unrolling depth.

Chapter 3

Methodology

3.1 Attacker Model

Before laying out the methodology for the FSMLock logic locking primitive, it is vital to reflect on the problem we aim to solve with this thesis and its related deliverables. As discussed in Section 1.2, this project aims to develop an automated toolset for the novel sequential logic locking FSMLock primitive and model its theoretical resource utilization and security. To quantify the security claim, we must define whom we consider the potential adversary via an attacker model. Without an attacker model, one cannot classify or rank the vulnerabilities in a system per their corresponding threat. Likewise, without a properly defined attacker model, one is ill-equipped to judge the effectiveness a preventative measure may or may not provide against a potential attack. In the article "A critical view on the real-world security of logic locking" by Engles et al., there is a harmonious discussion regarding the importance and absence of a consistent attacker model in logic locking literature. In light of this, we seek to openly and definitively classify ours in the following subsections.

We will delineate our attacker model with two critical attributes: adversarial assets and attack goals.

3.1.1 Adversarial Capabilities and Assets

As discussed in Section 2.2, many threats exist throughout the IC design lifecycle. The most severe of these threats is during the physical manufacturing process because foundries possess the tools, knowledge, and opportunity necessary to perform post and pre-manufacturing RE of the IC's netlist [38, 40], layout modifications [30], minimal mask modifications [21], Trojan insertion [28, 30], and even invasive probing of the internal signals of the manufactured IC or programmed FPGA [21, 74, 75]. Compared to other malicious parties, the differentiating factors that enable a malicious foundry to achieve these attacks are their capability to perform invasive attacks and their knowledge of the technology and cell libraries (that they incipiently produce). Moreover, malicious parties in other stages of the IC design lifecycle are not realistically capable of performing invasive attacks because the tools necessary are prohibitively expensive. The cost of operating a modern fabrication plant, including the tools used in invasive attacks, is upwards of $5 billion [10, 12].

For these reasons, while we consider a malicious foundry the most potent adversary to a logic locking scheme, we do not aim to fully secure our design against them. With the ability to perform invasive probing on the internal run-time (while the chip is powered on and in use) signals of an IC or FPGA, there is practically no piece of information out of reach. Instead, we aim to provide a robust logic locking methodology that maximizes output corruptibility without compromising the effective attack complexity that is secure against non-invasive, semi-invasive, and non-probing-based invasive attacks, as defined by [75]. As such, we will assume the capabilities of a malicious foundry, excluding run-time invasive probing attacks like those described in [74] when examining attack vectors. The described attacker is assumed to have the following assets at their disposal.

A1. The gate-level netlist and static memory contents of the locked design obtained via RE

A2. Multiple locked ICs or locked FPGA bitstreams obtained during regular production

A3. Multiple unlocked IC or FPGA systems obtained on the consumer market

A4. Access to the fabrication, including artifacts, such as the lithographic masks used to manufacture ICs

A5. State-of-the-art IC analysis equipment, i.e., testing equipment and tools to perform invasive analysis *excluding* run-time invasive probing against the target IC or FPGA

These assets are a modified and abridged version from the work of Engles et al. [21]. Modifications were required to consider FPGA systems and exclude invasive run-time probing attacks previously permitted. While this reduction in attacker strength may contradict what is desired in [21], we see the exclusion of run-time invasive attacks as a regrettable necessity. Protecting locked circuits against invasive probing attacks is out of the scope of this research, and we steer those looking to do so to other literature such as [74] and [75].

3.1.2 Attack Goals

An attack goal is the desired outcome of an attack. Three modified attack goals from the work of Engles et al. [21] will be used for this thesis.

G1. The adversary is able to unlock arbitrarily locked circuits without design modification

G2. The adversary is able to interfere with the fabrication process or bitstream configuration to disable or weaken the locking scheme, thus enabling the production of unlocked ICs and use of locked FPGA bitstreams after the inclusion of the adversary-induced modification

G3. The adversary is able to nullify the locking scheme on the netlist level, thus obtaining an effectively unlocked netlist

An IC/FPGA design is vulnerable to malicious hardware attacks if the adversary can accomplish one or more of these attack goals. Accordingly, the success of these three attack goals will be evaluated against the FSMLock primitive to quantify its security against existing attack vectors in Subsection 3.2.3. Only attack scenarios within the capabilities and assets of those discussed in Subsection 3.1.1 will be considered during this evaluation.

3.2 Design Outline

The FSMLock approach to ensuring the security of an IC/FPGA design throughout its lifecycle is modeled off of the sequential circuit shown in Figure 3.1. This figure illustrates a finite state machine (FSM). An FSM is a mathematical model that abstracts any sequential circuit into a finite set of potential states. A logical expression of the current state s and inputs a dictates which state the model will transition into next s_{next}. Likewise, a logical expression dictates the FSM's output y. For a Mealy FSM, the output depends on both the current state and the inputs; for a Moore FSM, the output depends solely on the current state. In hardware implementations of FSMs, it is common to use flip-flop registers as the state memory components and combinational logic gates to calculate the circuit's next state and output.

The first formal definition for an FSM used in this thesis is a set of state transitions

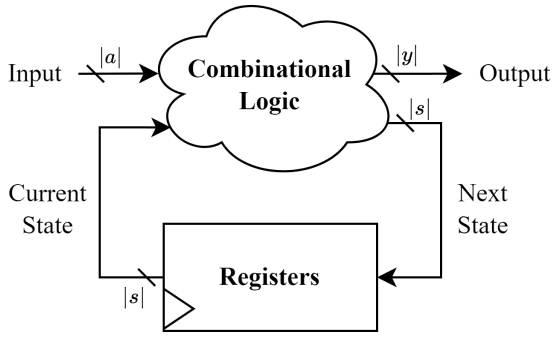

Figure 3.1: The abstraction of a sequential circuit as a finite state machine (FSM).

T, as defined in Equation 3.1.

$$T = \{t_0, t_1, ..., t_n\} \; s.t. \; (a, s, s_{next}, y) = t_n \in T \tag{3.1}$$

This definition is important because it is the input to the toolset described in Section 3.3. Each state transition $t_n \in T$ is defined as a 4-tuple (a, s, s_{next}, y) where a is the input that initiates the transition from state s to state s_{next} and y is the output during this transition. In the following, let $t_n.a$ denote the a element of the tuple t_n, i.e, the input which activates transition t_n. Similarly, let $t_n.s$, $t_n.s_{next}$, $t_n.y$ denote the state, next state, and output, respectively. Note that for a Moore FSM, all transitions that exist in T with the same current state s must have the same output y. Symbolically this definition for a Moore FSM is $\forall t_n, t_m \in T, \big(t_n.s = t_m.s \implies t_n.y = t_m.y\big) \iff$ "Moore FSM". It will be important for the designer to keep this definition in mind so that the intended Mealy/Moore FSM is distinguishable in the state transition table toolset input, as discussed in Section 3.3.

Figure 3.2 illustrates a state transition graph (STG) for an example Mealy FSM. We will use this STG to generate the corresponding state transition table (STT) in Table 3.1. Each node in the STG represents a state in the FSM, and each directed edge represents a transition. To generate the STT, where the set of rows is the state transition set T, each edge of the STG must be translated into a state transition

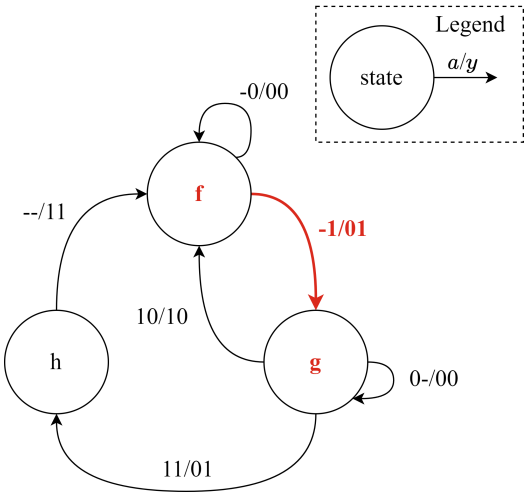

Figure 3.2: Mealy FSM example state transition Graph (STG). The bold red text is included for references made in Table 3.1 and Table 3.2.

$t \in T$. The label of the node which the edge points away from is the current state s, and the label of the node which the edge points toward is the next state s_{next}. The edge label is the input a, which initiates the transition and the output y during that transition; these are in the format a/y.[6] For some transitions, the symbol '-' is used as a value in the input binary string. This symbol represents a "don't care" input bit and could be 0 or 1 for the transition to occur. Therefore, for each don't care bit that exists in the input string, the effective number of absorbed transitions is doubled.

Table 3.1: State transition table (STT) of Mealy FSM STG example shown in Figure 3.2. The bold red text is in reference to that in Figure 3.2 and emphasizes the one-to-one connection between the edges of the STG and transition rows in the STT.

a	s	s_{next}	y
-0	f	f	00
-1	**f**	**g**	**01**
10	g	f	10
0-	g	g	00
11	g	h	01
--	h	f	11

Variants of an FSM are shown in Figure 3.3. In both Figure 3.3a and 3.3b, the

[6] In the STG representation of a Moore FSM the shared output y for all transitions leaving a state may instead be included on the second line of the node label.

combinational logic is replaced with a non-volatile memory (NVM) which operates as an addressable lookup table (LUT) of state entries, as will be defined in the following paragraph, collectively referred to as the state entry table (SET). On their own, the Figure 3.3 structures could be considered forms of FSM logic obfuscation, as defined in Subsection 2.1.2, because they provide functional equivalence with the original FSM structure shown in Figure 3.1 without restricting access. The memory-based LUT structure used in Figure 3.3a and 3.3b, used in place of combinational logic gates to calculate the next state and output values, aligns closely with the work [76] and [77]. But while the authors of [76] proposed using block memory devices to implement such LUT structure as a power-saving measure, we will elaborate on its potential as a logic locking primitive.

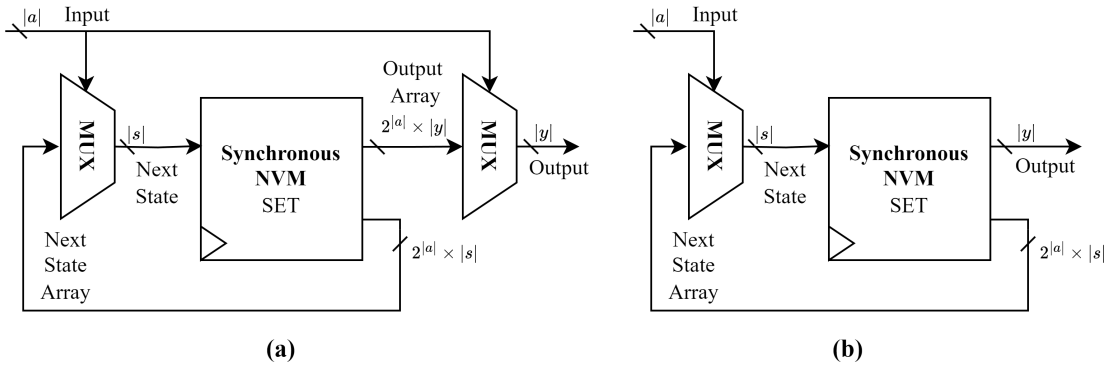

Figure 3.3: Variant of an FSM utilizing synchronous non-volatile memory (NVM), which operates as an addressable lookup table (LUT) of state entries–referred to as the state entry table (SET)–while targeting the (a) Mealy (Figure 3.4a) or (b) Moore (Figure 3.4b) state entry partitioning.

Although an individual state entry does not bijectively map to an individual state transition, the set of all state entries $E = \{e_0, e_1, ..., e_n\}$ does provide an alternative method of uniquely identifying FSMs. As such, set E is the second definition for an FSM used in this thesis, as defined in Equation 3.2.

$$E = \{e_0, e_1, ..., e_n\} \; s.t. \; (s, S_{next}, Y) = e_n \in E \tag{3.2}$$

Each state entry $e \in E$ represents an individual state and is a 3-tuple (s, S_{next}, Y) consisting of the current state s, an N-tuple of the next states $S_{next} = (s_0, s_1, ..., s_N)$, and an N-tuple of outputs $Y = (y_0, y_1, ..., y_N)$ where N is the number of unique inputs 2^a. In the example shown in Figure 3.2, $N = 2^2 = 4$ because the input a is encoded by a two-bit binary string ($|a| = 2$). The ordering of the S_{next} and Y N-tuples is significant because the next state and output values are respectively dependent on the elements $S_{next}.s_a$ and $Y.y_a$, indexed via the subscripted input a. The corresponding state entry table (SET), where the set of rows is the state entry set E, for the Figure 3.2 FSM example, is shown in Table 3.2. Note that the output y_a is not constant for all a values, confirming that the Figure 3.2 example is Mealy. On the other hand, Moore FSM outputs are not dependent on the input a, therefore all y values within Y for each state entry that exists in the set E of all state entries must be equivalent: $\forall e_n \in E, \forall y_n, y_m \in e_n.Y, (y_n = y_m) \iff$ "Moore FSM".

Table 3.2: The state entry table (SET) for the Mealy FSM STG example shown in Figure 3.2. The bold red text is in reference to that in Figure 3.2 and shows that there does not exist a bijective relationship between transitions in the STT, shown in Table 3.1, and entries in the state entry table (SET).

	$S_{next}.s_a$				$Y.y_a$			
s	$a = 00$	$a = 01$	$a = 10$	$a = 11$	$a = 00$	$a = 01$	$a = 10$	$a = 11$
f	f	**g**	f	**g**	00	**01**	00	**01**
g	g	g	f	h	00	00	10	01
h	f	f	f	f	11	11	11	11

This thesis uses two definitions for an FSM because, as shown in Figure 3.3 and Figure 5.2, it is natural to model the hardware implementation of a LUT-based FSM as a set of state entries E. However, it is more convenient for a system designer to provide the FSM as a state transition set T as the toolset input since don't care bits can be used to reduce the number of input fields. Also, the transition set T is more easily identifiable given its bijective relationship with the edges of the STG. The ability to map a set of state transitions T to a set of state entries E using the

bijective function $f : T \mapsto E$ plays a role in the automation process of the toolset, as discussed in Section 3.3.

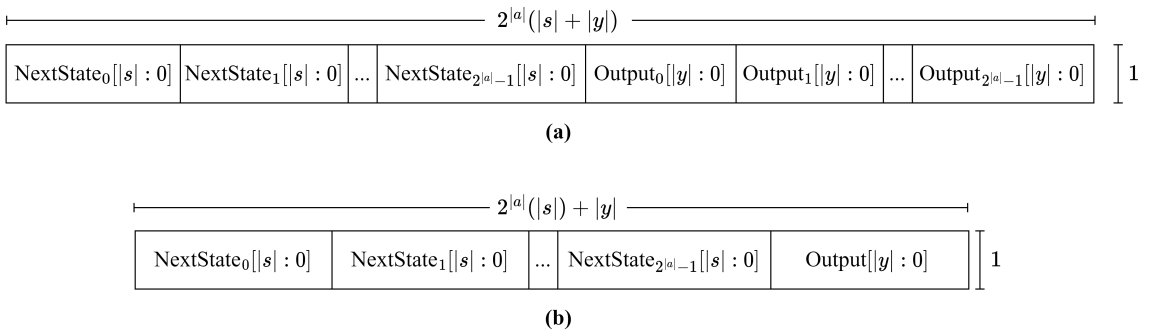

Figure 3.4: (a) The Mealy state entry partioning with $2^{|a|}$ next-state partitions and $2^{|a|}$ output partitions. (b) The Moore state entry partitioning with $2^{|a|}$ next-state partition and one output partition. Sizes are defined in terms of state bits $|s|$, output bits $|y|$, and input bits $|a|$.

