ABSTRACT

Title of dissertation:	DESIGNING EFFECTIVE LOGIC OBFUSCATION: EXPLORING BEYOND GATE-LEVEL BOUNDARIES
	Michael Jeffrey Zuzak Doctor of Philosophy, 2022
Dissertation directed by:	Professor Ankur Srivastava Department of Electrical and Computer Engineering

The need for high-end performance and cost savings has driven hardware design houses to outsource integrated circuit (IC) fabrication to untrusted manufacturing facilities. During fabrication, the entire chip design is exposed to these potentially malicious facilities, raising concerns of intellectual property (IP) piracy, reverse engineering, and counterfeiting. This is a major concern of both government and private organizations, especially in the context of military hardware. Logic obfuscation techniques have been proposed to prevent these supply-chain attacks. These techniques lock a chip by inserting additional key logic into combinational blocks of a circuit. The resulting design only exhibits correct functionality when a correct key is applied after fabrication. To date, the majority of obfuscation research centers on evaluating combinational constructions with gate-level criteria. However, this approach ignores critical high-level context, such as the interaction between modules and application error resilience. For this dissertation, we move beyond

the traditional gate-level view of logic obfuscation, developing criteria and methodologies to design and evaluate obfuscated circuits for hardware-oriented security guarantees that transcend gate-level boundaries.

To begin our work, we characterize the security of obfuscation when viewed in the context of a larger IC and consider how to effectively apply logic obfuscation for security beyond gate-level boundaries. We derive a fundamental trade-off underlying all logic obfuscation that is between security and attack resilience. We then develop an open-source, GEM5-based simulator called ObfusGEM, which evaluates logic obfuscation at the architecture/application-level in processor ICs. Using ObfusGEM, we perform an architectural design space exploration of logic obfuscation in processor ICs. This exploration indicates that current obfuscation schemes cannot simultaneously achieve security and attack resilience goals. Based on the lessons learned from this design space exploration, we explore 2 orthogonal approaches to design ICs with strong security guarantees beyond gate-level boundaries.

For the first approach, we consider how logic obfuscation constructions can be modified to overcome the limitations identified in our design space exploration. This approach results in the development of 3 novel obfuscation techniques targeted towards securing 3 distinct applications. The first technique is Trace Logic Locking which enhances existing obfuscation techniques to provably expand the derived trade-off between security and attack resilience. The second technique is Memory Locking which defines an automatable approach to processor design obfuscation through locking the analog timing effects that govern the function of on-chip SRAM arrays. The third technique is High Error Rate Keys which protect probabilistic

circuits against a SAT-based attacker by hiding the correct secret key value under stochastic noise. We demonstrate that all 3 techniques are capable of overcoming the limitations of obfuscation when viewed beyond gate-level boundaries in their respective applications.

For the second approach, we consider how architectural design decisions can influence hardware security. We begin by exploring security-aware architecture design, an approach where minor architectural modifications are identified and applied to improve security in processor ICs. We then develop resource binding algorithms for high-level synthesis that optimally bind operations onto obfuscated functional units to amplify security guarantees. In both cases, we show that by designing logic obfuscation using architectural context a designer can secure ICs beyond gate-level boundaries despite the presence of the rigid trade-off that rendered prior obfuscation techniques insecure.

DESIGNING EFFECTIVE LOGIC OBFUSCATION: EXPLORING BEYOND GATE-LEVEL BOUNDARIES

by

Michael Jeffrey Zuzak

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Dedication

To my wife and family, for standing by me throughout.

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Ankur Srivastava, for his mentorship. Without his dedication to my development, the late-night meetings, and his constant guidance and advice, I would have never pursued academia at all, nor achieved the academic and personal successes I have had along the way. His enthusiasm for research will always be both motivation and something to strive for as I move forward. It was always a pleasure to learn from him.

I would also like to thank Prof. Donald Yeung, Prof. Manoj Franklin, and Prof. Bruce Jacob for their advice and guidance throughout my PhD. The research and career discussions I had with each of them were invaluable.

I am grateful to my advisory committee, Prof. Manoj Franklin, Prof. Bruce Jacob, Prof. Donald Yeung, and Prof. Bobby Bhattacharjee, for the time and valuable feedback they have provided to improve this dissertation.

I also extend my thanks to my colleagues in Prof. Ankur Srivastava's research lab. I am grateful for the senior members, Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Yang Xie, and Zhiyuan Yang, for their camaraderie, guidance, and feedback. I am grateful for the junior members, Daniel Xing, Isaac McDaniel, Nina Jacobsen, Abir Akib, and Olsan Ozbay, for challenging me and bringing their infectious enthusiasm to every research discussion.

I would also like to express my gratitude to my friends, Ian Olver, Michael Titus, Rachel Cosgray, Kevin Merchant, Alana Merchant, Danny Kim, and Rebekah Kim, for their constant encouragement and support. They always found the time to watch presentation dry runs, read manuscripts, and give advice which strengthened my work and led to some great times in the process.

Finally, I would like to express my deepest gratitude to my wife, Manal Zuzak, and my parents, Gary Zuzak and Alison Zuzak. Their dedication to my success and their unwavering support throughout my life has made all of this possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADP	Area, Delay, Power
ALU	Arithemtic Logic Unit
BER	Bit Error Rate
DFG	Data Flow Graph
DI	Distinguishing Input
DIP	Distinguishing Input Pair
DRAM	Dynamic Random Access Memory
FPU	Floating Point Unit
FSM	Finite State Machine
FU	Functional Unit
HERK	High Error Rate Key
HLS	High-Level Synthesis
I/O	Input/Output
IC	Integrated Circuit
IP	Intellectual Property
LVI	Laser Voltage Imaging
MWP	Minterm-Wrong Key Pair
OP	Operation
ObfusGEM	Obfuscated GEM5
PI	Primary Input
PO	Primary Output
RM	Restore Multiplexer
RTL	Register Transfer Logic
SAT	Boolean Satisfiability Solver
SF	Stripped Functionality
SFLL	Stripped Functionality Logic Locking
SRAM	Static Random Access Memory
StatSAT	Statistical Boolean Satisfiability Attack
TDB	Tunable Delay Buffer
TLL	Trace Logic Locking

TPLUT	Tamper-Proof Look-Up Table
VLSI	Very Large Scale Integration
XNOR	Exclusive Negated Or
XOR	Exclusive Or

List of Publications

Journals:

1. **M. Zuzak**, A. Mondal, and A. Srivastava, "Evaluating the Security of Logic-Locked Probabilistic Circuits," in IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems (TCAD), 2021

2. Y. Liu, **M. Zuzak**, Y. Xie, A. Chakraborty, and A. Srivastava, "Robust and Attack Resilient Logic Locking with a High Application-Level Impact," in ACM Trans. on Design Automation of Electronic Systems (TODAES), 2021

3. **M. Zuzak**, Y. Liu, and A. Srivastava, "Trace Logic Locking: Improving the Parametric Space of Logic Locking," in IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems (TCAD), 2020

4. A. Chakraborty, N. Jayasankaran, Y. Liu, J. Rajendran, O. Sinanoglu, A. Srivastava, Y. Xie, M. Yasin, and **M. Zuzak**, "Keynote: A Disquisition on Logic Locking," in IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems (TCAD), 2019

5. D. Gerzhoy, X. Sun, **M. Zuzak**, and D. Yeung, "Exploiting Nested MIMD-SIMD Parallelism on Heterogeneous Microprocessors," in ACM Transactions on Architecture and Code Optimization (TACO), 2019

Conferences:

1. Y. Liu, **M. Zuzak**, D. Xing, I. McDaniel, P. Mittu, O. Ozbay, A. Akib, and A. Srivastava, "A Survey on Side-Channel-based Reverse Engineering Attacks on Deep Neural Networks," in Proceedings of the IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), 2022

2. **M. Zuzak**, Y. Liu, and A. Srivastava, "A Resource Binding Approach to Logic Obfuscation," in Proceedings of the Design Automation Conference (DAC), 2021 **(Best Paper Candidate)**

3. B. Tan, S. Garg, R. Karri, Y. Liu, **M. Zuzak**, A. Chakraborty, A. Srivastava, Et Al., "Independent Verification & Validation (IV&V) of Security-Aware CAD Tools," in Proceedings of the Design Automation Conference (DAC), 2021

4. **M. Zuzak** and A. Srivastava, "ObfusGEM: Enhancing Processor Design Obfuscation Through Security-Aware On-Chip Memory and Data Path Design," in Proceedings of the International Symposium on Memory Systems (MEMSYS), 2020

5. A. Mondal, **M. Zuzak**, and A. Srivastava, "StatSAT: A Boolean Satisfiability Attack on Logic Locking for Probabilistic Circuits," in Proceedings of the Design Automation Conference (DAC), 2020

6. Y. Liu, **M. Zuzak**, Y. Xie, A. Chakraborty, and A. Srivastava, “Strong Anti-SAT: Secure and Effective Logic Locking,” in Proceedings of the International Symposium on Quality Electronic Design (ISQED), 2020

7. Y. Liu, A. Mondal, A. Chakraborty, **M. Zuzak**, N. Jacobson, D. Xing, and A. Srivastava, “A Survey on Neural Trojans,” in Proceedings of the International Symposium on Quality Electronic Design (ISQED), 2020

8. **M. Zuzak**, M. Fitelson, S. Montano, and A. Srivastava, “Provable Detection and Location of Hardware Trojans with Linear Hybrid Cellular Automata,” in Proceedings of the Government Microcircuit Applications and Critical Technology Conference, 2020

9. **M. Zuzak** and A. Srivastava, “Memory Locking: An Automated Approach to Processor Design Obfuscation,” in Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2019

10. Z. Yang, **M. Zuzak**, and A. Srivastava, “HMCTherm: A Cycle-accurate HMC Simulator Integrated with Detailed Power and Thermal Simulation,” in Proceedings of the International Symposium on Memory Systems (MEMSYS), 2018

11. **M. Zuzak** and D. Yeung, “Exploiting Multi-Loop Parallelism on Heterogeneous Microprocessors,” in Proceedings of the International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG), 2017 **(Awarded Best Paper)**

Book Chapters:

1. Y. Liu, A. Mondal, A. Chakraborty, **M. Zuzak**, N. Jacobson, D. Xing, and A. Srivastava, “Neural Trojans,” in Encyclopedia of Cryptography, Security and Privacy, 2021

Technical Reports:

1. B. Tan, R. Karri, N. Limaye, A. Sengupta, ..., **M. Zuzak**, A. Srivastava, et al., “Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking,” in arXiv preprint arXiv:2006.06806, 2021

Posters:

1. **M. Zuzak**, “Designing Obfuscated Systems for Enhanced Hardware-Oriented Security,” at SIGDA Design Automation Conference (DAC) PhD Forum, 2021

2. **M. Zuzak**, “Securing Hardware in a Globalized Supply-Chain,” at ARCS Scholar Reception, 2020

3. **M. Zuzak**, “Building Functional ICs with Approximate Keys,” at CSAW’19 Logic Locking Conquest Finals, 2019

4. **M. Zuzak**, "Achieving Hardware Security: Design and Fabrication of Secure Integrated Circuits," at ARCS Scholar Reception, 2019

5. **M. Zuzak** and A. Srivastava, "Memory Locking: An Automated Approach to Processor Design Obfuscation," in Design Automation Conference (DAC), 2019

Chapter 1: Introduction

Custom IC design has surged in popularity for both military and commercial applications. As improving technology continues to drive demand for custom devices, the cost to create and maintain leading edge fabrication facilities increases as well. In 2008, a state-of-the-art foundry was estimated to cost \$5 billion [18]. By 2020, this number has more than tripled, with startup costs estimated at \$15-20 billion to setup a fabrication line for the smallest transistor size [89]. The substantial cost of these facilities has driven IC design companies to go *fabless*, relying on large unaffiliated foundries to fabricate their IP. This results in a number of security risks, which we consider throughout this dissertation.

1.1 Security Concerns During IC Fabrication

When a *fabless* business model is adopted by a design house, semiconductor fabrication is exported to an unaffiliated third party known as an *untrusted foundry*. By doing so, fabless design companies assume significant risk as untrusted foundries can reverse engineer, pirate, counterfeit, overproduce, or maliciously modify ICs [69]. We have included a block diagram of an extremely simplified fabless IC fabrication model in Figure 1.1.

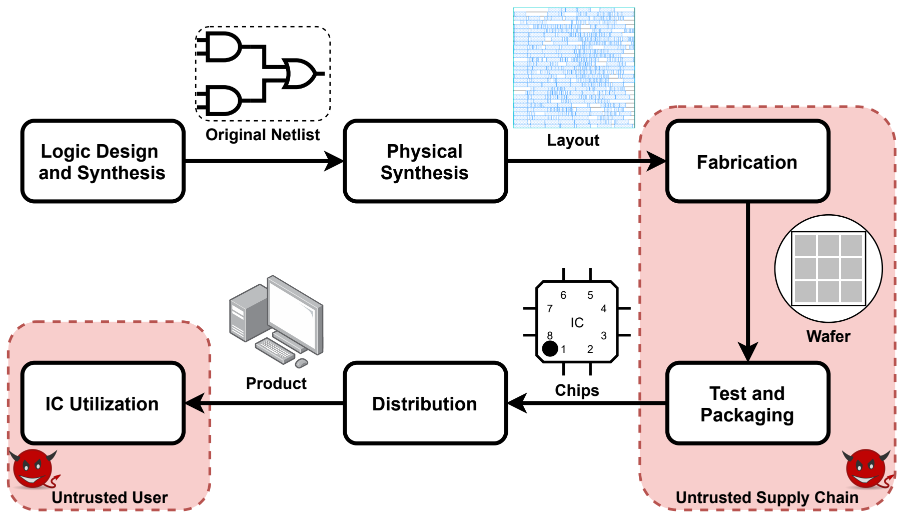


Figure 1.1: Overview of fabless business model for IC fabrication.

As shown in Figure 1.1, ICs are initially developed by a *design house*, who integrates both internal and purchased IP into a unified design to meet some specified functionality and performance requirements. This design undergoes *logic synthesis* to produce a netlist for the design. The *physical synthesis* process then converts this netlist into a layout that can be fabricated into an IC. At this point, a fabless business model distinguishes itself from a more traditional in-house fabrication model. A fabless design house exports their IC's physical design, in the form of GDSII files, to an unaffiliated foundry to fabricate it. The resulting wafers are then provided to a test and packaging facility to verify their functionality and incorporate them into a usable form factor. Finally, the packaged and tested ICs are returned to the design house for distribution, sale, or internal use.

There are 2 unique adversaries considered in Figure 1.1, an untrusted supply-chain entity (i.e. foundry or test facility) and an untrusted end-user. Regardless of the considered adversary, their goal (as considered in this dissertation) is fundamentally the same: to use design IP in an illegitimate or unauthorized fashion. Due to the cost and complexity of modern ICs, this design IP has substantial value, making such malicious activity especially attractive to an adversary. We list several commonly considered security threats brought about by the fabrication process outlined in Figure 1.1 below.

1. **Reverse-Engineering:** An untrusted foundry or malicious end-user can extract intimate design details (i.e. the netlist) from the GDSII files provided for fabrication [69]. This can be used either to gain competitive advantage or to enable piracy/counterfeiting of design IP.

2. **Piracy and Counterfeiting:** If a design is reverse engineered, design IP can be either counterfeited, integrated into counterfeit devices, or sold as-is on the gray market for lower prices.

3. **Overbuilding:** An untrusted foundry can produce additional unauthorized copies of a design that can be sold on the gray market.

4. **Hardware Trojans:** An untrusted foundry can maliciously alter a design prior to or during fabrication. For example, a kill-switch or security override procedure can be added [97]. These modifications are then present in all ICs fabricated by the foundry.

Each of these threats constitute a significant financial and security risk for IC design houses, especially in the context of military hardware. In this dissertation, we explore logic obfuscation as a means to mitigate hardware-oriented security concerns brought on by a fabless business model.

1.2 Hardware-Oriented Security Through Obfuscation

To mitigate the security risks identified in the previous section, researchers developed logic obfuscation (also called logic locking or logic encryption), a family of gate-level hardware security techniques that render IC functionality dependent on a secret key [64, 102, 70, 92, 94, 98, 103, 75, 104, 74, 68, 66, 111, 76, 33, 114, 45, 46, 34]. Without the correct secret key, logic obfuscation deterministically injects multi-bit errors at the output of any locked module. Given a sufficient error injection rate (i.e. the number of inputs that produce corrupt output compared to the size of the input space), a locked IC becomes unusable. By withholding the key from unauthorized users (e.g. an untrusted foundry), logic obfuscation can prevent the theft or unauthorized use of intellectual property (IP). Fundamentally, the goal of logic obfuscation is two-fold: 1) **Error Severity**: injecting sufficient error to render an IC unusable for any wrong key and 2) **Attack Resilience**: resisting attacks against it.

Despite logic obfuscation being proposed and implemented as a combinational, gate-level obfuscation technique, it cannot be evaluated at this level. This is the case because to ensure error severity, the first goal of logic obfuscation, gate-level error injection alone is insufficient. For error severity, gate-level errors must critically impact the application being run on an IC in order to prevent unauthorized use. Therefore, any effective obfuscation technique must induce sufficient gate-level error to thwart application-level IC functionality. This has been noted in related works as well, which argue that obfuscation cannot achieve security without considering an IC as a whole [114, 45, 46, 73, 60, 13, 58]. Despite research recognizing the importance of moving beyond gate-level boundaries, there has been no systematic evaluation of obfuscation or comprehensive approach to achieve security at this level.

This motivates the dissertation. The community has clearly shown that achieving security with logic obfuscation is necessarily a problem that must consider the entire IC. Therefore, considerations beyond the gate level must be taken into account in order to understand the security of logic obfuscation. In this dissertation, we aim to both evaluate security and develop the obfuscation techniques and design methodologies necessary to ensure that arbitrarily specified hardware security goals can be met in a broader IC, rather than solely within gate-level boundaries.

1.3 Contributions and Proposed Work

In this dissertation, we explore the hardware-oriented security of obfuscated ICs as a whole. This includes developing evaluation criteria, secure obfuscation techniques, and security-aware design methodologies. The major contributions of our work can be divided into the following 3 categories.

1.3.1 Evaluating the Security of Obfuscated Circuits Beyond Gate-Level Boundaries

For this research direction, we begin by deriving a fundamental trade-off between error severity and attack resilience, the 2 primary goals underlying all logic obfuscation, regardless of construction. This relationship forces integrated circuit (IC) designers to sacrifice the error severity of logic obfuscation to increase its attack resilience and vice versa.

We proceed by exploring the consequences of this trade-off by developing an architectural logic obfuscation simulation framework for processor ICs called ObfusGEM. First, we use ObfusGEM to simulate 9 benchmarks from the PARSEC [7] benchmark suite on a cycle-accurate GEM5 [8] model of 2 locked processor netlists (MIPS and 80186). We find that the identified trade-off prevents logic obfuscation configurations with feasible area, delay, and power overheads from achieving error severity and attack resilience in either core.

To generalize this result, we perform a Monte-Carlo-style design space exploration of logic obfuscation in a software model of 2 modern processor ICs (x86 and ARM A53). Specifically, we perform 50,400 trials incorporating unique locking configurations sweeping over the entire obfuscation design space throughout each processor. We find that state-of-the-art locking techniques are incapable of simultaneously achieving both error severity and attack resilience in either of these ICs when applied blindly at the gate level (as proposed by prior work [64, 102, 70, 92, 94, 98, 103, 75, 68, 66, 76, 33, 34, 74]). Because this trade-off exists regardless of logic obfuscation scheme, these results not only identify limitations of existing art, but also indicate that novel logic obfuscation techniques utilizing *conventional gate-level strategies/constructions* will also experience similar challenges.

1.3.2 Obfuscation Techniques for Security Beyond Gate-Level Boundaries

We proceed by developing 3 novel obfuscation techniques capable of overcoming the limitations identified by our design space exploration in the previous chapter. First, we propose Trace Logic Locking, a novel enhancement of gate-level logic locking which enables existing art to secure arbitrary length sequences of input minterms, referred to as *traces*. Doing so injects an additional degree of freedom into the parametric space of locking, enabling locking techniques to overcome the limitations of our derived trade-off. We both theoretically and empirically prove this by using Trace Logic Locking to enhance cutting edge obfuscation techniques. In 10 large benchmarks, we show that Trace-Logic-Locking-enhanced logic obfuscation provides exponentially stronger attack resilience than conventional locking with only modest additional overhead. Finally, we demonstrate the efficacy of Trace Logic Locking in a processor IC using architectural simulations. Despite prior art being unable to secure this IC, we find that Trace Logic Locking concurrently achieves strong error severity and attack resilience.

Second, we propose Memory Locking, an automatable logic obfuscation technique capable of denying application-level functionality to the adversary while maintaining SAT resistance. We target on-chip SRAM circuitry due to the 50-90% of transistor count dominated by SRAM-related circuitry in modern processors [59]. This creates significant flexibility in obfuscatable location and functionality. Additionally, the analog effects governing SRAM make it resistant to many proposed attack methodologies such as SAT-based attacks. We then demonstrate the application-level effectiveness of Memory Locking compared to prior art with ObfusGEM simulations.

Third, we propose High Error Rate Keys (HERK) to thwart StatSAT and other prominent attacks on probabilistic circuits. HERKs leverage high error wires, caused by probabilistic behavior, to hide the correct key under stochastic noise. HERKs can be integrated alongside prior deterministic logic obfuscation schemes for strong IP protection in probabilistic circuits. We demonstrate the efficacy of HERKs in several benchmark circuits, empirically verifying their resilience to StatSAT and other SAT-style attacks. As such, HERKs can exploit characteristics unique to probabilistic circuits to enable high-error obfuscation to be used without sacrificing SAT attack resilience, thereby enabling obfuscation configurations that are simultaneously high in error severity and attack resilience.

1.3.3 Design Methodologies for Security Beyond Gate-Level Boundaries

For our third research direction, we explore how high-level (i.e. architecture, application, system) design decisions can influence hardware-oriented security. First, we explore the possibility of a security-aware architecture design approach to enhance logic obfuscation techniques. To this end, we direct our attention to the most commonly proposed candidate modules for logic obfuscation: 1) the on-chip memory, such as cache controllers [13] or SRAM memory [114], and 2) the data path, such as ALUs [45, 46] or alternative compute units [73, 58]. Within these candidate locations, we propose and evaluate a quantitative, tool-driven design approach for both on-chip memory and data path architectures to enhance application-level security guarantees with logic obfuscation.

Next, we consider using the architectural context available during the resource binding phase of high-level synthesis (HLS) to co-design architectures and locking configurations capable of high error severity *and* SAT resilience simultaneously. To do so, we propose 2 security-focused binding/locking algorithms and apply them to bind/lock 11 MediaBench benchmarks. The resulting circuits showed a 26x and 99x average increase in the application errors of a fixed locking configuration while maintaining SAT resilience and incurring minimal overhead compared to other binding schemes.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides a survey of related/prior work on topics in hardware security and logic obfuscation. Chapter 3 provides background context and preliminary information on logic obfuscation and other relevant concepts relied on in this dissertation. Chapter 4 develops evaluation criteria for logic obfuscation, derives theoretical limitations based on this criteria, and presents the results of a design space exploration of logic obfuscation in processor ICs. Chapter 5 develops 3 novel logic obfuscation constructions with comprehensive hardware security guarantees beyond gate-level boundaries. Chapter 6 highlights 2 methodologies that utilize architectural design decisions to produce obfuscated ICs with hardware security guarantees beyond gate-level boundaries. Chapter 7 concludes the dissertation and suggests several directions for future research based on this work.

Chapter 2: Related Work

A large amount of work has been conducted to address the untrusted foundry problem. In this chapter, we provide a systematic overview of the work in this space. We begin with a review of the literature on logic obfuscation, a prominent research direction aimed at addressing the untrusted foundry problem. This review is relevant to and referenced in all chapters of this dissertation.

Following this review of prior logic obfuscation research, we highlight prior art on logic obfuscation pursuing security guarantees beyond gate-level boundaries. This work informs Chapter 4, where we introduce criteria to evaluate logic obfuscation beyond the gate level, considering the IC as a whole, and perform a design space exploration based on this criteria. Next, we provide an overview of current state-of-the-art logic obfuscation techniques, including research investigating obfuscation in probabilistic circuits. This work informs Chapter 5, where we develop 3 architecturally secure logic obfuscation constructions and evaluate them against this state-of-the-art. Finally, we consider work on logic obfuscation during the high-level synthesis (HLS) process. This work informs Chapter 6, where we explore security-aware architectural design methodologies.

2.1 Logic Obfuscation Techniques

To mitigate hardware security risks caused by an untrusted foundry (see Section 1.1), a family of hardware security techniques known as *logic obfuscation*, also called *logic locking* and *logic encryption*, has been proposed [5, 22, 64, 65, 70, 94, 92, 98, 114, 99, 103, 74, 75, 68, 66, 76, 102, 33, 34, 45, 46, 73]. A comprehensive survey of logic obfuscation research can be found in [11, 69, 79]. Logic obfuscation protects ICs from unauthorized use by inserting auxiliary combinational logic into IC modules. This added logic is designed to link locked module functionality to additional primary inputs, known as *key inputs*. By doing so, IC functionality becomes dependent on these key inputs. Only by applying the correct value to key inputs, known as the *secret key*, can correct IC functionality be unlocked. Fundamentally, the goal of logic obfuscation is two-fold: 1) **Error Severity**: injecting sufficient error to render an IC unusable for any wrong key and 2) **Attack Resilience**: resisting attacks against it.

Early research into logic obfuscation was dominated by approaches involving the insertion of key-driven accessory logic such as XOR/XNOR gates, multiplexers, or look-up tables into combinational modules within an IC [69]. In response to these techniques, an oracle-based attack, known as a SAT attack, was developed [88, 24] and expanded [83, 77, 4, 108, 23]. SAT attacks use a Boolean satisfiability (SAT) solver to iteratively eliminate all incorrect keys from the key-space, thereby locating the secret key. These attacks proved to be quite potent, quickly unlocking most existing locking techniques [5, 22, 64, 65, 70]. As a result, logic obfuscation began to incorporate SAT attack resilience guarantees [94, 92, 98, 114, 99, 103, 74, 75, 68, 66, 76, 106, 33, 45, 46]. Generally, these techniques can be characterized by 2 approaches: 1) non-digital obfuscation (i.e. delay locking [93], performance locking [105], mixed signal locking [32, 39], etc.) and 2) SAT resistant logic blocks (i.e. Anti-SAT block [94, 92], SARLock [98], CAS-Lock [76], SFLL [103, 75, 74], etc.). Recent work in [12] and [40] proposed the TimingSAT attack which was able to unlock delay-based obfuscation techniques such as [93] using a modified SAT-style attack procedure. On the other hand, many SAT-resistant techniques, such as Anti-SAT and SARLock, are resistant to conventional SAT attacks, but succumb to approximate SAT attacks, such as AppSAT [77] and Double DIP [83]. These approximate SAT-style attacks relax full functional correctness guarantees in favor of quickly finding a high-fidelity key. By relying on inherent error resilience properties of ICs, these approximate keys enable ICs to successfully execute most intended workloads, rendering locking ineffective [115, 114, 111].

Recent research has expanded on this result, suggesting that logic obfuscation cannot induce a sufficient number of errors to critically impact a locked IC (error severity) while maintaining resilience to a SAT attack (attack resilience) [81, 115]. This is due to a fundamental trade-off between the number of locked inputs and SAT attack resilience that has been identified underlying obfuscation [107, 109, 45, 46, 103]. To explore this relationship, researchers have provided limited derivations of it for specific techniques [103] and loose generic bounds for locking as a whole [107, 109, 45, 46]. This trade-off requires that locking corrupt only a small number of input/output pairs to be SAT resilient. A small number of locked inputs causes only a small number of errors, which are often inadequate to overcome the error resilience of ICs [42]. This creates a dilemma. High SAT resilience requires low combinational module corruption, however, we also need high application corruption for wrong keys. To overcome it, we must move beyond the gate-level, instead taking an architectural view of locking. This dilemma defines current state-of-the-art logic obfuscation research that aims to provide extremely tunable constructions that exhibit provable security guarantees within this trade-space [103, 75, 74, 111, 45, 46]. We further elaborate on current state-of-the-art logic obfuscation techniques in Section 2.4.

2.2 Attacks on Logic Obfuscation

The prevalence of logic obfuscation prompted the development of a variety of attacks against it. These attacks can be divided into two families: 1) logical and 2) physical. Logical attacks infer the key based on information leaked by the Boolean behavior or structural topology of the obfuscated circuit [88, 24, 4, 14, 85, 77, 83, 101, 96]. One of the most prolific logical attacks is the Boolean satisfiability (SAT) attack which identifies the secret key using a SAT solver to compare the logical function of a locked circuit and an unlocked (i.e. correctly-functioning) circuit, known as an oracle [88, 24, 4, 77, 83]. Conversely, physical attacks infer the key based on empirical side-channel data collected from an oracle IC [82, 37, 63]. These attacks often employ non-invasive radiation measurements to gather side-channel data. For example, electro-optical probing attacks employ an optical laser (i.e. a probe) to illuminate a die feature, usually a small number of transistors, and measure the corresponding reflected power. The index of refraction of the illuminated die feature is dependent on its sensitization (i.e. the signals applied to it). This allows attacks, such as [63], to analyze the spectral components of the power reflected by key register output buffers to infer the key of an obfuscated circuit. This is known as an electro-optical frequency mapping (EOFM) attack on obfuscation.