In the Figure 3.3 variant of an FSM, the address of the state entry is used to encode the current state s, i.e., the current state encoding s is used to address the state entry in the LUT of state entries. Therefore only the S_{next} and Y N-tuples must be stored at each memory location. To realize this, we propose the state entry partitioning shown in Figure 3.4a. This partitioning structure was chosen for its simplicity: the list of potential next states, sourced from S_{next}, followed by the list of potential outputs, sourced from Y. For the Moore state entry partitioning shown in Figure 3.4b, the list of outputs is replaced with the single value for that state. Functionally this distinction is not required since any Moore FSM could be modeled using the Figure 3.4a structure. However, Figure 3.4b is more memory efficient when modeling target Moore FSMs because the Y N-tuple of equivalent outputs can be replaced with a single copy.

Ultimately, the novel FSMLock logic locking primitive realized in this thesis emerges from the Figure 3.3 variant of an FSM and is directly inspired by the work [1]; if one could encrypt the memory contents of Figure 3.3, the identifiable next-state transitions S_{next} and output assignments Y would be incomprehensible to those

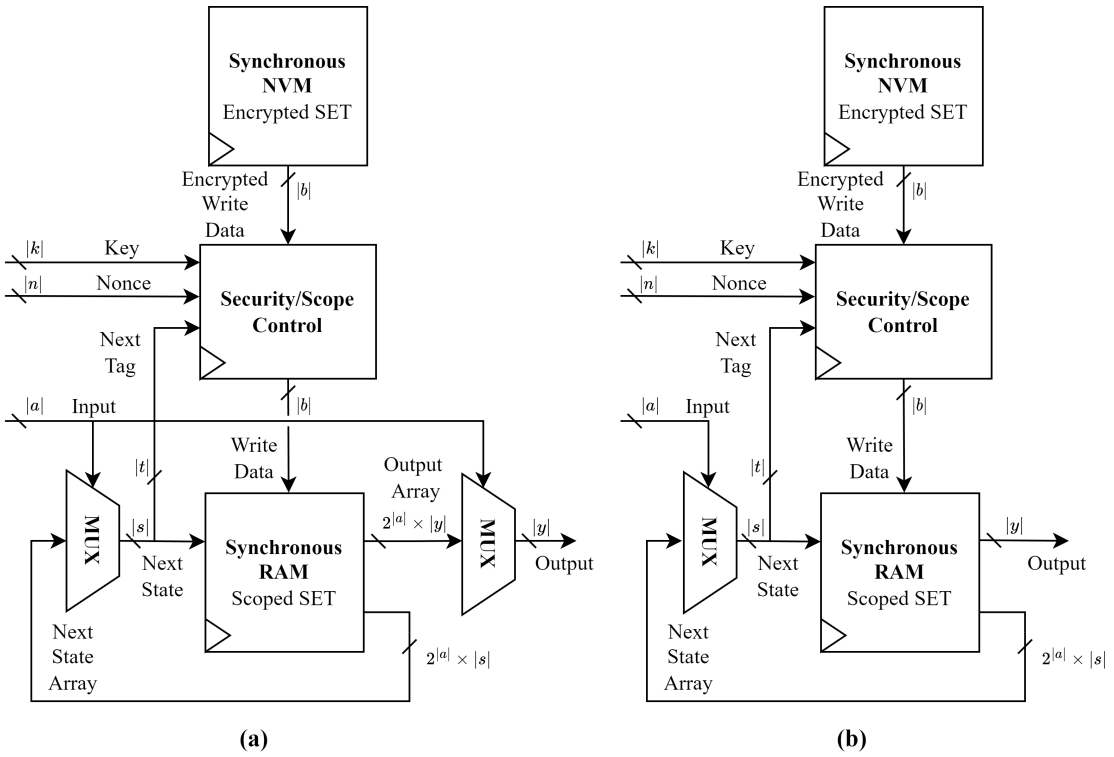

Figure 3.5: The model of the FSMLock primitive utilizing synchronous in-scope random access memory (RAM) while targeting the (a) Mealy (Figure 3.4a) or (b) Moore (Figure 3.4b) state entry partitioning.

fabricating the IC and inoperable by those attempting to use production units of a IC/FPGA without knowledge of the key. Furthermore, suppose only portions of the fully functional FSM are accessed at a given time through a scope control entity. In that case, this model can conceal even the number of reachable states at run time and prevent the entirety of the sequential logic from being available at once, further complicating a potential removal attack. High-level models of this system are shown in Figure 3.5a and Figure 3.5b. Respectively, these show the FSMLock primitive targeted toward the Mealy and Moore state entry partitionings.

To facilitate limited access to the entirety of the FSM at run time, the concept of state scope must be defined. States in scope have their corresponding state entry unencrypted. Contrariwise, the states which are encrypted are out of scope. Using the set of state entries $E = \{e_0, e_1, ..., e_n\}$ definition for an FSM, this means that $E_{in} =$

$\{e \mid e \in E \wedge$ "e is unencrypted"$\}$ and at any time during operation $E_{in} \subseteq E$. With scope control in mind, the proposed binary state encoding s can be partitioned into two pieces (tag t and index i), as shown in Figure 3.6. Using the assumption that the number of states in scope is constant and of value $2^{|i|}$, the tag marks which states share a scope, i.e., states with the same tag are those loaded into scope together. When the FSM transitions out of the loaded scope, the security/scope control entity must load and decrypt the next set of in-scope state entries E'_{in}. Without improvements, this process results in additional transition latency and is defined in Subsection 5.1.2.

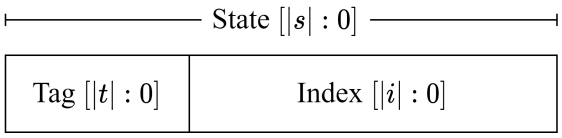

Figure 3.6: A state encoding with tag and index partitions. Sizes are defined in terms of state bits $|s|$, tag bits $|t|$, and index bits $|i|$..

Scope partitioning is only achievable due to the fundamental idea behind the FSMLock primitive, the first-of-its-kind classical encryption of the memory-based FSM SET contents. Through using a block cipher to load the locked circuitry in partitions, we contend that the FSMLock primitive is SAT resistant, like its combinational, SARLock [5], Anti-SAT [6], and LUT-Lock [57], and sequential, ReTrustFSM [69], counterparts while having increased resistance to removal attacks. The SAT-resistant properties come from the SAT hard nature of block ciphers [78], discussed later in Subsection 3.2.3, and the increased resistance to removal attacks is possible since there is presumably only enough memory to store the partitioned subset of the locked sequential logic at a time. Therefore, without profound design changes, one could not remove and replace the encrypted SET with its un-partitioned plaintext counterpart. Further, because only a portion of the sequential logic is stored in plain text at run time, there is no potential for an adversary to load out the entirety of the unencrypted SET memory at once. Aside from invasive run-time probing, which is already out of the coverage of the attacker model defined in Section 3.1, a cold

boot attack [79] could allow for memory contents to be read out of the in-scope SET. A cold boot attack is a type of side channel attack in which the attacker uses the phenomenon of memory remanence, potentially aided by the physical cooling of the chip, to read the data out of a dynamic RAM (DRAM) or static RAM (SRAM) device after it has been powered off. This attack vector is within the coverage of our attacker model defined in Section 3.1; as such, it is recommended that scope partitioning be used to hinder its use.

Although, assuming a trusted party is responsible for programming a unique key and configuration for each locked IC/FPGA device, one could then ignore the use of a key preprocessor and directly use the individual internal key at the chip boundary as shown in Figure 3.7a, we recommend the use of a PUF-based key preprocessor as shown in Figure 3.7b. The reasoning behind this recommendation is that given the attacker asset A1 "The gate-level netlist and static memory contents of the locked design obtained via RE" we must assume a potential adversary has access to the Encrypted SET. In such a situation, if the Individual Internal/Chip Key is leaked, the complete confidentiality of the locked circuit is compromised and enables attack goals G2 and G3. Figure 3.7a illustrates this scenario by coloring the Encrypted SET and Individual Internal/Chip Key in red, symbolizing that assets are obtainable to an adversary. To prevent this, a PUF-based key preprocessor can be utilized to translate the individual chip key into an unequivalent individual internal key. Considering the same attack scenario, such that the individual chip key and encrypted SET are known to the adversary, the confidentiality of the Encrypted SET would remain intact. This is because the adversary would be unable to generate the individual internal key from the individual chip key without knowledge of the PUF response. Assuming said response is unavailable at the chip boundary after the FSMLock configuration is programmed by a trusted party, an adversary would be unable to obtain it without invasive run-time probing, which is directly excluded from the attacker capabilities

and assets as stated in A5: "State-of-the-art IC analysis equipment, i.e., testing equipment and tools to perform invasive analysis *excluding* run-time invasive probing against the target IC or FPGA".

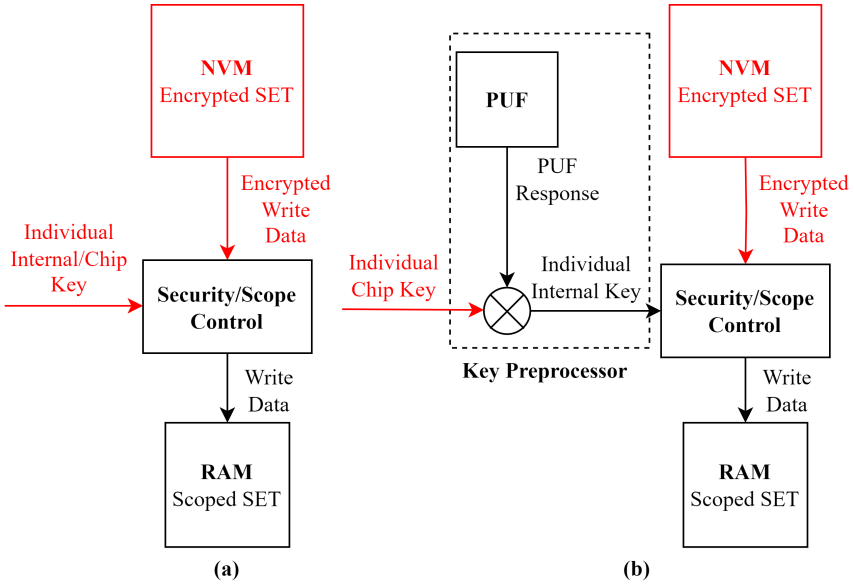

Figure 3.7: Simplified FSMLock primitive showing the use of (a) no key preprocessor and (b) a physically unclonable function (PUF) based key preprocessor. Assets assumed to be available to an adversary, given the list of capabilities and assets provided in Subsection 3.1.1, are colored in red.

In review, FSMLock is foremost a **logic locking** technique that operates on sequential circuits in the abstracted form of an FSM. The FSM is first obfuscated into a binary representation of the SET. Next, the binary SET is encrypted and stored in on-chip non-volatile memory–see Figure 3.5. At run time, the encrypted SET is decrypted in partitions with the internal chip key and loaded into the in-scope memory.

FSMLock is not a logic obfuscation technique, despite the intermediate obfuscation of the state logic into a LUT structure. This is because the encrypted SET elicits the requirement of a key to access the circuit's original functionality. Furthermore, because FSMLock stores all identifiable aspects of the sequential circuit in a classically encrypted memory, even those with direct access to the post-obfuscated HDL,

GDSII layouts, unencrypted configuration bitstreams, or locked production units will have no computationally feasible way to extract or modify the circuit functionality assuming a cryptographically secure cryptographic block cipher algorithm is used to encrypt the FSMLock SET contents.

3.2.1 Theoretical Resource Utilization

Resource utilization is a crucial design aspect because, as with most forms of logic locking, the introduction of FSMLock results in hardware utilization tradeoffs. The layout area of an IC is directly linked to the number of transistors and, consequently, the number of logical and memory components needed to fulfill the design, and as the layout area increases, so does the die cost. Therefore, it is economically beneficial for the FSMLock primitive to minimize resource utilization inflation. Likewise, FPGAs have a predefined finite number of LUTs, BMEMs, and FFs available for configuration. Therefore it is also advantageous to reduce resource utilization in an FPGA system.

The first constraint that dictates the theoretical resource utilization of the FSMLock primitive is the number of unique states, which must be obfuscated as state entries and stored in the in-scope and out-of-scope SET memories. In the scenario where the total number of unique states is unknown, for example, if the sequential circuit was not initially modeled following the general structure of an FSM showed in Figure 3.1, an upper bound on this quantity can be derived from the number of hardware registers which are to be absorbed into the FSMLock primitive. For any sequential circuit with $|s|$ memory registers, $2^{|s|}$ states are required to model it as a singular FSM, assuming all states are reachable and sequential binary state encoding is used with $|s|$ state encoding bits. This relationship can be inductively proven by considering the two possible register conditions: low or high. Therefore, the FSM abstraction of a single register has two states, and each time another register is added

to the arbitrary sequential, it doubles the potential permutations.

The theoretical out-of-scope SET memory size $Size_{out^T}$ is fundamentally related to the number of unique states because the out-of-scope SET holds all state entries, and each state entry must be stored in an exclusive location. Therefore, the required SET memory depth increases linearly with each $e \in E$. Hence it increases exponentially with the number of memory registers, or state bits $|s|$, in the original sequential circuit. This is illustrated in Figure 3.8 and modeled in Equation 3.3. The depth of the out-of-scope SET is exactly equal to the number of states because the Mealy and Moore state entry partitions shown in Figure 3.4 have a depth of 1, i.e., they only require a single memory location per state entry.

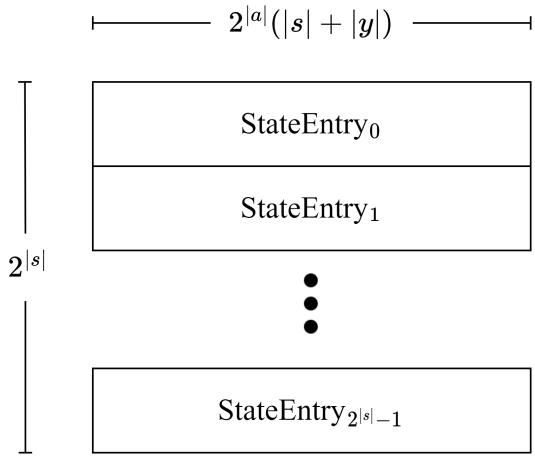

Figure 3.8: Abstract memory structure holding the out-of-scope SET. The depth of the memory is $2^|s|$, and the width of the memory is $2^{|a|}(|s| + |y|)$ such that $|a|$ is the number of input bits, $|s|$ is the number of state bits, and $|y|$ is the number of outputs bits.

$$2^{|s|} = Depth_{out^T} \tag{3.3}$$

The in-scope depth similarly linearly depends on the number of state entries it stores. But since only $2^{|i|}$ states are ever in scope with one another, and $|i| <= |s|$, that means the depth of the in-scope region will always be lesser than or equal to the

out-of-scope region. See Equation 3.4 for the in-scope memory depth.

$$2^{|i|} = Depth_{in^T} \tag{3.4}$$

The second constraint that dictates memory utilization is the width of the in-scope and out-of-scope SETs. Equation 3.5 expresses the rate at which the number of input bits $|a|$, state bits $|s|$, and output bits $|y|$ increases the memory width of the Mealy state entry partitioning illustrated in Figure 3.4a and hence the SET memory.[7] The width of in-scope and out-of-scope SET is equivalent to that of the state entry because, by definition, the SET is a table comprised of state entries.

$$2^{|a|}(|s| + |y|) = Width_{in^T} = Width_{out^T} \tag{3.5}$$

The number of input bits $|a|$ is exponentially proportional to the number of unique input permutations. Correspondingly, $2^{|a|}$ state encodings of bit length $|s|$ and output vectors of bit length $|y|$ must be stored in each state entry, exponentially increasing the width.

$$Depth_{in^T} \times Width_{in^T} = 2^{|i|} \times 2^{|a|}(|s| + |y|) = Size_{in^T} \tag{3.6}$$

$$Depth_{out^T} \times Width_{out^T} = 2^{|s|} \times 2^{|a|}(|s| + |y|) = Size_{out^T} \tag{3.7}$$

Multiplying the depth and width, a model of the theoretical in-scope and out-of-scope SET memory size requirements can be found, as shown respectively in Equation 3.6 and Equation 3.7.

3.2.2 Theoretical Performance Impact

The performance of the post-locked hardware design is also important to ensure that the original design's functionality remains unaffected and/or is acceptable to the

[7] For the Moore state entry partitioning illustrated in Figure 3.4b, the entry width is instead expressed with $2^{|a|}|s| + |y| = Width_{in^T} = Width_{out^T}$.

system designer. The two pillars of the FSMLock primitive that alter the performance/functionality of the original circuit are decryption latency and memory propagation delay.

3.2.2.1 Decryption Latency

The time it takes to decrypt a scoped region results in initial power-up downtime, and if scope partitioning is used ($|t| \neq 0$), it also results in latency during state transitions that change the scope, i.e., those which change the tag bits in the state encoding. The latency introduced during the decryption of the next scoped region is linearly proportional to the number of cipher blocks that must be decrypted. Therefore, assuming a cipher implementation is used with a fixed latency of $CipherLatency$ clock cycles, the number of clock cycles of latency at power up and during all scope transitions is modeled by Equation 3.8. Any additional latency caused during the power-up procedure, such as round key generation, is not accounted for in Equation 3.8.

$$Latency = CipherRounds \times CipherLatency \tag{3.8}$$

Because the system designer specifies the number of tag bits $|t|$ and index bits $|i|$, the number of $CipherRounds$, i.e., the number of blocks in each scope, will likely be initially unknown. Therefore, to calculate latency using Equation 3.8, one would need to first solve for $CipherRounds$. This can be done through observations of two values: the number of state entries that share a scope $SEinScope$ and the number of state entries per block $SEinBlock$. The first of these values is easy to determine since, by definition, each state encoding that shares a tag is in scope with each other, meaning $SEinScope = 2^{|i|}$. The latter value, $SEinBlock$, is itself dependent on two other factors: the cipher block size $|b|$ and the state entry size $|e|$. One can divide the block size by the state entry size to determine the number of state entries per block. For the Mealy and Moore state entry partitionings shown in Figure 3.4, the state entry size is

equivalent to the state entry width because each state entry partitioning has a depth of 1. This is not the case for the Mealy† state entry partitioning introduced later in Section 5.1, therefore the width and depth of the state entry will need to be multiplied to determine the size $|e|$. Also, to aid in the implementation, the current rendition of the FSMLock automation toolset requires all state entry widths to be padded to the nearest power of 2; the reading behind this is later covered in Subsection 4.2.1. As such, the function $l(x) = 2^{\lceil log_2(x) \rceil}$ will be defined to aid in the clarity of Equation 3.9, which steps through the calculations needed to solve for $CipherRounds$.

$$\begin{aligned} CipherRounds &= \left\lceil SEinScope \Big/ SEinBlock \right\rceil \\ &= \left\lceil 2^{|i|} \Big/ \frac{|b|}{l(|e|)} \right\rceil \\ &= \left\lceil \frac{2^{|i|} \times l(|e|)}{|b|} \right\rceil \end{aligned} \tag{3.9}$$

3.2.2.2 Memory Propagation Delay

The propagation delay of the memory structure used to store the in-scope states may limit the operational clock frequency of the locked sequential circuit and, in turn, the surrounding logic. Likewise, in an FPGA design, the program clock used to drive the configurable logic blocks would be confined to the maximum clock frequency of the memory structure used to store the in-scope SET.

For example, in the Xilinx Artix-7 series of FPGA devices, which was used to measure the performance statistics provided later in Subsection 4.2.2, the maximum BMEM clock frequency is 288MHz [80]. This is less than the global clock tree maximum clock frequency of 394MHz to 628MHZ, depending on speed grade [81]. That being said, given the mesh network of interconnects and logic within an FPGA, one is unlikely to achieve the maximum global clock tree frequency even before logic locking is applied. Nonetheless, the inclusion of FSMLock primitives has the potential to

reduce the maximum achievable clock frequency.

Because the propagation delay through a memory component is processing node and memory architecture-specific, no quantified model of FSMLock's impact on clock frequency can be provided. Instead, we assert that it is limited to the system's memory speed, and the memory delay should be considered by system designers when implementing the FSMLock primitive.

3.2.3 Security Claims

We claim that the theoretical security characteristic of the FSMLock primitive is equivalent to that of the block cipher used to encrypt the SET memory configuration. Therefore assuming the cryptographic security of the AES cipher [82] and counter mode of the operation [20], the case studies discussed in Section 4.1 have an effective brute force time complexity of $O(2^{128})$ due to the 128-bit cipher key k.

We will support this claim by addressing each attack goal listed in Subsection 3.1.2, identifying existing attack vectors that aim to achieve said goal, and describing the design aspects of the FSMLock primitive that nullify said attacks. All attack vectors cited in the background Subsections on logic locking methodologies, 2.3.5 and 2.3.6, will be considered in this review.

Attack goal one (G1) states "The adversary is able to unlock arbitrarily locked circuits without design modification". G1 is only achievable given the existence of 1) a global chip key or 2) an individual chip key and a global internal key combination that utilizes a non-preimage resistant key pre-processor to generate the internal key. In either of these scenarios, the global chip key and global internal key, respectively, would need to be discovered by the adversary, presumably through a key extraction attack such as SAT [58, 64] or key propagation [4]. FSMLock is resistant to this attack goal because it does not fall into either of these categories. Instead, it uses an individual internal key and, further, a distinct device-specific individual internal key

when the recommended PUF-based key preprocessor shown in Figure 3.7b is utilized. Consequently, even if an adversary could unlock a singular IC/FPGA through disclosure of the individual chip key, the key would be useless at unlocking other already produced systems.

Attack goal two (G2) states "The adversary is able to interfere with the fabrication process or bitstream configuration to disable or weaken the locking scheme, thus enabling the production of unlocked ICs and use of locked FPGA bitstreams after the inclusion of the adversary-induced modification". Such interference covers attack vectors that aim to hard-code the correct global internal key through IC minimal mask manipulation [21], or FPGA bitstream modifications [15] and those which change the initial state of the locked sequential FSM (initial state patching) [8, 69]. The first is impossible for the same reason G1 is unachievable. FSMLock does not use a global internal key; therefore, an adversary could not produce additional units through sole interference with the fabrication process. This protection is achieved using the PUF-based key preprocessor design shown in Figure 3.7b, preventing the chip-specific encrypted SET from being used in other devices. On the other hand, if the PUF-based key preprocessor is omitted, an adversary could obtain the encrypted SET from an unlocked device (with a known individual chip/internal key), as permitted given asset A1, and load it into newly manufactured ICs or locked bitstream configurations. Hence, the individual chip/internal key would become a global chip/internal key. Therefore, for this security analysis, we will assume that the PUF-based key preprocessor from Figure 3.7b is used since it nullified the aforementioned attack, as explained in Section 3.2.

The latter, initial state patching, does not apply to FSMLock because FSMLock does not rely on implicit external secrecy, as defined in ReTrustFSM [69] and reviewed in Subsection 2.3.6. Rather like ReTrustFSM, FSMLock utilizes explicit external secrecy in the form of a key. Therefore, any attack which changes the initial state of

the FSM will only result in additional corruption since there is no "obfuscation" states like that of HARPOON [66] or Dynamic State Deflection [68] to navigate through at startup.

Attack goal three (G3) states "The adversary is able to nullify the locking scheme on the netlist level, thus obtaining an effectively unlocked netlist". To nullify the locking scheme on the netlist level would require two prerequisites: the adversary can identify the locking primitive in the RTL netlist, and said primitive can be removed and possibly replaced with the original logic if circuit withholding was performed without affecting the functionality of the overall IC/FPGA design. We do not aim to camouflage the FSMLock primitive such that it is unidentifiable in the netlist. Considering the FSMLock primitive uses easily identifiable memory structures and RE technologies such as the automation of FSM extraction from gate-level netlists [45, 46, 47] exist, it would be reckless to assume it is and will remain computationally infeasible to identify the FSMLock primitive within an IC/FPGA design. Therefore, we openly assume it is possible for an adversary, with the capabilities and assets defined in Subsection 3.1.1, to identify, isolate, and remove the FSMLock primitive and corresponding block cipher.

Instead, FSMLock relies on the inability to meet the second prerequisite to inhibit an adversary from achieving G3. Specifically, inferring or solving for the withheld circuit information would be computationally infeasible because, to our knowledge, there is currently no attack on the AES algorithm that reduces the cipher's attack time complexity. This includes resistance to SAT attacks such as Fun-SAT [64] and RANE [65]. Block ciphers such as AES are resistant to SAT attacks because a property of the AES, and other provable secure block ciphers, is that it is computationally infeasible to determine the inputs of a cipher from its outputs when the key is unknown [78]. This property is also exploited in the work of [2] and [24] to prevent SAT attacks using a cipher-based one-way random function. FSMLock differs from [2] and [24] because

FSMLock uses the AES block to directly encrypt the FSM contents as opposed to pre-processing the chip key. FSMLock is, therefore, less susceptible to removal attacks because the AES decryption block is directly essential to the system's functionality. If the AES engine is removed, there will be nothing to decrypt the next-state and output logic withheld in the encrypted SET memory configuration. Likewise, if the FSMLock primitive is removed said SET logic would be missing from the resulting design. In conclusion, with no way to RE the withheld FSM logic, there is no way to nullify the locking scheme; hence G3 is unachievable.

Since none of the attack goals are achievable given the adversarial capabilities and assets listed in Subsection 3.1.1, we assert that the FSMLock primitive is secure to the degree of the cryptographic block cipher algorithm used. Assuming no side channel vulnerability is found, this implies a brute force attack time complexity of $O(2^k)$.

3.3 Tool Automation

For the scope of this thesis, we sought to develop the FSMLock logic locking primitive described in Section 3.2. To enable this, we developed an automated toolset that translates pre-partitioned FSM, represented as an STT, into an encrypted SET memory configuration file. Combining the memory configuration file with the static HDL template generates the FSMLock hardware model primitive for the target FSM that is both synthesizable and can be simulated to verify its functionality. Figure 3.9 illustrates the flow of the automation toolset. Note that the system designer must only provide the STT, key, and nonce. The automated toolset uses this information to generate the set of state entries E and populate the SET, as shown in Figure 3.8.

The automated toolset performs no partitioning of the FSM. As such, the system designer must provide the fixed set of state encodings in the STT tool input. The choice of state encoding is significant because, as mentioned in Subsection 3.2.2, the

transition between scopes has a performance impact. Considering the scope partitioning scheme illustrated in Figure 3.6, it would be best for states that are often transitioned between to share the same tag so that they are in scope with each other and do not incur frequency transitional latency.

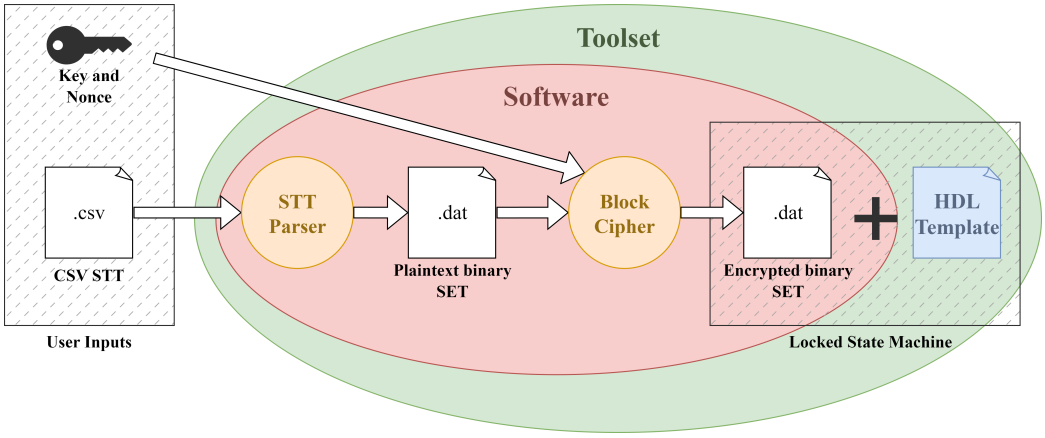

Figure 3.9: The automation toolset data flow diagram. System designer inputs are shown in the leftmost dotted box. Inputs include the desired key/nonce and a comma-separated value (CSV) formatted representation of the state transition table (STT). Toolset outputs are shown in the rightmost box. Outputs include the encrypted state entry table (SET) memory data file and an HDL template which takes the path to the encrypted SET memory data file as a generic input.

Also, because the automation toolset for this thesis is designed only to generate the hardware representation of FSMLock primitive, it is unaware of the larger circuit that the targeted FSM is a part of. As such, the placement strategy for the FSMLock logic locking primitive is outside the scope of the toolset and this thesis.

To aid in the toolset's overall goal, the primary purpose of the software portion of the automation toolset is to read an STT and generate a binary representation of the SET. The bijective function $f : T \mapsto E$ for translating the set of state transitions $T = \{t_0, t_1, ..., t_n\}$, in an STT, into the set of state entries $E = \{e_0, e_1, ..., e_n\}$, in a SET, is shown in Algorithm 1. After which, all that is required of the software is to represent the SET as a binary sequence, encrypt it, and store it in a data file that the HDL can load into the instantiated encrypted memory.

Lastly, the automation toolset aids in deploying the FSMLock primitive by pro-

Algorithm 1 Translation Algorithm $f : T \mapsto E$

Require: $(\forall t_\alpha, t_\beta \in T), (t_\alpha.s = t_\beta.s), (t_\alpha.a = t_\beta.a) \iff t_\alpha = t_\beta$
Ensure: $(\forall e \in E), (\forall s \in e.S_{next}) \implies s \neq \emptyset$
1: $E \leftarrow \{e_0, e_1, ..., e_{2^S-1}\}$
2: **for all** $t \in T$ **do**
3: $A \leftarrow getInputs(t.a)$ # Set of all matching inputs with a considering '-' bits
4: **for all** $a \in A$ **do**
5: $E[t.s].S_{next}[a] \leftarrow t.s_{next}$
6: $E[t.s].Y[a] \leftarrow t.y$
7: **return** E

viding an HDL template, as shown in Figure 3.9. This template refers to the HDL file provided by the FSMLock HDL library, fsmlock_top, and automatically inserts the required security/scope control logic, block cipher engine, and multiplexors shown in Figure 3.5 into the hardware design. To use the template, it should be instantiated alongside a memory component storing the encrypted SET. Figure 3.10 illustrates the HDL hierarchy of the fsmlock_top HDL template. Instances automatically inserted by the HDL template are shaded in gray, and the accompanying encrypted SET memory, enrypted_state_entry_brom, is also shown.