Countermeasures have been developed to address both logical and physical families of attacks. For example, logical attacks, such as the SAT attack [88, 24, 4, 77, 83], are thwarted by adding SAT-resilient [94, 75, 74, 103] or SAT-Hard [33, 34] instances. These are Boolean functions that yield an unduly complex circuit when viewed through a SAT-based lens. Conversely, physical attacks, such as electro-optical probing [82, 63], rely on a rigid approach capable of extracting leakage only from key registers. A designer can include scatterers or noisy components alongside key registers to blur the side-channel and mitigate these attacks [82, 61].

2.3 Moving Beyond Gate-Level Security with Logic Obfuscation

Despite logic obfuscation being proposed and implemented as a gate-level locking technique, it cannot be evaluated at this level. This is the case because to ensure error severity, the first goal of logic obfuscation, gate-level error injection alone is insufficient. For error severity, gate-level error must critically impact the application being run on an IC. Therefore, any effective obfuscation technique must induce sufficient gate-level error to thwart application-level IC functionality. Logic obfuscation research has begun to recognize that security must move beyond the gate-level, extending the the IC as a whole, by noting that obfuscation cannot achieve security without considering an IC's unique architecture [114, 45, 46, 73, 60]. However, despite the presence of research that recognizes the importance of moving beyond gate-level considerations for security, there has been no systematic evaluation of logic obfuscation or design approach to achieve security at this higher level. For example, the work in [73] secures a large, multi-million gate chip with a focus on system-level impact, however, it performs only qualitative analysis to do so. Alternatively, [58, 60] merely recognizes the need for architectural considerations, but does not delve into the specifics of obfuscation at this level. On the other hand, [45, 46, 13, 114] provide a limited evaluation of their proposed techniques in a specific architecture, but do not provide any methods to design or verify architecturally secure obfuscation configurations. Therefore, while these works provide substantial evidence that architectural potency is necessary, they fall short of providing a quantitative understanding of architectural security or a method to reliably achieve it.

2.4 State-of-the-Art Logic Obfuscation Techniques

There exists a diverse array of state-of-the-art logic obfuscation techniques [111, 76, 33, 74, 103, 75, 45, 46]. The locking techniques with the most traction in the community can be classified into 2 families, denoted 1) critical minterm locking and 2) exponential SAT iteration runtime locking. Critical minterm locking schemes include techniques such as Stripped Functionality Logic Locking (SFLL) [103, 75, 74], Strong Anti-SAT [45, 46], and Trace Logic Locking [111]. These techniques allow an IC designer to select specific *critical input minterms* in a locked module and force them to produce errant output for a large subset of wrong keys. Such techniques usually guarantee that the expected number of SAT iterations scales exponentially in key length. Exponential SAT iteration runtime locking schemes include techniques such as Full-Lock [33], InterLock [34], LoPher [71], and Cross-Lock [78]. These techniques cause the runtime of successive SAT attack iterations to increase exponentially.

2.5 Probabilistic Circuits and Obfuscation

Probabilistic circuits, in terms of architecture [110, 27], design methodology [36, 91], and modeling [3, 43], have been heavily studied. Probabilistic IP is most often considered in signal processing, machine learning, or embedded applications where error tolerance is high and power savings are required to meet specifications [27]. Within these applications, arithmetic logic (e.g. adder, multiplier, etc.) is among the most commonly considered targets for probabilistic behavior [110, 27]. Arithmetic circuits are some of the largest combinational blocks in modern systems, making them ideal obfuscation candidates [104, 112]. This makes the obfuscation of probabilistic design IP an attractive option. On the other hand, the use of probabilistic gates has been explored as a possible obfuscation technique as well.

For example, [55] and [56] propose using probabilistic giant spin Hall effect gates in order to obfuscate a circuit. The authors argue that the non-deterministic behavior of these probabilistic gates mitigate deterministic SAT attack strategies, enabling stronger security guarantees. However, [56, 51] demonstrated that modified SAT-style attacks could be launched to unlock obfuscation schemes relying on these probabilistic gates.

2.6 Boolean Satisfiability Attacks Against Probabilistic Circuits

The conventional SAT attack [88, 24, 4] is good for deterministic circuits, but not for probabilistic ones because the latter exhibits inconsistent behavior which can be detrimental to the progress of the attack. Research has suggested exploiting these limitations of the conventional SAT attack by inserting probabilistic gates within otherwise deterministic circuits to protect them [55]. In response to these limitations, the work in [56] proposes a Probabilistic SAT (PSAT) attack to handle this.

In [56], the authors first show that the probabilistic behavior of the gates in the oracle misguides the attacker (and hence a SAT solver) as they can be tricked into using a wrong output for a DI. This causes the standard SAT attack, described in Sec. 3.3, to fail even at barely significant error levels[1]. They then propose

[1]An approximate SAT attack proposed in both [77, 83] also cannot be applied to unlock a probabilistic circuit because it too requires a deterministic oracle.

a modified version of the SAT attack, called Probabilistic SAT (PSAT), wherein the oracle is queried multiple times for a certain DI instead of just once. This is done to circumvent the inconsistency of the oracle's output. If the most frequently occurring output pattern is dominant (see [56] for the meaning of dominant), it is considered the correct output[2]. Otherwise, one of the output patterns is sampled with a probability equal to its frequency of occurrence and is chosen as the correct output. The PSAT attack produces better results than the conventional SAT attack. The StatSAT attack was introduced in [51, 113] to improve upon the PSAT attack and enable obfuscated probabilistic IP with a higher error rate to be successfully attacked. Such attacks demonstrate that an untrusted foundry can steal the IP of approximate/probabilistic chips potentially employed in systems wherein probabilistic computing is a sought-after framework through SAT-style attack methodologies [56, 51, 113].

2.7 Logic Obfuscation During High-Level Synthesis

Prior work has explored high-level synthesis (HLS) in the context of logic obfuscation [58, 104, 17, 57, 31, 53, 112]. The TAO technique [58] suggested transformations to obfuscate a design during HLS. However, TAO assumes a restrictive attacker model where the adversary cannot have access to a working chip. This limits the use of TAO to a small subset of logic obfuscation use cases where the IC

[2]A correct output is what the oracle would output if it was not probabilistic.

is never distributed beyond the design house (e.g. government fabrication of military ICs). Doing so restricts the use of the SAT attack [88, 24, 83, 77, 4, 108, 23], which quickly unlocks the high-error locking used by TAO [88], limiting its utility.

SFLL-HLS proposes an extra HLS step to identify IC modules with a sufficient number of inputs to support locking [104]. Then, based on simulations of the RTL design, they tune the size of their locking configuration to ensure security. While this approach occurs during HLS, it does not directly integrate with HLS algorithms to inform either the design's RTL, or the configuration of logic obfuscation to improve security. Rather, it serves as an argument that supports architectural consideration for logic obfuscation.

DECOY presents a tighter integration with HLS, however, it still does not integrate into any phase/algorithm of HLS [17]. Instead, DECOY adds an HLS step to partition the design into critical and non-critical IP. Critical IP is then implemented in a separate eFPGA, with non-critical IP implemented in an ASIC. As a result, the eFPGA, which exhibits strong reverse-engineering protection, obfuscates the critical IP. While this yields security, it also introduces substantial design complexity and overhead. Such an approach is often untenable.

While both SFLL-HLS and DECOY recognize the importance of HLS's context, they fail to capitalize on the RT-level design decisions made during this phase. Instead, they rely on standard HLS algorithms that optimize for parameters such as switching activity [16], or register re-use [30]. This is a missed opportunity as these algorithms can make RT-level design decisions to optimize and inform supply-chain security instead, as we later show in Chapter 6.

Chapter 3: Preliminaries

This chapter provides a brief background on topics explored in this dissertation. We begin the chapter with a basic introduction to logic obfuscation (also called logic locking or logic encryption) constructions. We then formalize the untrusted foundry attacker model considered in this work and introduce the mathematics guiding Boolean satisfiability-based attacks, which are potent under this attacker model. Next, we provide an overview of the state-of-the-art stripped-functionality logic locking (SFLL) technique, which is used for comparison and evaluation throughout this work. We also consider obfuscation and attack strategies against obfuscation in the context of probabilistic circuits, an application which is addressed in Chapter 5. Finally, we briefly introduce the high-level synthesis (HLS) process, which is modified to enable obfuscation-aware design in Chapter 6.

3.1 Logic Obfuscation

Logic obfuscation (also called logic locking or logic encryption) is a diverse family of combinational hardware security techniques aimed at preventing unauthorized IC use, such as piracy, counterfeiting, overproduction, and reverse engineering, by untrusted elements in an IC's fabrication supply chain [64, 70, 94, 92, 98, 114, 99, 103, 74, 75, 68, 66, 76, 102, 33, 73, 77, 83, 81, 80, 88, 4, 108, 45, 46, 111, 33, 34].

It is characterized by the introduction of accessory combinational logic into modules within an IC. This accessory logic is driven by both internal logic signals and an added set of primary inputs, known as *key inputs*, which are driven by a tamper-proof memory included in the design. Following fabrication, this tamper-proof memory is loaded with a user-defined value known as the *secret key* of the logic locking construction.

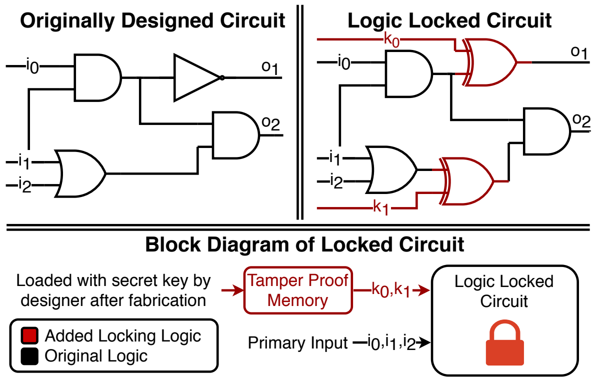


Figure 3.1: Configuration of a logic locked module.

Through this construction, the functionality of a locked module becomes dependent on this secret key. This means that an IC will exhibit incorrect functionality (i.e. deterministic multi-bit error injections) whenever some key other than the correct secret key is applied. By sufficiently corrupting IC functionality, a locked IC becomes unusable. Hence, by withholding the secret key from any unauthorized user, logic locking will render the IC unusable for these entities, restricting unauthorized use. In order to be successful, logic locking must achieve 2 primary goals: 1) **Error Severity**: injecting sufficient error to render an IC unusable for any wrong key and 2) **Attack Resilience**: resisting attacks against it. Error severity is generally related to the *wrong key error rate* of the logic locking construction, defined as the average number of inputs that produce errant outputs for a wrong key compared to the total possible input combinations. Attack resilience is usually defined as the time or number of attack iterations required to recover the secret key for a given locking construction. A generic example of a logic locked module is shown in Figure 3.1.

3.2 Attacker Model

In this dissertation, we consider the most common adversary used in recent research to assess hardware-oriented security in the presence of an untrusted foundry [94, 92, 98, 114, 99, 103, 74, 75, 68, 66, 76, 102, 33, 73, 77, 83, 81, 80, 88, 4, 108, 45, 46, 111, 76, 33, 34]. Specifically, we consider an adversary who takes some strategy utilizing:

1. A locked netlist of the IC, C, which can be obtained through reverse engineering the GDSII file provided for fabrication [69].
2. A correctly-keyed, black-box oracle IC, C_o. This can be obtained through the open market or IC testing facilities. The adversary can query the black box oracle with an input and record the correct output, denoted as $y \leftarrow C_o(x)$.

The goal of the attacker (e.g. an untrusted foundry) is to create an IC sufficient for sale or IP piracy. Because ICs are designed to run a set of specific applications, any successful defense strategy must ensure critical failures in these workloads for wrong keys. Therefore, the attacker's goal is to obtain an IC capable of running these specific applications. In this dissertation, we quantify this with application failure rate and mean time to failure. A higher failure rate or shorter time to failure indicates more secure locking. While such success criteria allows experimental evaluations, it is too informal to enable theoretical derivations. To this end, we formally define the attacker's goal to be finding a key, k, such that $\{\forall x, C(x,k) = C_o(x)\}$.

3.3 SAT-Based Attacks

In response to logic locking, a Boolean satisfiability attack (SAT attack) was proposed which quickly unlocked most existing logic locking art [88, 24, 83, 77, 4, 108, 23]. The goal of this attack was to locate a key (k) that when applied to the locked IC (C), yielded identical output (y) to an unlocked oracle IC (C_o), regardless of input (x). Hence, a key satisfying $\{\forall x, C(x,k) = C_o(x)\}$. To perform this attack, we must convert the locked circuit to conjunctive normal form (CNF): $C_{cnf}(x,k,y)$. This form evaluates to true only if an assignment of x, k, and y can be found such that $y = C(x,k)$. By using a CNF-SAT solver on the CNF circuit, the attack proceeds as follows:

1. An input (x_{di}) and 2 keys (k_1, k_2) must be found such that when this input is applied to the locked circuit, each key produces a different output (y_1, y_2).

   $$C_{cnf}(x_{di}, k_1, y_1) \wedge C_{cnf}(x_{di}, k_2, y_2) \wedge (y_1 \neq y_2) \tag{3.1}$$

   This input, x_{di}, is called a *distinguishing input (DI)*.

2. The DI is applied to C_o and the output, y_{di}, is recorded.

3. During each iteration, a pair of keys k_1, k_2 must be found which produce correct output for all previously located DIs (x_j) along with an additional new DI, x_{di}.

$$\begin{aligned} C_{cnf}(x_{di}, k_1, y_1) \wedge C_{cnf}(x_{di}, k_2, y_2) \wedge (y_1 \neq y_2) \\ \bigwedge_{j=1}^{i-1} (C_{cnf}(x_j, k_1, y_j) \wedge C_{cnf}(x_j, k_2, y_j)) \end{aligned} \tag{3.2}$$

4. The SAT solver operates on (3.2) until it is unsatisfiable. This indicates that no further DIs remain. A final key is found which matches the oracle's output on all tested DIs. This key is functionally equivalent to the correct key.

3.4 Stripped Functionality Logic Locking (SFLL)

SFLL is a prominent gate-level locking scheme under the SAT attack model [103, 75, 73, 74]. At the core of SFLL is the idea of functionality stripping, defined as the incorrect and permanent alteration of the output produced when specific inputs are applied to a locked module. This stripped functionality is then corrected by some added logic, known as the *restore unit*, when a correct key is applied. By functionality stripping more or smaller minterms, the wrong key error rate of an SFLL construction can be modified. This makes SFLL a uniquely tunable locking construction. The work in [103, 75, 73, 74] introduces multiple locking constructions, however, we focus on SFLL-Fault [75].

SFLL-Fault [75] is a fault injection and automatic test pattern generation (ATPG) driven strategy to implement the SFLL-Flex construction outlined in [103]. The construction of SFLL-Fault consists of a functionality stripped module and a tamper-proof look-up table (TPLUT) as the restore unit. In SFLL-Fault, the TPLUT is indexed by the secret key. When the input to the locked module matches the index of the TPLUT (secret key), a restore signal is provided which inverts the locked module's output. In the presence of a correct key, the restore signal will correct output errors induced by stripped functionality. In the presence of a wrong key, the restore signal will corrupt correct outputs, causing more error.

3.5 Estimating Bit Error Ratio (BER) in Probabilistic Circuits

Consider a 2-input gate G with input signal probabilities p_1 and p_2 and input error probabilities ϵ_1 and ϵ_2. These inputs can be erroneous if they are outputs of probabilistic gates. Further, let ϵ_g be the error probability of G itself, which is the probability that G produces wrong output irrespective of its inputs. The error probability ϵ at the output of G can be modeled in terms of the above known quantities using Boolean Difference Calculus [50, 87]. This is done by propagating the errors at the inputs of a gate to its output.

For any circuit, the output BERs can be estimated for any input vector and gate error ϵ_g using the Boolean Difference Calculus [50]. It is done by modeling the error probability at the output of each gate in terms of its input signal and error probabilities. This propagates the errors at the inputs of a gate to its output. For a 2-input AND gate, the output error rate is:

$$\epsilon = \epsilon_g + (1 - 2\epsilon_g)(\epsilon_1 p_2 + \epsilon_2 p_1 + \epsilon_1\epsilon_2(1 - 2(p_1 + p_2) + 2p_1p_2)) \tag{3.3}$$

Notice that this error probability (and that of an OR gate) depends on both the input error and signal probabilities, whereas that of the XOR gate depends only on the error probabilities. This is because the AND and OR gates have skewed outputs, whereas the XOR gate has symmetric outputs.

$$\epsilon_{XOR} = \epsilon_g + (1 - 2\epsilon_g)(\epsilon_1 + \epsilon_2 - 2\epsilon_1\epsilon_2) \tag{3.4}$$

By chaining these equations together such that they match a circuit's topology, an input vector can be propagated through them to estimate each output's BER for that input vector.

3.6 StatSAT Attack on Logic Obfuscation in Probabilistic Circuits

Obfuscated probabilistic IP is a challenging target for a SAT-based adversary. To show this, consider a common approach to probabilistic adders [110, 43], where high-order bits are implemented deterministically and low-order bits probabilistically. If we assume the 8 least significant output bits use probabilistic logic, thereby producing a correct output 90% of the time (independent of other outputs), we can calculate the probability of an entirely correct output pattern to be $0.9^8 < 50\%$. Thus, the circuit is more likely to produce an incorrect output pattern than a correct one. However, the average effective error (i.e. the mean difference between the probabilistic and deterministic sum) is only $\sum_{i=0}^{7} 0.1 * 2^i = 25.5$ for this circuit. If we consider the full range of a 16-bit adder, this error amounts to only a small deviation (0.04%) from the intended sum. These properties combine to create a design with a small effective error that produces incorrect output patterns with a high probability. The high likelihood of incorrect output makes deterministic attacks (e.g. SAT), which rely on discovering the intended I/O relationship, challenging.

The Probabilistic SAT (PSAT) attack was developed to overcome the challenges of SAT-style attacks against obfuscated probabilistic IP [56]. To do so, the PSAT attack queries the black-box oracle circuit multiple times for a certain DI instead of just once. This is done to circumvent the inconsistency of the oracle's output. If the most frequently occurring output pattern is dominant (see [56] for the meaning of dominant), it is considered the correct output[1]. Otherwise, one of the output patterns is sampled with a probability equal to its frequency of occurrence and is chosen as the correct output. The PSAT attack produces better results than the conventional SAT attack, however, was still shown to fail at quite low error rates in [51]. The work in [51, 113] expands upon the PSAT attack by developing the StatSAT attack to overcome the following limitations of PSAT.

- The PSAT attack treats output patterns from the oracle as a whole and considers only a single correct pattern for any DI in the SAT-CNF formula. However, a probabilistic circuit (e.g. the black-box oracle) may not produce the correct output with the highest frequency. For example, if output patterns 0110, 0010, and 0001 are produced by the oracle for a given DI 11, 7 and 2 times, then 0110 will be considered to be the correct output by PSAT since it is dominant, even if the correct output is 0010.

[1]A correct output is what the oracle would output if it was not probabilistic.

- For a circuit with many primary outputs (PO), the probability that a fully correct output is produced may be exponentially small [50]. As a result, PSAT fails to return any key when the number of POs are large, or as the error levels in the circuit increase.

The StatSAT attack overcomes these limitations by introducing a “don’t care” condition into a conventional SAT attack [88]. This “don’t care” condition is used to leave primary output values with a high uncertainty and/or bit error rate (BER) unspecified in DIs present in Eqn. 3.2, which guides the SAT attack (see Sec. 3.3 for a description of the conventional SAT attack procedure). This makes it statistically unlikely for the StatSAT attack to latch a logically incorrect value, caused by probabilistic behavior, into Eqn. 3.2. Note that if an incorrect value were to be latched into Eqn. 3.2, this would cause the correct key to be eliminated, thereby causing the SAT attack to fail.

As a result of introducing a “don’t care” condition for high BER/uncertainty primary outputs in DIs, it is possible that an insufficient number of primary output bits in a DI are specified to enable the attack to progress (i.e. to eliminate further keys from the keyspace). In this case, the StatSAT attack would stall. To address this, the StatSAT attack leverages a forking mechanism whereby the SAT attack forks into two instances, one with each possible logical value (0/1) for an unspecified primary output. Doing so produces two independent SAT attack instances where there previously was one and ensures that at least one of them characterizes the intended logical behavior of the circuit. Ideally, the forked instance with the incorrect logical value specified will produce an unsatisfiable formulation, allowing this forked SAT instance to be quickly terminated and eliminated from consideration. If this does not occur, all keys returned by independent SAT attack instances are evaluated for closeness to the behavior of the correctly-keyed, black-box oracle circuit. The secret key producing behavior most closely mirroring the behavior of the oracle circuit is then chosen as the correct secret key. Through such a procedure, the StatSAT attack is shown to be capable of unlocking obfuscated probabilistic circuits, even with reasonably large error rates [51, 113]. This leaves obfuscated probabilistic IP vulnerable to theft, reverse engineering, and piracy. A comprehensive explanation of the StatSAT attack is in [51, 113].

3.7 High-Level Synthesis (HLS)

HLS is an automated design process that converts a behavioral description of a digital system into an RT-level design. HLS consists often utilizes 3 design optimizations: allocation, scheduling, and binding. Allocation determines the type and number of resources necessary to implement a design. After this design optimization, a list of functional units (FUs) (e.g. adders, memories, etc.) to implement a design is created. Scheduling partitions a design into control steps, called operations, which can be completed in one clock cycle. After this design optimization, a schedule, usually represented as a scheduled data flow graph (DFG), is created. In the DFG, nodes are operations that must be completed and edges are dependencies between operations. Binding maps (or *binds*) each operation to an FU allocated during the allocation phase. Common binding approaches target 1) minimizing required registers/multiplexers [30] and 2) minimizing switching activity [16, 86]. During this design optimization, knowledge of the ICs input space is assumed to be known. This allows the power ramifications of binding decisions to be evaluated [16, 86].

Chapter 4: Evaluating the Security of Obfuscated Circuits Beyond Gate-Level Boundaries

To begin this chapter, we diverge from the common assumption in logic obfuscation research that gate-level security guarantees are adequate to thwart IP theft [64, 102, 70, 94, 92, 98, 103, 75, 104, 74, 68, 66, 76, 33, 74, 34]. Instead, we argue that the attacker's goal is to sufficiently unlock a system so that it can be used to execute target applications successfully. While the prior view is prevalent, recent years have seen views which move beyond the gate-level becoming more common (see Section 2.3 for a literature review). Notice that this higher-level (i.e. architecture, application, system) view of obfuscation does not necessarily require that individual locked modules be fully unlocked and error free. This subtle change in attacker model requires entirely new evaluation methods and criteria capable of assessing the impacts of obfuscation in an IC as a whole, rather than at the gate-level. In this chapter, we aim to formalize these methods and provide a robust exploration of logic obfuscation beyond the gate-level. We later apply these methods and results to inform design techniques that provide strong hardware-oriented security guarantees in an IC as a whole.

We begin by presenting a theoretical derivation of the limits of logic obfuscation techniques. To protect against an untrusted foundry, logic obfuscation must 1) inject sufficient error to ensure critical application failures for any wrong key (error severity) and 2) resist any attack against it (attack resilient). Recently, an inverse relationship between the error severity and the SAT attack resilience of logic obfuscation has been identified [107, 109, 45, 103]. To explore this relationship, researchers have provided limited derivations of it for specific techniques [103] and loose generic bounds for locking as a whole [107, 109, 45]. Unfortunately, the specificity/weakness of these results provides little insight into the impact of such a relationship on logic obfuscation design goals. Therefore, we begin this chapter by rigorously deriving the exact relationship, rather than a loose bound, between the average error injection rate and the number of SAT attack iterations required to unlock logic obfuscation. As a result of this derivation, once a locking configuration's error rate is fixed, the corresponding SAT attack resilience can be directly quantified (and vice versa). Therefore, our derivation defines provable limits on conventional logic obfuscation.

The derived trade-off relates gate-level, rather than architectural, criteria. To understand how this trade-off affects security, we must assess its impact at the architecture level. To do so, we developed obfuscated GEM5 (ObfusGEM), a comprehensive logic obfuscation simulation framework based on the GEM5 simulator [8]. ObfusGEM is an open-source tool[1] to enable the design and evaluation of logic obfuscation techniques in processor ICs as a whole (i.e. at the architecture and

[1]ObfusGEM is available at: "*https://github.com/mzuzak/ObfusGEM*"

application level). We used ObfusGEM to quantify the architecture-level impact of our derived trade-off by simulating 9 benchmarks from the PARSEC [7] benchmark suite on a cycle-accurate GEM5 [8] model of 2 locked processor netlists. Our results show that the identified trade-off prevents logic obfuscation configurations with feasible area, delay, and power overheads from achieving error severity and attack resilience simultaneously. Because this trade-off exists regardless of logic obfuscation scheme, these results not only identify limitations of existing art, but also indicate that novel logic obfuscation techniques utilizing conventional constructions (that remain bounded by this same trade-off) will also experience these limitations.

To build on this result, we broaden our view and present a more holistic design space exploration of logic obfuscation in modern processor ICs. For this exploration, we obfuscated 14 common modules in a software model of an ARM and x86 processor core. We then assess the impact of logic obfuscation by performing Monte-Carlo simulations sweeping over the obfuscation design space. In total, we perform 50,400 Monte Carlo trials, each with a unique logic obfuscation configuration. By combining the results of our theoretical derivation and this experimental design space exploration, we find that logic obfuscation techniques applied with solely gate-level criteria, as is proposed by prior work [64, 102, 70, 94, 92, 98, 103, 75, 104, 74, 68, 66, 76, 33, 74, 34], are entirely inadequate to secure modern processors. This formalizes a central theme of this work: a challenging trade-off exists between the design goals for all combinational logic obfuscation techniques. In order to achieve a high amount of application corruptibility, the gate-level corruptibility must be very high. However, a high gate-level corruptibility inevitably makes the locked IC susceptible to SAT-type attacks.

4.1 Deriving the Parametric Space of Logic Locking

We begin by identifying and deriving a parametric space which exists underlying *every* logic obfuscation technique. Specifically, we show that an increase in the wrong key error rate of a fixed logic obfuscation construction, while improving error severity, must reduce the average number of SAT queries required to find an unlocking key. As a result, an IC secured with logic locking that simultaneously achieves the highest error severity and SAT resilience can be shown to be to have an infeasible design overhead.

While prior work has identified this trade-off for specific techniques [103] or as loose asymptotic bounds applying to more generic logic locking constructions [107, 109, 45], it has not yet been precisely quantified. Hence, our work constitutes the first derivation that directly quantifies, without reliance on asymptotic bounds, the wrong key error rate (error severity) and SAT attack resilience of any logic locking construction.

Let the input and key of an arbitrary locked module be of length n and $|k|$ bits respectively (2^n total inputs and $2^{|k|}$ total keys). There exists c correct keys for the locking construction, therefore, $2^{|k|} - c$ incorrect keys exist which corrupt the output corresponding to q inputs on average. Each of these inputs, on average, produces corrupt output for x wrong keys. This implies that the average wrong key error rate, ϵ, is $\epsilon = q/2^n$. With this notation, we define Lemma 4.1.

Lemma 4.1. The average number of wrong keys corrupting the output of each input minterm, x, is defined as $x = (2^{|k|} - c)\epsilon$.

Proof. Given the arbitrary logic locked module which we have defined, let us refer to (x_j, k_i) as a *minterm-wrong key pair (MWP)* if the input minterm x_j produces corrupt output for the wrong key k_i. Based on this, we can write the equation: $|MWP| = (2^{|k|} - c)q = 2^n x$, so, $x = (2^{|k|} - c)\epsilon$ $\square$

Theorem 4.2. The expected number of SAT attack queries required to unlock an arbitrary logic locked module, λ, is:

$$\lambda = \left\lceil \frac{log\left(\dfrac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{\epsilon(2^{|k|} - c)(2^{|k|} - c - 1)}\right)}{log\left(\dfrac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{2^{|k|} - c - 1}\right)} \right\rceil \tag{4.1}$$

Proof. A SAT-based adversary attacking an arbitrary technique will locate the unlocking key when all wrong keys within the key-space are eliminated. To accomplish this, we assume the SAT-based attacker randomly samples the input space for DIs[2]. The merits of this assumption are discussed in Section 5.1.3.1. Therefore, in each SAT query, the attacker selects a DI and removes all wrong keys that corrupt the

[2]An input is not a DI if it does not produce corrupt output for any wrong key. So, if locking never corrupts the output for some input, that input should be excluded from the input space for this derivation as it is not a valid DI

output for this DI that have not previously been eliminated. Let a_i be the expected total number of wrong keys eliminated up to iteration i. Hence, the expected number of wrong keys eliminated by SAT iteration i is $a_i - a_{i-1}$.

Lemma 4.1 indicates that each DI enables x wrong keys to be eliminated on average. However, some portion of these x keys could have been eliminated during prior SAT iterations and cannot be eliminated again. This must be addressed. By definition, a DI selection must eliminate *at least* 1 undiscovered wrong key to be valid. Therefore, a given SAT iteration eliminates 1 wrong key and a fraction of the $x - 1$ remaining wrong keys that have not been eliminated by prior DIs. Because DIs are randomly selected from the input-space, excluding the 1 wrong key we are guaranteed to eliminate, the likelihood for any wrong key not having been eliminated equals the ratio of # wrong keys that have not been eliminated, $(2^{|k|} - c - a_{a-1} - 1)$, to # total wrong keys, $(2^{|k|} - c - 1)$. Therefore, $a_i - a_{i-1}$ is defined by:

$$a_i - a_{i-1} = 1 + (x-1) \cdot \frac{2^{|k|} - c - a_{i-1} - 1}{2^{|k|} - c - 1} \tag{4.2}$$

We continue by simplifying the above form:

$$a_i = \beta \cdot a_{i-1} + x \quad \text{where} \quad \beta = 1 - \frac{x-1}{2^{|k|} - c - 1} \tag{4.3}$$

Let us create an intermediate variable, δ, defined as $x = \delta - \beta\delta$, which can be substituted into the above equation:

$$a_i - \delta = \beta \cdot (a_{i-1} - \delta) \quad \text{where} \quad \delta = \frac{x}{1-\beta} \tag{4.4}$$

Let us define the expected number of SAT queries necessary for a successful attack as λ. Using this, we can form the following set of equations that define the expected number of eliminated keys after each SAT query.

$$\begin{aligned}
a_1 - \delta &= \beta(a_0 - \delta) \\
a_2 - \delta &= \beta(a_1 - \delta) \\
&\vdots \\
a_\lambda - \delta &= \beta(a_{\lambda-1} - \delta)
\end{aligned} \tag{4.5}$$

Prior to launching a SAT attack, no wrong keys are eliminated. This provides an initial condition, $a_0 = 0$, enabling induction to be applied. By induction, we arrive at:

$$a_\lambda = \delta + \beta^\lambda \cdot (a_0 - \delta) = (1 - \beta^\lambda)\delta \tag{4.6}$$

For a successful SAT attack, all wrong keys must be eliminated, therefore, $a_\lambda = 2^{|k|} - c$.

$$2^{|k|} - c = \delta + \beta^\lambda \cdot (a_0 - \delta) = (1 - \beta^\lambda)\delta \tag{4.7}$$

We substitute for the intermediate terms, β and δ, and solve for λ. λ is a positive integer so we require a ceiling function.

$$\lambda = \left\lceil \frac{log\left(\dfrac{2^{|k|} - c - x}{x(2^{|k|} - c - 1)}\right)}{log\left(\dfrac{2^{|k|} - c - x}{2^{|k|} - c - 1}\right)} \right\rceil \tag{4.8}$$

Finally, we apply Lemma 4.1 to arrive at the final form:

$$\lambda = \left\lceil \frac{log\left(\dfrac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{\epsilon(2^{|k|} - c)(2^{|k|} - c - 1)}\right)}{log\left(\dfrac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{2^{|k|} - c - 1}\right)} \right\rceil \tag{4.9}$$

□

Let us briefly explore an approximate form of this result. Assume that the total number of keys ($2^{|k|}$) is much greater than the number of correct keys (c). If this were not the case, there is a sizable probability that a random key guess would produce a functional IC, making these configurations largely useless. This allows us to assume $2^{|k|} - c \approx 2^{|k|}$. Similarly, let us assume $2^{|k|} \gg 1$, so $2^{|k|} - 1 \approx 2^{|k|}$. This yields:

$$\lambda \approx 1 - \frac{log(\epsilon \cdot 2^{|k|})}{log(1 - \epsilon)} \tag{4.10}$$

However, as $\epsilon(2^{|k|} - c) \to 1$, the removed c and -1 terms become increasingly relevant, degrading Equation 4.10. Equation 4.10 should be avoided in this case.

4.1.1 Understanding the Derived Parametric Space

Prior to analyzing this result, we emphasize its fundamental nature. Because the key length and the number of correct keys are generally fixed locking constraints, Theorem 4.2 quantifies a direct relationship between 2 primary goals of locking: wrong key error rate (error severity) and SAT resilience. Therefore, in addition to proving that these 2 primary goals are in direct contention, this also enables an IC designer to directly quantify the provable security of their locking technique regardless of construction. By doing so, one can consciously trade between the error severity and SAT attack resilience of locking to ensure the design of a provably secure locking configuration.

To analyze the derived result, we visualize the parametric space created by Theorem 4.2 as a line in Figure 4.1. In the figure, we have fixed the key length ($|k|$=16) and number of correct keys (c=1). This was done because key length is generally determined by the allowable design overhead and number of correct keys is determined by the locking construction utilized. We note, however, that the shape of the plot is nearly identical regardless of the value selected for these parameters. Finally, note that the number of SAT attack iterations cannot be larger than the size of a locked module's input space. As described in Section 3.3, a SAT-based attacker selects specific input combinations as DIs which are used to eliminate all incorrect keys. Once all possible inputs have been selected, the SAT attack has provably eliminated all incorrect keys. Therefore, regardless of the key length, the number of SAT iterations will never exceed the size of the locked module's input space. These restrictions produce the parametric space in Figure 4.1.

Despite the opacity of the form of Theorem 4.2, the resulting parametric space is quite intuitive. The trade-off between error severity and SAT attack resilience can be characterized by a monomial inverse relationship. We note that point-function-based locking techniques (such as [98, 102]) reside at the far right of Figure 4.1, achieving the maximum possible SAT resilience and minimum possible wrong key error rate. This is unsurprising as these techniques were introduced to maximize SAT attack resilience. On the other hand, high error rate logic locking techniques (such as [5, 22, 64, 65, 70]) reside at the far left side of Figure 4.1, achieving a high error rate and an extremely low SAT attack resilience. Once again, this is unsurprising as these techniques were designed prior to the SAT attack [88] and therefore did not consider SAT attack resilience.

Finally, we have evaluated several locking configurations to experimentally support Theorem 4.2. To this end, we locked an 8-bit adder (n=16) using Anti-SAT [94, 92] and SARLock [98]. Each netlist was then attacked with the open-source SAT attack from [88]. The resulting number of SAT queries to unlock each netlist was compared to the expected number of SAT queries from Theorem 4.2. We have plotted this comparison in Figure 4.1. The empirical SAT queries closely match the expected SAT queries. Hence, Theorem 4.2 appears to correctly quantify the relationship between average error rate and SAT resilience.

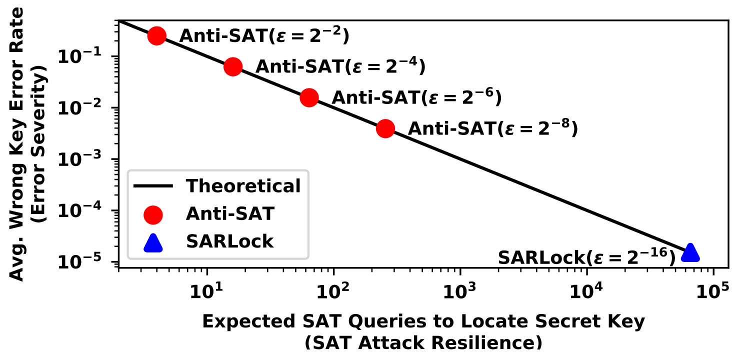


Figure 4.1: Inverse relationship between error severity and SAT resilience for logic locking ($|k|$ =16, c=1) from Theorem 4.2.

4.1.2 Understanding SAT Attack Iteration Runtime

Theorem 4.2 quantifies the number of SAT attack iterations necessary to unlock logic locking. Prior research, such as [94, 92, 98, 99, 103, 74, 75, 76, 45, 46], has relied upon this metric to demonstrate SAT resilience. However, recent works, such as Full-Lock [33, 34], have taken an alternative approach to SAT resilience, attempting to make the runtime of successive SAT iterations scale exponentially. To consider this, we expand our view to total SAT attack runtime, modeled by $T_{SAT} = \sum_{i=1}^{\lambda} T(i)$, where $T(i)$ is the runtime of SAT iteration i.

Each SAT attack iteration must solve an NP-complete problem. Hence, no efficient algorithm to solve each SAT query exists, only a variety of heuristics. Thus, the time to solve each SAT query ($T(i)$) is variable and specific to 1) the design topology, 2) the Boolean SAT solver algorithm, and 3) the specifications of the machine running the attack. Hence, the runtime of each SAT query is unpredictable and empirically dependent. Regardless, the derivation in Theorem 4.2 holds true, even for Full-Lock-style schemes.

An analysis of SAT iteration runtime is necessary for a full view of prior art. However, due to the empirical nature of this metric, we must rely on experimental analysis. So, we have implemented Full-Lock in the $\sim$ 10k gate b14 benchmark from ITC'99 [19]. The resulting SAT attack runtime for each configuration is in Figure 4.2. Notice that the SAT runtime did increase exponentially in the size of Full-Lock. However, we still unlocked each configuration within 10 minutes, despite

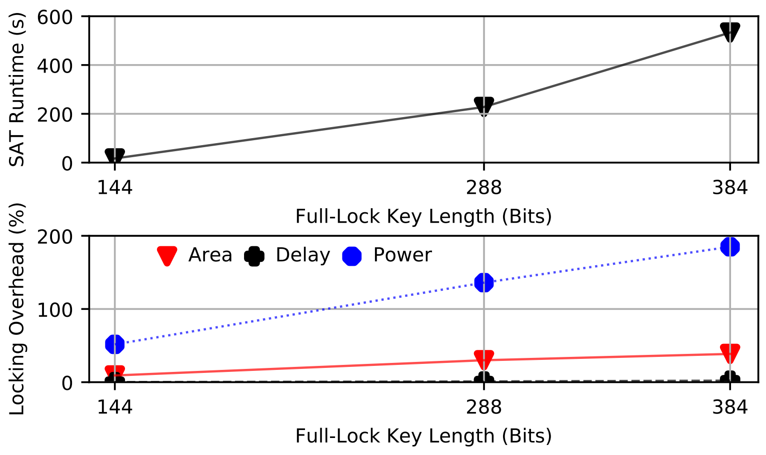


Figure 4.2: SAT attack runtime with corresponding area, delay, and power overhead for Full-Lock [33] in b14 netlist.

the presence of large keys, up to 384 bits. Despite exponentially increasing SAT runtime, Full-Lock did not exhibit significant SAT resilience for reasonably sized configurations.

We also calculated the overhead of each Full-Lock configuration using the Cadence Encounter RTL Compiler and the Synopsys 90nm SAED library. Full-Lock showed large overheads, with nearly a 200% increase in power and 60% increase in area for the largest tested scheme. This supports our assertion in Section 4.1 that design overhead restricts locking from simultaneously being error severe and SAT resilient.

4.2 ObfusGEM Simulation Framework

Later in this chapter, we empirically evaluate the trade-off derived in Sec. 4.1 to assess its ramifications with respect to the architecture-level attacker model defined in Sec. 3.2. However, such an evaluation requires some way to quantify the error severity of a locking construction, which varies among configurations and locked ICs. This currently does not exist. Therefore, prior to presenting the results of our design space exploration, we first introduce the ObfusGEM Simulation Framework[3] that we have developed for this work to enable the design space exploration of logic obfuscation beyond the gate-level, at the architecture/application level. ObfusGEM is an open-source tool-set which allows users to apply logic locking techniques to custom processor netlists, attack them with cutting edge attack methodologies, and then simulate custom workloads on the resulting ICs in a precise, cycle-accurate fashion. By observing any locking induced workload failures in these ICs, the application level security of logic locking can be quantified. Therefore, ObfusGEM enables the quantitative exploration of logic locking at the application level. As noted in Section 2.3, prior work has identified the importance of architecture/application level considerations for logic locking. ObfusGEM serves as the first systematic way to explore these considerations, regardless of locking technique. Throughout this work, we will utilize the ObfusGEM framework to enable us to both design and evaluate secure logic obfuscation techniques and architecture design methodologies.

[3]An open-source copy of the ObfusGEM Simulation Framework can be found at: "*https://github.com/mzuzak/ObfusGEM*".

4.2.1 ObfusGEM Supported Attacker Models

ObfusGEM is attacker model agnostic. This allows the user to evaluate any attacker model they consider realistic. Also, because ObfusGEM operates on real models of locked ICs, it can utilize any attack methodology or locking approach without modification.

4.2.2 Overview of the ObfusGEM Framework

To introduce ObfusGEM, we start with a brief overview. A block diagram of the process to quantify locking at the application level is in Figure 4.3. We discuss each step below.

1. A netlist is selected and logic locked. Any number or combination of modules can be locked within an IC.

2. Any attack (SAT/SMT [88, 4], structural [84, 96], removal [100], etc.) can be applied to the IC to locate a key. We note that a real netlist is used so any proposed attack can be applied without modification. This allows the effectiveness of specific attacks against logic locking to be quantified at the application level.

3. The attacker's key is applied to each locked module and a fault analysis locates any corrupted input minterms within any locked module. This essentially creates a truth table defining the functionality of each locked module.

4. The on-chip memory/processor architecture, intended IC workloads, locking configuration, and any corrupted minterms for each module are specified in configuration files.

5. The ObfusGEM simulator, described in Section 4.2.3, uses these configuration files to perform cycle-accurate simulations of a locked and an unlocked oracle processor running specified workloads. By tracking the divergence of these cores, the effects of locking are measured.

6. Steps 2-5 are repeated in a Monte Carlo fashion, randomizing parameters including the simulated application, applied locking key, or adversarial attack methodology.

By aggregating the results of many Monte Carlo simulations, the application failure rate and the mean time to failure can be calculated for a locked processor. Additionally, because ObfusGEM is based on the GEM5 simulator [8], performance and power analysis details can also be obtained for the locked IC. Given sufficient Monte Carlo trials, these data points quantify the usability and overhead of a locked processor after attacked by an untrusted foundry. Because the attacker's goal is to produce an IC sufficient for open market sale or piracy, the more usable an IC, the more successful the attacker. Therefore, the data produced by ObfusGEM directly quantifies the architectural effectiveness of logic locking. We use ObfusGEM for the remainder of this work, first to systematically explore the logic locking design space for both on-chip memory and data path locking and then to enable the tool-driven security-aware design approach that we introduce for on-chip memory and data path design in Section 6.2.

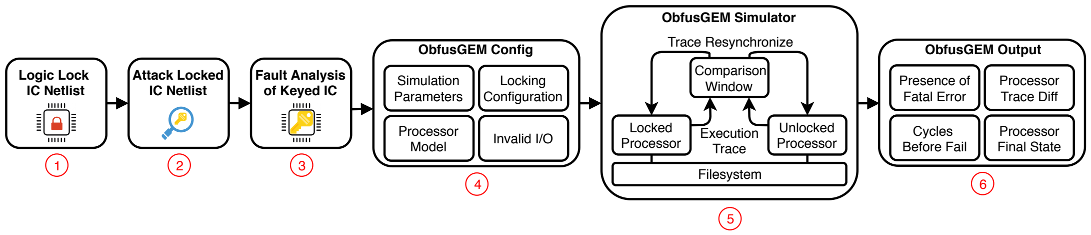


Figure 4.3: Block diagram of the ObfusGEM simulation framework.

4.2.3 Simulator Overview

Now, we discuss the simulator block of the ObfusGEM framework, displayed as step 5 of Figure 4.3. To construct this simulator, we relied upon stochastic fault injection research by the error resilience community [42, 52]. Building upon the outline laid out in [52], we implemented our own custom fault injection simulator with one critical difference: faults injected by logic locking are deterministic, not stochastic in nature. This means that locked primary inputs must always inject error when applied as input to a locked module. This mitigates the impact of error detection and many other error recovery procedures relied upon in the error resilience community.

Specifically, we perform application level simulations of a locked and an unlocked instance of an identical processor in GEM5 [8]. In the locked simulation instance, corrupted module output (located via a fault analysis of the locked netlist) are mapped to a deterministic error state. As errors are injected, their severity is classified by comparing the processor state of the locked and unlocked GEM5 simulation over a variable number of clock cycles. A divergence of the locked from the unlocked core which is not corrected or rendered latent in the variable clock cycle window is classified as an unrecoverable/critical application error. If the locked processor returns to a state exhibited by the unlocked processor at any point after the fault injection, we re-synchronize each processor trace (to reset the variable timing window and enable new faults to be analyzed) and consider the fault to be masked and therefore architecturally irrelevant.

4.2.4 Relationship to Prior Art

The concept of error resilience simulation is not new. It has been heavily studied by the research community, primarily focusing on either radiation-induced soft errors or manufacturing defect related errors. To facilitate soft error research, a series of error resilience simulators have been developed [54, 25]. While the ObfusGEM simulator relies heavily on the lessons learned by these tools, they are not interchangeable. This is due to the difference between the probabilistic, on-chip memory errors caused by cosmic rays which are modeled by soft error simulators and the deterministic, gate level errors caused by logic locking which are modeled by ObfusGEM. To effectively model soft errors, these simulators "fast-forward" through a random number of processor clock cycles, maintaining only the overall processor state. In the extremely unlikely probability of an error, the processor execution is halted, a small number of bit-flips are injected into on-chip memory, and a detailed simulation mode proceeds until the impact of the injected error is determined.

As described in Section 4.2.3, the functionality of ObfusGEM differs significantly. Because the errors injected by ObfusGEM are deterministic, rather than probabilistic, we are unable to use a “fast-forward” mode. Deterministic error injection requires that the current state of each module is maintained at all times and compared to any corrupted I/O during each clock cycle. Prior simulators lack the detailed tracking and comparison mechanisms necessary for deterministic error injection. Additionally, even if comparison logic was added, the detailed simulation mode would be necessary at all times. This would yield prohibitively slow performance.

Alternatively, there exists fault simulators focused on modeling errors related to manufacturing defects or degradation [9, 67]. While these simulators do explore deterministic error injections, they rely on quite detailed netlist modeling to do so. In general, fault simulators utilize full-scale Verilog simulations of the IC under test. These detailed models make the simulation of operating systems or workloads, as is necessary for logic locking research, prohibitively slow. ObfusGEM, while based on a netlist representation, simulates only a functional model of the IC using GEM5. This not only reduces execution time, but also enables the user to easily model and modify IC architectures. By leveraging the customizability of the GEM5 simulator in ObfusGEM, one can easily explore the effect of architecture design on supply chain security.

4.3 Assessing Security in Processors Beyond the Gate Level

In Section 4.1, a rigid parametric space was identified that placed 2 primary goals of locking, error severity and SAT resilience, into contention. Because effective locking must achieve both goals, this raises major security concerns. To continue our work, we use the ObfusGEM framework to empirically explore the consequences of our derived trade-off on the effectiveness of obfuscation. To do so, we use the tunability of SFLL-Fault [103, 75, 74] to lock processor ICs with locking that sweeps over our derived parametric space. By evaluating the security of each locking configuration, we explore the resulting design space. We find that the trade-off between error severity and SAT resilience renders logic locking with a feasible design overhead unable to thwart an untrusted foundry attacker.

To arrive at this assertion, we locked victim netlists with a variety of locking configurations. To select victim netlists, we aggregated the benchmarks used by several logic locking works and assessed commonality [103, 94, 92, 98]. In these works, processor logic constituted 74% of evaluated netlists. Of the processor logic tested, data and control path netlists were roughly equally represented. Therefore, we evaluated both control and data path locking. Specifically, we locked the control and data path of a RISC (MIPS) and CISC (80186) core to provide a cross-section of processor logic.

Locking Location: Within the control path, we locked the instruction decoder because it was the largest control path module in both processors. Within the data path, we locked the adder circuit. This was due to the prevalence of ALU netlists evaluated by prior art (57% of data path benchmarks).

Locking Configuration: Each netlist was locked using SFLL-Fault [103, 75, 74]. Using the tunability of SFLL-Fault, we incorporated a series of constructions within each benchmark that swept over the derived parametric space. Because this same design space exists underlying all logic locking techniques, this experiment allows us to characterize the design space of logic locking as a whole.

4.3.1 Logic Locking Attack Methodology

After locking each netlist, we evaluated their security with respect to both error severity and attack resilience (specifically SAT resilience). To evaluate error severity, we must quantify the usability of each IC in the case of a wrong key. To do so, we used the ObfusGEM simulator [115] to simulate 9 benchmarks from the PARSEC [7] benchmark suite on a cycle-accurate GEM5 [8] model of each locked netlist. We outline our experimental setup in greater detail in Section 5.1.5.3. A high failure rate for these benchmarks indicates that a locked IC is thoroughly unusable when a wrong key is applied and therefore exhibits strong error severity guarantees. To measure SAT resilience, we attacked each netlist using an open-source SAT attack [88]. By measuring the iterations and execution time required to recover the secret key, we quantify the SAT susceptibility of each locking configuration.

Control Path: We launched a SAT attack on the locked netlist with the lowest achievable wrong key error rate SFLL-Fault configuration. The evaluated MIPS controller had 16 primary inputs and the evaluated x86 controller had 15 primary inputs (after removing pass-through inputs). Therefore, the evaluated SFLL-Fault constructions utilized a 16-bit and a 15-bit key that stripped a single 16-bit and 15-bit input minterm. This locking configuration corresponds to the highest possible SAT attack resilience achievable by stripping a single minterm with SFLL-Fault. Despite being the largest control circuit, the decoder is small, allowing it to be unlocked via SAT attack.

	SAT Runtime	
Locked Circuit	**Control Path**	**Data Path**
MIPS (RISC)	108493 sec	893.2 sec
80186 (CISC)	95193.4 sec	993.4 sec

Table 4.1: SAT attack runtime for processor logic.

The runtime of each attack against the control path is in Table 4.1. Each attack successfully located the key within 48 hours. We note that even a worst-case, non-logic-locking-type approach, such as removing and replacing the netlist with a LUT, could only require a maximum of 2^{16} and 2^{15} SAT attack iterations to unlock the circuit based on Theorem 4.2. This corresponds to the case that each input must be selected as a DI. While not ideal, it is entirely possible to brute-force this number of SAT iterations given the small size of the controller logic. Therefore, because we selected the largest control path circuitry for locking and incorporated the most SAT resilient SFLL-Fault configuration of this form, it appears that SFLL-Fault is unable to protect the control path against a SAT attack.

Data Path: The adder circuit's input size enables extremely low error rate SFLL-Fault configurations. Because SAT resilience is inversely related to wrong key error rate, this implies that extremely strong SAT resilience can be achieved by locking. However, the goal of locking is two-fold. Alongside SAT resilience, locking must also achieve error severity. Because these 2 goals are in contention, we must locate an SFLL-Fault configuration sufficient to achieve both.

To do so, we used our simulation framework to identify the minimum wrong key error rate SFLL-Fault construction capable of achieving error severity. This minimum error rate locking construction corresponds to the maximum achievable SAT resilience for a locking configuration exhibiting error severity. Therefore, if this construction can be unlocked using a SAT attack, a locking configuration capable of simultaneously achieving SAT resilience and error severity does not exist.

We aggregated the results of these simulations for the locked 80186 netlist in Figure 4.4. To visualize the parametric space between the failure rate of common workloads (error severity) and SAT resilience, we have related the workload failure rate of each SFLL-Fault construction to the average number of SAT queries required to unlock it. This results in a map of the parametric space between error severity and SAT resilience. For both netlists, a non-zero workload error rate occurs at a wrong key error rate of 0.02%, corresponding to a 4096 (2^{12}) query SAT attack in Figure 4.4.

This indicates that a minimum error rate exceeding 0.02% is necessary to achieve any error severity. To accomplish this, we designed an SFLL-Fault configuration to strip a 13-bit input using a 13-bit key.

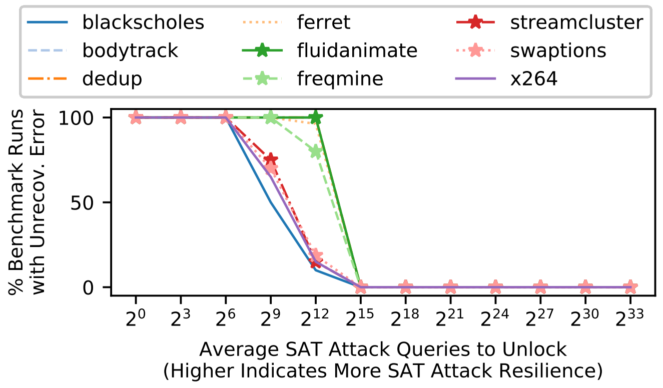


Figure 4.4: Empirically derived relationship between error severity and SAT attack resilience in locked 80186 processor data path.

We launched a SAT attack against each netlist configured with this minimal error rate SFLL-Fault construction which still achieved error severity. The runtime of each attack is in Table 4.1. Each attack found the secret key within 1 hour, indicating that a logic locking configuration of this form that is capable of providing both SAT resilience and error severity within the data path of either IC likely does not exist. Note that security could be achieved by greatly increasing the number of stripped inputs, however, doing so requires added restore logic for each stripped input. This makes such an approach infeasible in terms of area, delay, and power overhead. Prior work also notes the infeasibility of this approach [81]. Hence, our results indicate that a logic locking configuration capable of both error severity and SAT attack resilience with a feasible design overhead likely does not exist for this core.

4.3.2 Limitations Imposed by the Parametric Space of Locking

Evaluated SFLL-Fault configurations could not achieve both error severity and SAT resilience in either the control path or the data path. This was due to the trade-off between these 2 objectives. While each netlist was only locked with SFLL-Fault, we have proven that this trade-off between error severity and SAT resilience exists underlying *every* conventional logic locking technique[4] (Section 4.1). Additionally, we used the inherent tunability of SFLL-Fault to design locking configurations which swept throughout the parametric space. Therefore, our result is not a limited example in which cutting edge logic locking was insecure, but a demonstration of the underlying limits imposed on logic locking by its rigid parametric space. As a result, simply proposing novel logic locking constructions (which remain bounded by this trade-off) will do little to overcome these limits. Instead, we must explore ways to expand or bypass this parametric space, rather than operate within it.

[4]Logic locking could achieve security by sufficiently scaling key length. For example, portions of the circuit can be replaced with re-configurable logic (e.g. an FPGA), or SFLL-Fault can strip a sizable portion of an IC's inputs. These approaches are a theoretically viable way to achieve error severity/SAT resilience. This can be confirmed by Theorem 4.2. However, this does not weaken our assertion. These approaches are infeasible due to their tremendous design overhead. Prior work, such as [81], arrives at a similar conclusion.

4.4 Exploring the Design Space of Processor Design Obfuscation

To expand on the findings of the previous section, we relax our requirement for real hardware netlists and instead adopt software models to perform a more generalized design space exploration of logic obfuscation. Relaxing the requirement for real hardware allows us to use more modern testbed processors that lack publicly available netlists. Software models also allow us to quickly lock target processors in more diverse locations because we no longer must re-design the netlist to lock each module. This allows us to evaluate a large number of locking configurations and locations in more modern hardware. However, because we lack locked netlists to launch attacks against, we are restricted to theoretical calculations of attack susceptibility (which is inherently netlist dependent). To perform the more general design space exploration presented in this section, we model SFLL-Fault [75, 74, 103], as it is the most prevalent locking technique which remains unbroken[5]. As later discussed in Section 4.4.3, despite using SFLL-Fault for this exploration, our results are applicable not just to SFLL-Fault, but general to any combinational approach to logic obfuscation, regardless of construction. Therefore, this experiment serves as an exploration of combinational logic locking as a whole.

[5]Several works have shown that structural traces unique to SFLL can be exploited to recover the secret key [84, 96]. While these approaches have not successfully unlocked SFLL-Fault, they have unlocked earlier SFLL-based techniques, such as SFLL-HD, indicating some limitations of the SFLL family.

To launch this exploration, we incorporate a variety of locking configurations which sweep over the error injection rate of locking and quantify the corresponding effectiveness (error severity) of each locking construction. As noted in Section 2.1, SAT attack resilience (attack resilience) is inversely related to the error injection rate of locking. Therefore, we can calculate the corresponding SAT resilience of each locking configuration based on error injection rate using results from Section 4.1. We note that this relationship between error injection rate and SAT attack resilience exists underlying all combinational logic locking techniques, not just SFLL-Fault. Therefore, this experiment aims to identify locking configurations capable of inducing critical application failures (error severity) while maintaining SAT attack resilience within our processor testbeds. In other words, this experiment identifies secure locking constructions capable of securing each IC as a whole, not just at the gate level.

4.4.1 Experimental Methodology

Testbed Processors: An x86 (out-of-order, embedded core) and an ARM A53 processor were used. Note that due to the proprietary nature of both of these cores, we were forced to use software, rather than hardware, models. These models were functional representations developed within the GEM5 simulator. However, all methods and data presented could be performed equivalently on a hardware model of the core.

Locking Configuration: SFLL-Fault[75] based locking configurations sweeping over error injection rate were applied to each processor. To do so, SFLL-Fault was configured to lock a single, randomly selected input cube of varying length. By scaling the length of the locked input cube, a varying number of minterms could be corrupted, thereby scaling the error injection rate. Each of the 14 modules (see Figure 4.5) were locked and evaluated independently (i.e. one at a time). Note that the candidate locked modules were selected from 3 categories: 1) on-chip memory, 2) data path, and 3) control path. As we have noted before, on-chip memory [114, 13] and data path locking [45, 46, 73, 58] have been the most commonly suggested locking candidates in prior literature. Therefore, we focused our attention on modules within the on-chip memory and data path.