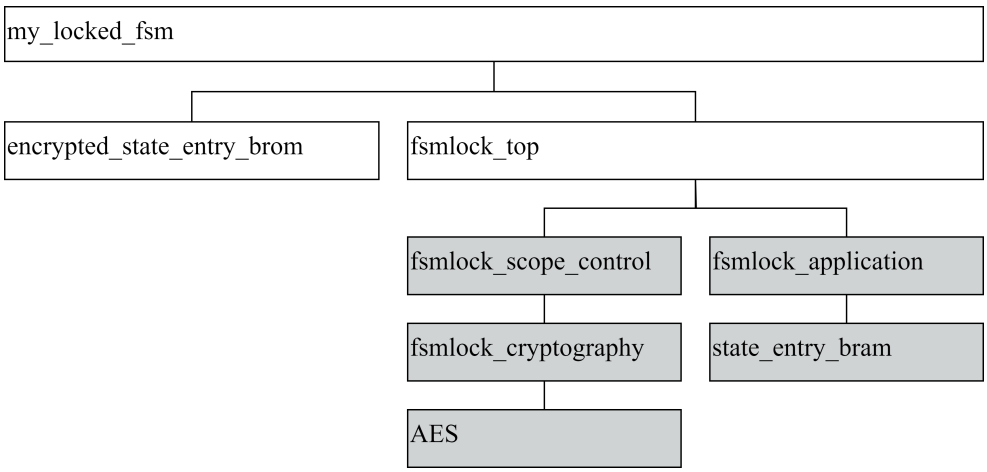

Figure 3.10: Hierarchy of the logic locking primitive HDL template including the encrypted block memory component. Blocks that are grayed out represent the entities automatically instantiated via the fsmlock_top entity, i.e., the HDL template shown in Figure 3.9.

Note that two memory entities are required: one manually instantiated, which

stores the encrypted SET, and another automatically generated by the FSMLock toolset and stores the in-scope portion of SET. The encrypted FSM SET is always available in the enrypted_state_entry_brom while the state_entry_bram contains only the in-scope entries E_{in}.

Chapter 4

Results

4.1 Debut of Case Studies

To test the performance and resource utilization characteristics of the FSMLock primitive, two case studies were chosen. These case studies represent different FSM scenarios in which the FSMLock primitive could be applied.

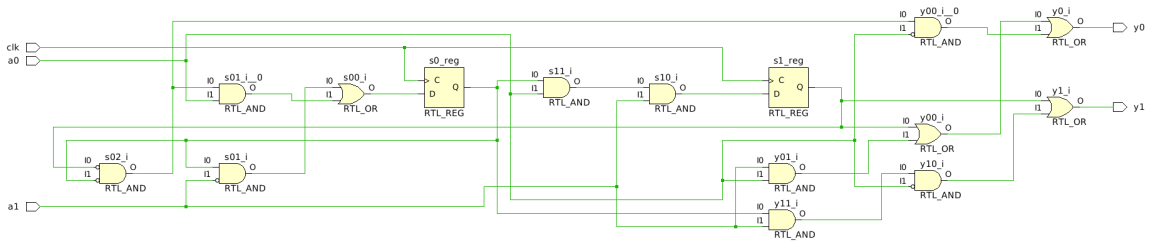

Figure 4.1: Circuit diagram for the Figure 3.2 Simple FSM example.

The first is the Mealy FSM example shown in Figure 3.2, referred to as the "Simple" example. This example illustrates the potential to lock a small chunk of sequential logic. This is possible because mealy FSMs can model any sequential logic, as discussed in Section 3.2. For example, see Figure 4.1 for the circuit diagram of the Simple FSM. With the ability to conceal chunks of mixed sequential and combinational circuitry in mind, we contend that instances of the FSMLock primitive can lock small chunks of logic through a design, effectively locking the larger IC/FPGA system they are a part of. Although evaluation and experimentation of a placement

strategy for such a locking approach are out of the scope of this thesis, we contend, like that of Full-Lock [7], that even the random insertion/placement of such primitives would sufficiently protect against sequential SAT attacks such as Fun-SAT [64]. This is because the FSMLock primitive is SAT-resistant due to the complexity of the AES cipher algorithm, similar to how the CLN provides Full-Lock SAT resistance without increasing the SAT iterations. On the other hand, strategies such as fault analysis-based [54] and corruptibility-based [29] placement may be beneficial if the system designer is interested in ensuring output corruptibility and Trojan resistance, respectively.

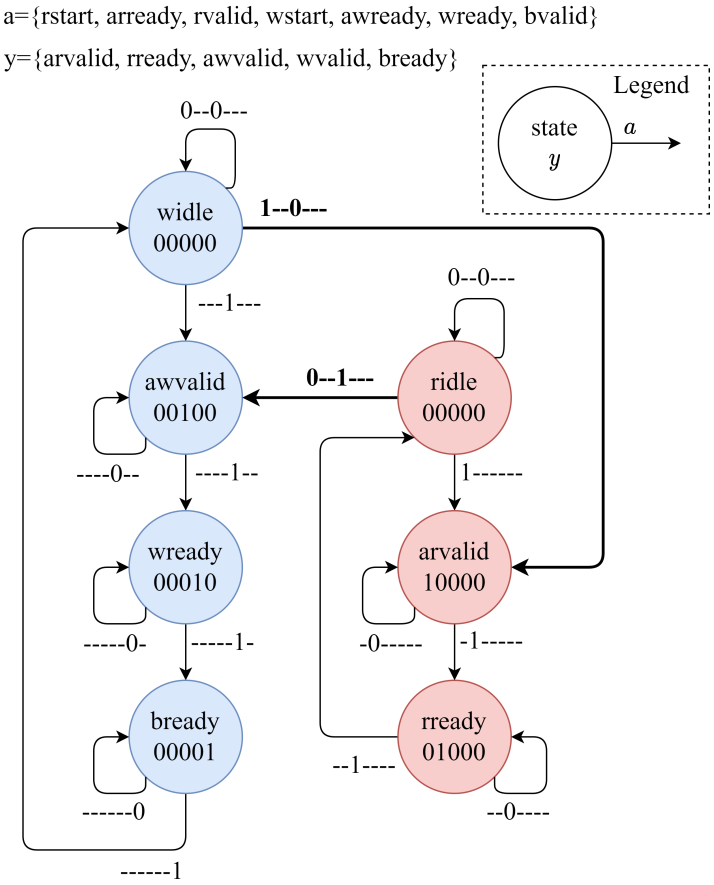

Figure 4.2: State transition graph (STG) for the master AXI lite (m_AXIL) controller FSM example. The node coloring illustrates the proposed scope partitioning for this example, and two idle states are included with state partitioning in mind to improve performance. Bold edges are emphasized to illustrate transitions that change the scope.

The second case study chosen is a master **A**dvanced e**X**tensible **I**nterface (AXI)

lite (m_AXIL) controller. Figure 4.2 illustrate the chosen Moore STG for this controller. This study was included because it represents a non-real-time FSM, i.e., it does not need to respond to events within predictable and specific time constraints. This is possible because the AXIL interface natively uses a handshake protocol with the entities connected to its inputs and outputs. Therefore, the FSMLock primitive can be used to lock the FSM control logic *with scoping* without invoking the required modification to the surrounding logic. Non-real-time FSMs include all FSMs which do not have a specific latency requirement between the FSM inputs and outputs. The states in Figure 4.2 are highlighted in blue or red to illustrate how scope partitioning can best be utilized in this m_AXIL example. With the proposed partitioning in mind, two idle states, "widle" and "ridle" were included to improve the performance of the m_AXIL controller. Doing so prevents the occurrence of decryption latency, as defined in Subsection 3.2.2, during repeated read and write operations.

Table 4.1: State transition table (STT) for the m_AXIL FSM example shown in Figure 4.2. Rows in bold indicate a change in scope considering the colored partitioning illustrated in Figure 4.2.

a	s	s_{next}	y
0--0---	widle	widle	00000
---1---	widle	awvalid	00000
1--0---	**widle**	**arvalid**	**00000**
----0--	awvalid	awvalid	00100
----1--	awvalid	wready	00100
-----0-	wready	wready	00010
-----1-	wready	bready	00010
------0	bready	bready	00001
------1	bready	widle	00001
0--0---	ridle	ridle	00000
1------	ridle	arvalid	00000
0--1---	**ridle**	**awvalid**	**00000**
-0-----	arvalid	arvalid	10000
-1-----	arvalid	rready	10000
--0----	rready	rready	01000
--1----	rready	ridle	01000

The STT for the m_AXIL example is provided in Table 4.1. As previously defined,

each row in the STT is bijectively mapped to each edge in the Figure 4.2 STG. Two transitions (i.e. rows) are in emphasized bold because these are transitions that change the scope of the FSM and are, therefore, subject to decryption latency, as discussed in Subsection 3.2.2. Lastly, note that the input vector in each row of the transition table has several don't care bits, represented by the '-' symbol. These don't care bits play an essential role in the input multiplexing improvement later listed in Subsection 5.1.2.

4.2 Design Characteristics

From the case studies detailed in the previous section, several results can be summarized regarding the performance and utilization characteristics of the FSMLock primitive.

4.2.1 Resource Utilization

The quantified model for the theoretical memory utilization of the locked circuit, presented in Subsection 3.2.1, provides insight into the required width and size of the in-scope and out-of-scope memory structures used to store the encrypted and unencrypted SETs required for the FSMLock primitive. What it does not account for, though, are device and implementation-specific limitations along with the refinements provided by the FSMLock toolset, made possible via consideration of the pre-existing bottleneck on out-of-scope memory reads due to the block cipher size $|b|$. As such, this section will translate the theoretical utilization values to what is predicted given our hardware implementation of the FSMLock primitive, provided in the automation toolset, and then compare the predicted utilization values against the post-implementation experimental utilization values. The purpose of the predicted utilization values is to illustrate what the expected output of our toolset could provide, while the experimental values show what was actually inferred given the

current condition of the hardware model provided in the automation toolset. Given the tweaking of the HDL models within the automation toolset, we believe all predicted values are achievable. Post-implementation testing was performed using the automated toolset described in Section 3.3 within the Xilinx Vivado 2019.1 EDA and with a Nexys A7-100T board utilizing the Artix-7 series XC7A100T FPGA part.

When translating from the memory structure of the theoretical SETs to the predicted memory structure, the first difference that one might see is that all theoretical SET widths $Width_{in^T}$ and $Width_{out^T}$ have been padded to a power of two. This design choice was made to ease the calculations required within the generic fsmlock_scope_control and fsmlock_application instances shown in Figure 3.10. To accommodate this, a partition of unused space is inserted between the next-state partitions and output partition(s) in the Mealy and Moore state entry encodings illustrated in Figure 3.4. The length of the unused partition is equal to the difference between the theoretical state entry width $Width_{in}^{T}$ and the next largest power of two. The next largest power of two can be found through utilization of the function $l(x) = 2^{\lceil log_2(x) \rceil}$ as demonstrated in Equation 4.1, calculating the in-scope predicted SET width $Width_{in^P}$.

$$l(Width_{in^T}) = l(2^{|A|}(|s| + |y|)) = Width_{in^P} \tag{4.1}$$

While the predicted width increases, the depth of the in-scope SET must remain unchanged to ensure the same number of $2^{|i|}$ state entries remain in-scope together and can each be read in a single memory read. Equation 4.2 re-affirms that there is no change to depth.

$$Depth_{in^T} = 2^{|i|} = Depth_{in^P} \tag{4.2}$$

But since the predicted width is increased ($Width_{in^P} \geq Width_{in^T}$) while the depth remains unchanged ($Depth_{in^P} = Depth_{in^T}$), that consequently means that the size

of in-scope SET must increase as well ($Size_{in^P} \geq Depth_{in^T}$). This is modeled in Equation 4.3.

$$Depth_{in^P} \times Width_{in^P} = 2^{|i|} \times l(2^{|A|}(|s| + |y|)) = Size_{in^P} \tag{4.3}$$

The next difference between the theoretical and predicted SET memory utilization is that the out-of-scope SET memory ($Width_{out^P}$) is changed to a fixed width. Precisely, the width of the out-of-scope SET memory is fixed to the cipher block size $|b|$. For the examples listed in Subsection 4.1, $|b| = 128$. Note this change does not imply that the state entry partitioning size $|e|$ is a different size in the out-of-scope memory. Rather, the out-of-scope SET memory is flattened so each read contains only a fraction of a single state entry partition when $|e| > |b|$ or several state entry partitions with one possibly fractured if $|e| \leq |b|$. This change is made to improve hardware resource utilization. As later discussed and modeled in Equation 4.8, the amount of BMEM primitives required is positively correlated with the memory width and depth. Therefore, since the block cipher of fixed width $|b|$ is already a bottleneck that prohibits more than one block from being decrypted at a time, there is no reason to increase the out-of-scope memory width $Width_{out^P}$ greater than that of $|b|$. In limiting the out-of-scope memory width, the predicted BMEM utilization decreases.

$$|b| = Width_{out^P} \tag{4.4}$$

Since the change to the out-of-scope memory $Width_{out^P}$ flattens the SET instead of lengthening each state entry like the change to the in-scope $Width_{in^P}$, it does not affect the size of the out-of-scope SET memory. The caveat is that $Size_{out^P}$ must still increase, given the overarching increase in state entry partitioning size $|e|$ due to the previously discussed inclusion of unused padding bits in the state entry e. Equation 4.5 models the predicted out-of-scope SET memory size, $Size_{out^P}$. The maximum

operator is included to ensure that, for smaller target FSMs, the predicted size of the out-of-scope SET is at least that of the width.

$$\max(2^{|s|} \times l(2^{|A|}(|s| + |y|)), Width_{out^P}) = Size_{out^P} \tag{4.5}$$

Finally, the depth of the predicted out-of-scope SET, $Depth_{out^P}$, can be determined using the calculated $Width_{in^P}$ and $Size_{out^P}$, as shown in Equation 4.6.

$$\frac{Size_{out^P}}{Width_{out^P}} = Depth_{out^P} \tag{4.6}$$

Now that the predicted width, depth, and size of the in-scope and out-of-scope SET memories have been determined, the predicted resource utilization of each FSMLock primitive generated by the automation toolset can also be modeled. As stated, the Mealy and Moore state entry partitionings shown in Figure 3.4 store all state entry information within a singular memory location. This means a single memory read is required to obtain the output and next-state logic, and such information should not change until the next transition occurs. Hence a synchronous memory structure with a width greater than the state entry width $|e|$ must be used for the in-scope SET. In FPGA devices, BMEM primitives are the most efficient way to implement large amounts of synchronous memory because they are hard IP blocks that do not require the fabric LUT and FF resources. Considering this, BMEM will be targeted by the FSMLock primitive while using the Mealy and Moore state entry partitionings. Equation 4.7 models the theoretical number of in-scope BMEM primitives required for an FSMLock primitive in terms of the required $Width_{in^P}$ and $Size_{in^P}$ of the in-scope SET. For the Mealy state entry partitioning, the in-scope $Width_{in^P}$ and $Size_{in^P}$ are

respectively modeled in Equations 4.1 and 4.3.

$$\max(\max(Width_{in^P}, Width_{out^P}) \pmod{72}, Size_{in^P} \pmod{(36 \times 2^{10})}) = BMEMs_{in} \tag{4.7}$$

The premise behind Equation 4.7 is that the number of BMEM devices is dictated by either a constraint on the memory width or size. Each Xilinx BMEM primitive is 36kbit in size and has a memory width of 72 bits, as defined in the Xilinx PG058 [80]. Therefore, SETs with widths larger than 72 bits must be spread across a minimum of $Width \pmod{72}$ BMEM primitives. Likewise, SETs with sizes larger than 36kbit must be spread across a minimum of $Size \pmod{(36 \times 2^{10})}$ BMEM primitives. Whichever one of these is larger is the theoretical number of in-scope BMEMs required for in-scope FSMLock SET. Further, because the in-scope SET memory must support storing data in the the out-of-scope $Width_{out^P}$ and reading state entries in the in-scope $Width_{in^P}$, the maximum of these values must be used when determining the in-scope BMEM utilization.