Feasibility of Locking Configuration: We note that this locking configuration is consistent with state-of-the-art logic locking research. We make several observations regarding this:

1. Recent work on architectural locking considers locking only a single module which contains the critical IP to be protected [73, 103, 75, 76, 13]. This is consistent with our decision to independently lock each module.

2. Cutting edge locking, such as [103, 13, 94, 92, 98, 73, 111, 76, 75], propose constructions which distribute error throughout a locked module's input space. This is consistent with our random cube selection approach. By doing so, we both maximize the SAT attack resilience of the locking construction and

prevent cryptographic information leakage in the case that an adversary has knowledge of a module's input space (e.g. knowledge of "*protected input cubes*" for SFLL allows an adversary to recover the secret key in linear time [103]).

3. Cutting edge techniques, such as [103, 74, 73, 94, 92, 98], propose integrating their low error locking constructions alongside a high error locking technique, such as [102]. Taking this so-called "*compound*" approach allows a designer to achieve both high error severity and SAT attack resilience simultaneously. However, recent research has shown that high error locking techniques in compound locking constructions can be easily removed, leaving only low error locking within the module [77, 83, 81, 80]. This is consistent with our choice not to pair SFLL-Fault with a high error locking technique, as it could be easily removed.

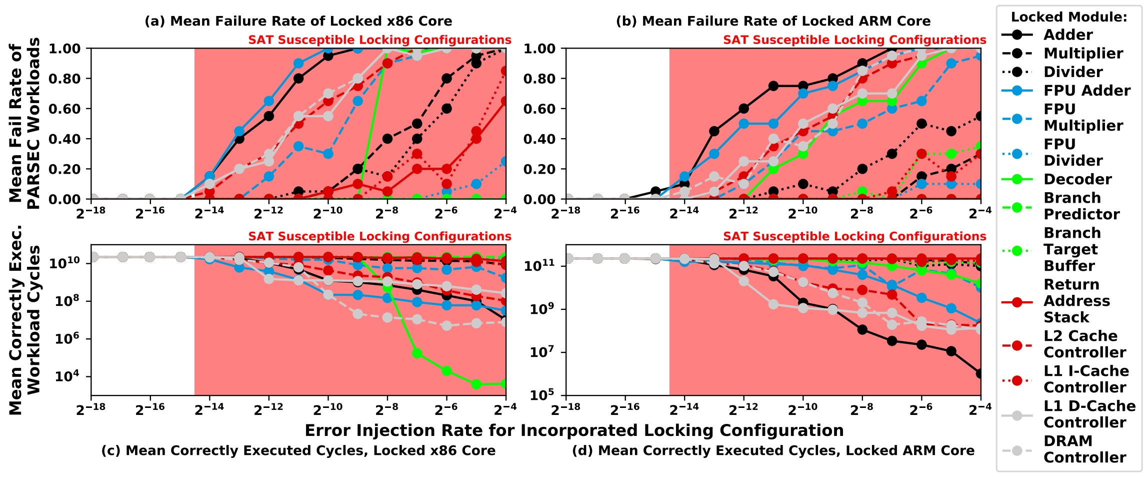


Figure 4.5: ObfusGEM results quantifying the application-level security of locking in an x86 and ARM A53 core.

ObfusGEM Configuration: Using the ObfusGEM simulator, 120 Monte Carlo simulations were performed for each locking configuration in each processor. A locking configuration consists of a locking location and an associated error injection rate (e.g. the x86 core adder locked by an SFLL-Fault configuration with an error rate of 0.01%). For each Monte Carlo iteration, a wrong key was randomly selected and 1 of 3 benchmarks from the PARSEC benchmark suite[6] [7] were simulated on the core. Because the error rate of SFLL-Fault is uniformly distributed across all wrong keys (i.e. each wrong key has the same error rate), each random key selection produces a locking configuration with an identical error rate and different locking-corrupted minterms. Note that this is the weakest possible attacker. The attacker randomly selects a key with no attempt to minimize the error injection rate or intuit the correct key. In total, this makes 50,400 Monte Carlo trials[7].

Monte Carlo Simulation Count: We selected the number of Monte Carlo trials performed for each locking configuration empirically. To do so, we performed 1,500 Monte Carlo trials for 4 error rates (2^{-10}, 2^{-9}, 2^{-8}, 2^{-7}) applied to 4 locking locations (FPU Adder, Multiplier, L1 D-Cache Controller, Decoder) respectively. We considered the application failure rate after 1,500 iterations to be the *true* application failure rate of each locking construction. Based on this, we located

[6]PARSEC benchmarks [7] are designed to be a cross-section of common processor workloads, therefore, they serve as a good measurement of a locked processor's ability to do useful work at the application level.

[7]50,400 total trials = 2 ICs * 14 locking locations * 15 SFLL-Fault error rates * 120 Monte Carlo trials

the number of Monte Carlo trials in which each locking configuration converged to an error rate within $\pm 5\%$ of the *true* application failure rate. This was 120 Monte Carlo trials per configuration.

4.4.2 Quantifying SAT Attack Resilience

The work in Section 4.1 defines an inverse mathematical relationship between the error injection rate of logic locking and the average number of SAT iterations necessary to unlock it. For a successful attack on logic obfuscation, a SAT attack must locate the correct key in a reasonable time. Because SAT attack runtime is both netlist and attack formulation dependent, the expected number of SAT attack iterations alone does not provide sufficient context to accurately characterize SAT susceptibility. Fortunately, the results presented in the work on SFLL-Fault [103, 75] provide a strong intuition for SAT runtime.

Yasin et al. provided an empirical analysis of SAT attack effectiveness against varying error rate SFLL constructions in a series of benchmark circuits. For their experiment, they incrementally lowered the error injection rate of SFLL and evaluated the corresponding SAT attack runtime. The lowest error injection rate SFLL configuration which could be successfully SAT attacked within 48 hours required 2^{14} SAT iterations. Therefore, we define any locking configuration unlocked in $\leq 2^{14}$ iterations as SAT susceptible.

4.4.3 Analysis of Design Space Exploration

We have aggregated the results of the experiment described in Section 4.4.1 in Figure 4.5. Within the figure, the region constituting SAT susceptible locking configurations have been shaded in red for clarity. Fundamentally, the goal of our design space exploration was to locate an architecturally secure locking configuration, defined as a locking configuration which simultaneously 1) induces critical application failures for any wrong key (error severity) and 2) maintains SAT attack resilience. In the figure, the first goal of logic locking (error severity) is quantified by the PARSEC benchmark failure rate. If a locking configuration with a given error rate induces a high failure rate for PARSEC benchmarks, the configuration has successfully derailed application functionality. The second goal of logic locking (maintaining SAT resilience) is quantified by the red-shaded SAT susceptible region. If a given error rate locking configuration resides outside of this region, it is deemed SAT resilient. Therefore, based on our results, there does not exist an SFLL-Fault configuration capable of simultaneously achieving both goals.

While each netlist was only locked with SFLL-Fault, this same trade-off between error severity and SAT resilience exists underlying *every* locking technique [111, 107, 109, 45]. Therefore, because all logic locking techniques restrict unauthorized use with the same fundamental functionality, namely deterministically corrupting the output corresponding to some portion of the input space, these results can be generalized to logic locking as a whole. Any alternative logic locking technique configured with a given, randomly distributed error injection rate can be expected to achieve a similar error severity to that of SFLL-Fault. Additionally, given the generalized nature of the results derived in [111, 107, 109, 45], this error severity can be expected to correspond to a similar number of SAT attack iterations necessary to unlock the locking construction, hence a similar SAT attack resilience. Therefore, the results of this design space exploration are not a limited example in which only SFLL-Fault was insecure when viewed beyond the application level, but a demonstration of the underlying limitations of logic locking when viewed at the application level.

This result is quite alarming. **Fundamentally, it indicates that state-of-the-art locking applied with only gate-level considerations (as proposed by most cutting edge art [64, 102, 70, 94, 92, 98, 103, 75, 68, 66, 76, 33]) is inadequate to thwart an untrusted foundry attacker, regardless of locking location or configuration.** To further exacerbate this result, we note that the weakest possible attacker model was utilized. The untrusted foundry simply selected a random wrong key with no attempt to attack it or to minimize error within the locked IC. Even given this extremely weak attacker model, state-of-the-art logic locking was unable to achieve application level security within either IC, constituting a massive security risk.

4.5 Conclusions

Based on the results in this section, we argue that the trade-off between error severity and SAT attack resistance is a real one. Additionally, in the specific trade-space characterized by Figure 4.5, there does not exist a configuration which guarantees the IC designer both SAT resistance and application-level security using current state-of-the-art gate-level locking techniques. These results highlight that we must explore methodologies beyond conventional gate-level logic obfuscation techniques in order to achieve strong hardware-oriented security in practice.

Chapter 5: Obfuscation Techniques for Security Beyond Gate-Level Boundaries

In the previous chapter, we demonstrated that logic obfuscation, as currently defined, is limited in achieving both error severity and attack resilience simultaneously. This is due to the rigid parametric space between the average wrong key error rate of a locking construction and the number of SAT attack iterations necessary to unlock it. As a result, once a locking configuration's error rate is fixed, the corresponding SAT attack resilience can be directly quantified (and vice versa). From our design space exploration in the previous chapter, we found that in order to achieve gate-level error rates sufficient for the denial of application-level functionality, one must design a locked circuit which is inherently SAT susceptible. In this chapter, we propose 3 non-conventional approaches that tweak the standard operation of logic obfuscation to favorably alter the defined trade-off. As a result, we propose 3 novel obfuscation constructions capable of achieving both error severity and attack resilience simultaneously in 3 distinct applications.

First, we propose Trace Logic Locking (TLL), a novel enhancement of gate-level logic locking which enables existing art to secure arbitrary length sequences of input minterms, referred to as *traces*. Doing so injects an additional degree of freedom into the parametric space of locking, enabling locking techniques to overcome the limitations of our derived trade-off. We both theoretically and empirically prove this by using TLL to enhance cutting edge locking. In 10 large benchmarks, we show that TLL-enhanced logic locking provides exponentially stronger attack resilience than conventional locking with only modest additional overhead. Finally, we demonstrate the efficacy of TLL in a processor IC using ObfusGEM simulations. Despite prior art being unable to secure the evaluated processor ICs, we find that TLL concurrently achieves strong error severity and attack resilience.

Second, we propose Memory Locking, an automated logic obfuscation technique capable of denying application-level functionality to the adversary while maintaining SAT resistance. To do so, Memory Locking targets on-chip SRAM circuitry due to the 50-90% of transistor count dominated by SRAM-related circuitry in modern processors [59]. This creates significant flexibility in obfuscatable location and functionality. Additionally, the analog effects governing SRAM make it resistant to many proposed attack methodologies such as SAT-based attacks. We then demonstrate the effectiveness of Memory Locking compared to prior art with application-level simulations using ObufsGEM.

Third, we propose High Error Rate Keys (HERK) to thwart both StatSAT and other prominent attacks on probabilistic circuits. HERKs leverage high error wires, caused by probabilistic behavior, to hide the correct key under stochastic noise. HERKs can be integrated into prior deterministic logic obfuscation schemes for strong IP protection in probabilistic circuits. We demonstrate the efficacy of HERKs in several benchmark circuits, empirically verifying their resilience to StatSAT and other SAT-style attacks. As a result, using HERKs allows high-error obfuscation to be used without sacrificing SAT attack resilience, thereby enabling obfuscation configurations that are simultaneously high in error severity and attack resilience to be configured. This allows strong security to be achieved beyond the gate-level in probabilistic applications in particular.

5.1 Trace Logic Locking (TLL)

To counter the rigidity of logic locking's parametric space, we developed Trace Logic Locking (TLL), a novel logic locking enhancement which injects an additional degree of freedom into the parametric space of locking. TLL achieves this by locking a sequence, or *trace*, of inputs. This differs from conventional locking which locks a set of inputs. Because both a set and a trace of inputs can be locked simultaneously and independently, TLL can be integrated into any conventional logic locking technique. As we show both theoretically and experimentally in Section 5.1.3.1/5.1.5, doing so causes the SAT attack resilience of a TLL-enhanced technique to vary exponentially in locked trace length. This is a major contribution. The derived parametric space of logic locking requires a reduction in the average wrong key error rate for an improvement in SAT attack resilience. However, by utilizing TLL, SAT attack resilience can be achieved by scaling locked trace length, thereby disentangling the average wrong key error rate and attack resilience of a locking configuration.

5.1.1 Foundations of TLL

Prior to detailing a construction of TLL, let us formalize its notation and intended functionality. Let us assume that conventional locking (e.g. SFLL-Fault) has been applied to some arbitrary combinational module in an IC which receives an input ($x \in X$) on each clock cycle. Additionally, let us assume that some incorrect key (k_i) has been provided to the incorporated locking. In this case, logic locking will corrupt the output of some subset of the input space, $X_i \subseteq X$, such that a fixed incorrect output is produced whenever an input, $x \in X_i$, is applied. The inputs in X_i depend only on k_i.

In this work, we refer to a set of inputs occurring over l clock cycles as a *trace* of length l. TLL is designed to modify a conventional locking construction (e.g. SFLL-Fault) to lock a trace of length l. We refer to this as l-state TLL. To do so, TLL-enhanced locking must corrupt the output of a different set of inputs, $X_i \subseteq X$, on l different clock cycles. This is achieved by injecting the notion of state into conventional locking. Hence, l-state TLL incorporates an l state finite state machine (FSM) within the locking construction where each state corresponds to a unique X_i. Therefore, the FSM's state determines the currently locked inputs. We refer to the set of inputs locked in FSM state m as X_i^m. Hence, when using TLL, X_i^m depends on both k_i and the FSM's state.

5.1.1.1 TLL as a Logic Locking Enhancement

As noted, conventional locking secures a set of inputs (X_i) which are dependent only on the value of k_i. This functionality is combinational, entirely lacking a sequential component. TLL, on the other hand, is entirely sequential in nature as it secures input traces. Therefore, because conventional locking lacks a sequential component, TLL can be integrated alongside any conventional locking technique. Doing so introduces a sequential component to locking without altering the underlying combinational functionality of the conventional locking technique. We refer to this as *enhancing* a locking technique with TLL. By doing so, a locking construction is created where X_i^m is dependent on both the value of k_i, determined by the conventional locking technique, and the current state of the locking construction (m), determined by TLL.

Presenting TLL as a logic locking enhancement, rather than a unique locking construction, is quite advantageous. It allows TLL to leverage the strongest existing conventional techniques, while still expanding the parametric space of locking. In fact, because TLL only adds a sequential component to a conventional locking construction, it does not modify the underlying combinational functionality of conventional locking at all. This means that TLL expands the parametric space of locking, thereby exponentially improving SAT resilience, while maintaining any other security guarantees (e.g. removal resistance) of the underlying locking technique. However, because TLL is a locking enhancement, its construction depends on the technique it enhances. Moving forward, we utilize SFLL-Fault [103, 75, 74] to formalize a construction of TLL.

SFLL-Fault was chosen as it is currently the most prevalent logic locking technique which remains unbroken. However, there are limitations to the SFLL family. For example, structural traces unique to SFLL have been shown to be exploitable to reconstruct the secret key [84, 96]. While these traces have not yet been used to successfully unlock SFLL-Fault, they have been used to unlock SFLL-HD, indicating some limitations. While other techniques, such as [114, 68, 66, 76, 33], have been shown to be strong alternative locking constructions, they have not yet gained the prevalence of SFLL. Therefore, we present a locking construction for TLL based on SFLL-Fault. However, any alternative conventional locking technique could have been utilized as an equally valid backbone for TLL.

5.1.1.2 Comparison of TLL and FSM-Based Locking

Despite TLL's sequential nature, it differs substantially from FSM-based locking schemes, such as [15, 2, 21, 20, 26]. To distinguish TLL and sequential locking, we note 2 key differences:

1. **Target Circuitry:** FSM-based locking obfuscates an IC by altering its control FSM. To do so, a series of key authentication states are added to the control FSM to validate the key. For a wrong key, the controller enters a permanent obfuscation mode utilizing dummy states with incorrect functionality. For a

correct key, the controller enters the intended FSM region, enabling correct functionality. Hence, FSM-based locking schemes obfuscate IC control flow. Rather than modifying the control path, TLL instantiates a separate FSM that directly modifies a combinational logic locking scheme. When the key is incorrect, rather than inducing errant control flow, TLL induces combinational errors in the logic locked module.

2. **Intended Use:** FSM-based locking achieves stand-alone security. While these schemes can be paired with other locking art, they operate independently. TLL must be closely integrated with a logic locking scheme because it cannot induce error on its own. TLL relies on the underlying locking scheme for error, making TLL an enhancement for logic locking, rather than a stand-alone scheme.

This leads to 2 key advantages over FSM-based schemes:

1. **Exponential Improvement in SAT Resilience:** While FSM-based locking can be integrated alongside logic locking, it operates separately, allowing it to be attacked separately. For example, automated reverse engineering attacks [26] can isolate the control FSM to infer the key of FSM-based locking. Similarly, a logic locked module can be isolated and SAT attacked separately from the locked control FSM. TLL integrates tightly into logic locking. As derived in Thm. 5.2, this requires both schemes be attacked together, exponentially improving SAT resilience.

2. **Reverse Engineering Attack Resilience:** FSM-based locking is susceptible to reverse engineering. In [26], the authors outline these attacks on prominent FSM-based schemes [15, 2, 21, 20]. These attacks are potent due to the ease of identifying and reverse engineering FSMs, enabling the authors to infer errant control flow, thus the secret key. However, TLL 1) does not target an IC's control FSM and 2) does not rely on errant control flow for security. So, unlike FSM-based locking, TLL's FSM topology does not encode the key, thus FSM reverse engineering is irrelevant.

Therefore, we consider TLL-enhanced and FSM-based locking to be fundamentally different hardware security schemes.

5.1.2 Enhancing SFLL-Fault With TLL

In this section, we formalize a construction of TLL to enhance SFLL-Fault ($\text{TLL}_{SFLL-Fault}$) [103, 75, 74]. For the remainder of this work, we rely on this construction to evaluate TLL. We begin by introducing a limited example of 2-state TLL which we later generalize to a fully tunable construction.

5.1.2.1 Enhancing SFLL-Fault With 2-State TLL

Assume an arbitrary combinational module receives input, $i \in I$, on each clock cycle. We refer to a sequence of l inputs applied over l clock cycles as a *trace* of length l. Therefore, 2-state TLL locks a trace of length 2. We emphasize that the intended functionality of a TLL locked module must remain combinational despite the sequential nature of TLL's locking.

For the 2-state TLL construction which we introduce in this section, let us assume that we intend to corrupt the output of a single cube within the locked module for each of the 2 TLL states. To illustrate this, let us arbitrarily refer to the locked trace inputs as i_0 followed by i_1. In this case, i_0 *or* i_1 produces incorrect output in the locked module for a given clock cycle, never both. The currently locked input switches between i_0 and i_1 whenever the input to the module matches a portion of the secret key. This yields functionality such that the order of locked inputs in the trace is critical, but the number of cycles between locked inputs is irrelevant. Figure 5.1A shows a block diagram of 2-state TLL.

From the block diagram, notice that 2-state TLL consists of a stripped functionality (SF) module, a restore unit, and an XOR gate. The functionality of these components relies on a secret key, denoted as the concatenation of 2 independent *subkeys*, $k = (k_1, k_0)$. The correct secret key corresponds to the locked inputs of the trace, $k = (i_1, i_0)$. For the remainder of the section, we describe the functionality of TLL in depth by considering each of its 3 components.

SF module: SF is defined as the re-design of a given set of inputs within a combinational module to produce incorrect output. In 2-state TLL, the SF module contains 2 SF inputs corresponding to the locked trace, namely the inputs i_0 and i_1. However, we note that 2-state TLL only has 1 SF input enabled within the design during each clock cycle. To achieve this, both SF inputs in the module are mapped to a common intermediate value, X, rather than two separate and unrelated incorrect outputs as is done by [103]. This intermediate value is then mapped to the correct output for *either* i_0 or i_1. Finally, because both inputs are mapped to the same intermediate, X, it is impossible for both inputs to have correct output during a given clock cycle. To select between mapping X to the correct output for i_0 or i_1, additional logic called the "restore multiplexer" (RM) is added. Specifically, the select line of the RM determines whether i_0 or i_1 is mapped to correct output. This select line is controlled by the restore unit state.

Restore Unit: The restore unit is located below the SF module in Figure 5.1A and consists of a 2-state finite state machine (FSM). As noted, the current state of this FSM controls the select line of the RM, hence, the restore unit determines the current SF inputs within the locked module. State transitions occur within this FSM when the current input to the locked module matches one of the two secret subkeys (k_0 or k_1), with k_0 used in state 0 and k_1 used in state 1. Hence, the state sequencing of this FSM is determined by the secret subkey corresponding to TLL's restore unit state. Building upon the foundation built by SFLL-Fault [75], this secret subkey should be stored within a tamper-proof look-up table (TPLUT) for protection. This TPLUT, as defined in [103], contains a secret subkey (k_0 or k_1),

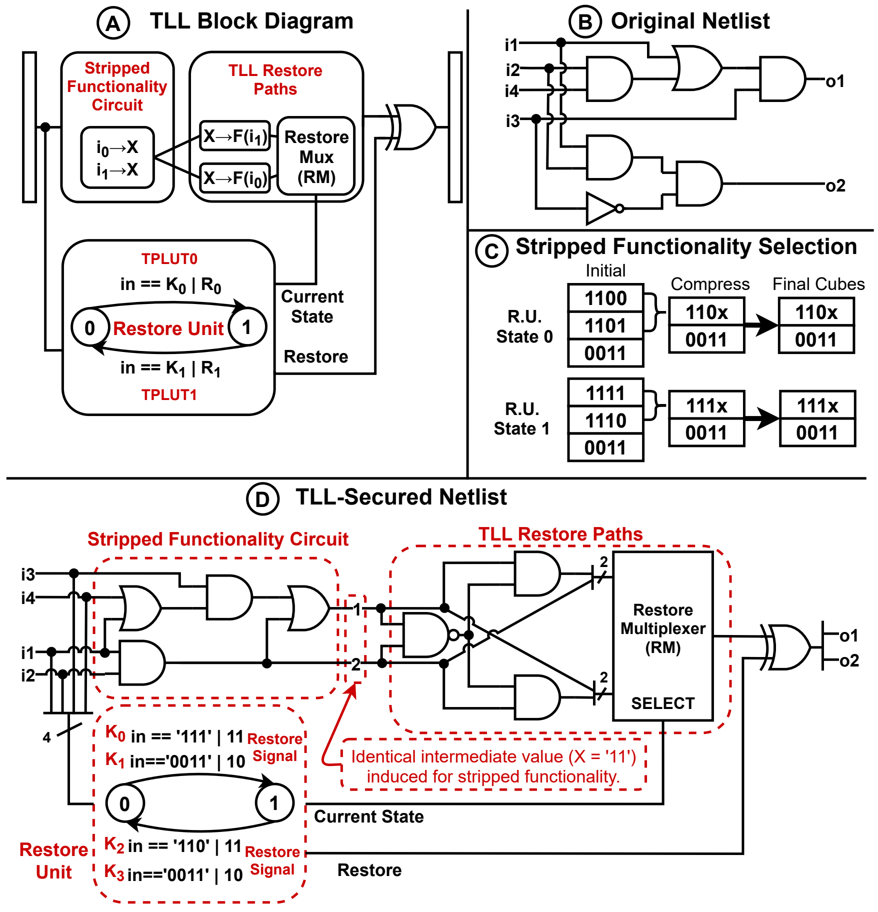


Figure 5.1: 2-state TLL-secured module. A) Block diagram of 2-state TLL for input sequence "i_1, i_0". B) Original c17 netlist. C) Stripped functionality selection and compression for TLL. D) C17 netlist secured with 2-state TLL.

acting as the index, and a restore signal, acting as the output. Additionally, when a state transition occurs (i.e. when the currently active secret subkey matches the input to the locked module), a restore signal (R_s) is applied to TLL's XOR gate, altering the module's output.

When properly keyed, the TLL restore unit will correct output corruption induced by SF inputs by applying the restore signal to the XOR gate located at the output of the SF module. In the case of a wrong key, the restore unit will inject error by applying a restore signal which corrupts otherwise correct outputs corresponding to non-SF inputs. In this case, output corruption is present in the locked IC not only for SF inputs, but also for these non-SF inputs.

XOR Gate: The final component of 2-state TLL is the XOR gate on the output of the locked module. As noted previously, when a restore unit state transition occurs, a restore signal is applied to this XOR gate which modifies module output.

Note that a TLL-secured module remains combinational. Only the restore unit is sequential. Additionally, only 1 input in the trace has SF on a given clock cycle. The SF input switches when the currently enabled secret subkey matches the input to the module, thus transitioning the restore unit FSM to the next state. This allows 2 separate locked inputs to exist without any increase in the wrong key error rate of the locked module for a given clock cycle. As we show in Section 5.1.3, by utilizing TLL to enhance SFLL-Fault, locking constructions can be created which exhibit equivalent error rates and exponentially stronger SAT resilience than SFLL-Fault alone. This allows TLL to expand the parametric space of SFLL-Fault. We summarize our construction of TLL-enhanced SFLL-Fault below.

1. TLL protects a sequence of 2 inputs, i_0 followed by i_1
2. At any given time, only a single SF input is locked
3. The RM select line dictates which SF input is locked
4. The restore unit corrects SF errors when the secret subkey matches the current SF input

5.1.2.2 Example TLL$_{SFLL-Fault}$ Implementation

To clarify TLL's implementation, we have locked the c17 circuit from ISCAS'89 [10] in Figure 5.1B-D. The un-locked c17 circuit is shown in Figure 5.1B. To implement 2-state TLL, the designer first selects candidate SF inputs for each TLL restore unit state. In Figure 5.1C, we have selected 3 candidate SF minterms for each state: (1100, 1101, 0011) and (1111, 1110, 0011) for restore unit state 0 and 1, respectively. Notice that the same inputs can be selected as SF inputs in multiple states (e.g. 0011). Now, we can optionally compress these minterms into smaller cubes to reduce design overhead. In Figure 5.1C, the minterms (1100, 1101) and (1111, 1110) are compressed into single cubes during this compression step.

Given our list of SF cubes, we can implement TLL in the c17 netlist with the following process. First, each SF input must be mapped to some identical intermediate value ('X'). In this case, 'X' is '11'. With 'X' defined, we can perform standard combinational optimization to synthesize the SF circuit. The resulting SF circuit is labeled "Stripped Functionality Circuit" in Figure 5.1D. Now, 2 restore paths, one for each restore unit state, must be added to the output of the SF circuit to correct the SF inputs. Hence, in restore unit state 0, the intermediate value, '11', must be mapped to the intended output '01' and in restore unit state 1, '11' must be mapped to '10'. These paths are denoted "TLL Restore Paths" in Figure 5.1D. At their output, a 2-input multiplexer, called the "Restore Multiplexer", connects these 2 paths to an XOR gate added at the output of the module.

Finally, we add TLL's restore unit. To do so, an FSM with a single state for each trace index must be included (i.e. 2 states). In Figure 5.1D, we have labeled TLL's restore unit as "Restore Unit". The state of the restore unit dictates the currently active restore path by driving the select line of the RM. Notice that the secret subkeys applied to the restore unit dictate its functionality. Whenever the active subkey matches the primary input, the restore unit FSM changes its state and applies a restore signal to the XOR gate at the module's output. For a correct key, this signal corrects errant outputs due to SF. For a wrong key, this signal corrupts otherwise correct outputs.

5.1.2.3 Example 2-State TLL Functionality

Assume that the 2-state TLL configuration in Figure 5.1A is used to lock a 2-bit input module. Therefore, the possible input space can be represented by i_3, i_2, i_1, i_0. Within this module, the trace i_0 followed by i_1 is locked. This configuration would yield correct functionality with the key $k = i_1, i_0$ and incorrect functionality otherwise. An error map exhaustively describing this 2-state TLL-secured circuit is in Table 5.1.

The top row of this table enumerates the possible TLL secret keys as a combination of 2 subkeys. Below this row, each key combination is enumerated in the form: {subkey for restore unit state 1, subkey for restore unit state 2}. On the side of the table, each possible restore unit state and primary input combination is enumerated. As an example, let us assume we want to know TLL's output in the following situation: the restore unit is in state 1, the applied key is i_2, i_0, and the primary input is i_2. To do so, we find the intersection of the row for the current restore unit state/primary input value and the column corresponding to the current secret key. This intersection is a green-shaded cell in Table 5.1. The ✗ in this cell indicates that TLL would produce an errant output in this scenario. Alternatively, if the primary input i_3 were applied, instead of i_2, the corresponding a ✓ symbol indicates that TLL provides correct output for this scenario.