Although in its theoretical format, the out-of-scope BMEM count for the out-of-scope SET could be similarly modeled by removing the $Size_{in}$ parameter of Equation 4.7, we contend a more lenient constraint can be imposed on the width. The resulting model is shown in Equation 4.8.

$$\max(|b| \pmod{72}, Size_{out^P} \pmod{(36 \times 2^{10})}) = BMEMs_{out} \tag{4.8}$$

As previously discussed, the width of the out-of-scope SET memory is not constrained by the state entry width. This is because the block cipher size $|b|$ already bottlenecks the maximum amount of data that can be processed during reads. Therefore, there is no need for the width of the out-of-scope memory to be wider than $|b|$. Accordingly, Equation 4.8 is only limited by $|b|$ and $Size_{out^P}$.

Table 4.2: Memory utilization table for Simple (Figure 3.2) and m_AXIL (Figure 4.2) FSM case studies. Theoretical and experimental BMEM, LUT, and FF resource utilization accounts for only the components used to implement the memory components within the FSMLock primitive, i.e., does not include memory resources required to implement the chosen block cipher or other logic.

Example	**Stage**	**In Scope**						**Out Scope**					
		Width	Depth	Size	BMEM	LUT	FF	Width	Depth	Size	BMEM	LUT	FF
Simple	Theo.	16	4	64	-	-	-	16	4	64	-	-	-
	Pred.	16	4	64	2	0	0	128	1	128	2	0	0
	Exp.	16	4	64	4	0	0	128	1	128	0	88	128
m_AXIL	Theo.	389	4	1556	-	-	-	389	8	3112	-	-	-
	Pred.	512	4	2048	6	0	0	128	32	4096	2	0	0
	Exp.	512	4	2048	16	0	0	128	32	4096	2	0	0

Finally, given a model for the predicted memory resource utilization of the FSM-Lock primitive, it can be compared against what was experimentally seen in the case studies reviewed for this thesis. Table 4.2 lists the theoretical memory requirements, predicted memory resources, and experimentally measured memory resources, while Table 4.3 lists the parameters for each example.

Table 4.3: Parameters for the Simple (Figure 3.2) and m_AXIL (Figure 4.2) FSM case studies.

	Simple	**m_AXIL**
State Entry Partitioning	Mealy	Moore
Tag bits ($\|t\|$)	0	1
Index bits ($\|i\|$)	2	2
Input bits ($\|a\|$)	2	7
Output bits ($\|y\|$)	2	5

Interestingly, although the experimental memory width, depth, and size align with our predictions, the BMEM utilization is much higher. Also, for the Simple example, the out-of-scope SET memory was inferred to be distributed instead of BMEMs. The reasoning behind the inference of distributed memory is likely because the Vivado synthesis tool recognized that few LUTs are required for the 128×1 and justly prioritized saving two BMEM primitives over 88 LUTs. This, therefore, points to no fault of the automation toolset HDL template but instead points to the flexibility provided through the behavioral description of the memory structure. On the other hand, the in-scope BMEM resource inflation is quite troubling. After further investigation, the source of this inflation appears to be caused by the tool's inability to utilize both ports of each BMEM during the synthesis of traditionally modeled asymmetric block memory ports. The current implementation of the FSMLock primitive requires asymmetric read and write ports because the out-of-scope SET, of width $Width_{out^P} = |b|$, must be processed by the cipher block and loaded into the in-scope SET memory of width $Width_{in^P}$, i.e., when $Width_{out^P} \neq Width_{in^P}$ the read and write ports of the in-scope memory are asymmetric. Although no work has been done to remedy

this problem, potential solutions include artificially increasing the in-scope memory $Width_{in^P}$ such that it is equal to $Width_{out^P}$ when $Width_{in^P} < Width_{out^P}$ or buffering decrypted out-of-scope reads when $Width_{in^P} > Width_{out^P}$, such that, only a singular write happens to the in-scope SET after $Width_{in^P}$ bits of data have been decrypted.

Although memory utilization has been the primary area of discussion regarding resource utilization, it is not the only factor that dictates hardware utilization; the surrounding control logic required for the FSMLock primitive must also be added to a locked system. In Xilinx FPGA devices, like the XC7A100T FPGA part used for testing, the reconfigurable fabric mesh is comprised of configurable logic blocks and, embedded within them, slices. In each slice, there exist 6-input LUTs that can be used to implement an arbitrary function up to 6 variables in size. In addition to the basic LUTs, slices contain three multiplexers (F7AMUX, F7BMUX, and F8MUX) and FFs. The multiplexers are used to combine LUTs and provide any function of seven or eight inputs in a slice, and the FFs allow for the modeling of sequential circuitry [83]. For the FSMLock primitive, the use of this surrounding fabric is required to implement the chosen cryptographic cipher (AES counter mode was used during the testing of the Simple and m_AXIL examples) and scoping control logic. These respectively exist within the fsmlock_cyrptography and fsmlock_scope_control instances shown in the Figure 3.10 HDL hierarchy. Likewise, when synchronous memory is used, as is the case with the Moore and Mealy state entry partitionings, the next-state and output multiplexers shown in Figure 3.5 must also be instantiated in the fabric. These multiplexers are inferred within the fsmlock_application instance shown in the Figure 3.10 HDL hierarchy. To document these additional sources of resource utilization for the Simple and m_AXIL examples, Tables 4.4 and 4.5 are included.

Although these tables illustrate the experimental resource utilization experienced, we claim that the predicted memory utilization values previously shown in Table 4.2 provides a better estimate of the achievable memory utilization of the FSMLock

primitive. The increased number of in-scope BMEM primitives displayed in both Table 4.4 and Table 4.5 along with the inferred distributed memory in the place of the predicted encyrpted_state_entry_brom BMEM in Table 4.4 are byproducts the Vivado synthesis tool not properly interpreting what hardware resources the HDL template provided in the automation tools intends to generate. As previously stated, through tweaking the HDL template or manual instantiation of the intended structure of Xilinx BMEM primitives required for the in-scope SET memory, predicted resource utilization values can be achieved.

Table 4.4: Post-synthesis resource utilization hierarchy of the locked Simple example (Figure 3.2). All resources required within the fsmlock_cryptography instance comprise the HDL molded AES cryptography block.

	LUT	FF	F7MUX	F8MUX	BRAM
my_locked_fsm	640	400	0	0	11
fsmlock_top	552	272	0	0	11
fsmlock_application	7	0	0	0	4
state_entry_bram	0	0	0	0	4
fsmlock_scope_control	545	272	0	0	0
fsmlock_cryptography	540	269	0	0	7
encrypted_state_entry_brom	88	128	0	0	0

Table 4.5: Post-synthesis resource utilization hierarchy of the locked m_AXIL example (Figure 4.2). All resources required within the fsmlock_cryptography instance comprise the HDL molded AES cryptography block.

	LUT	FF	F7MUX	F8MUX	BRAM
my_locked_fsm	674	285	49	24	25
fsmlock_top	674	285	49	24	23
fsmlock_application	105	0	49	24	0
state_entry_bram	0	0	0	0	16
fsmlock_scope_control	569	284	0	0	7
fsmlock_cryptography	556	275	0	0	7
encrypted_state_entry_brom	0	0	0	0	2

4.2.2 Performance Impact

As discussed in Subsection 3.2.2, two main performance impacts exist with introducing the FSMLock primitive: decryption latency and memory propagation delay. Because the latter is processing node and memory architecture-specific, we will focus the discussion on decryption latency results.

Each example circuit exhibited power-up latency documented in Table 4.6. This latency comes from two sources, the initial power-up sequence of the selected cipher algorithm, such as static key expansion (if applicable), and the scope decryption latency modeled in Equation 3.8. Both examples experience latency at power-up, and the m_AXIL example experiences occasional latency during run-time. The m_AXIL example is different because it utilizes scope partitioning and is subject to decryption latency for each transition between scopes during run-time. The intelligent partitioning performed on the m_AXIL example shown in Figure 4.2 ensures that such latency only occurs when performing reads after writes and vice versa. While it is not necessary to partition the m_AXIL example following this structure, we believe it is the best considering the likeness of performing multiple reads or writes in a row is assumed to be high.

Table 4.6: Latency table for the Simple (Figure 3.2) and m_AXIL (Figure 4.2) FSMs. Because the counter mode AES cipher used during experimentation required 18 cycles for each encryption round, the decryption latency cycle count was found by multiplying the number of rounds by 18. For other ciphers and/or implementations, the decryption latency cycle count will need recalculating with the new cipher latency.

	Simple	**m_AXIL**
Key Expansion (cycles)	26	26
Decryption Latency (rounds)	1	16
Decryption Latency (cycles)	18	288

The number of decryption latency rounds, i.e., the number of cipher blocks which must be decrypted to load the next in-scope SET ($CipherRounds$), can be solved for using Equation 3.9 or through the simplified format shown in Equation 4.9. Both

equations model the same logic but use different inputs, which may be more readily available given the current stage in the FSMLock design process. Mainly, Equation 4.9 requires the $Size_{in}^{P}$ variable to be precomputed.

$$\left\lceil \frac{Size_{in}^{P}}{|b|} \right\rceil = CipherRounds \tag{4.9}$$

After utilizing Equation 3.9 or 4.9, one can multiply $^{Block}/_{Scope}$ by the selected cipher cycle count required during each decryption round to determine the decryption latency in cycles. In the case of the counter mode AES cipher used during experimentation, the cipher required 18 cycles; hence, how the decryption latency cycle count was determined for the Simple and m_AXIL examples shown in Table 4.6.

Subsection 5.1.2 will later discuss how through potential improvements to the FSMLock automation toolset, the decryption latency caused by the introduction of the FSMLock primitive can be minimized. This is done in two ways: decreasing the number of rounds or decreasing the discernible number of cipher latency cycles.

Chapter 5

Closing

5.1 Future Work

5.1.1 Resource Utilization Improvements

The first area of future work for the FSMLock methodology is improvements in resource utilization. Improvements in resource utilization are important, as determined via the case studies discussed in Chapter 4.

5.1.1.1 Mealy† State Entry Partitioning

The first potency improvement is support for new state entry partitioning. The initially proposed Mealy and Moore state entry partitionings, shown in Figure 3.4, are simple because they contain all state entry information encoded in e in a singular addressable memory location, i.e., they have a memory depth of 1. To reiterate briefly, the state entry e is a 3-tuple (s, S_{next}, Y) consisting of the current state s, an N-tuple of the next states $S_{next} = (s_0, s_1, ..., s_N)$, and an N-tuple of outputs $Y = (y_0, y_1, ..., y_N)$ where N is the number of unique inputs. Considering each state entry has a depth of 1, the only factor which dictates the size of the state entry partitioning e is the width of the encoded state entry information.

Having all state entry information in a single memory location can be a problem for some target FSMs because, as modeled in Equation 3.5 and experimentally shown

in Chapter 4, there quickly becomes a feasibility concern regarding the required memory width of the in-scope FSMLock unencrypted memory. This stems from the fact that the S_{next} and Y N-tuples scale exponentially with the number of inputs a and, in both the Mealy state entry partitioning (Figure 3.4a) and Moore state entry partitioning (Figure 3.4b), the exponentially increasing S_{next} N-tuple must be stored at each location. Further, the exponentially increasing Y N-tuple must also be stored for the Mealy state entry partitioning.

In FSMs which a large number of inputs, $|a| > 4$, the required memory bus width quickly grows larger than feasible in most IC/FPGA systems. In response, one could use a state entry partitioning that spans multiple memory locations (has a depth > 1). Figure 5.1b shows an example of a state entry partitioning that spans $2^{|a|}$ memory locations. This partitioning will be referred to as the Mealy† state entry partitioning.

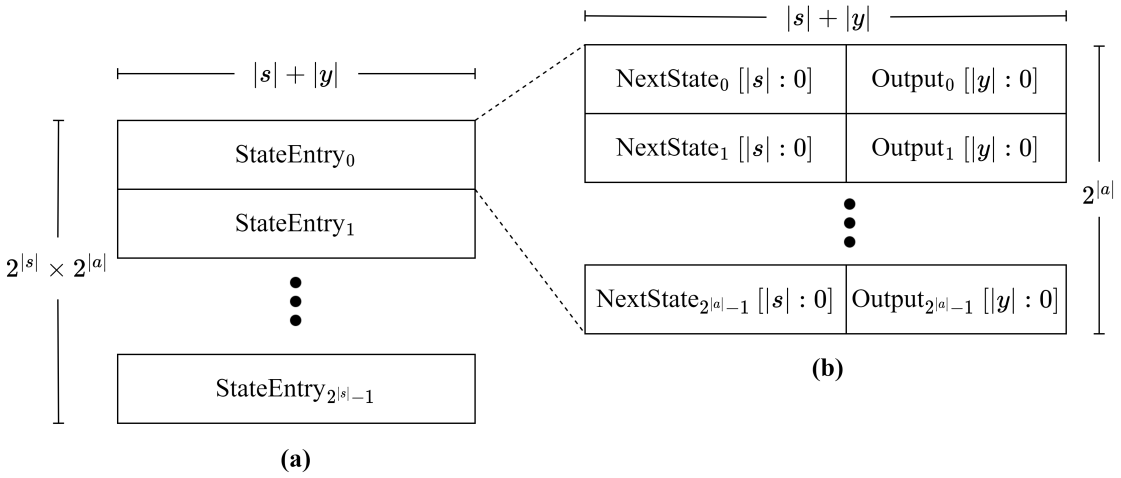

Figure 5.1: The proposed (a) new state entry table (SET) that reduces memory width through the use of a (b) new Mealy† state entry partitioning. The Mealy† partitioning reduces SET memory width while utilizing the same amount of memory (i.e., has the same size) as the Mealy partitioning shown in Table 5.2, through increasing the memory depth of each state entry by a factor of $2^{|a|}$.

While the Mealy† state entry partitioning successfully removes the exponential $2^{|a|}$ scaling factor from the width of the Mealy state entry partitioning, as highlighted through a comparison between the old and new theoretical width models, shown

respectively in Equation 3.5 and Equation 5.1, it comes with two drawbacks.

$$(|s| + |y|) = Width_{in^T} = Width_{out^T} \tag{5.1}$$

The first drawback is that the depth of the required SET is increased by a factor of $2^{|a|}$ since now every state entry in the SET spans a depth of $2^{|a|}$ memory locations. Equations 5.2 and 5.3 model the memory depth of out-of-scope and in-scope SETs while using the Mealy† state entry partitioning.

$$2^{|a|} \times 2^{|s|} = Depth_{out^T} \tag{5.2}$$

$$2^{|a|} \times 2^{|i|} = Depth_{in^T} \tag{5.3}$$

This trade-off directly opposes the decrease in width; therefore, the required in-scope and out-of-scope memory sizes of the resulting Mealy† FSMLock primitive are equal to that of the initially proposed Mealy state entry partitioning modeled respectively by Equation 3.6 and Equation 3.7. Note that there is no Moore† equivalent of the Mealy† state partitioning like that of Figure 3.4b because, by design, each uniquely addressable location is responsible for the current output. Therefore, in modeling a Moore FSM using the Mealy† state entry partitioning, the singular output value for each state must be accessible in each of the $2^{|a|}$ potential memory locations. Therefore, there would be no way to decrease the state entry partitioning size for targeted Moore FSMs, and there exists no reason to define a Moore† state entry partitioning.