5.1.2.4 A Generalized Construction of TLL$_{SFLL-Fault}$

We initially presented a simplified construction of TLL-enhanced SFLL-Fault as a special case. A more generalized form offers the IC designer scalability in both wrong key error rate and locked trace length. Expanding to the generalized TLL$_{SFLL-Fault}$ construction relies on the same principles of functionality stripping and a sequential restore unit. We discuss the modifications necessary to tune each of these parameters separately, but present a unified TLL$_{SFLL-Fault}$ construction that enables scaling in both trace length and error rate. A block diagram of SFLL-Fault enhanced with TLL is in Figure 5.2.

Scaling TLL$_{\mathbf{SFLL-Fault}}$ Error Rate

The IC designer can scale the error rate of this TLL construction by incorporating additional SF inputs within the locked module for a restore unit state. The functionality stripping of additional inputs leads to a higher error rate which

R.U.		**key input $\mathbf{k = k_1, k_0}$**							
State	**in**	i_0, i_0	i_0, i_1	i_0, i_2	i_0, i_3	i_1, i_0	i_1, i_1	i_1, i_2	i_1, i_3
0	i_0	✓	✗	✗	✗	✓	✗	✗	✗
0	i_1	✓	✗	✓	✓	✓	✗	✓	✓
0	i_2	✓	✓	✗	✓	✓	✓	✗	✓
0	i_3	✓	✓	✓	✗	✓	✓	✓	✗
1	i_0	✗	✗	✗	✗	✓	✓	✓	✓
1	i_1	✗	✗	✗	✗	✓	✓	✓	✓
1	i_2	✓	✓	✓	✓	✓	✓	✓	✓
1	i_3	✓	✓	✓	✓	✓	✓	✓	✓

R.U.		**key input $\mathbf{k = k_1, k_0}$**							
State	**in**	i_2, i_0	i_2, i_1	i_2, i_2	i_2, i_3	i_3, i_0	i_3, i_1	i_3, i_2	i_3, i_3
0	i_0	✓	✗	✗	✗	✓	✗	✗	✗
0	i_1	✓	✗	✓	✓	✓	✗	✓	✓
0	i_2	✓	✓	✗	✓	✓	✓	✗	✓
0	i_3	✓	✓	✓	✗	✓	✓	✓	✗
1	i_0	✓	✓	✓	✓	✓	✓	✓	✓
1	i_1	✗	✗	✗	✗	✗	✗	✗	✗
1	i_2	✗	✗	✗	✗	✓	✓	✓	✓
1	i_3	✓	✓	✓	✓	✗	✗	✗	✗

Table 5.1: Error map for a module with a 2-bit primary input secured with 2-state TLL. The locked trace is $in = i_1, i_0$. Correct key functionality is denoted with highlighted cells.

yields increased output corruption in an improperly keyed IC. To compensate for the added SF inputs, the restore unit must be modified to locate and restore newly locked inputs.

To modify the restore unit, we incorporate additional subkeys to match the additional SF inputs. In the worst case, a new subkey must be included for each SF input. However, combinational optimization techniques can be applied to combine subkeys or reduce the number of bits and hence reduce hardware overhead. See [47, 103, 74, 75] for proposed security aware synthesis algorithms which enable the combinational optimization of TLL.

Error rate can also be scaled by decreasing the length of the subkey. A shorter subkey matches fewer bits of the primary input thereby increasing the percentage of locked inputs within the module. Because we now consider the possibility of a subkey length (s) less than the length of the primary input (n), error rate becomes slightly more complex. We define the number of subkeys currently being compared to the primary input to be c throughout the remainder of this work. This means the error rate of the locked module is $c \cdot 2^{n-s}/2^n$.

The described modifications have been applied within Figure 5.2. Notice that during any given restore unit state, 2 primary inputs are stripped within the locked module. This modification provides double the error rate of 2-state TLL in case of a wrong subkey for any state.

Scaling TLL$_{\mathbf{SFLL-Fault}}$ Trace Length

The length of the trace secured by TLL can be expanded from the presented 2-state case as well. Note that by expanding the trace length, exponentially stronger security guarantees against SAT-based attackers can be provided. See Section 5.1.3 for proof of this claim. In order to expand locked trace length, our construction must include more restore unit states. The number of states and the length of the restricted trace must be equal. $\lceil log_2(l) \rceil$ restore multiplexers (RM1 and RM2 in Figure 5.2) must be included in the design as well. These RMs determine the currently exposed SF inputs in the module.

By combining these 2 modifications, we create a variable trace length TLL construction. An example of a TLL construction for a locked trace of length 4 is presented in Figure 5.2. Notice that the most significant bit (MSB) of the restore unit's state dictates which SF inputs are currently exposed in the module (i_0, i_1, i_2, i_3 with state 0/1, or i_4, i_5, i_6, i_7 with state 2/3). The least significant bit (LSB) of the state controls RM2. Depending on the select line of RM2, half of the stripped functionality created by RM1 is restored. Whenever any secret subkey matches the current input to the module, a state transition occurs. This causes a restore signal to be applied to the XOR gate which either corrects or corrupts the output depending on whether the secret subkey is correct.

Generalized Tunable TLL$_{\mathbf{SFLL-Fault}}$ Construction

A block diagram of a TLL$_{SFLL-Fault}$ construction scaled in both trace length and error rate is contained in Figure 5.2. It differs from the 2-state case contained in Figure 5.1 in 3 ways:

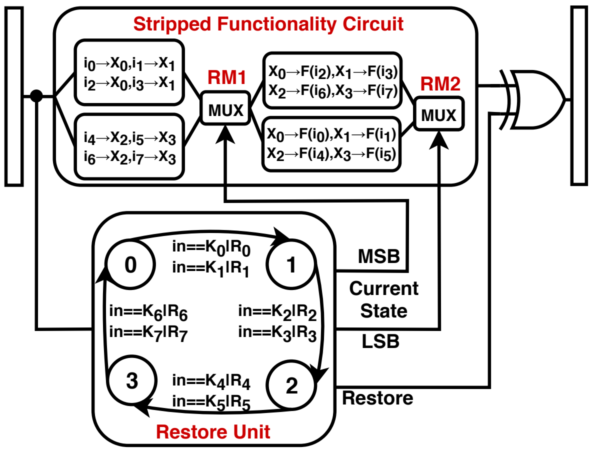

Stripped Functionality Circuit
$i_0 \to X_0, i_1 \to X_1$
$i_2 \to X_0, i_3 \to X_1$
$i_4 \to X_2, i_5 \to X_3$
$i_6 \to X_2, i_7 \to X_3$
RM1
MUX
$X_0 \to F(i_2), X_1 \to F(i_3)$
$X_2 \to F(i_6), X_3 \to F(i_7)$
$X_0 \to F(i_0), X_1 \to F(i_1)$
$X_2 \to F(i_4), X_3 \to F(i_5)$
RM2
MUX
0
1
2
3
in==K_0|R_0
in==K_1|R_1
in==K_2|R_2
in==K_3|R_3
in==K_4|R_4
in==K_5|R_5
in==K_6|R_6
in==K_7|R_7
MSB
Current
State
LSB
Restore
Restore Unit

Figure 5.2: 4-state TLL configuration with 2 locked primary inputs per cycle. Secured input sequence: $i_0 \vee i_1, i_2 \vee i_3, i_4 \vee i_5, i_6 \vee i_7$.

1. Increased states in restore unit (trace length scaling)
2. Additional restore multiplexer, RM1 (trace length scaling)
3. Additional SF inputs in locked module (error rate scaling)

By modifying these details, both the wrong key error rate and the locked trace length of our TLL construction can be altered yielding tunable security guarantees.

5.1.3 Mathematical Foundations of TLL$_{SFLL-Fault}$

The goal of TLL is to expand the parametric space of locking. As shown in Section 4.1, this parametric space is created by the trade-off between wrong key error rate (error severity) and SAT attack resilience. To this point, we have argued TLL achieves this by injecting trace length into this trade-off, thereby disentangling the direct relationship between error severity and SAT resilience. In this section, we prove this claim for our presented TLL construction. Specifically, we show that the

SAT resilience of our presented construction varies in both wrong key error rate (as is the case with all conventional locking) and trace length. Proving this assertion means that wrong key error rate can be increased (improving error severity) without degrading SAT resilience by increasing locked trace length. Hence, TLL injects trace length into the parametric space of locking, thereby expanding it.

The secret key for our TLL construction, $k = k_{l-1}, k_{l-2}, ..., k_0$, is a concatenation of several independent keys corresponding to separately locked primary inputs within the locked trace of length l. We will refer to each of these concatenated keys as *subkeys* of length $c \cdot s$ bits, where c is the number of locked inputs for each position in the trace and s is the length of each locked input. Each of these subkeys correspond to a particular position within the locked trace. Additionally, each position in the locked trace corresponds to a state within the TLL restore unit. Without loss of generality, we will assume that k_0 corresponds to restore unit state 0, k_1 corresponds to restore unit state 1, and so on. For brevity, each TLL construction is presented as a triplet, TLL(s,l,c).

5.1.3.1 SAT Resilience of TLL$_{SFLL-Fault}$

The SAT resilience of a logic locking technique is defined as the probability of a SAT attack successfully recovering the secret key within q queries. To derive this, we assume that the attacker uniformly samples the input space for DIs. Previous research in logic locking has relied upon this assumption in attack resilience proofs such as [103, 45, 99]. We experimentally verify our resulting derivations in Section

5.1.5 to ensure these assumptions are reasonable. Note that despite a focus on the SAT attack presented in [88], our result holds against a series of other SAT-based attackers such as [83, 77, 4, 108, 23].

The goal of a SAT-based attacker is to select all c SF inputs within the locked module as DIs for each of the $i = 1..l$ indices of the trace. By selecting each of the c SF inputs for a given trace index, all possible wrong subkeys for that trace index are eliminated from the keyspace. If the attacker does this for all l subkeys, corresponding to the l trace indices (restore unit states), all wrong subkeys will be eliminated. By concatenating the remaining subkeys, the adversary constructs the correct secret key. We note this approach constitutes a so-called "unrolling" attack where TLL's state machine is unrolled into a series of combinational locking configurations which are then solved by a SAT attack.

We begin characterizing SAT resilience with a derivation of the probability of selecting all c SF inputs within q selections for a given trace index by uniformly sampling the input space. This corresponds to the necessary SAT queries to locate a secret subkey by the above methodology.

Lemma 5.1. The probability of selecting all c SF inputs from an input-space of size 2^s within q queries is $\mathrm{P} = \binom{2^s-c}{q-c}/\binom{2^s}{q}$

Proof. We have a budget of q queries. An input combination of length s constitutes a query. Any input combination is sampled with equal probability. Therefore, there are $\binom{2^s}{q}$ possible ways of doing this. A successful attack is the case in which all c SF inputs are selected in these q queries. Regardless of which order these c inputs are selected, c out of the q queries must be SF inputs. Hence, $q - c$ must be from the set of non-SF inputs. The number of possible non-SF inputs is $2^s - c$. The number of ways in which $q - c$ selections can be made from these $2^s - c$ non-SF choices is $\binom{2^s-c}{q-c}$. Therefore, the probability of locating all c SF inputs within q queries is:

$$P = \binom{2^s - c}{q - c} \Big/ \binom{2^s}{q} \quad , \quad (q \geq c) \tag{5.1}$$

□

Using this result, we derive the SAT resilience of TLL(s,l,c).

Theorem 5.2 (SAT Resilience of TLL)**.** The probability of an adversary unlocking a TLL(s,l,c) locked module within q SAT queries per restore unit state is $P = (\binom{2^s-c}{q-c}/\binom{2^s}{q})^l$.

Proof. As discussed, the goal of a SAT-based attacker is to select all c SF inputs as DIs for each of the $i = 1..l$ indices of the trace. This process recreates the entire secret key. This is because selecting all c SF inputs for each trace index, i, as a DI eliminates all possible wrong subkeys for all i. The remaining subkeys can be concatenated to yield the correct secret key. To accomplish this, the attacker proceeds as follows.

1. Within the locked module, initialize the restore unit state to 0. Note that the restore unit state corresponds to a specific index of the trace. Additionally, all the SF minterms, c, for each trace index, i, are independent. Hence, the SAT attack for each index can occur independently of other indices.

2. The attacker applies a SAT attack against the module in restore unit state 0. On termination, the attack returns the secret subkey for restore unit state 0. The probability of finding the correct subkey in q_0 SAT queries is given by Lemma 5.1: $P_0 = \binom{2^s-c}{q_0-c}/\binom{2^s}{q_0}$.

3. The remaining $l-1$ restore unit states must be attacked to find the remaining $l-1$ secret subkeys. This unlocks the circuit as a whole. For each remaining restore unit state ($i = 2 \dots l$), the adversary will initialize the restore unit to that state and repeat the attack. The probability of finding the correct subkey in q_i SAT queries for the i-th trace index is given by Lemma 5.1: $P_i = \binom{2^s-c}{q_i-c}/\binom{2^s}{q_i}$.

This methodology reconstructs the secret key for a TLL locked module. Therefore, the probability of reconstructing the secret key of a TLL(s,l,c) locked module within q SAT queries per restore unit state is:

$$P = \prod_{i=0}^{l-1} P_i = \prod_{i=0}^{l-1} \binom{2^s-c}{q_i-c}/\binom{2^s}{q_i} \qquad \text{or, equivalently:}$$

$$P = \left(\frac{\binom{2^s-c}{q-c}}{\binom{2^s}{q}}\right)^l, \; q = q_0 = ... = q_{l-1} \tag{5.2}$$

□

Theorem 5.3. The probability of an adversary unlocking a TLL(s,l,c) locked module in q SAT queries per restore unit state exponentially decays in the length of the trace, l.

Proof. From Theorem 5.2, the probability of unlocking TLL(s,l,c) within q queries per restore unit state is:

$$P = \left(\binom{2^s - c}{q - c} \Big/ \binom{2^s}{q} \right)^l \tag{5.3}$$

We can expand this form:

$$= \left(\frac{(2^s - c)!(2^s - q)!q!}{(q-c)!(2^s-q)!(2^s)!} \right)^l = \frac{q^l (q-1)^l ... (q-c+1)^l}{(2^s)^l (2^s-1)^l ... (2^s - c + 1)^l}$$

For a successful SAT attack, $c \leq q \leq 2^s$. Hence, $q/2^s \leq 1$, $(q-1)/(2^s-1) \leq 1$,...,$(q-c+1)/(2^s-c+1) \leq 1$. Therefore,

$$\frac{q(q-1)...(q-c+1)}{(2^s)(2^s-1)...(2^s-c+1)} \leq 1 \tag{5.4}$$

This means that Equation (5.3) exponentially decays in l. □

Let us approximate Theorem 5.2 for clarity. To do so, assume that the number of SF inputs (c) is small. Large numbers of SF inputs quickly yield infeasible design overheads. This means $q!/(q-c)! \approx q^c$ and $(2^s - c)!/2^s! \approx 2^{-c \cdot s}$, yielding the form:

$$P \approx \left(\frac{q}{2^s} \right)^{c \cdot l} \tag{5.5}$$

With this result, we return to our motivation: the trade-off underlying logic locking between error severity and SAT attack resilience requires locking configurations capable of protecting ICs from an untrusted foundry attacker to have an infeasible design overhead. This is because the SAT resilience of logic locking only scales efficiently in wrong key error rate (Section 4.1). As shown by Theorem 4.2, this result applies to all logic locking. However, we designed TLL to inject another parameter, trace length, into logic locking schemes that otherwise lack this parameter. When we consider the impact of TLL's trace length in Theorem 5.2, we find that it yields a locking scheme where SAT resilience is dependent on both wrong key error rate *and* trace length. Because the SAT susceptibility of TLL exponentially decays in the length of the trace, it can be used to efficiently achieve SAT attack resilience and error severity. This ensures that when an IC designer fixes some wrong key error rate necessary for error severity, they can always choose a value of l where SAT resilience is also guaranteed. This constitutes an expansion of the parametric space of logic locking and is the primary contribution of TLL.

To conclude, we note that our derivations are specific to the TLL$_{SFLL-Fault}$ construction. However, any conventional locking enhanced with TLL will exhibit the same functionality. Namely, a locking construction whose currently locked inputs depend on both the restore unit state and the applied wrong key. Because the underlying functionality is identical, the ramifications of TLL will also be identical. Hence, TLL will create the same increase in SAT attack resilience for any trace length scaling, expanding the parametric space of locking.

5.1.3.2 Removal Resistance of TLL$_{SFLL-Fault}$

There have been several proposed removal-type attacks against logic locking techniques such as [47, 100]. When considering TLL$_{SFLL-Fault}$, these attacks can be split into 2 categories: SF removal and restore unit removal. In deriving the resistance of TLL to each of these attacks, we assume the designer has incorporated TLL into the locked module using a security aware synthesis algorithm, such as [47].

First, we consider a removal attack on the SF module. Because SF is added to a module through re-design, it cannot be removed through logic removal. A re-design attack is needed. We defer our analysis of this more complex attack to Section 5.1.3.3. Next, we consider TLL$_{SFLL-Fault}$(s,l,c)'s security against a restore unit removal attack. In this attack, the attacker has located and removed TLL's restore unit.

Theorem 5.4 (Restore Unit Removal Attack Resistance of TLL)**.** In the case of a restore unit removal attack, $c \cdot 2^{n-s}$ errors remain in a TLL(s,l,c) locked module.

Proof. Assume that an adversary has found and removed all restore unit logic within a TLL-secured module. The remaining circuit contains any errors induced through SF minterms. There exists $l \cdot c$ SF minterms in the locked module which induce $lc \cdot 2^{n-s}$ errors. $(l-1)c \cdot 2^{n-s}$ of these errors are corrected by the current RM state yielding $c \cdot 2^{n-s}$ errors present within the module after a restore unit removal attack. □

Finally, we consider the case in which TLL$_{SFLL-Fault}$'s state is fixed (e.g. by removing alternate states). We note that any active SF in the circuit can only be corrected by a restore unit state transition. Hence, an attack which fixes TLL into a single state is unable to restore any SF inputs in the circuit. This means that Theorem 5.4 applies in this case as well and $c \cdot 2^{n-s}$ errors are present in the module. Additionally, because TLL cannot restore errors without changing states, these errors cannot be recovered, regardless of the secret key applied.

5.1.3.3 Re-Design Resistance of TLL$_{SFLL-Fault}$

Let us address a re-design attack on TLL$_{SFLL-Fault}$, where the adversary alters the logic of a locked gate-level netlist to unlock it. Note that this attack is outside of the scope of a more traditional SAT-based attacker model, such as ours (or those proposed in [94, 92, 76, 45, 98]), as it requires significant alteration of the IC's underlying logic. We discuss such an approach because it can be used to weaken TLL$_{SFLL-Fault}$, but it cannot fully unlock the circuit and is costly.

Let us consider an adversary who has located TLL's FSM using the semi-automated reverse-engineering techniques outlined in [26]. Removing this FSM results in $c \cdot 2^{n-s}$ unrecoverable errors (Theorem 5.4). To correct these errors, the attacker can re-design the gate-level netlist to include additional restore logic. Without loss of generality, let us assume a LUT is added with an entry for each of the c SF inputs. This yields a topology consistent with SFLL-Fault [75]. Hence, if the c SF inputs are located, they can be corrected through this added LUT. This approach bypasses the sequential aspect of $\text{TLL}_{SFLL-Fault}$, but there remains 3 key limitations:

1. The circuit is not unlocked. $c \cdot 2^{n-s}$ SF-induced errors remain. To correct these errors, each SF input must be located and entered in the added LUT. A SAT attack can locate these SF inputs, but its complexity scales exponentially.
2. This attack requires netlist modification. After removing TLL's FSM, a LUT must be added. Gate-level changes force the attacker to layout, close timing, verify, etc. the IC. Performing a new tape-out is resource intensive.
3. A new mask must be created to overbuild/counterfeit the IC. This is costly. If the modified mask is used to fabricate ICs for the design house, device tampering is obvious both 1) functionally, through the presence of logically irrelevant key-bits, and 2) visually (i.e. de-layering), through major changes in key logic. This provides a watermark and allows the designer (or any future IP user) to detect tampering.

Thus, the limited and costly nature of such an attack against $\text{TLL}_{SFLL-Fault}$ makes its utility and profitability doubtful.

5.1.3.4 Structural Resilience of $\text{TLL}_{SFLL-Fault}$

Now, we consider a structural attack against $\text{TLL}_{SFLL-Fault}$ launched as follows: 1) locate the restore multiplexer (RM), 2) replace it with a XOR gate to create a miter, and 3) use a SAT tool to locate differences between restore paths. These

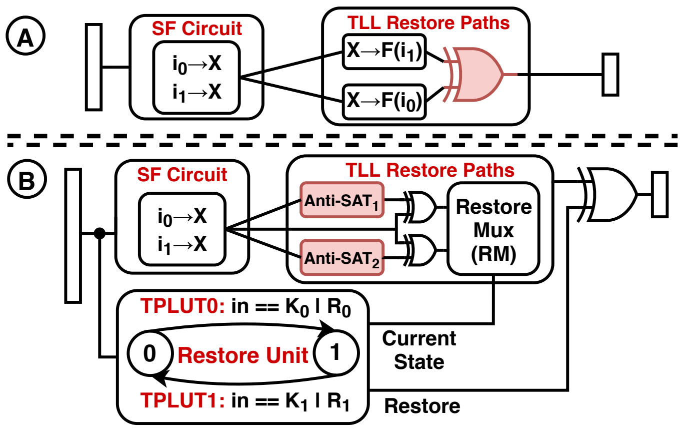


Figure 5.3: A) Configuration of structural miter attack on $TLL_{SFLL-Fault}$ B) Miter-attack-resistant $TLL_{SFLL-Fault}$.

differences are SF inputs, which can be used to construct secret subkeys. Notice that this attack exploits the fact that one can infer the key of SF-based locking constructions through knowledge of SF inputs. Hence, such an attack is only valid against TLL extensions of SFLL-style techniques. This attack is shown in Figure 5.3A.

In a base TLL$_{SFLL-Fault}$ locking configuration, such an attack is successful. To ward against it, a designer can incorporate additional combinational locking within each restore path. Let us consider replacing each restore path with a buffer locked by an Anti-SAT block [94, 92], as shown in Figure 5.3B. In this case, the key applied to each Anti-SAT block determines the functionality of each restore path. Hence, the suggested miter attack now only identifies currently corrupted inputs for each Anti-SAT block, rather than the SF inputs in the circuit. This modified design no longer encodes SF inputs in restore paths, hence, no longer leaks the SF minterms.

To unlock this modified configuration, the adversary still must determine all SF inputs. With this knowledge, they can 1) determine a correct key for each Anti-SAT block, which restores the necessary SF inputs for each restore path and 2) set each TLL subkey to restore the remaining SF cubes. Thus, the SAT resilience derived in Thm. 5.2 holds and TLL$_{SFLL-Fault}$ maintains security against this miter-based structural attack.

5.1.4 Enhancing Alternative Techniques With TLL

TLL is an enhancement of existing logic locking art. However, to this point, we have only provided a single concrete TLL construction, TLL$_{SFLL-Fault}$. Therefore, to show how TLL can be used to achieve equivalent functionality in alternative locking techniques, we present another example of TLL-enhanced logic locking, TLL$_{Anti-SAT}$. A block diagram of TLL$_{Anti-SAT}$ is included in Figure 5.4.

In the figure, 2 Anti-SAT blocks are incorporated within the circuit. These blocks serve as the core element of Anti-SAT locking, injecting error within the circuit for a specific, key-driven minterm [94, 92]. Note that each Anti-SAT block has been provided an entirely independent subkey (K_0, K_1), which serves as the secret key of TLL$_{Anti-SAT}$. A restore multiplexer has been incorporated to select the Anti-SAT block currently injecting error within the circuit. This multiplexer is controlled by the current state of the TLL restore unit, which switches between states on 2 arbitrary (e.g. randomly selected) input minterms (in0, in1).

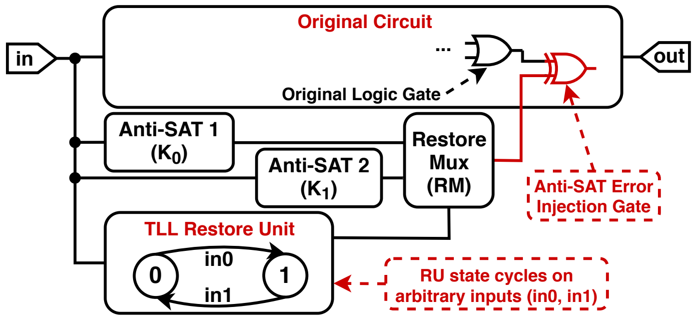


Figure 5.4: 2-state TLL-enhanced Anti-SAT construction.

This construction uses 2 Anti-SAT blocks to lock a trace of length 2. Because each Anti-SAT block employs an independent key, each index in the trace corresponds to a different secret subkey. By applying the approach in Theorem 5.2, we can show that increases in trace length exponentially scale the SAT resilience of Anti-SAT. At the core of each presented construction is the same underlying principles and structures, which allow TLL to provably expand the parametric space of the underlying locking scheme. To enhance an arbitrary locking scheme the following criteria must be met:

1. l independent logic locking configurations with independent keys must be integrated into a single module.

2. An FSM must be integrated into this same netlist, which enables only 1 of the l independent locking configurations for each state. Each of the l locking configurations must be enabled in at least one state.

Notice that the 2 presented TLL schemes meet these criteria:

- **TLL$_{\mathbf{SFLL-Fault}}$:** 1) l complete SFLL-Fault configurations are integrated. Each of the l configurations use an independent subkey and separate SF inputs. 2) Each TLL state enables a separate subkey and set of SF inputs.

- **TLL$_{\mathbf{Anti-SAT}}$:** 1) l Anti-SAT blocks are added to the circuit with independent keys. 2) Each TLL state enables only one of the l Anti-SAT blocks present in the design.

By the approach in Theorem 5.2, a logic locking scheme that meets these criteria can be shown to exponentially increase SAT resilience as trace length is scaled. Thus, a scheme that meets these criteria has been successfully enhanced by TLL.

5.1.5 Experimental Analysis of TLL$_{SFLL-Fault}$

We provide an experimental analysis of TLL$_{SFLL-Fault}$. This analysis is broken into 3 components. First, we demonstrate that TLL expands the parametric space of logic locking by validating its theoretical SAT resilience in a series of benchmarks. Our results show that the empirical SAT resilience of TLL-locked circuits closely match the theory derived in Section 5.1.3. Second, we use the same benchmarks to characterize TLL's overhead and the effects of trace length scaling. Based on our experiments, we found that TLL achieves exponentially stronger security than a comparable SFLL-Fault configuration while only incurring an overhead of +3.8%, -0.8%, and +3.5% for area, delay, and power. Finally, we provide an architectural example of TLL by locking the 80186 processor netlist that was un-securable using prior art. Using our locking methodology, we show that TLL can simultaneously achieve both error severity and SAT resilience with detailed architecture-level simulations of the locked IC. For these experiments, we used the 10 largest benchmarks of ISCAS'89 [10] and ITC'99 [19] to evaluate TLL. These netlists are identical to those used by SFLL [103] to enable a direct comparison.

5.1.5.1 Experiment 1: SAT Resilience of TLL$_{SFLL-Fault}$

We characterized the SAT resilience of TLL within our 10 benchmark circuits to experimentally verify the theory derived in Section 5.1.3.1. Specifically, we aimed to evaluate whether the probabilistic derivation in Theorem 5.2 is indeed practically valid, thereby empirically verifying TLL's ability to expand the parametric space of locking. To this end, we experimented with 3 TLL instances, TLL(s=11,l={1,2,3},c=1). For each TLL instance, we constructed 4 different locking configurations by randomly selecting alternative sets of minterms to lock. Hence, for each trace length we had 40 (4 locking configurations * 10 benchmarks) unique TLL netlists. We SAT attacked each netlist using the SAT attack from [88] and recorded the number of SAT queries required to unlock each TLL instance. With this data, we computed the probability of a SAT attack terminating within q SAT queries per restore unit state. Figure 5.5 displays the observed SAT resilience alongside the theoretical derivation from Theorem 5.2. The experimental/theoretical results closely match, supporting our derived results. Notice as trace length is increased, the experimental SAT susceptibility of TLL exponentially decays. This supports Theorem 5.3 which proves that the SAT resilience of TLL exponentially improves for a linear increase in trace length.

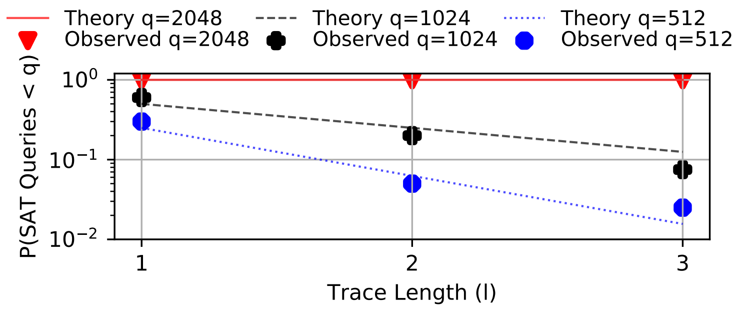


Figure 5.5: Comparison of the theoretical and experimental SAT resilience achieved with TLL(s=11,l={1,2,3},c=1).

5.1.5.2 Experiment 2: ADP Overhead of TLL$_{SFLL-Fault}$

We characterized the area, delay, and power (ADP) overhead of our presented TLL construction. To do this, we incorporated TLL(s=128,l=2,c=2) within each benchmark circuit. The corresponding post-mapping overhead was determined using the Cadence Encounter RTL Compiler with the Synopsys 90nm SAED library. Additionally, we evaluated an SFLL-Fault implementation with the same error rate (s=128,c=2) for comparison. The ADP overhead for each benchmark is in Figure 5.6. On average, we found the ADP overhead of TLL to be 9%, 1.1%, and 7.2%, respectively. When compared to an equivalent error rate implementation of SFLL-Fault, TLL demonstrated a modification of +3.8%, -0.8%, and +3.5% in ADP overhead. Therefore, when SFLL-Fault is enhanced with TLL, exponentially stronger SAT resilience is achieved with only a small additional ADP overhead.

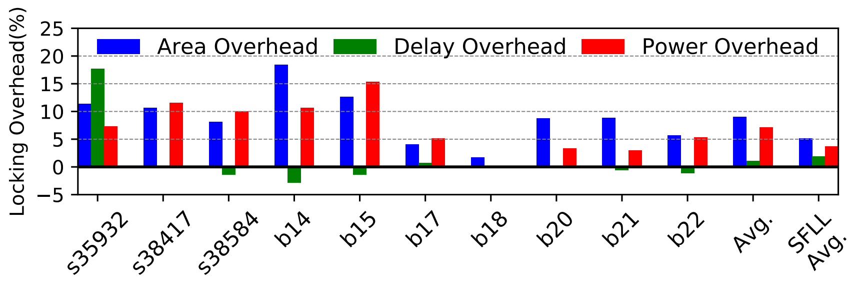


Figure 5.6: ADP overhead of TLL(s=128,l=2,c=2). The average ADP overhead of an equal error rate SFLL(s=128,c=2) construction is shown for reference.

We have characterized the overhead associated with trace length scaling as well. To do so, we calculated the design overhead of TLL(s=128,l={2,3,4},c=2) within each of our 10 benchmark circuits. These results have been aggregated in Figure 5.7. Notice that the average design overhead of TLL increased by +2%, +0.8%, and a +1% for area, delay, and power as trace length was increased from l=2 to l=3. As we further increased trace length from l=3 to l=4, a slightly smaller ADP overhead increase of +1.8%, +0.9, and +0.7% was observed. This small reduction in added area and power overhead was due to the increased combinational optimization made possible as more functionality was stripped from the circuit. This implies that as trace length continues to scale, the overhead due to added TLL logic (i.e. LUT entries and restore unit states) will be increasingly offset by additional combinational optimization. To conclude, this experiment indicated that a linear increase in trace length yields a slightly sub-linear increase in ADP overhead. However, for this increase in overhead, TLL was shown to provide an exponential increase in SAT resilience. Hence, TLL provides an efficient trade-off between ADP overhead and security.

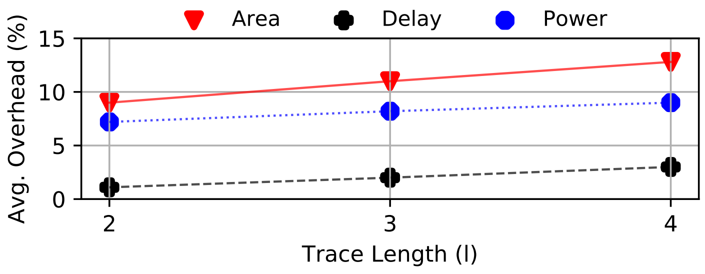


Figure 5.7: Average area, delay, and power overhead of trace length scaling for TLL(s=128,l={2,3,4},c=2).

5.1.5.3 Experiment 3: Security of TLL Beyond the Gate Level

We used TLL to lock the 80186 core found to be un-securable by logic locking in Section 4.3. By evaluating TLL in this netlist, we explore its ability to secure real-world ICs.

TLL Configuration:

We began by designing a TLL construction capable of achieving security within the data path of our 80186 netlist. Note that conventional logic locking was unable to achieve security in this netlist (Section 4.3). Using the design space exploration in Figure 4.4, we were able to quantify the wrong key error rate necessary for error severity. From the figure, this corresponds to the wrong key error rate of an SFLL-Fault configuration requiring between 1024 and 4096 SAT queries to unlock on average. According to Theorem 4.2, this corresponds to an error rate of 0.02% and 0.08%, respectively.

We designed 2 TLL constructions, TLL(s=17,l=?,c=16) and TLL(s=17,l=?,c=64), to exceed an error rate of 0.02% and 0.08%. Next, we selected a trace length which allows TLL to achieve strong SAT resilience given each wrong key error rate. To do so, we aggregated the SAT resilience of TLL for varying trace lengths in Figure 5.8. Given a trace length of 8, there exists a negligible probability (2^{-128} and 2^{-512} respectively) of a SAT attack locating a correct subkey in $< 2^{16}$ queries per restore unit state. This constitutes extremely strong SAT resilience. Therefore, we used a trace length of 8 for each construction. Finally, we randomly selected input minterms for locking and incorporated TLL(s=17,l=8,c=16) and TLL(s=17,l=8,c=64) into the data path of the 80186 netlist. To evaluate each IC, we the ObfusGEM simulator [115].

Simulation Framework:

Using the ObfusGEM simulator, we performed cycle-accurate simulations of locked ICs to quantify the rate with which a given locking construction induced critical failures in IC workloads. This failure rate serves as a measurement of the error severity of a given locking configuration. For these simulations, we chose 9 PARSEC benchmarks [7] to serve as a reasonable cross-section of common computing applications.

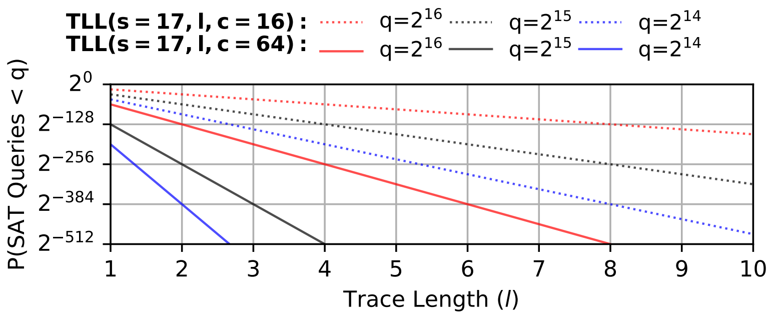


Figure 5.8: SAT attack resilience of a locked 80186 netlist with varying trace length TLL constructions.

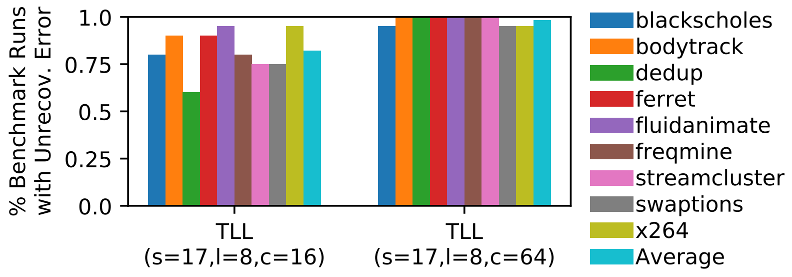


Figure 5.9: Experimentally derived error severity for TLL-secured 80186 netlist running PARSEC workloads.

Experimental Results:

We quantified the error severity of each TLL construction with our simulation framework. Specifically, we performed 40 Monte Carlo simulations of each PARSEC benchmark with a different, randomly-selected wrong key for each trial. Hence, each Monte Carlo trial induced output corruption for a different set of input minterms in the processor. The percent of PARSEC benchmark runs with unrecoverable errors for each TLL configuration is in Figure 5.9 alongside the corresponding ADP overhead in Table 5.2. Based on the simulation results, both TLL configurations achieved error severity. The first configuration, TLL(s=17,l=8,c=16), achieved an average benchmark failure rate of over 80%. This result implies that an untrusted foundry pirating this locked IC would find 80% of workloads to fail. The second TLL configuration, TLL(s=17,l=8,c=64), showed even stronger error severity, causing 97% of workloads to fail given a wrong key. We now turn our attention to SAT resilience. In Figure 5.8, we have aggregated the SAT resilience achieved by each TLL construction over varying trace lengths. Because the SAT resilience of TLL-enhanced locking techniques grows exponentially in trace length, relatively short trace lengths achieved extremely strong SAT resilience. Given our selected trace length of 8, there exists a negligible probability (2^{-128} and 2^{-512} respectively) of a SAT attack successfully locating a subkey in less than 2^{16} SAT queries per restore unit state. This indicates strong SAT resilience within each netlist.

TLL Construction	**Area**	**Delay**	**Power**
TLL(s=17,l=8,c=16)	3.30%	0.00%	3.42%
TLL(s=17,l=8,c=64)	12.40%	6.43%	15.6%

Table 5.2: ADP overhead for TLL-secured 80186 core.

Finally, we note that the first TLL configuration, TLL(s=17,l=8,c=16), achieved security with minimal design overhead, with only a ∼3% degradation in area/power and no delay overhead. The second TLL configuration, TLL(s=17,l=8,c=64), exhibited a higher overhead, however, provided much stronger security guarantees to compensate for this additional overhead. Therefore, both TLL constructions simultaneously achieved strong error severity and SAT resilience guarantees with only modest design overhead degradation. The conclusions of this experiment can be summarized as follows:

1. Due to the identified parametric space, logic locking techniques with a feasible overhead were unable to achieve error severity and SAT resilience in this netlist (Sec. 4.3).

2. By enhancing conventional locking with TLL, a locking configuration was designed and empirically shown to achieve both error severity and SAT resilience. Therefore, through trace length scaling, TLL expanded the parametric space of locking, overcoming the limits of prior art.

5.1.6 Conclusion

We proposed Trace Logic Locking (TLL) to expand upon the trade-off between error severity and attack resilience derived in Section 4.1. TLL is a provably secure and scalable enhancement to existing logic locking techniques which locks a sequence of primary inputs, known as a *trace*. By locking traces, TLL expands the parametric space of logic locking. This allows an IC designer to achieve both error severity and SAT resilience by varying trace length. We provided a theoretical and empirical demonstration of this. The low overhead nature of TLL was verified as well in 10 benchmark circuits. Finally, TLL was used to secure a real-world 80186 core. Through architectural simulations, we showed that TLL achieved both error severity and SAT resilience simultaneously.

5.2 Memory Locking

We continue by proposing another logic obfuscation technique capable of comprehensive security guarantees despite the trade-off between error severity and attack resilience. However, instead of injecting an additional degree of freedom into the parametric space of locking, we instead take a different approach by targeting non-combinational modules. We introduce an automated and attack-resistant obfuscation technique, called Memory Locking, which targets on-chip SRAM. We then demonstrate the application-level effectiveness of Memory Locking through ObfusGEM simulations of obfuscated processors.

5.2.1 Memory Locking Construction and Implementation

To remedy the issues underlying combinational obfuscation presented in the previous chapter, we propose Memory Locking, a logic obfuscation technique focused on denying on-chip SRAM functionality to the adversary. The SRAM circuit is targeted for 3 reasons. 1) SRAM circuitry dominates processor area and is involved in most processor functionality. This provides flexibility in obfuscatable location and functionality. 2) SRAM contains a delicate analog/timing balance. This leads to discrete-domain attack resilience and easily induced errors. 3) SRAM arrays are generated with design automation which can be leveraged to incorporate Memory Locking.

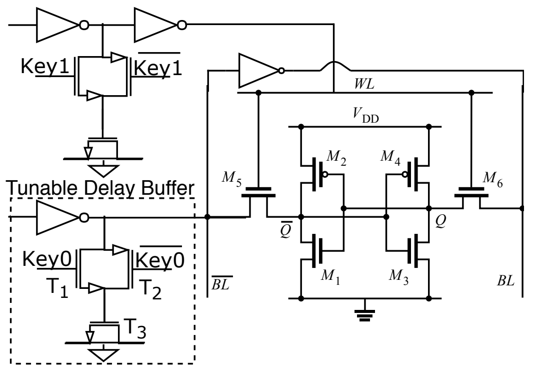


Figure 5.10: Memory-locked SRAM cell.

More specifically, Memory Locking is the insertion of tunable delay buffers (TDB), contained in the box in Figure 5.10, within buffer step-up chains of the bit-lines and word-lines of on-chip SRAM arrays. These TDBs enable the key-driven modification of the parasitic capacitance on these lines thereby altering the analog signature (drive strength, power leakage, timing, etc.) of any attached SRAM cells.

Throughout the remainder of this work, we will rely on an abstraction called the ϕ value of an SRAM cell, which characterizes the current state of analog parameters a cell is operating in. We use this because an SRAM cell is designed for multiple interrelated parameters including relative word/bit-line timing, power leakage, cell sizing, drive strength, and cycle timing. The raw values of each of these variables are irrelevant as Memory Locking relies primarily on the divergence of these variables from the values the SRAM cell was designed for. Fundamentally, memory locks act to upset the SRAM cell state from the unique designed for state, $\phi_{correct}$, to an non-unique incorrect state, $\phi_{incorrect}$, when improperly keyed.

5.2.1.1 Memory Locking Example

A basic example of a memory locked circuit is contained in Figure 5.10. First, we focus on the functionality of the labeled TDB circuit, T1-T3, located on the bit-line of Figure 5.10. When a '1' is applied as key0 to the pass transistors (T1 and T2), a parasitic capacitance is created through the gate/source-drain of T3 and added to the bit-line. Applying '0' as key0 would not connect the parasitic capacitance (T3) to the circuit. At a fundamental level, by modifying the key, Memory Locking acts to modify parasitics on the bit/word-line. This added parasitic alters the bit/word-line drive-strength, leakage, timing, and other analog parameters. Utilizing this, when a correct key is applied to the IC, the signal modifications created by memory locks induce a designed for state. In an improperly keyed IC, an unintended parasitic is introduced which upsets the analog equilibrium of the SRAM array and induces errors.

Now, let us assume the memory locked SRAM circuit displayed in Figure 5.10 was designed to have a correct key value of '10.' If an untrusted foundry were to fabricate this circuit and incorrectly apply the key '11,' both a write error and read error state would be induced. In the case of a write, no value would be stored in the SRAM cell because the pass transistor (M6) is no longer able to overpower the value held in the SRAM cell inverter (M3 and M4) due to the increased bit-line capacitance. Note that this error is due to the drive-strength of the bit-line rather than bit-line timing. While relaxing timing might help to alleviate this error, a sufficiently large TDB would restrict the bit-line charge from ever overpowering the SRAM cell, inducing an error regardless of timing. In the case of a read, a read error would occur as the SRAM cell would be unable to pull the bit-lines apart rapidly enough to ensure accurate sense amplifier functionality, once again due to the increased capacitance. Again, as is the case for a write, a sufficiently large TDB would yield a bit-line too capacitive to be overpowered by the SRAM cell, inducing an error regardless of timing. Each alternate wrong key results in similar analog and timing issues within the locked cell.

5.2.2 Relationship to Prior Work

Notice that Memory Locking relies on similar structures to delay locking [93], but utilizes them differently. TDBs are utilized in Memory Locking to modify the analog signature of an SRAM array (i.e. power leakage, drive strength, etc.) rather than to create timing violations within combinational paths as was the case in [93]. Memory Locking targets bit/word-lines because these lines must have sufficient drive strength to deliver power to overwrite internal SRAM transistors on a write, but also capable of being overpowered by these same transistors on a read for accurate functionality. By modifying analog parameters with TDBs and SRAM ϕ values, Memory Locking creates security through inducing a challenging analog design problem with many interdependent variables rather than the combinational timing focus of [93].

5.2.2.1 Locking Large Scale SRAM Arrays

As we scale Memory Locking, a security limitation presents itself. Assuming the SRAM array was symmetric, the ϕ value for every cell is identical and therefore each SRAM cell will have the same correct key. An adversary could reverse-engineer the correct key for one cell and replicate this key for all other cells, unlocking the whole SRAM array. This is due to *lack of diversity* in ϕ values. Because of this simple intuition and the symmetry of SRAM arrays, Memory Locking as described would be limited in key-space and security.

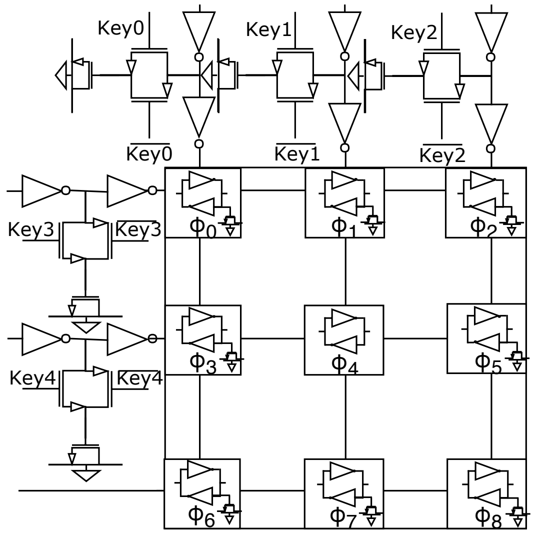


Figure 5.11: 100% unique ϕ value memory locked 3x3 SRAM.

We can create diversity through ensuring multiple unique $\phi_{correct}$ values throughout the SRAM array. By placing non-keyed parasitics within SRAM cells through small, but *random* deviations in cell parameters, the designer can create unique $\phi_{correct}$ values for each cell (or set of cells). Each of these $\phi_{correct}$ values have a unique correct key which should be at least 1 bit, but could be longer. As we linearly increase the number of unique $\phi_{correct}$ values, the attacker will face an exponential increase in the searchable key-space, therefore an exponential increase in the required reverse engineering effort. Finding the correct key for a cell in isolation does not guarantee a solution for cells with alternate $\phi_{correct}$ values.

By designing for multiple $\phi_{correct}$ values throughout the array, the adversary is forced to redesign the SRAM array in its entirety in the process of solving for the correct key. This is unrealistic given that 50-90% of the transistor count on modern CPUs is devoted to SRAM circuitry [59]. In Figure 5.11 we show the topology of a memory locked SRAM array containing a unique $\phi_{correct}$ value for each SRAM cell.

5.2.2.2 Memory Locking Implementation

Memory layout is generally performed with an SRAM compiler. These tools strive to lessen the time required to layout the large area of SRAM cells that span modern ICs. We propose a methodology for the design automation of a memory locked SRAM array leveraging the feature set of openRAM [28], an open-source SRAM compiler. Despite our focus on openRAM, nearly every modern SRAM compiler shares the functionality necessary to implement this methodology.

To automate a memory locked layout, two additional custom blocks, a TDB and a parasitic capacitance with parameterizable sizing, must be included in the SRAM compiler. First, the IC designer would tile an array of SRAM cells for a specific architectural block (i.e. register file). Depending on the level of security desired, a certain set of unique ϕ values are chosen. These values are distributed randomly across the SRAM cells. Based on the ϕ value, each cell is redesigned to include a certain internal parasitic capacitance.

Following array layout, TDBs are added on bit/word-lines utilizing the compiler's timing analysis tool to size these locks. Note that each unique ϕ value corresponds to a unique Memory Locking formulation and requires at least 1 key bit. The total number of key-bits corresponds to the total number of unique ϕ values in the SRAM array. Following said modifications, the IC designer proceeds with a standard design flow, adding peripheral circuitry as required. This automated methodology yields a memory locked SRAM with a given key.

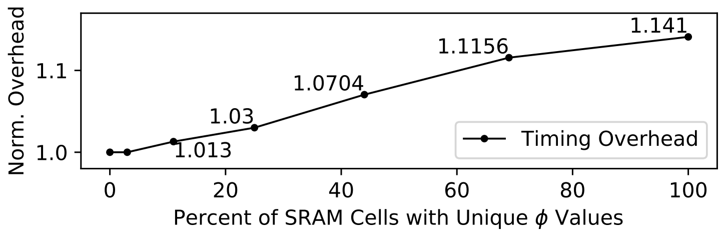


Figure 5.12: Memory locking security/timing overhead trade-off.

Leveraging the proposed methodology, we designed a 6x6 SRAM array to explore the effects of Memory Locking in a slightly larger circuit. This array was designed using FreePDK15 [6], an open-source, 15nm, predictive-process library and simulated using HSpice. With these tools, we were able to design a 15nm SRAM cell, tile it into a memory locked 6x6 array, and add peripheral circuitry to the array. By cycling the inputs to our decoder, we were able to simulate memory traffic with accurate timing and control signals.

By sweeping over the percentage of unique ϕ values present in the array, we are able to quantify the timing overhead of Memory Locking. Because SRAM cycle time is dictated by the capacitance of the circuit, increases in unique ϕ values cause a degradation of the minimum cycle time of the array. We quantified the relationship between unique ϕ values and circuit timing degradation in Figure 5.12. Note that a linear increase in the percent of unique ϕ values yields a linear increase in timing overhead while exponentially increasing key-space.

5.2.3 Security Analysis of Memory Locking

5.2.3.1 Tool-Driven Approach

The automated nature of Memory Locking might drive concerns of an adversary using similar tools to determine the correct key for a memory locked SRAM. While the attacker has access to circuit details, they would need to perform detailed row/column simulations for any SRAM cell with a unique ϕ value and a memory lock on an associated bit/word-line. For each cell, the attacker must apply a key to the array and then read/write both a '1' and a '0' at each locked cell to verify it. To verify that a read stability error does not occur, the attacker must perform two consecutive reads for each word-line while storing both a '1' and a '0'. To verify the write capability from any state, the adversary must write each cell from '1' to '0' and '0' to '1'. This process would require 9 read/write operations and hence 9 clock cycles in the best-case to achieve. This series of read/writes must take place for each locked cell in the SRAM array for each key guess. This implies that unlocking a register file, one of the smallest on-chip SRAM arrays, with a 64-bit key, corresponding to 64 memory locks with unique ϕ values, and 64, 64-bit registers would require over $6.8 * 10^{23}$ cycles to unlock. This is unrealistic.

5.2.3.2 SAT Based Approach

SAT-based formulations, including those which target timing based locking, such as TimingSAT [40, 12], do not apply to Memory Locking. This is due to the feedback loop and internal state present in each SRAM cell. This feedback loop leads to a recursive and thus unresolvable SAT formulation. We direct the reader to Lemma 1 from [108] which states that if a feedback loop in a circuit is stateful, SAT attack will enter an infinite loop. Given that memory is inherently stateful, Memory Locking becomes SAT-unresolvable.

5.2.3.3 Removal Attack

In this case, we assume the adversary has removed all TDBs from SRAM circuitry. This creates a non-functional circuit as the SRAM array was designed to function given only the correct key configuration which includes added parasitic capacitance from the memory locks.

5.2.3.4 Redesign Based Attacks

One can argue that locked, on-chip SRAM arrays could be selected from the layout and replaced by an unlocked SRAM array of the same size. However, such an attack is also unrealistic. The primary concern is that 50-90% of transistor count within modern processors is involved in SRAM circuitry [59]. Even with the help of SRAM automation tools, searching through, removing, and then redesigning this portion of IC transistor count is a massive undertaking. On-chip memory is distributed in several smaller modules such as pipeline registers, register files, branch predictors, etc. A complete "find and replace" of all these locked modules with unlocked modules while matching the timing, area, etc. characteristics is quite a challenging endeavor. We also emphasize that conventional locking approaches are subject to such ”find and replace” attacks as the functionality of targeted combinational modules, such as an adder, multiplier, instruction decoder, etc., is generally known.

5.2.4 Memory Locking Security Beyond the Gate Level

5.2.4.1 Simulator Overview

For this work, we leveraged ObfusGEM to *close the loop* between module-level obfuscation and its application-level impact. To do so, we incorporated various locking techniques within an 80186 processor netlist. A fault analysis of the netlist was performed for a given key and the errant minterms remaining in the circuit were incorporated into the GEM5 simulator model of the core. Workloads could then be run on the GEM5 model of the locked netlist to evaluate the application-level impact of module-level locking.

We configured our simulator to mimic a logically obfuscated 80186 core running a Linux operating system. We performed our architectural benchmarking using benchmarks from the PARSEC benchmark suite [7] and aggregated the results of 40 monte-carlo simulation runs of each benchmark and processor configuration to quantify the impact logic obfuscation had on timing overhead, error rate, and error severity. We define error severity as the number of operations successfully executed until an unrecoverable error occurs in the processor, indicating the amount of work completed before obfuscation related errors derail the core. We define error rate as the percent of benchmark runs which do not complete successfully due to errors injected from logic obfuscation.

5.2.4.2 Simulation Results

We have investigated the architectural impact of both state-of-the-art gate-level locking and Memory Locking. In this section, we present the simulation results derived using the methodology laid out in Section 5.2.4.1. For space, we only include results for the optimal configuration of each technique in Figure 5.13. In our consideration of Memory Locking, we target on-chip memory modules used for the integer register file, floating point register file, data cache tag, and data cache data field. We model Memory Locking using randomly selected keys of various lengths at each of these locations. We initially consider Memory Locking in a single location.

Stand-Alone Memory Locking: Based on simulation results for each Memory Locking location, locking the data cache tag performed optimally. As seen in Figure 5.13, any key length greater than 32-bits yielded 100% application-level error rates for each workload within the first 100 operations executed. This implies that minimal useful work could be performed. Compare this to the performance of conventional logic obfuscation in Figure 4.5 where 0% application-level error rates were recorded for SAT-resilient obfuscation configurations.

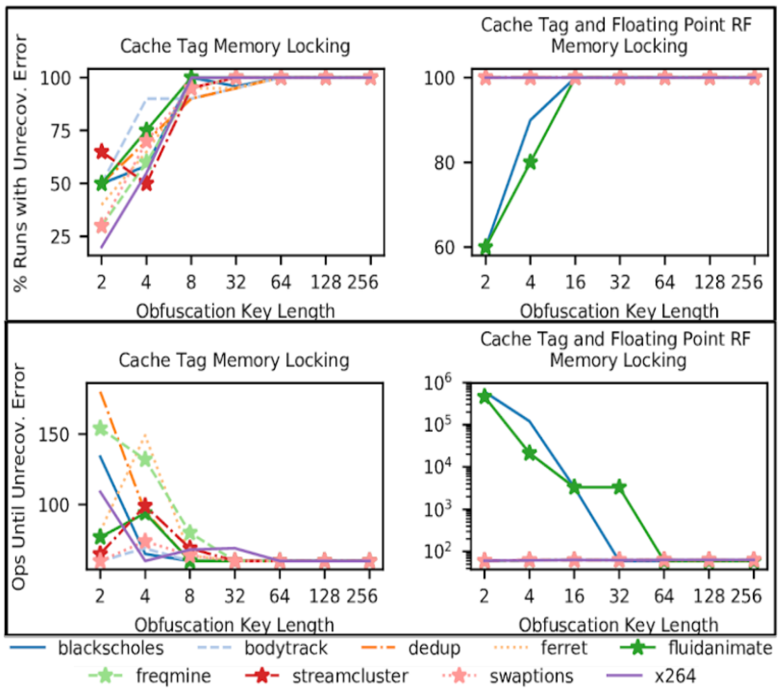


Figure 5.13: Error rate and severity results for optimal Memory Locking configurations of 80186 core running PARSEC suite.

In addition to the application-level security of Memory Locking, the incurred timing overhead was also much lower than the SFLL locked processor as shown in Figure 5.14. For brevity, we have only shown results for 100% unique ϕ value Memory Locking, the worst case timing overhead. These significantly reduced overheads are due to the inclusion latency hiding techniques (i.e. out-of-order processing, cache-banking, etc.) within the IC. These techniques, architected to hide memory latency, were able to amortize the 14.1% latency overhead from Memory Locking for all locations except integer register file locking. This is unsurprising as the integer register file latency dictates the cycle time of this processor; therefore, no

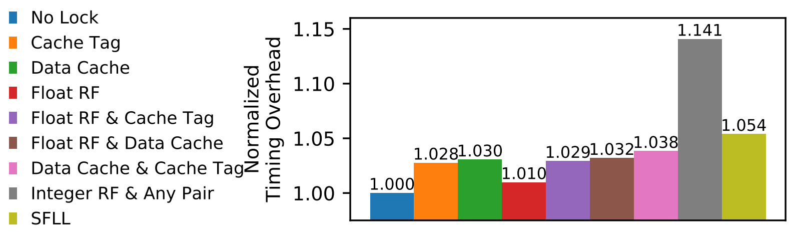


Figure 5.14: Average obfuscation timing overhead on 80186 core running PARSEC.

overhead could be amortized. Data cache tag Memory Locking exhibited only a 2.8% runtime overhead compared to the 5.4% overhead reported by SFLL, a significant improvement.

Paired Location Memory Locking: We performed pairwise simulations of all combinations of 2 Memory Locking locations with an equal key distribution to investigate the effect of locking diversity on application-level security. We found the optimal pairwise configuration to be cache tag and floating point register file Memory Locking. Multiple locking locations as a whole appear to perform better than their single location counterpart. This is likely due to the diversity in processor functionality locked, thereby inducing more diversity in errors. When considering the optimal stand alone and pairwise Memory Locking configuration, as shown in Figure 5.13, the benefit of diversity seems reduced. Cache tag and floating point register file locking with a 128-bit key caused 100% of workloads to encounter an unrecoverable error within the first 100 operations. This performance is similar to stand alone cache tag locking; however, notice that the pairwise application-level error rate is higher regardless of key length. When considering the amortized timing overhead in Figure 5.12, the optimal pairwise and stand alone Memory Locking configurations perform similarly, with paired locking exhibiting a 2.9%

overhead compared to the 2.8% overhead of stand alone data cache tag locking. As before, Memory Locking appears to significantly outperform SFLL regardless of configuration.