The second drawback is that the Mealy† state entry partitioning requires asynchronous memory. This is because the output and next state values must update asynchronously with the a input used to address the memory. Expressly, the next state signal is registered at the rising edge of the clock, concatenated with the asynchronous input bits, and together used as the address of the in-scope FSMLock mem-

ory. The asynchronous updated output y is directly used as the FSM output and, as previously alluded, to prevent the output of targeted Moore FSMs from changing with the asynchronous input, all $2^{|a|}$ copies of each state's outputs must be identical. Figure 5.2 illustrates an obfuscated FSM utilizing asynchronous NVM as previously described. Figure 5.2 is analogous to the synchronous Figure 3.3 used with the Mealy and Moore state entry partitionings.

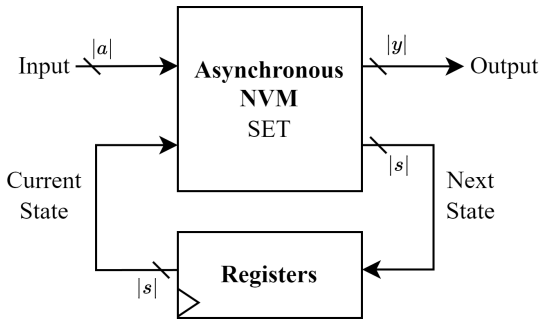

Figure 5.2: Variant of an FSM utilizing asynchronous non-volatile memory (NVM), which operates as an addressable lookup table (LUT) of state entries–referred to as the state entry table (SET)–while targeting the Mealy† (Figure 5.1b) state entry partitioning.

Considering the requirement of an asynchronous in-scope memory, the synchronous FPGA BMEM primitives targeted during the case studies shown in Chapter 4 can not be used. Instead, memory structures that support asynchronous reads are mandatory with the Mealy† state entry partitioning. In the case of SRAM FPGAs, distributed memory can be used in place of BMEM. Distributed memory is comprised of the special LUTs within the SLICEM slices of the FPGA fabric and can be read asynchronously [9]. The obvious downside of using distributed memory is that it decreases the LUTs available for other circuitry in the FPGA system.

Although distributed memory increases the required LUT utilization, it has other benefits. Primarily, a benefit to the Mealy† state entry partitioning and the use of asynchronous memory is the removal of the resource-costly next state and output multiplexers. When a synchronous memory is used to choose which state in the S_{next} N-tuple to transition into next, a multiplexer with $|a|$ control lines is required, as

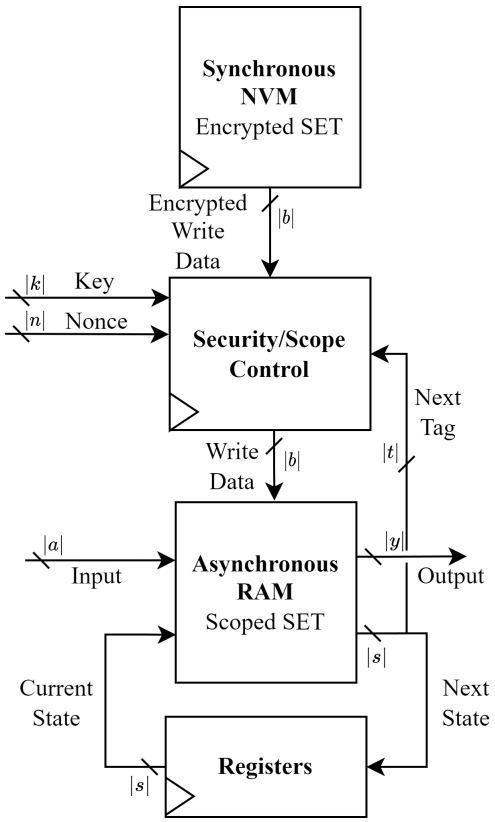

Figure 5.3: The model of the FSMLock primitive utilizing asynchronous in-scope random access memory (RAM) while targeting the Mealy† (Figure 5.1b) state entry partitioning.

seen in Figures 3.5. Likewise, when a Mealy FSM is targeted, an additional MUX must be used to select the output. Such multiplexers are resource intensive because of the large number of control lines ($|a|$ bits), which switch the signals in both the next state ($|s|$ bits) and output ($|y|$ bits) binary strings. When the Mealy† state entry partitioning is used, the asynchronous memory structure shown in Figure 5.3 represents the FSMLock primitive and is notably void of the large next-state and output multiplexers present in the synchronous memory structure shown in Figure 3.5.

5.1.1.2 Input Multiplexing

A second and potentially supplementary approach to decreasing memory width is input multiplexing. This technique is discussed in the work of I. Garcia-Vargas et

al. aptly titled "ROM-Based Finite State Machine Implementation in Low Cost FPGAs" [77] and is a way to reduce memory utilization of memory-Based FSMs. Since FSMLock is also an implementation of a memory-based FSM, the input multiplexing technique proposed in [77] is directly applicable in reducing memory size for the Mealy, Moore, and Mealy† state entry partitioning schemes and also width in the case of the initially proposed Mealy and Moore state entry partitioning schemes.

Input multiplexing decreases memory utilization by reducing the number of inputs that must be abstracted within the SET memory representation of the FSM. The process begins by observing the STT, where the number of state effective inputs (SEI) for each state can be determined. The number of SEI for a given state is defined as the maximum number of non-don't care bits for all transitions out of the state, i.e., "inputs which [have] influence on a particular state" [77]. In the scenario where the maximum SEI for all states is lesser than the number of primary inputs ($\max(SEI) < |a|$), the FSMLock primitive can be modeled using the lower number of $\max(SEI)$ bits as the new number of input bits. This is achievable because not all inputs are needed during every transition, and multiplexers can be placed to selectively choose which inputs feed directly into the memory-based FSM. Additional outputs from the locked FSM are required to control these newly added multiplexers. Even so, the width of the Mealy and Moore state entry partitions is linearly proportional to the number of output bits $|y|$ and exponentially proportional to the number of input bits $|a|$; therefore, the tradeoff still results in a decrease in memory width and size. For the Mealy† state entry partitioning, input multiplexing will instead increase the memory width since the width, as modeled by Equation 5.1, has no dependence on $|a|$ but a linear relationship with $|y|$. Nevertheless, the same decrease in memory size occurs, and because of the drastic memory width decrease provided by the Mealy† state entry partitioning, the increased width is unlikely to be a problem. An example showing the increase in memory width while introducing input multiplexing to an

FSM encoded Mealy† state entry partitioning is shown in Table 5.2.

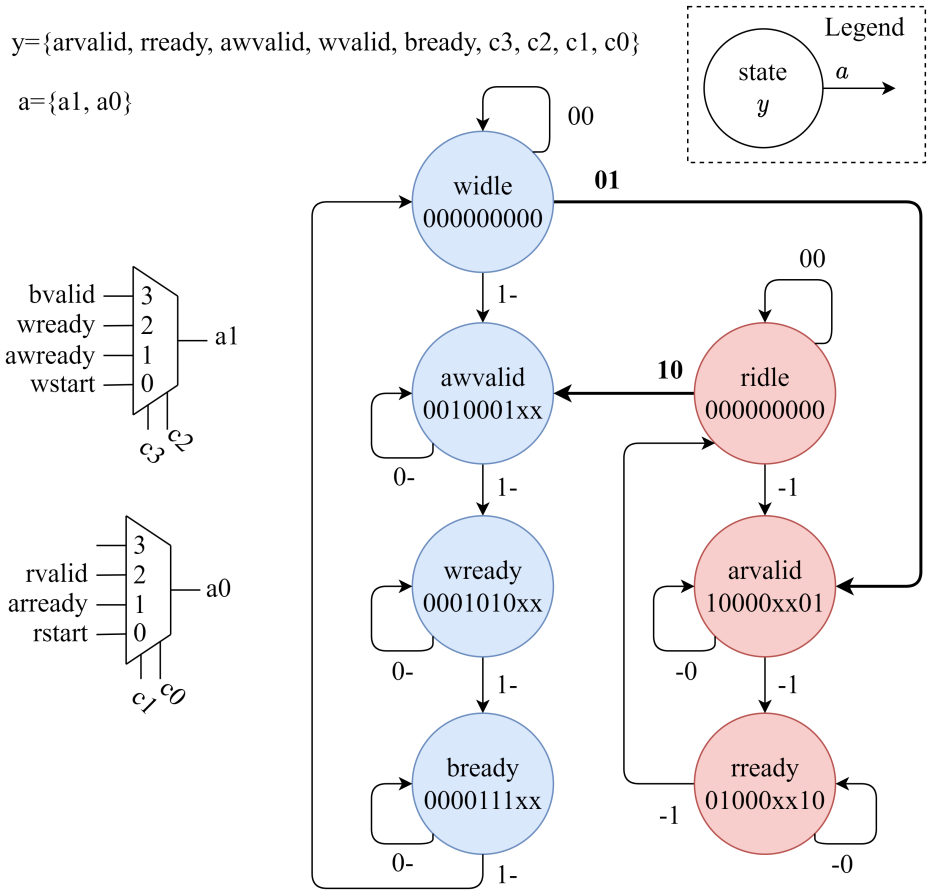

Figure 5.4: State transition graph (STG) for the master AXI lite (m_AXIL) controller FSM example after input multiplexing has been performed.

The m_AXIL case study lends itself well to input multiplexing because it has a maximum SEI size of 2 (As can be found through observation of Table 4.1)–much lower than the original $|a|$ length of 7. Using this information, a pair of multiplexers can be added to the design, as is shown alongside the new STG for the m_AXIL example illustrated Figure 5.4. The four additional outputs needed to control the multiplexers are also included in Figure 5.4, along with the corresponding values for these outputs during each state so as to retain the original functionality of the m_AXIL example. Table 5.1 is the resulting STT after modification to the m_AXIL STG. The last four bits of the y binary string are the newly added multiplexer control signals, c3, c2, c1, and c0, illustrated in Figure 5.4.

Table 5.1: State transition table (STT) for the m_AXIL FSM example after input multiplexing has been performed.

a	s	s_{next}	y
00	widle	widle	000000000
1-	widle	awvalid	000000000
01	**widle**	**arvalid**	**000000000**
0-	awvalid	awvalid	0010001xx
1-	awvalid	wready	0010001xx
0-	wready	wready	0001010xx
1-	wready	bready	0001010xx
0-	bready	bready	0000111xx
1-	bready	widle	0000111xx
00	ridle	ridle	000000000
-1	ridle	arvalid	000000000
10	**ridle**	**awvalid**	**000000000**
-0	arvalid	arvalid	10000xx01
-1	arvlid	rready	10000xx01
-0	rready	rready	01000xx10
-1	rready	ridle	01000xx10

The memory utilization characteristics of the original m_AXIL example are shown in Table 5.2 alongside the theoretical characteristics of the example after the input multiplexing and Mealy† state entry partitioning improvements were applied. Experimental results for the Moore state entry partitioning with input multiplexing improvement were possible because the modified STT shown in Table 5.1 was manually generated. Afterward, the modified STT could be entered into the automation toolset to generate the FSMLock primitive representation of the STG shown in Figure 5.1. Finally, in firmware, the multiplexers required for the input multiplexing were manually inserted into the hierarchy alongside the fsmlock_top instance shown in Figure 3.10. The effective FSM parameters for each of the m_AXIL example configurations is shown in Table 5.3.

Table 5.2: Memory utilization table for the m_AXIL example (Figure 4.2) with input multiplexing (Figure 5.4), theoretical application of the proposed Mealy† state entry (SE) partitioning (Figure 5.1), and both improvements at once. Theoretical and experimental BMEM, LUT, and FF resource utilization accounts for only the components used to implement the memory components within the FSMLock primitive, i.e., does not include memory resources required to implement the chosen block cipher or other logic.

Example (Improv.)	**Stage**	**In Scope**						**Out Scope**					
		Width	Depth	Size	BMEM	LUT	FF	Width	Depth	Size	BMEM	LUT	FF
m_AXIL (None)	Theo.	389	4	1556	-	-	-	389	8	3112	-	-	-
	Pred.	512	4	2048	6	0	0	128	32	4096	2	0	0
	Exp.	512	4	2048	16	0	0	128	32	4096	2	0	0
m_AXIL (MUX)	Theo.	21	4	84	-	-	-	21	8	168	-	-	-
	Pred.	32	4	128	2	0	0	128	2	256	2	0	0
	Exp.	32	4	128	4	0	0	128	2	256	0	128	128
m_AXIL (SE)	Theo.	8	512	4096	-	-	-	8	1024	8192	-	-	-
	Pred.	8	512	4096	0	70	0	128	64	8192	2	0	0
	Exp.												
m_AXIL (MUX & SE)	Theo.	12	16	192	-	-	-	12	32	384	-	-	-
	Pred.	16	16	256	0	12	0	128	4	512	2	0	0
	Exp.												

Table 5.3: Parameters for the m_AXIL example (Figure 4.2) with input multiplexing (Figure 5.4), theoretical application of the proposed Mealy† state entry (SE) partitioning (Figure 5.1), and both improvements at once.

	m_AXIL (None)	**m_AXIL** (MUX)	**m_AXIL** (SE)	**m_AXIL** (MUX & SE)
State Entry Partitioning	Moore	Moore	Mealy†	Mealy†
Tag bits ($\|t\|$)	1	1	1	1
Index bits ($\|i\|$)	2	2	2	2
Input bits ($\|a\|$)	7	2	7	2
Output bits ($\|y\|$)	5	9	5	9

Aside from decreased BRAM utilization due to the introduction of input multiplexing, the LUT utilization of the fsmlock_application instance (listed in Table 5.4) is also decreased. The new value of 5 LUTs is drastically reduced compared to the 105 LUTs shown in Table 4.5. The LUTs used by the fsmlock_application instance that are not included within the nested state_entry_bram instance comprise the resources required to model the next-state and output multiplexers illustrated in Figure 3.5. These are reduced when input multiplying is introduced since the resource-costly next state and output multiplexers, as covered in the discussion of the Mealy† state entry partitioning, are dictated in size by the number of inputs a into the FSMLock primitive. Therefore, since input multiplexing reduces the number of effective inputs to SEI, less logic is required for these multiplexers. Further, the additional input multiplexer(s) have a significantly smaller hardware footprint since they switch a small number of inputs ($\leq |a|$) using the minimal control signals added to the output y. For example, in the muxed m_AXIL, only two LUTs were required to implement the two input multiplexers added to the design, netting a 98 LUT decrease compared to the non-input multiplexed design (if the unexpected encrypted_state_entry_brom instance was instead correctly inferred as a block memory component).