Combinational Obfuscation: In Section 4.3, we used SFLL [103, 75, 74] and Anti-SAT [94, 92] to lock 2 processor netlists using the methodologies proposed by the authors. We demonstrated a successful attack methodology to unlock the control path circuitry and an approximate attack to partially unlock the data path circuitry. Even with gate-level error remaining in the data path circuitry, simulation results demonstrated a 0% application-level error rate, as seen in Figure 4.4.

5.2.5 Conclusion

In this section, we proposed a logic obfuscation technique to obfuscate SRAM circuitry, called Memory Locking, and a corresponding automated implementation methodology. Using ObfusGEM, we evaluated Memory Locking and state-of-the-art combinational logic obfuscation at the architecture-level. Based on our results for the simulated architecture, we found 64-bit cache tag/64-bit floating point register file combined Memory Locking, or 128-bit cache tag Memory Locking to perform optimally. However, other architectures and techniques may result in different optimal configurations.

5.3 High Error Rate Keys (HERK)

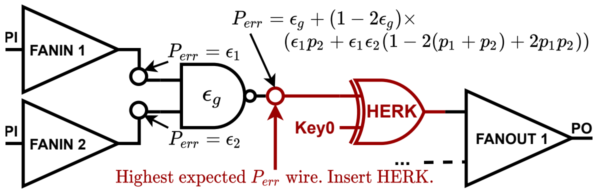
To continue, we propose a third logic obfuscation scheme aimed at securing probabilistic circuits in particular. As noted in Section 3.6, prior work, such as [51, 56, 113], has demonstrated that probabilistic IP is vulnerable to piracy and reverse engineering via SAT-style attacks. As a result, a SAT-concerned designer must utilize low-error logic obfuscation techniques to prevent SAT-style attacks (see Section 4.1). Based on the simulation results presented in Section 4.4, this severely limits the efficacy of obfuscation when considered beyond the gate level. In this section, we introduce an obfuscation technique for probabilistic circuits, known as High Error Rate Keys (HERK), which utilizes the inherent probabilistic/uncertain behavior in a probabilistic circuit to hide the correct secret key under stochastic noise. Such an approach enables high-error obfuscation schemes to resist SAT-style attacks within probabilistic circuits. As a result, designers can configure obfuscation within probabilistic circuits capable of simultaneously achieving both a high SAT attack resilience and error severity. This allows strong comprehensive security to be achieved, thereby protecting probabilistic IP from piracy and reverse engineering. After introducing HERKs in this section, we empirically affirm their security guarantees against the StatSAT attack, a SAT attack proven to be capable of unlocking probabilistic IP in [51, 113].

5.3.1 Overview of High Error Rate Keys (HERK)

We now introduce our novel logic locking scheme called High Error Rate Keys (HERK) to counter the StatSAT attack. HERKs are key gates inserted on high error rate wires in probabilistic circuits. Such an approach hides the correct key value (i.e. correct HERK function) under high probabilistic noise present at the insertion point. This makes it extremely hard for a SAT solver to infer the correct key (i.e. correct function) of any logic locking influenced by HERK function. We show that this leads to an exponential increase in StatSAT runtime for a linear increase in the number of inserted HERKs. Our proposed approach distinguishes itself from prior locking art by using naturally occurring high-error locations unique to probabilistic circuits to achieve security. As we show, by combining HERKs with deterministic locking, both SAT-style attacks and other prevalent attacks on locking can be mitigated to protect previously vulnerable probabilistic design IP.

The proposed HERK structure is shown in Figure 5.15. To implement the construction in this figure, a designer must first identify locations in the circuit that exhibit a high error rate (i.e. wires where probabilistic behavior makes wrong signal values likely). The error rate at each location in the circuit can be calculated via Boolean Difference Calculus (see Sec. 3.5). However, the calculated error rate on a wire is dependent on the considered input pattern. Ideally, HERKs will be inserted at high error rate locations for a large set of input patterns. We propose the following approach to find high error locations for HERK insertion.

First, some set of random inputs (or input probabilities) is selected. Next, Boolean Difference Calculus is used to calculate the BER at each location in the (correctly-keyed) circuit for each input. The BER calculation is then averaged over all inputs for each location. The wires with the highest average BER likely exhibit error for a sizable subset of the considered inputs, making them ideal locations for HERK insertion. Finally, a XOR gate is inserted on the highest average BER wires, driven by the identified wire and an added key input. A XOR gate is used due to its minimal error masking properties compared to other gates (i.e. any single input error necessarily results in an output error for a XOR gate). The added XOR gate is known as a *High Error Rate Key (HERK)*. We propose inserting many HERKs into the circuit in this fashion to ensure security.



PI
FANIN 1
PI
FANIN 2
$P_{err} = \epsilon_1$
$P_{err} = \epsilon_2$
ϵ_g
$P_{err} = \epsilon_g + (1 - 2\epsilon_g)\times$
$(\epsilon_1 p_2 + \epsilon_1 \epsilon_2 (1 - 2(p_1 + p_2) + 2p_1 p_2))$
HERK
Key0
FANOUT 1
PO
...
Highest expected P_{err} wire. Insert HERK.

Figure 5.15: Sample HERK insertion into probabilistic circuit.

In probabilistic circuits, high error wires often exhibit error rates that exceed 50% for specific input patterns [50]. This can be observed in the initial work proposing StatSAT as well, where the worst-case BER among only outputs (not all wires) exceeds 50% in 20 of 23 examined circuits [51, 113]. HERKs inserted at these locations exhibit extremely high error rates. This is by design as HERKs aim to hide their key value under probabilistic noise on a high error rate wire. Thus, in any oracle circuit, the actual functionality of the HERK is extremely hard to infer.

This obfuscates not only the key value (correct function) of the HERK gate, but also the key value (correct function) for any locking structures that rely/influence HERK functionality. This property of HERKs is shown to ensure StatSAT resilience.

5.3.2 Evaluation of High Error Rate Keys (HERK)

We now assess the security guarantees and design ramifications of HERKs. Specifically, we consider 2 prevalent attack families against logic locking (StatSAT and removal) alongside a discussion on design overhead and implementation. Our goal is to show how HERKs can leverage high-error locations that naturally occur in probabilistic circuits, the feature differentiating HERKs from prior art, to secure probabilistic IP.

5.3.2.1 StatSAT Resilience

The StatSAT attack is a security threat for locking in probabilistic circuits [51, 113]. An overview of the StatSAT attack is in Sec. 3.6. We propose HERKs to counter this threat. To show this, remember that StatSAT assigns a "don't-care" state for any primary output (PO) whose BER exceeds a user-specified BER threshold for a considered DI. This eliminates the possibility of using a wrong value for the PO, which would exclude the correct key from consideration. However, an unspecified PO also cannot be used to eliminate wrong keys either. HERKs exploit this feature of StatSAT by inserting key gates at high error locations for a set of inputs. Whenever any of these high error inputs are selected as a DI during the

StatSAT attack, the HERK gate has high output error. This error propagates to one or more POs. Remember, the error rate of both HERKs and POs commonly exceed 50% for these high error inputs [50]. Thus, these inputs likely result in the BER threshold being exceeded and the PO being specified as “don’t-care”. StatSAT handles this in 3 ways that each degrade attack performance.

1. **StatSAT Forks:** High error inputs to HERKs leads to unspecified POs. An unspecified PO cannot be used to eliminate keys. Hence, StatSAT cannot eliminate keys for the HERK (or any locking dependent on the HERK). If few (or no) keys can be eliminated, the SAT solver will return the same DI for the next iteration. This causes StatSAT to fork. Thus, an exponential number of SAT instances in the number of POs affected by HERKs are required to unlock the circuit. By inserting each HERK to affect at least 1 unique PO, the SAT instance count will scale exponentially in the number of HERKs.

2. **StatSAT Cannot Fork:** If the StatSAT instance limit (N_{inst}) is reached, further forking is prohibited. StatSAT force proceeds by guessing the most likely PO value. If wrong, the correct key is eliminated from consideration.

3. **Low Quality CNF Clauses:** Defined POs can still be used to eliminate wrong keys. However, because HERK-related POs are unspecified, the resulting CNF cannot be used to infer the key value for the HERK (or any HERK-influenced logic). This reduces the number of keys that can be eliminated for a DI, requiring that more DIs be found (i.e. more SAT iterations) to infer HERK-dependent keys and unlock the circuit.

Each possible way that StatSAT handles HERKs for high error inputs causes performance degradation. In the worst case, the correct key is excluded from the key space, or the number of SAT instances to unlock the circuit is exponential in the number of HERKs. In the best case, many more SAT iterations are needed. As a result, we experimentally show that StatSAT runtime scales exponentially in the number of HERKs.

5.3.2.2 HERK Attack Resilience

We consider 4 ways that an attacker could attempt to tune StatSAT (or the locked circuit) to mitigate HERKs. For each case, either 1) the IC designer can make it impossible for such an attack to succeed, or 2) the attack is severely limited in practicality.

1. **Raise BER Threshold**: The BER threshold can be increased such that the BER at any PO will not exceed it. Thus, high error inputs do not produce an unspecified PO. However, a high BER threshold risks incorrectly guessing a PO's value, excluding the correct key. This is why the PSAT attack [56] often fails to find the correct key.

2. **Ignore HERKs (i.e. Removal)**: One could assume because HERKs correspond to high error (nearly arbitrary) points in the circuit that simply ignoring them should not have much effect. However, error is input dependent [50]. Nodes may be high error for some inputs and critical for others. HERKs

greatly impact functionality for low-error input patterns, which are likely critical, thus they cannot be removed. We consider removal in detail in Sec. 5.3.2.4.

3. **Increase Allowable SAT Instances**: To avoid excluding the correct key due to SAT instance limits, one could greatly increase this limit. However, the required number of SAT instances is exponential in the number of HERK gates. Such an approach quickly becomes infeasible.

4. **Reduce Allowable SAT Instances**: To avoid an exponential number of SAT instances, the instance limit could be kept low. However, once the limit is reached, StatSAT "*force proceeds*" by guessing poorly defined PO values. If a wrong value is assumed, the correct key is eliminated.

5.3.2.3 Experimental Analysis

Now, we experimentally evaluate the StatSAT resilience of HERKs. We implemented HERKs alongside conventional locking (SFLL/SLL) in each of the benchmarks used to evaluate StatSAT in [51]. The characteristics of each benchmark circuit is summarized in Table 5.3. To implement HERKs within these StatSAT evaluated benchmarks, we selected 1000 random input patterns and used Boolean Difference Calculus [50] to estimate the error on each wire in the circuit. A HERK (XOR gate) was then inserted, driven by the wire containing the highest average error and an added key input. A separate set of 1000 inputs were used for each HERK insertion and no 2 HERKs were inserted at the same location. Additionally, we considered only the lowest gate error (ϵ_g) version of each circuit. This corresponds to the lowest PO BER, hence, it is the hardest for HERKs to secure. HERK security at a smaller gate error rate implies security for higher error rates in a given circuit. Because we use Boolean Difference Calculus to estimate BERs, HERK insertion can occur prior to introducing probabilistic behavior to the circuit. In fact, as long as there is a fixed relationship between the error in each gate, the optimal HERK insertion point at a lower gate error will always correspond to the optimal HERK insertion point at a higher gate error rate. Thus, with a fixed gate error distribution for all gates, HERK insertion can be done without knowledge of the exact gate error rate in a design. We recommend such an approach to enable the designer to scale probabilistic behavior such that HERKs do not impact design error specifications.

Circuit	Obfuscation		Key	Tot.	Locking		Gate Error	Comp-
Name	Scheme	PIs	Inputs	Gates	Gates	POs	Rate (%)	lexity
c3540	SFLL-HD [103]	50	16	2434	736	22	1.25	Low
c7552	SFLL-HD [103]	207	16	4826	1163	108	2.0	Med.
seq	SFLL-HD [103]	41	16	5326	260	35	6.0	Med.
ex1010	SLL [64]	10	253	4231	732	10	0.4	Med.
b14	SFLL-HD [103]	277	16	11156	1307	299	0.5	High
b15	SFLL-HD [103]	485	16	15410	1312	519	0.2	High

Table 5.3: Locked benchmark circuit characteristics.

We launched the StatSAT attack against each benchmark containing 1 to 4 HERKs. The uncertainty and BER thresholds (U_λ and E_λ) for StatSAT was set to the highest possible threshold capable of recovering the correct key for the baseline

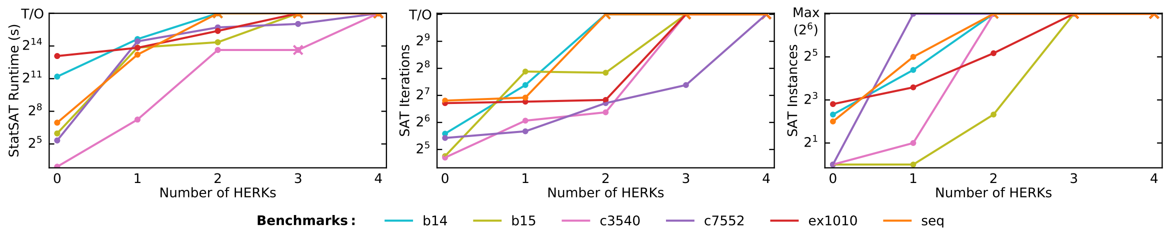


Figure 5.16: Runtime, SAT iterations, and SAT instances required by StatSAT to unlock each HERK-secured benchmark circuit. An ‘x’ for any data point indicates that StatSAT was not able to locate a functionally correct key.

(i.e. 0 HERK) circuit. This ensures that increasing the BER threshold cannot be used to bypass HERK-based locking as the correct key would not be recovered. For a given run, if StatSAT did not recover a correct key with its uncertainty and BER threshold, we lowered each in 1% increments and relaunched the attack. This continued until either a correct key was found, the SAT instance limit was reached, or a 30 hour timeout was reached. The SAT instance limit (N_{inst}) was set to 128 to avoid unbounded exponentiation.

The resulting StatSAT runtime, SAT iterations, and SAT instances required to unlock each benchmark are in Fig. 5.16. An ‘x’ for any data point indicates that StatSAT did not find a correct key. We make 3 observations from these results.

1. No correct key could be located in 30 hours when 4 HERKs were applied, regardless of the considered benchmark circuit. This indicates that strong StatSAT resilience can be achieved with only a few HERKs.

2. The 3 mechanisms by which StatSAT responds to a HERK-induced unspecified PO can be readily observed for each benchmark. 1) The number of SAT instances increases exponentially in the number of HERKs. 2) Once the SAT instance limit is reached, StatSAT cannot locate the correct key (due to wrong PO values being assumed for suppressed forks). 3) The SAT iterations needed to locate the key increases with the number of HERKs.

3. StatSAT runtime increases exponentially in the number of HERKs. This is the key takeaway. It indicates that even a small number of HERKs causes an infeasible StatSAT runtime. Hence, our experimental results indicate that HERKs provide strong StatSAT resilience.

5.3.2.4 Removal Resistance

A removal attack is a common structural attack where the adversary locates and removes locking structures from a circuit to recover a functional chip [100, 95, 47]. HERKs resist such an attack. Consider an attacker who finds and removes any inserted HERKs in a circuit. This attacker hopes to exploit the fact that HERK insertion points have a high error rate for a set of inputs, thus their removal minimally impacts functionality.

This assumption ignores the input-dependence of error rate. While HERK insertion points do exhibit high error for some set of inputs, this does not extend to all inputs. Consider an AND gate fed by independent, error-prone inputs with a 1% error probability. In this case, the input '00' requires 2 simultaneous input errors to produce errant output, a probability of 0.01%. Conversely, input '11' requires only a single error on either input for errant output, nearly a 2% probability. Therefore, while HERKs exhibit high error (or nearly arbitrary output) for certain inputs, their error can be quite small for other input patterns that sensitize the circuit differently.

If a HERK is removed from the design, it has a small impact on function for high error inputs, but a large impact on function for low error inputs. Probabilistic circuits are designed to ensure that input patterns critical to IC performance exhibit minimal error [49, 29, 1]. Thus, low-error inputs impacted by HERK removal are likely function-critical, rendering the IC unusable. This thwarts removal-type attacks against HERKs.

5.3.2.5 Overhead Analysis

HERK implementation requires that a single XOR gate be inserted per key bit. Each key bit leads to an exponential increase in StatSAT runtime, hence, only a few HERKs are needed to ensure strong security. Each considered benchmark had $1,000$-$10,000$ gates. Thus, the area, delay, and power overhead caused by inserting a small number of HERK gates is negligible because it is amortized into a much larger benchmark circuit. Compare this to alternate locking schemes that use large locking structures, such as "almost" one-to-one switch-boxes for Full-Lock/Interlock [33, 34], block-ciphers for LoPher [71], or Hamming Distance units and tamper-proof LUTs for SFLL-style schemes [103, 75, 74]. Because of these large added locking structures, alternate schemes require overheads constituting sizable percentages of the base circuit's parameters. Thus, HERK overhead is negligible compared to prior art, an important advantage of HERKs.

5.3.2.6 Implementing HERKs Alongside Prior Art

To evaluate HERKs, we considered them in tandem with other locking. This was intentional. We do not propose HERKs as stand-alone locking, but rather as an extension to traditional locking schemes aimed specifically at StatSAT protection for probabilistic IP. As previously shown, probabilistic design IP faces a security threat from StatSAT-style attacks. HERKs serve a vital purpose in thwarting these attacks.

By using HERKs in a compound fashion, locked circuits exhibit the best security guarantees of both approaches. Prior art, such as SFLL [103, 75, 74], Full-Lock [33, 34], or others [111, 76], ensures tunable and provable security against a variety of attacks, while HERKs uniquely apply the properties of probabilistic circuits for StatSAT resilience. Because HERKs do not alter or interact with traditional locking structures, the security guarantees of both techniques are maintained. For this reason, we consider HERKs to be a logic locking extension for probabilistic circuits, similar to trace logic locking [111] for deterministic circuits, rather than a stand-alone scheme.

5.3.3 Conclusion

This section proposes High Error Rate Keys, a logic locking technique that can be combined with traditional locking to resist both StatSAT and other prominent attacks. HERKs leverage high error points in probabilistic circuits to hide the correct key value (i.e. correct function) behind probabilistic noise. This causes StatSAT runtime to scale exponentially in the number of HERKs. By adding a sufficient number of HERKs to the circuit, the runtime of a SAT attack can be sufficiently scaled to achieve security, despite the presence of obfuscation with a high average wrong key error rate. Hence, by adding HERKs alongside high-error locking, both error severity and SAT attack resilience can be simultaneously achieved in probabilistic circuits despite the trade-off identified in Chapter 4.

Chapter 6: Design Methodologies for Security Beyond Gate-Level Boundaries

As demonstrated in Chapter 4, conventional gate-level approaches to logic obfuscation applied using gate-level criteria are unable to secure an IC when viewed beyond these boundaries (i.e. at the architecture, application, or system level). This is due to the inherent trade-off between the average error rate and the SAT attack resilience of conventional obfuscation techniques. To address this, we presented 3 non-conventional obfuscation techniques in Chapter 5 that were capable of favorably tweaking this trade-off to meet security goals beyond gate-level boundaries in particular applications. In this chapter, we take an orthogonal approach. Rather than modifying logic obfuscation techniques, we instead look to architectural design modifications to achieve security goals with conventional, gate-level obfuscation schemes. To do so, we first explore the impact of architectural design decisions on hardware security. As a result of this exploration, we propose 2 security-aware architecture design methodologies capable of designing ICs with strong error severity *and* SAT attack resilience simultaneously.

For our first approach, we explore the possibility of security-aware architecture modifications to enhance the hardware-oriented security of logic obfuscation. To this end, we direct our attention to the most commonly proposed candidate modules for logic obfuscation: 1) the on-chip memory, such as cache controllers [13] or SRAM memory [114], and 2) the data path, such as ALUs [45, 46] or alternative compute units [73, 58]. Within these candidate locations, we identify the factors limiting the effectiveness of logic obfuscation. Based on these limiting factors, we identify design decisions that can be made to amplify the hardware security of on-chip memory and processor data path modules. Finally, we propose and evaluate a quantitative, tool-driven design approach for both on-chip memory and data path architectures to achieve strong security guarantees that transcend gate-level boundaries while using prior logic locking schemes.

For our second approach, we target the resource binding phase of the high-level synthesis (HLS) process to enable the design of obfuscated architectures with strong hardware-oriented security guarantees. For this approach, we leverage the architectural context available during resource binding to co-design architectures and locking configurations with high corruption (error severity) *and* SAT resilience simultaneously. To do so, we develop 2 security-focused binding/locking algorithms and apply them to bind/lock 11 MediaBench benchmarks. The resulting circuits showed a 26x and 99x increase in the application errors (error severity) of a fixed locking configuration while maintaining SAT resilience and incurring minimal overhead compared to other binding schemes. Each locking scheme applied post-binding was unable to achieve a high application error rate (error severity) and SAT resilience simultaneously.

6.1 Factors Limiting Security

To mitigate the identified security risks through security-aware architecture design, we first must understand the underlying architecture and application level factors limiting the effectiveness of locking from our design space exploration in Section 4.4. Based on the experiments presented in Section 4.4, we identified 2 primary factors limiting the architectural effectiveness of locking, namely 1) module input space non-uniformity and 2) processor error resilience. We discuss each in turn.

6.1.1 Input Space Non-Uniformity

Within a processor, inputs to each module are generally heavily skewed towards a small subset of the input-space. This is due to the tendency of processors to repeatedly access the same set of data and resources, a heavily studied phenomenon referred to as the *principle of locality*. Additionally, because data and resource utilization is dictated by the application being run on a processor, the input-space of each module is not only skewed, but also application-specific. This means that locking applied independently of IC architecture/application (as is generally proposed) cannot account for the architecture/application-dependent input space of a module. Finally, as we have noted, the portion of the input space that is corrupted by logic locking must be limited to ensure SAT resilience [111, 45, 107, 109]. Therefore, it is unlikely that any of the tiny set of application-agnostic inputs corrupted by logic locking will ever actually occur within the skewed, application-dependent set of inputs applied to the locked module. This makes the likelihood of locking induced errors negligible, greatly limiting the efficacy of locking configured independently of IC architecture.

To empirically support the above claim, we have used ObfusGEM to characterize the input-space of an adder within our x86 core running 9 benchmarks from the PARSEC benchmark suite. Several observations from this experiment are below.

1. Workloads used $10^{-34}\%$ to $10^{-31}\%$ of the input space.

2. A histogram of input utilization for the Blackscholes benchmark is in Figure 6.1. This input-space is skewed between ± 4096, with $> 95\%$ of inputs in this range.

3. Only $\sim$13,000 inputs are shared between each workload.

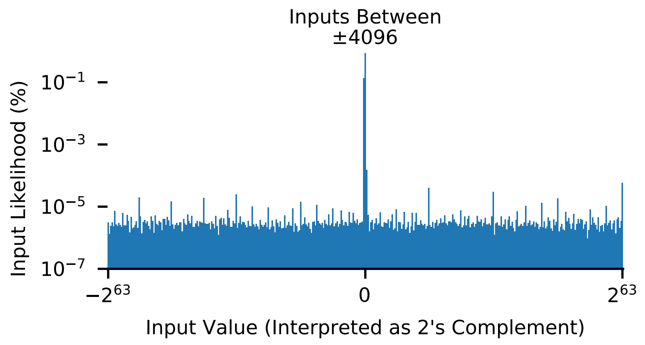


Figure 6.1: X86 core adder input utilization for Blackscholes.

These results support the claim that a small/skewed subset of a module's input space is used by the application. Additionally, the relatively small input overlap between benchmarks indicates that a module's input-space is quite application-dependent as well. Finally, we reiterate that to achieve SAT resilience, the number input minterms corrupted by locking must be severely limited. By combining these results with the $10^{-34}\%$ to $10^{-31}\%$ input space utilization identified empirically, we confirm that the probability of a corrupted minterm actually being applied to a locked module is indeed nearly negligible. Therefore, effective locking must account for both IC architecture and the applications being run on an IC to achieve the error severity necessary for security.

6.1.2 Processor Error Resilience

Substantial computer architecture research has shown that many ICs, especially processors, mask an overwhelming majority of module level errors when viewed architecturally [52, 72]. For example, the work in [72] showed that over 97% of random, radiation-induced soft errors vanished within a tested IBM POWER6 core. On top of the architectural error resilience of ICs, most common applications have been shown to be error resilient as well [42, 25]. For example, common media and AI benchmarks mask as much as 46% of module level errors injected when considering application output [42]. This means that even when logic locking induced errors occur, there still exists a sizable probability that this error will be simply masked and rendered architecturally irrelevant. Therefore, in addition to ensuring module level error is injected, effective locking must also ensure that injected error will derail IC functionality.

6.2 Security-Aware Architecture Design

In this section, we look for ways to mitigate the factors limiting security identified in Section 6.1. Given that these identified factors exist outside of the locked module itself, beyond gate-level boundaries, we look to an IC's architecture to overcome the identified limitations. We explore the possibility of so-called *security-aware* architecture design to improve the security of logic obfuscation, regardless of technique. Fundamentally, we are suggesting that architecture design decisions should consider not only traditional parameters (i.e. area, delay, power, performance, etc.), but also hardware security. To this end, we propose and evaluate security-aware architecture design, a tool-driven approach, based upon the ObfusGEM simulator, to identify and evaluate minor architecture design modifications capable of improving the impact of locking in the IC as a whole. In particular, we look to the most effective locking candidates identified in Figure 4.5, namely the on-chip memory and data path, to improve hardware security. For these components, we quantify the goals of a design approach that favors hardware security in these components. Then, we apply this design approach to redesign candidate locking locations in each IC and evaluate the efficacy of the modified design using ObfusGEM.

6.2.1 Design Methodology

To both demonstrate and evaluate a security-aware architecture design approach, we implement it in our x86 and ARM A53 processor ICs. To this end, we proceed as follows.

1. **Identify Factors Limiting Logic Locking:** Using the cycle-accurate, architectural data provided by ObfusGEM, we identify the factors limiting architectural security. This is highlighted in Section 6.1 for our tested ICs.

2. **Identify Candidate Design Modifications:** Minor architectural design modifications for the on-chip memory and data path that are capable of mitigating any limiting factors must be identified. We perform this in Section 6.2.1.1 for our x86 and ARM A53 core.

3. **Implement and Evaluate Security-Aware Changes:** Using the quantitative, architectural lens provided by ObfusGEM, the efficacy of each identified change must be quantified and tuned, ensuring that sufficient security guarantees are achieved beyond gate-level boundaries in the IC as a whole. We perform this in Section 6.2.2 for our tested ICs.

Fundamentally, this approach relies on the quantitative lens provided by the ObfusGEM simulator to both identify and apply these security-driven modifications. Throughout the remainder of this work, we demonstrate how a security-focused approach to on-chip memory and data path design can exponentially improve security. This enables our previously insecure ICs to achieve strong security guarantees that transcend gate-level boundaries.

6.2.1.1 Identifying Candidate Design Modifications

As shown in Section 6.1, the limitations of locking can be partially attributed to 1) input space non-uniformity and 2) processor error resilience. This means that any design decision which 1) increases the number of uses of (i.e. utilization) or unique input minterms applied to (i.e. diversity) a locked module or 2) amplifies the impact of locking induced errors at the application level will enhance security. We continue by identifying a series of architectural changes which achieve either of these security-focused design goals.

6.2.1.2 Increasing Locked Module Input Utilization/Diversity:

When input utilization/diversity is increased, a larger percentage of a locked module's input space is used. Increased input space utilization increases the likelihood that a locked minterm will be applied to the module, increasing the likelihood of a locking induced fault injection. Many architectural decisions can achieve this:

- For cache controller locking, increasing cache associativity increases both the length and diversity of cache tags (increases input diversity of locked module).

- For memory controller locking, utilizing a write through (rather than write back) cache will increase write frequency, increasing memory controller use (increase utilization). However, we note that this change has a large number of side effects for an IC, likely making it unreasonable in practice.

- For functional unit (FU) locking, smart scheduler design can either favor locked FUs or ensure that corrupted I/O pairs are likely to be scheduled to locked FUs (increase utilization).
- The number of FUs (i.e. adder, FPU, etc.) can be increased and locked with different locking configurations corrupting different inputs (increases diversity of locked inputs).

6.2.1.3 Amplifying the Impact of Locking:

By amplifying the impact of locking induced errors, locking is more likely to overcome architecture/application error resilience. Many design choices can achieve this, for example:

- Locking to ensure that unrecoverable faults are induced for incorrect keys. Examples include locking the FPU to throw a *divide by 0* exception (fatal error), or locking the branch predictor to force a branch to NULL (fatal security exception). Therefore, when a wrong key is applied, any logic locking induced fault injection will cause an error with a critical application impact.
- Locking to ensure fault propagation. For example, locking multiple cache controllers so that locking induced errors in low level caches trigger block write-back to high level caches/main memory. Increasing error propagation throughout memory reduces the odds of error masking.