The primary takeaway from Table 5.2 is that the best configuration for the m_AXIL example is using the Moore state entry partitioning with input multiplexing. It minimizes the required in-scope and out-of-scope memory size and is the only

Table 5.4: Post-synthesis resource utilization hierarchy of the locked input multiplexed m_AXIL example (Figure 5.4). All resources required within the fsmlock_cryptography instance comprise the HDL molded AES cryptography block.

	LUT	FF	F7MUX	F8MUX	BRAM
my_locked_fsm	695	403	0	0	11
fsmlock_top	565	275	0	0	11
fsmlock_application	5	0	0	0	4
state_entry_bram	0	0	0	0	4
fsmlock_scope_control	559	274	0	0	7
fsmlock_cryptography	552	270	0	0	7
encrypted_state_entry_brom	128	128	0	0	0

configuration not limited by the 128-bit cipher block size. Therefore, transitions between the write and read scopes will require only one round of the cipher, and hence the lowest decryption latency is achieved. The number of rounds required and the corresponding decryption latency is shown in Table 5.5.

Table 5.5: Latency table for improved m_AXIL example configurations. Because the counter mode AES cipher used during experimentation required 18 cycles for each encryption round, the decryption latency cycle count was found by multiplying the number of rounds by 18. For other ciphers and/or implementations, the decryption latency cycle count will need recalculating with the new cipher latency.

	m_AXIL (None)	**m_AXIL** (MUX)	**m_AXIL** (SE)	**m_AXIL** (MUX & SE)
Key Expansion (cycles)	26	26	26	26
Decryption Latency (rounds)	16	1	32	2
Decryption Latency (cycles)	288	18	576	36

While the original Moore state entry partitioning was most beneficial for m_AXIL example, that does not mean it will always be the best. Particularly, if the target FSM is Mealy and therefore uses the originally proposed Mealy state entry partitioning, the Mealy† state entry partitioning is guaranteed to result in the same in-scope and out-of-scope memory sizes while also reducing the in-scope memory width. Therefore, when transitioning from the Mealy state entry partitioning to the Mealy† state entry partitioning, one is guaranteed to have the same number of decryption latency rounds/cycles while benefiting from the decreased in-scope memory width. As such,

the Mealy† state entry partitioning would be very beneficial for target FSMs which are limited because of their width but have a shallow depth while using the Mealy state entry partitioning. Figure 5.5 later illustrates how like-sized FSMs can be locked with the Mealy† state entry partitioning using a significantly smaller memory size.

In extension to input multiplexing, we suggest that wire entanglement [60] could be used along it. Wire entanglement is the process of introducing a programable routing block that entangles the desired inputs and unnecessary noise signals such that, without knowledge of the explicitly secret programable routing, an attacker would be unable to identify which inputs to the routing block are original signals and which are noise [60]. These programmable routing blocks could be added to the design before the multiplexers shown in Figure 5.4, such that each routing block's inputs are connected to all of the FSM inputs and the outputs are connected to the unmodified input multiplexers. Then only with the proper programming of the routing block would the original signals needed for that multiplexer be passed through. This results in a design that multiplexes the inputs to the FSM, hence resulting in the aforementioned resource utilization improvements without disclosing what input pairs have the potential to influence a particular state.

In comparison, using fixed input multiplexing, an attacker with access to the netlist, presumably obtained via RE, could infer information about the targeted FSM by inspecting which input signals are multiplexed together and are never used simultaneously. This is not the case after the inclusion of wire entanglement because an attacker would be unable to separate desired inputs from the noise. The use of wire entanglement was not experimented with or modeled during this thesis, but we believe that it may be an area of interest for those who wish to preserve the complete confidentiality of the targeted FSM while taking advantage of the memory width and size improvements introduced via input multiplexing.

5.1.2 Performance Improvements

The next area of future work for the FSMLock methodology is performance improvements. As discussed in Subsection 3.2.2, there are fundamentally two aspects of the FSMLock primitive that govern performance: decryption latency and memory propagation delay. Since the latter is foremost dependent on memory architecture, we will focus on decreasing decryption latency. Equation 3.8 models the latency at power up (excluding any cipher-specific process such as round key generation) and during transitions between FSMLock scopes. From this equation, it can be inferred that there are two ways to decrease latency, reduce the number of cipher blocks required for each scoped region or decrease the cipher latency, i.e., decrease the time it takes to decrypt the next scoped region.

Assuming the size of the cipher block is fixed, to reduce the number of ciphers block required for each scoped region, one must instead aim to decrease the size of each scoped region. Consequently, a decrease in memory utilization directly results in increased performance, and we recommend that the techniques covered in Subsection 5.1.1 be considered not only to reduce memory width and size but also for performance gains. Table 5.5 and the surrounding discussion explain how the resource improvements to the m_AXIL example positively affect the decryption latency.

Also, take heed that increasing the number of state machine partitions with the intention of decreasing the size of each scoped region and therefore improving performance is unlikely to result in the desired behavior. While doing so effectively reduces the size of each scoped region and decreases decryption latency, the overall performance of the locked target FSM will stay relatively the same. As the number of scoped regions increases, so does the likeliness of transitioning between scopes during a transition, which incurs decryption latency. In summary, an increased partition count will reduce the duration of decryption latency but increase the frequency at which it is experienced. It is up to the system designer to decide whether this behavior

is advantageous compared to less frequent but longer periods of FSM downtime.

Of course, one could also disregard scope partitioning entirely, therefore opting to decrypt the entirety of the target FSM at power-up. This will maximize the latency value modeled in Equation 3.8, since the number of blocks in scope will be equal to the number of total blocks, and maximizes the required in-scope memory resources but ensures that said latency will only occur once and prior to run time. In such a configuration, FSMLock can be utilized to lock real-time sequential circuitry, as shown in the Simple example (Figure 3.2). The only limitation is the potential for an extended period of downtime during the initial unlocking/decryption process if the SET size is particularly large.

Other approaches to improve performance include the introduction of a ping-pong buffer [84] and the use of an encryption mode of operation such as counter [20] that allows for pre-computing of the cipher keystream. A ping-pong buffer overlaps the computations that cause decryption latency during regular FSM operation by allowing the Security/Scope Control unit to preemptively decrypt a scope that the target FSM will likely transition into next. Therefore, no decryption delay will occur if the preemptively decrypted scope is correctly chosen. Instead, the FSMLock primitive switches which buffer is treated as the currently scoped SET. Such a configuration will likely require additional logic, such that the Security/Scope Control unit can intelligently choose which scope to preemptively decrypt. Also, double the amount of in-scope memory resources will be required since two buffers are needed for the ping-pong structure. Pre-computing the AES keystream reduces decryption latency by front-loading the computationally intensive block cipher computations before decrypting the corresponding scoped region. Therefore, when the corresponding region must be decrypted, only an XOR operation must be performed to obtain the original plain text SET [20]. This improvement will double the required out-of-scope memory resources of the FSMLock primitive.

5.1.3 Automation Improvements

The final area of future work for the FSMLock methodology is automation improvements. Foremost, the inference of the target FSM from a preexisting RTL netlist would drastically reduce the burden on a system design when utilizing the FSMLock toolset. As shown in Figure 3.9, the expected input to the FSMLock automation toolset is a pre-formatted STT. Therefore, in its current condition, the system designer must generate the STT for the FSM before locking. This time-intensive task requires a deep understanding of the HDL source because the system designer must translate the HDL inferred states and conditional blocks into a list of transitions. Hence, it would drastically reduce the burden on the system developer if the automation toolset could infer an FSM from a pre-existing HDL hierarchy or RTL netlist instead.

The technology required for this improvement exists and has already been discussed in this thesis in Subsection 2.2.2, when covering background on RE techniques, and brought up again in Subsection 2.3.6 when the attacks on existing sequential logic locking techniques were discussed. Expressly, FSM extraction from gate-level netlists [45, 46, 47, 8] could be used to infer strongly connected FSM register components in a flattened RTL design and perform boolean function analysis on the associated feedback logic to determine the boolean representation of the next-state and output logic. After control registers, next-state, and output logic has been identified and removed, the post-improved automation toolset could use the boolean representation of the removed logic to generate the STT input, which natively interfaces with the pre-existing FSMLock toolset or directly generate the SET and its encrypted binary memory configuration representation.

Furthermore, the automatic identification of FSMs from the RTL netlist may be helpful in utilizing the FSMLock methodology in sequential circuits that were not originally modeled as an FSM. This is because FSM extraction from gate-level netlists

[8] operates on the flattened RTL and therefore uses the structural properties of an FSM to identify them. The identified FSMs from the [8] algorithm, including false positives which meet the structural criteria for an FSM but were never modeled as such, could then be considered potential candidates for the FSMLock primitive.

Other automation improvements include a remedy to the BMEM inflation demonstrated in Tables 4.2 and 5.2, the introduction of the Mealy† state entry partitioning into the software and firmware components of the toolset, and the automatic insertion of input multiplexers when the maximum number of SEI is less than the originally defined FSM input count. No experimental results were produced using the Mealy† state entry partitioning because of said lack of support from both the software and firmware side of the toolset. Further, all input multiplexed examples covered in Subsection 5.1.1 were manually configured before locking through the creation of a modified STT, as shown in Table 5.1, because the current condition of the toolset does not take the number of SEI into consideration. In firmware, the multiplexers required for input multiplexing were manually inserted into the hierarchy alongside the fsmlock_top instance shown in Figure 3.10.

5.2 Use Case Recommendations

Considering the resource and performance characteristics of the FSMLock primitive, we foremost recommend that FSMLock be used on targeted FSMs with a small number of input bits a. More precisely, we recommend that FSMLock be used on target FSMs with small $\max(SEI)$ since input multiplexing has proven to be an effective method for reducing input bits, and its use is recommended alongside the FSMLock primitive. Target FSMs that best take advantage of input multiplexing have a $\max(SEI) << |a|$. See Subsection 5.1.1 for a discussion on the calculation of $\max(SEI)$. Figure 5.5a illustrates what $\max(SEI)$, $|s|$, and $|y|$ bits parameters a targeted FSM can have while fitting within a predetermined number of Xilinx 36kbit

BMEM devices. Each Xilinx 36kbit BMEM device has a memory width of 72 bits, as defined in the Xilinx PG058 [80], and this full width is assumed to be utilized. Figure 5.5b likewise illustrates the valid max(SEI), $|s|$, and $|y|$ bit parameters for the Mealy† state entry partitioning scheme using three distributed RAMs configurations. These distributed RAM configurations were selected to best match the valid parameter curves in the like row Mealy and Moore state entry partitioning schemes. Notably, similarly sized target FSMs can be locked using the Mealy† state entry partitioning while utilizing a substantially smaller memory size–approximately 72 times smaller in the third row of Figure 5.5. The corresponding LUT resource utilization for these three configurations is emphasized in bold in Table 5.6.

Only max(SEI), $|s|$, and $|y|$ combinations that lie below the curves illustrated in Figure 5.5a are possible given the constraint on BMEM primitive count. Similarly, only combinations below the curves illustrated in Figure 5.5b are possible given the resource utilization dictated by the distributed RAM configuration Width $\times$ Depth selected.

Table 5.6: LUT utilization for distributed memory configurations. Results are from post-implementation runs of the Xilinx Distributed Memory Generator IP Version 8.0 [9] configured as a single port RAM with non-registered inputs and outputs while targeting the Artix-7 xc7a100t part.

		Depth				
		64	256	1024	2048	4096
Width	16	16	64	276	552	1104
	32	32	128	548	1096	2192
	64	64	256	1092	2184	4368
	128	128	**512**	**2180**	4360	**9482**

Additional configurations are shown in Table 5.6 to emphasize the flexibility of distributed memory. Unlike BMEM primitives with a fixed size, distributed memory is implemented directly in the fabric. Therefore, resource utilization can be more granularly adjusted as the total memory size decreases. For example, an FSMLock primitive that utilizes the Mealy state entry partitioning will require a minimum of

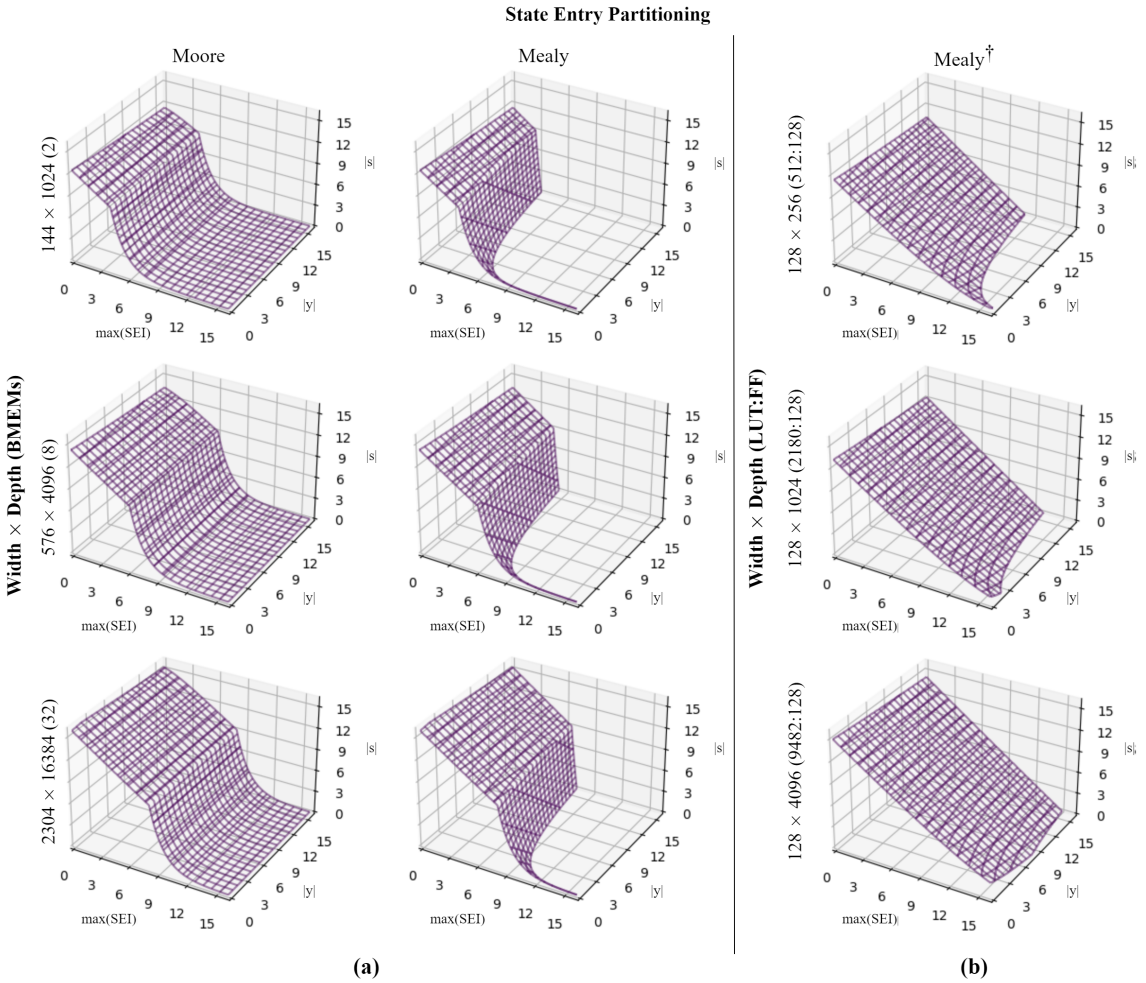

Figure 5.5: 3D graphs of the constraint on maximum state effective input bits $\max(SEI)$, state bit count $|s|$, and output bit count $|y|$ depending on (a) the number of Xilinx 36kbit block memory (BMEM) primitives allowed for or (b) the chosen distributed memory configuration from Table 5.6. Only configurations below each curve are possible for the chosen state entry partitioning scheme and resource constraints. The (a) Mealy and Moore state entry partitions, introduced in 3.4, are included alongside (b) the Mealy† proposed in Section 5.1.

two BMEM primitives for all memory widths between 72 and 144. In comparison, the LUT utilization of the same FSMLock primitive targeted towards the Mealy† state entry partitioning will continually decrease as the total required width and depth decreases.

5.3 Conclusion

To reiterate, the FSMLock primitive is a novel approach to logic locking which utilizes a block cipher to restrict access to and observation of locked sequential logic. Using a classical block cipher directly contributes to the outstanding security characteristics displayed by FSMLock. The explicit external secrecy of a block cipher key, incorporated through the abstraction of the target FSM sequential logic as encrypted memory contents, results in impressive security characteristics that resist the attack goals listed in Subsection 3.1.2 given the adversarial capability listed in Subsection 3.1.1. Unlike other sequential logic locking schemes that utilize encryption for key preprocessing [2, 24], FSMLock is less susceptible to removal attacks since the encryption engine is directly responsible for the decryption of the abstracted sequential logic. Without the encryption engine or the FSMLock primitive, the IC/FPGA system will cease to operate correctly.

The development of and related research efforts towards the FSMLock methodology have proven to be successful. Although it was determined that not all target FSM are suitable for locking via the FSMLock primitive assuming a fixed resource allocation budget for locking circuitry (Figure 5.5), we believe that this thesis and its related deliverables provide a clear overview of the FSMLock methodology, including its strengths in security and tradeoffs in resource utilization and performance. In doing so, we achieve our objectives listed in Section 1.2. Furthermore, considering the various future works outlined in Section 5.1, we believe that the applicability of the FSMLock primitive can be greatly extended beyond what is possible, given the current condition of the automation toolset. As such, there are clear paths forward for the FSMLock methodology, and we anticipate continuing improvements.

Bibliography

[1] M. Lukowiak, M. Kurdziel, S. Farris, and S. Radzisozwski, "System and method for obfuscation of sequential logic through encryption," U.S. Patent filed, 2022.

[2] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On Improving the Security of Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1411–1424, 2016.

[3] J. J. Rajendran and S. Garg, *Hardware Protection through Obfuscation*, D. Forte, S. Bhunia, and M. M. Tehranipoor, Eds. Cham: Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-49019-9_3

[4] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," in *DAC Design Automation Conference 2012*, 2012, pp. 83–89.

[5] 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)*, 2016, pp. 236–241.

[6] Y. Xie and A. Srivastava, "Anti-SAT: Mitigating SAT Attack on Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 2, pp. 199–207, 2019.

[7] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-Lock: Hard Distributions of SAT instances for Obfuscating Circuits using Fully Configurable Logic and Routing Blocks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[8] M. Fyrbiak, S. Wallat, J. Déchelotte, N. Albartus, S. Böcker, R. Tessier, and C. Paar, "On the Difficulty of FSM-based Hardware Obfuscation," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, pp. 293–330, 2018.

[9] AMD, *Distributed Memory Generator v8.0 Product Guide (PG063)*, AMD, 11 2015. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg063-dist-mem-gen

[10] J. Rajendran, O. Sinanoglu, and R. Karri, "Regaining Trust in VLSI Design: Design-for-Trust Techniques," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1266–1282, 2014.

[11] "IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)," *IEEE Std 1735-2014 (Incorporates IEEE Std 1735-2014/Cor 1-2015)*, pp. 1–90, 2015.

[12] D. R. Collins, "TRUST, A Proposed Plan for Trusted Integrated Circuits," Defense Advance Research Projects Agency Microsystems Technology, Fairfax, VA, Tech. Rep. 0704-0188, Mar. 2006.

[13] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit Integrated Circuits: A Rising Threat in the Global Semiconductor Supply Chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[14] G. Mura, R. Murru, and G. Martines, "Analysis of Fake Amplifiers," in *2021 IEEE 32nd International Conference on Microelectronics (MIEL)*, 2021, pp. 131–134.

[15] M. Ender, A. Moradi, and C. Paar, "The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1803–1819. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ender

[16] B. Miller, "CIA Triad," http://blog.electricfork.com/2010/03/cia-triad.html, 2010, accessed: 2022-08-29.

[17] L. L. P. D.E. Bell, "Secure Computer System: Unified Exposition and Multics Interpretation," The MITRE Corporation, Bedford, MA, Tech. Rep. ESD-TR-75-306, Mar. 1976.

[18] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," MIT Laboratory for Computer Science, Cambridge, MA, Tech. Rep., 1987.

[19] N. I. of Standards and Technology, "An Introduction to Information Security," U.S. Department of Commerce, Washington, D.C., Tech. Rep. National Institute of Standards and Technology Special Publication 800-12r1, 2017.

[20] ——, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques," U.S. Department of Commerce, Gaithersburg, MD, Tech. Rep. National Institute of Standards and Technology Special Publication 800-38A, 2001.

[21] S. Engels, M. Hoffmann, and C. Paar, "A critical view on the real-world security of logic locking," *Journal of Cryptographic Engineering*, vol. 12, no. 3, pp. 229–244, 2022.

[22] Y. Alkabani, F. Koushanfar, and M. Potkonjak, "Remote activation of ICs for piracy prevention and digital right management," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 674–677.

[23] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending Piracy of Integrated Circuits," in *2008 Design, Automation and Test in Europe*, 2008, pp. 1069–1074.

[24] J. Blocklove, S. Farris, M. Kurdziel, M. Łukowiak, and S. Radziszowski, "Hardware Obfuscation of the 16-bit S-box in the MK-3 Cipher," in *2021 28th International Conference on Mixed Design of Integrated Circuits and System*, 2021, pp. 104–109.

[25] J. Vosatka, *The Hardware Trojan War: Attacks, Myths, and Defenses*, S. Bhunia and M. M. Tehranipoor, Eds. Cham: Springer International Publishing, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-68511-3_2

[26] "Intel FPGAs and Programmable Solutions," https://www.intel.com/content/www/us/en/products/programmable.html, accessed: 2023-06-06.

[27] "FPGAs & 3d ICs," https://www.xilinx.com/products/silicon-devices/fpga.html, accessed: 2023-06-06.

[28] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan, "Hardware Trojan Attacks: Threat Analysis and Countermeasures," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1229–1247, 2014.

[29] S. Dupuis, P.-S. Ba, G. Di Natale, M.-L. Flottes, and B. Rouzeyre, "A novel hardware logic encryption technique for thwarting illegal overproduction and Hardware Trojans," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, 2014, pp. 49–54.

[30] T. Perez, M. Imran, P. Vaz, and S. Pagliarini, "Side-Channel Trojan Insertion - a Practical Foundry-Side Attack via ECO," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[31] D. Ziener, J. Pirkl, and J. Teich, "Configuration Tampering of BRAM-based AES Implementations on FPGAs," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018, pp. 1–7.

[32] T. Güneysu, I. Markov, and A. Weimerskirch, "Securely Sealing Multi-FPGA Systems," in *Reconfigurable Computing: Architectures, Tools and Applications*, O. C. S. Choy, R. C. C. Cheung, P. Athanas, and K. Sano, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 276–289.

[33] M. Fyrbiak, S. Strauß, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, "Hardware reverse engineering: Overview and open challenges," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 88–94.

[34] U. H. 98th Congress 2nd Session. (1984, Apr. 26) H.R.5525, Semiconductor Chip Protection Act of 1984. [Online]. Available: https://www.congress.gov/bill/98th-congress/house-bill/5525?r=1&s=1

[35] Wikipedia, "Semiconductor Chip Protection Act of 1984," Accessed Oct. 17, 2022 [Online]. [Online]. Available: https://en.wikipedia.org/wiki/Semiconductor_Chip_Protection_Act_of_1984#cite_note-9

[36] M. G. Rekoff, "On reverse engineering," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, no. 2, pp. 244–252, 1985.

[37] G. L. Zhang, B. Li, B. Yu, D. Z. Pan, and U. Schlichtmann, "TimingCamouflage: Improving circuit security against counterfeiting by unconventional timing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 91–96.

[38] R. Torrance and D. James, "The State-of-the-Art in Semiconductor Reverse Engineering," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 333–338.

[39] M. Schobert, "Degate," https://www.degate.org/, accessed: 2022-09-07.

[40] R. S. Rajarathnam, Y. Lin, Y. Jin, and D. Z. Pan, "ReGDS: A Reverse Engineering Framework from GDSII to Gate-level Netlist," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 154–163.

[41] "Synopsys EDA Tools," https://www.synopsys.com/, accessed: 2022-09-12.

[42] "Cadence," https://www.cadence.com/, accessed: 2022-09-12.

[43] "Siemens EDA Software," https://eda.sw.siemens.com/en-US/, accessed: 2022-09-12.

[44] J.-B. Note and É. Rannaud, "From the bitstream to the netlist," in *Symposium on Field Programmable Gate Arrays*, 2008.

[45] Y. Shi, C. W. Ting, B.-H. Gwee, and Y. Ren, "A highly efficient method for extracting FSMs from flattened gate-level netlist," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 2610–2613.

[46] T. Meade, S. Zhang, and Y. Jin, "Netlist reverse engineering for high-level functionality reconstruction," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016, pp. 655–660.

[47] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, "Gate-level netlist reverse engineering for hardware security: Control logic register identification," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 1334–1337.

[48] L. N. Nguyen, C.-L. Cheng, M. Prvulovic, and A. Zajić, "Creating a Backscattering Side Channel to Enable Detection of Dormant Hardware Trojans," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 7, pp. 1561–1574, 2019.

[49] A. Kahng, J. Lach, W. Mangione-Smith, S. Mantik, I. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking techniques for intellectual property protection," in *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, 1998, pp. 776–781.

[50] ——, "Constraint-based watermarking techniques for design IP protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1236–1252, 2001.

[51] S. Wallat, M. Fyrbiak, M. Schlögel, and C. Paar, "A look at the dark side of hardware reverse engineering - a case study," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 95–100.

[52] M. Schmid, D. Ziener, and J. Teich, "Netlist-level IP protection by watermarking for LUT-based FPGAs," in *2008 International Conference on Field-Programmable Technology*, 2008, pp. 209–216.

[53] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Logic encryption: A fault analysis perspective," in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 953–958.

[54] 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.

[55] S. M. Plaza and I. L. Markov, "Solving the Third-Shift Problem in IC Piracy With Test-Aware Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 961–971, 2015.

[56] Y.-W. Lee and N. A. Touba, "Improving logic obfuscation via logic cone analysis," in *2015 16th Latin-American Test Symposium (LATS)*, 2015, pp. 1–6.

[57] H. Mardani Kamali, K. Zamiri Azar, K. Gaj, H. Homayoun, and A. Sasan, "LUT-Lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 405–410.

[58] 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)*, 2015, pp. 137–143.

[59] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Logic Barriers," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 66–75, 2010.

[60] S. Khaleghi, K. D. Zhao, and W. Rao, "IC Piracy prevention via Design Withholding and Entanglement," in *The 20th Asia and South Pacific Design Automation Conference*, 2015, pp. 821–826.

[61] S. Roshanisefat, H. M. Kamali, and A. Sasan, "SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware," *CoRR*, vol. abs/1804.09162, 2018. [Online]. Available: http://arxiv.org/abs/1804.09162

[62] H. Zhou, R. Jiang, and S. Kong, "CycSAT: SAT-based attack on cyclic logic encryptions," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 49–56.

[63] M. E. Massad, S. Garg, and M. Tripunitara, "Reverse engineering camouflaged sequential circuits without scan access," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 33–40.

[64] Y. Hu, Y. Zhang, K. Yang, D. Chen, P. A. Beerel, and P. Nuzzo, "Fun-SAT: Functional Corruptibility-Guided SAT-Based Attack on Sequential Logic Encryption," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021, pp. 281–291.

[65] S. Roshanisefat, H. Mardani Kamali, H. Homayoun, and A. Sasan, "RANE: An Open-Source Formal De-obfuscation Attack for Reverse Engineering of Logic Encrypted Circuits," *Great Lakes Symposium on VLSI*, 2021.

[66] R. S. Chakraborty and S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[67] ——, "Security against hardware Trojan through a novel application of design obfuscation," in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, 2009, pp. 113–116.

[68] J. Dofe and Q. Yu, "Novel Dynamic State-Deflection Method for Gate-Level Design Obfuscation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 273–285, 2018.

[69] M. S. Rahman, R. Guo, H. M. Kamali, F. Rahman, F. Farahmandi, and M. Tehranipoor, "ReTrustFSM: Toward RTL Hardware Obfuscation-A Hybrid FSM Approach," *IEEE Access*, vol. 11, pp. 19 741–19 761, 2023.

[70] F. Koushanfar, "Provably Secure Active IC Metering Techniques for Piracy Avoidance and Digital Rights Management," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 51–63, 2012.

[71] A. R. Desai, M. S. Hsiao, C. Wang, L. Nazhandali, and S. Hall, "Interlocking Obfuscation for Anti-Tamper Hardware," in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2459976.2459985

[72] L. Gong and O. Diessel, "ReSim: A reusable library for RTL simulation of dynamic partial reconfiguration," in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–8.

[73] "Partial Reconfiguration Design Simulation," https://www.intel.com/content/www/us/en/docs/programmable/683834/21-3/partial-reconfiguration-design-simulation.html, accessed: 2023-06-23.

[74] H. Wang, D. Forte, M. M. Tehranipoor, and Q. Shi, "Probing Attacks on Integrated Circuits: Challenges and Research Opportunities," *IEEE Design & Test*, vol. 34, no. 5, pp. 63–71, 2017.

[75] M. Weiner, *Hardening Digital Circuits Against Invasive Attacks with On-chip Delay Measurements*. Technische Universität München, 2020. [Online]. Available: https://mediatum.ub.tum.de/doc/1547380/1547380.pdf

[76] A. Tiwari and K. Tomko, "Saving power by mapping finite-state machines into embedded memory blocks in FPGAs," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, 2004, pp. 916–921 Vol.2.

[77] I. Garcia-Vargas, R. Senhadji-Navarro, G. Jimenez-Moreno, A. Civit-Balcells, and P. Guerra-Gutierrez, "ROM-Based Finite State Machine Implementation in Low Cost FPGAs," in *2007 IEEE International Symposium on Industrial Electronics*, 2007, pp. 2342–2347.

[78] O. Goldreich, *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001.

[79] J. A. Halderman, S. D. Schoen, N. Heninger, William, Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," in *CACM*, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:7770695

[80] AMD, *Block Memory Generator v8.4 Product Guide (PG058)*, AMD, 8 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen

[81] ——, *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics (DS181)*, AMD, 2 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet

[82] N. I. of Standards and Technology, "Advanced Encryption Standard (AES)," U.S. Department of Commerce, Gaithersburg, MD, Tech. Rep. Federal Information Processing Standards Publication 197, 2001, updated: 2023.

[83] AMD, *7 Series FPGAs Configurable Logic v1.8 Block User Guide (UG474)*, AMD, 9 2016. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB

[84] Y.-M. Joo and N. McKeown, "Doubling memory bandwidth for network buffers," in *Proceedings. IEEE INFOCOM '98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No.98*, vol. 2, 1998, pp. 808–815 vol.2.