6.2.2 Evaluating Security-Aware Design

Now that we have identified a series of security-aware design modifications for both on-chip memory and data path components, we continue by implementing and evaluating these design decisions. To this end, we leveraged ObfusGEM to design and evaluate security-aware modifications to our x86 and ARM A53 cores. As shown in Section 4.4, cutting edge locking was unable to protect either architecture. Therefore, for success, we must implement minor architectural design decisions within the on-chip memory and data path that sufficiently improve the application level security of incorporated logic locking art so that both error severity and attack resilience can be achieved simultaneously. For this section, we targeted several on-chip memory and data path modules and developed security-aware architectures capable of amplifying hardware security for each. The evaluation of each proposed design proceeded as follows.

1. The location under test was locked with configurations identical to those in Section 4.4.1 (i.e. the same randomly selected input cube of varying length was locked). Therefore, for this experiment, only the architecture of the IC was modified compared to Section 4.4.1.

2. Security-aware architecture modifications were applied.

3. ObfusGEM compared the application level effects of locking within the modified and un-modified processor.

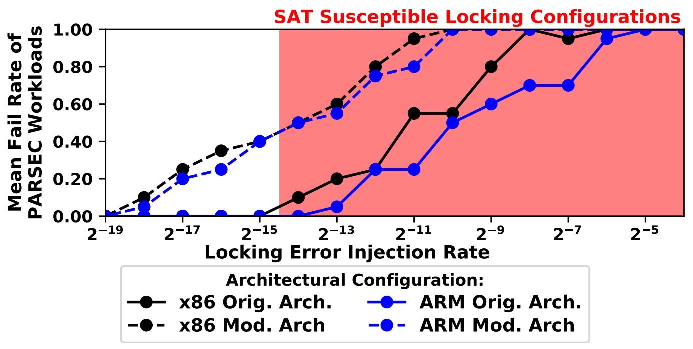


Figure 6.2: The effect of on-chip memory hierarchy redesign on the security of L1 D-cache controller locking.

6.2.2.1 Experiment 1: Security-Aware On-Chip Memory Design

To evaluate the effectiveness of a security-aware design approach for the on-chip memory of a processor, we chose to redesign the L1 D-cache. To this end, we selected the locking configuration implemented within the L1 cache controller of each processor in Section 4.4 as our candidate locking configuration. We then redesigned the L1 D-cache of the IC to amplify hardware security.

Specifically, we made 2 design decisions simultaneously. 1) The associativity of the cache was increased. This increases the cache tag length and the number of unique inputs applied to the cache controller's tag logic (input diversity). 2) The locking was designed to map locked minterms to fatal errors at the output of the cache controller. Both the x86 and ARM core were initially designed with 2-way set associative L1 D-caches. For this experiment, we increased the associativity of this cache to 8-way set associative. Additionally, the locking configuration was modified to produce an invalid tag whenever a locked input was applied to the cache controller. Other than these changes, all other aspects of the on-chip memory and locking configuration were fixed. ObfusGEM results for this experiment are in Figure 6.2.

A similar approach can be taken to enhance locking within higher level caches as well. To demonstrate this, we narrowed our focus to solely the ARM A53 processor testbed. The cache and DRAM control logic within this processor is more complex than the selected x86 core, thereby allowing more design modifications without significant redesign. For this experiment, we attempted to redesign the on-chip memory hierarchy to amplify both L2 cache controller locking and DRAM controller locking. To do so, we enabled hardware pre-fetching within both the L1 D-cache and L2 cache. By enabling pre-fetching, both the diversity and number of minterms applied to both the cache controller and DRAM controller can be increased. We also enabled speculative execution within the core, enabling the processor to execute instructions based on conditional branch predictions. Once again, this modification both increases the diversity and amount of traffic occurring within the on-chip memory system. We have aggregated ObfusGEM simulation results quantifying the hardware security achieved by locking both the L2 cache controller and the DRAM controller in our ARM A53 core with both pre-fetching and speculative execution enabled in Figure 6.3.

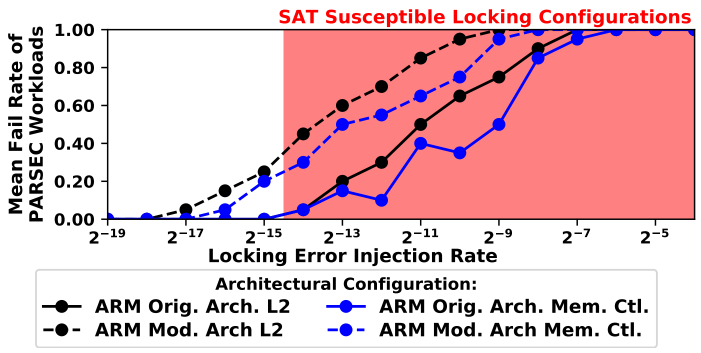


Figure 6.3: The effect of on-chip memory hierarchy redesign on the hardware security of L2 cache controller and DRAM controller logic locking.

6.2.2.2 Experiment 2: Security-Aware Data Path Design

To evaluate the effectiveness of a security-aware design approach for the data path of a processor, we chose to redesign the floating point unit. To this end, we selected the floating point adder locking configuration from Section 4.4 as our candidate locking configuration. We then redesigned the floating point unit of the IC to amplify the achievable hardware security.

To improve floating point adder locking, we explored 2 architectural approaches. 1) We increased the number of floating point adder functional units (FUs) within the core. Each FU was then independently locked for a separate, randomly chosen input cube. 2) We implemented a *smart scheduler*, a redesigned version of each processor's out-of-order scheduler which favors locked FUs (only if that FU was available) for any operation on locked input minterms. For both the x86 and ARM processor, only a single FPU adder was included within the design. Therefore, for

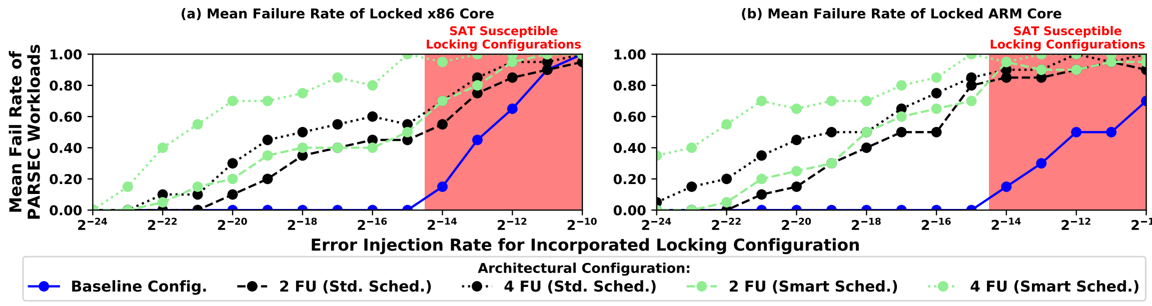


Figure 6.4: The effect of modified FU count and scheduler redesign on the application level security of FPU adder locking.

our evaluation, we increased the number of floating point adders to both 2 and 4 for both cores. ObfusGEM results for both designs (alongside the baseline from Section 4.4) are in Figure 6.4.

A similar approach can be applied to other data path modules as well. To demonstrate this, we performed the same experiment on the second best data path locking configuration, the integer adder. To amplify hardware security in this module, we increased the number of integer adders in each core from 2 to 4. We then locked each of these additional adder circuits independently for a randomly chosen input cube. The ObfusGEM simulation results quantifying the impact of this design change on the hardware security achieved by integer adder locking is included in Figure 6.5.

6.2.2.3 Experimental Design Overhead

By their very nature, the architectural changes we have implemented will impact the design parameters (i.e. area, delay, power, performance) of each device. While this is not ideal, we note that strong hardware security guarantees are crucial

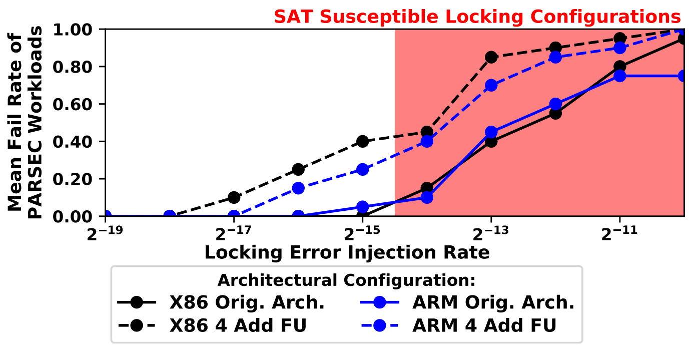


Figure 6.5: The effect of modified FU count on the application level security of integer adder locking.

to ensuring both the IP and integrity of a device. Therefore, we argue that strong hardware security guarantees are a necessary component of IC design. We have aggregated the design overhead of the proposed architectural design modifications in Tables 6.1 and 6.2. Runtime was modeled with ObfusGEM through the runtime of PARSEC benchmarks. Processor power and area were estimated based on GEM5 data using the McPAT modeling framework [41] with a 32nm technology library. Note that the estimated overhead of logic locking each module with SFLL-Fault is included within the design overhead as well. To do so, we added the average area and power overhead of SFLL-Fault from [75] within the McPAT model of any locked design component. For each experiment, we fixed the clock rate for each architecture because scaling clock frequency would generally be considered a severe design modification. For this reason, no degradation was seen in the clock rate of any design.

	X86 Core	ARM Core		
	L1	L1	L2	DRAM
	D-Cache	D-Cache	Cache Cont.	Cont.
Area	9.1%	5.5%	4.1%	4.3%
Peak Power	2.2%	1.2%	3.3%	3.4%
Runtime	-1.2%	-0.1%	0.7%	0.7%
Clock Rate	0.0%	0.0%	0.0%	0.0%

Table 6.1: Design overhead for x86 and ARM core redesigned with a security-aware on-chip memory architecture. Note that these numbers include locking overhead in addition to the overhead of any architectural redesign.

6.2.2.4 Analysis of Security-Aware Designs

As seen in Figures 6.2-6.5, each proposed security-aware redesign yielded locking configurations that reliably derailed processor functionality (achieved error severity) with exponentially smaller error injection rates. The trade-off identified in [103] proves that a linear decrease in error injection rate yields a linear increase in the SAT attack resilience of SFLL-Fault. This means that locking can obtain exponentially stronger SAT attack resilience while still maintaining equivalent error severity in these modified architectures. Because a secure locking configuration must achieve both error severity and attack resilience simultaneously, this constitutes an exponential improvement in the security of logic locking. In fact, through the security-aware design of both the on-chip memory and data path of the IC, logic locking critically impacted PARSEC benchmarks (achieved error severity) with error

Security-Aware X86 Data Path Redesign Overhead					
	Int. Adder Locking	**FPU Adder Locking**			
		No Smart Sched.		**Smart Sched.**	
	4 FU	**2 FU**	**4 FU**	**2 FU**	**4 FU**
Area	11.7%	12.1%	24.8%	12.1%	24.8%
Peak Power	9.3%	10.3%	20.7%	10.3%	20.7%
Runtime	-0.9%	-8.7%	-10.8%	-8.7%	-10.8%
Clock Rate	0.0%	0.0%	0.0%	0.0%	0.0%

Security-Aware ARM Data Path Redesign Overhead					
	Int. Adder Locking	**FPU Adder Locking**			
		No Smart Sched.		**Smart Sched.**	
	4 FU	**2 FU**	**4 FU**	**2 FU**	**4 FU**
Area	11.5%	12.3%	25.3%	12.3%	25.3%
Peak Power	9.2%	9.6%	20.2%	9.6%	20.2%
Runtime	-0.7%	-2.4%	-2.5%	-2.4%	-2.5%
Clock Rate	0.0%	0.0%	0.0%	0.0%	0.0%

Table 6.2: Design overhead for x86 and ARM core redesigned with a security-aware data path architecture. Note that these numbers include locking overhead in addition to the overhead of any architectural redesign.

injection rates residing outside the red-shaded SAT susceptible region. Therefore, our security-aware design modifications enabled locking to simultaneously achieve both error severity and SAT attack resilience in both cores. In Section 4.4, the same locking configurations were unable to achieve security in each un-modified IC. Hence, each design approach was successful in allowing a designer to achieve hardware security.

Despite this positive result in both cases, there are clear differences between on-chip memory hierarchy and data path redesign for this purpose. First, we note that the strongest hardware security was achieved through our redesign of the FPU adder. In this case, non-zero application failure rates could be observed with error injection rates of 2^{-20} in all cases. However, this increase required a significant increase in the both the area and power consumption of the design, with over a 20% increase in the worst case. While it is possible that some processor designs could absorb this level of design overhead to achieve hardware security, this additional overhead would likely be unfeasible in most cases. A similarly large overhead can be seen in the integer adder redesign as well. Hence, both evaluated data path designs induced substantial overhead.

Comparatively, the on-chip memory redesign yielded slightly smaller security improvements, but with substantially less design overhead. For example, our L1 D-cache redesign allowed L1 cache controller locking to achieve error severity with an error injection rate of 2^{-17}. This error injection rate resides well outside of the SAT susceptible region. However, this design required less than a 10% area overhead in the worst case and only a 1-2% increase in peak power. This level of design overhead is much more manageable.

We found that the design overhead of proposed on-chip memory modifications was much lower than data path modifications due to the more subtle changes available to the IC designer in memory hierarchy design. In our redesign of the data path, we relied on simply increasing the number of functional units within the device, an extremely coarse design modification. However, the on-chip memory hierarchy had a wide array of candidate design modifications, such as associativity, cache size, pre-fetching, or hierarchy organization. Each of these design modifications can be finely tuned to greatly impact the environment a cache or DRAM controller operates in. This makes each of these design changes ideal for a hardware security-aware design approach. For example, by enabling hardware pre-fetching and speculative execution, 2 features already available within the hardware, we were able to increase DRAM controller traffic by 17.1%. This increase in traffic substantially increased the utilization and diversity of minterms applied to the locked module, exponentially improving hardware security, while imposing minimal area/power/performance overhead.

Therefore, it appears that the on-chip memory hierarchy serves as a more viable candidate for logic locking. Unlike the data path, the complexity of the on-chip memory system enabled a diverse array of modifications capable of substantially enhancing hardware security. While we obviously did not explore every aspect of locking within the on-chip memory hierarchy, our results demonstrate the promise of a memory-focused logic locking approach.

6.2.2.5 Summary of Security-Aware Design

To conclude, our on-chip memory and data path redesign in the x86 and ARM A53 core exponentially improved application level hardware security. While the design overhead of the data path redesign approach was substantial in many cases, the on-chip memory redesign exhibited a much more modest increase in design overhead. The success of this approach supports the results of prior research noting the on-chip memory hierarchy as an ideal locking candidate [13, 114]. However, we note that achieving security in either of these components of an IC required some architectural tuning in both testbed processors. This result not only demonstrates the importance of IC architecture for hardware security with logic locking, but also emphasizes the importance of a security-aware approach to architecture design.

Our security-aware approach was made possible by ObfusGEM, which both enabled us to identify the factors limiting logic locking and to design/evaluate changes to mitigate these factors. Therefore, while we have shown that an ObfusGEM-driven, security-aware design approach can achieve strong application level security within custom ICs, we also note that the presented results are just a small slice of the security-aware designs made possible by ObfusGEM. To this end, we have released the ObfusGEM simulator alongside this work to enable others in the research community to identify alternative security-aware design modifications and implementation methodologies capable of achieving the application level security currently missing from cutting edge logic locking approaches.

6.2.3 Conclusion

In Section 6.1, we identified input-space non-uniformity and error resilience as 2 of the factors which limited locking in ICs. We then proposed security-aware architecture design to overcome these limitations. Using ObfusGEM, we performed security-aware architecture design to design the on-chip memory and data path of 2 processor testbeds that were insecure with prior art (Section 4.4). Our proposed security-aware on-chip memory and data path designs were shown to exponentially improve security. In the case of on-chip memory redesign, these exponential improvements incurred only a modest design overhead, serving as a viable approach to allow locking to achieve strong security guarantees capable of transcending gate-level boundaries in these previously insecure devices.

6.3 A Resource Binding Approach to Logic Obfuscation

Finally, we consider an alternate approach to produce obfuscated architectures with hardware-oriented security guarantees [112]. Namely, we aim to use architectural knowledge available during the resource binding phase of high-level synthesis (HLS) to both design and lock an overall IC against an untrusted foundry. As we show, by using security-aware resource binding algorithms, we can achieve both objectives of high application corruption (error severity) *and* SAT resilience (attack resilience) simultaneously. In this section, we develop these security-aware resource binding algorithms and then demonstrate their efficacy in 11 Mediabench benchmark circuits.

6.3.1 Motivational Example: Security-Aware Binding

Given the findings of prior research, outlined in Section 2.3, there is a strong need to think *beyond the module* when obfuscating ICs. If we follow the conventional wisdom of pursuing module-level locking after the gate-level configuration of modules are fixed we get stuck in the dilemma of choosing high corruption versus high SAT resilience. Achieving both objectives is critical. In this section, we show that through "smart" obfuscation-aware resource binding decisions, we can indeed satisfy both of these competing needs.

In the context of design obfuscation, binding impacts security by mapping operations onto functional units (FU). This mapping determines the types of values (input minterms) typically processed on each FU. Because locking corrupts output for specific locked inputs, this greatly impacts the lock-ability of an IC. Let us consider an example to show the utility of a security-aware binding approach.

6.3.1.1 Motivational Example: Overview

Consider the scheduled data flow graph (DFG) in Figure 6.6A. This DFG represents some behavioral code segment where each node is an operation that must be applied to the data and each edge shows the flow of data between operations. The DFG in the figure requires 2 cycles to execute (clk 1 and 2). During the first (second) cycle, 2 operations, OP_A and OP_B (OP_C and OP_D), must be completed. Without the loss of generality, let us assume that each operation in the figure is an add. To implement this DFG in hardware, 2 adder FUs are necessary to execute the 2 concurrent add operations.

Resource binding specifies the mapping of the 4 add operations onto the 2 allocated adder FUs. In Figure 6.6B, we show the 2 possible bindings for the scheduled DFG. For each binding, the green (red) circled operations are mapped to FU 1 (FU 2). Let us assume that a security-oblivious binding algorithm has selected binding 1 for the design. After binding, a designer has decided to lock FU 1 to protect the design. In this case, the best solution would be to lock a large majority of input minterms to ensure the highest corruption at the application level. However, due to the SAT resilience constraint, let us assume we can only lock a single input minterm, randomly selected to be x. If we are aware of the input distribution for each operation, a common assumption for HLS [86, 16], we can determine the expected number of occurrences of arbitrary input minterms for each operation in the DFG during a typical workload. We have aggregated the expected number of times input minterm x and y are applied to each operation during a typical workload at the bottom of Figure 6.6A.

Because we know the expected occurrences of input x for each operation, we can estimate the number of locked inputs applied to our locked adder (FU 1). This is the number of application errors caused by the locking scheme. Notice that FU 1 in binding 1 executes OP_A, which is expected to operate on input x 6 times, and OP_C, which is expected to operate on input x 0 times. This means that the bound/locked circuit is expected to inject $6 + 0 = 6$ errors. We have aggregated the expected occurrences of minterm x and y for each FU-binding combination below Figure 6.6B. Let us consider how security-aware binding could increase the number of error injections.

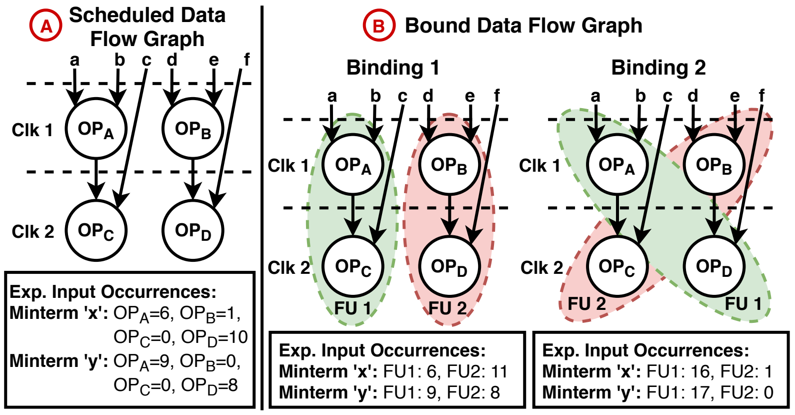

A
Scheduled Data Flow Graph
a b c d e f
Clk 1
OPA
OPB
Clk 2
OPC
OPD
Exp. Input Occurrences:
Minterm 'x': OPA=6, OPB=1, OPC=0, OPD=10
Minterm 'y': OPA=9, OPB=0, OPC=0, OPD=8
B
Bound Data Flow Graph
Binding 1
Binding 2
FU 1
FU 2
Exp. Input Occurrences:
Minterm 'x': FU1: 6, FU2: 11
Minterm 'y': FU1: 9, FU2: 8
Exp. Input Occurrences:
Minterm 'x': FU1: 16, FU2: 1
Minterm 'y': FU1: 17, FU2: 0

Figure 6.6: Sample scheduled DFG and corresponding binding solutions.

6.3.1.2 Example 1: Obfuscation-Aware Binding

Consider the case where the logic locking configuration has been specified prior to resource binding. Following our prior example, this means that FU 1 will lock the input minterm x. However, instead of binding in a security-oblivious fashion, let us instead bind the DFG in Figure 6.6A to maximize the number of times the locked input (x) will be applied to the locked FU (FU 1). In this case, binding 2 would be selected, resulting in $6+10=16$ application errors over a typical workload. Such an approach has 2 key advantages. 1) The number of errors injected by logic locking is more than doubled (16 vs. 6) compared to our security-oblivious binding approach. Because the number of locked inputs is static, this results in a substantial increase in the corruption caused by logic locking *without* compromising SAT resilience. 2) Errors are now injected during both clock cycles of the schedule (clk 1 and 2) instead of only one (clk 1). This opportunity for consecutive error injections increases the likelihood of critically impacting the application. By binding to maximize the application errors, a locking configuration that simultaneously causes substantially more *and* higher quality application errors is produced.

6.3.1.3 Example 2: Binding-Obfuscation Co-Design

In addition to selecting which operations are bound to each FU, let us also decide which input minterms to lock. Previously, we assumed that only input x could be locked. If we simultaneously consider the locked inputs and the binding in order to maximize the number of application errors, we will lock input y in FU 1 for binding 2. Notice that input y does not have either the highest total number of expected occurrences, or the most occurrences for a single operation. However, this locking configuration causes $9+8=17$ application errors during a typical workload. Not only does this 1) increase the number of error injections by over 2x (17 vs. 6) compared to our security-oblivious approach and 2) inject error during both cycles of the schedule, but it also results in more errors than any configuration locking input x could achieve. Thus, a co-design approach can further improve the number and quality of application errors caused by locking.

6.3.1.4 Security-Aware Binding Problem Formulations

In both examples, the architectural context available during resource binding enabled us to create locked circuits causing over 2x more application errors. This allows a designer to decrease the number of locked inputs, while simultaneously increasing the application errors caused by the locking construction. Essentially, *security-aware* binding decisions enable us to achieve higher attack resilience *and* higher corruption simultaneously. Based on each example, we specify a problem formulation that is addressed in the remainder of the work.

1. **Obfuscation-Aware Binding:** For this problem, we assume that modules have already been locked to secure a known set of error-causing locked inputs. Based on this configuration, we map operations onto FUs (bind) to maximize application errors.

2. **Binding-Obfuscation Co-Design:** For this problem, we simultaneously select the binding and the locked input minterms to maximize the application errors caused by locking. Once the binding and locked inputs are selected, any critical minterm locking technique can be used to create the locking construction.

6.3.2 Problem 1: Obfuscation-Aware Binding

The *obfuscation-aware binding* problem assumes that the allocation/scheduling phases of HLS have occurred and a SAT-resilient locking configuration (i.e. one that locks a sufficiently small number of inputs) has been specified for the allocated FUs. The locking specification must include 1) the number of FUs locked, 2) the locking scheme used, and 3) the locked inputs. We also assume that critical minterm locking schemes, such as SFLL-rem [74], have been used so that locked inputs are static between wrong keys. Now, given a list of FUs, a scheduled DFG, and locking details, we must map each operation to an FU such that the application errors caused by the locking construction are maximized. Doing so ensures that IC corruption is maximized while maintaining SAT resilience (because the locking construction was chosen to be SAT resilient a priori). To address this, we have developed an objective cost function to quantify the application errors caused by locking for a fixed binding.

6.3.2.1 Obfuscation-Aware Objective Cost Function

Suppose that we have scheduled and bound a DFG onto FUs, some of which have been locked using critical minterm locking. We aim to quantify the impact of these locked FUs on the error of the DFG. We capture this error by counting the number of times a locked input is evaluated by a locked FU during the DFG's execution. The objective is to maximize these error injecting events through appropriate binding decisions. Let us define matrix K to represent the occurrence of each locked input for each operation. The number of times the locked input m is applied for operation n is $K_{m,n}$. One way to calculate K for a given DFG is to simulate the execution of the DFG for "typical" input traces, or applications. These are commonly assumed to be available during HLS [86, 16]. Given an input trace for the DFG, we can perform time simulation to calculate the number of times a given locked input is applied to each operation.

Based on K, we define an objective cost function to inform binding that quantifies the expected number of application errors for a given locking configuration in a bound DFG. To do so, assume that some set of L FUs have been locked. Each locked FU, $l \in L$, locks a set of inputs M_l and binds a set of operations N_l. The expected number of application errors caused by this locking configuration is:

$$E = \sum_{l \in L} \sum_{m \in M_l} \sum_{n \in N_l} K_{m,n} \tag{6.1}$$

6.3.2.2 Obfuscation-Aware Binding Algorithm

Using the cost function in Eqn. 6.1, we develop a binding algorithm that maps operations to FUs such that the number of application errors (i.e. when locked inputs are applied to locked FUs) is maximized. Consider a scheduled DFG, S, which spans s clock cycles. A set of resources, R, has been allocated to bind the DFG. While we make no assumptions as to the type (e.g. adder, multiplier, etc.) of the resources and operations, we do assume that they are all the same type. Thus, any of the resources in R can execute any operation in the DFG. By handling each operation/resource type separately, this assumption can be made without the loss of generality. Of these R resources, a subset, L, has been locked ($L \subseteq R$). Each $l \in L$ locks a set of critical inputs M_l, which are pre-determined.

During each cycle t ($t \leq s$), a set of concurrent operations $N_t \in S$ are scheduled. Binding requires us to map each operation in N_t to one of the allocated FUs (i.e. $|R| \geq |N_t|$). Consider the first cycle of the DFG, $t = 1$. To bind the operations at $t = 1$ (N_1), we build a weighted bipartite graph, $B_1 = (R \cup N_1, E_1)$. Each vertex $r_i \in R$ is an FU. Each vertex $n_j \in N_1$ is an operation. If r_i can bind n_j (i.e. the FU r_i is available and can run operation n_j), an edge of weight $w_{i,j}$ is added. This should be the case for all r_i-n_j pairs, so a complete bipartite graph is produced. The weight, $w_{i,j}$, is:

$$w_{i,j} = \sum_{m \in M_i} K_{m,j} \tag{6.2}$$

where M_i is the set of locked inputs for FU i and $K_{m,j}$ is the expected occurrences of locked input $m \in M_i$ for operation j. Therefore, $w_{i,j}$ is the number of times locked inputs will be applied to resource i if operation j is bound to it. Note that the edge weights connected to non-locked FUs will be 0. Now, we solve the max weight bipartite matching problem for B_1, which can be solved optimally in P-time. The resulting matching maps (binds) each operation during clock $t = 1$ to an available FU.

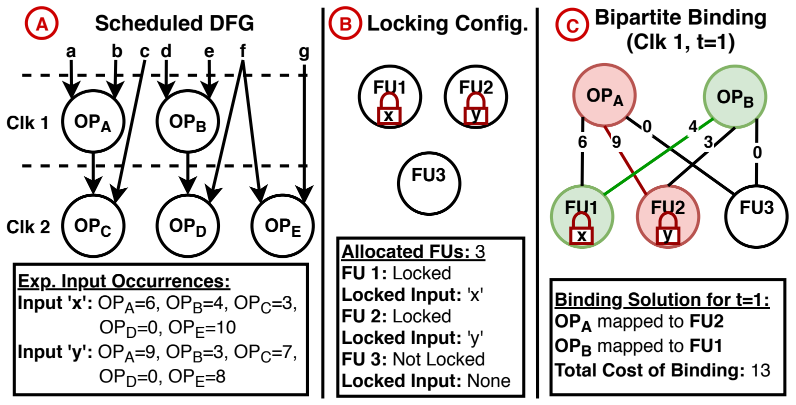


Figure 6.7: Obfuscation-aware binding algorithm for clock 1 (t=1).

To demonstrate this algorithm, consider the DFG in Figure 6.7A, which spans 2 clocks. There are 3 FUs, $R = \{FU1, FU2, FU3\}$, allocated to bind this DFG, shown in Figure 6.7B. Of these FUs, 2 are locked, $L = \{FU1, FU2\}$, with locked inputs $M_{FU1} = \{x\}$ and $M_{FU2} = \{y\}$. For this DFG's typical input trace, the number of times each locked input (x and y) was applied to each operation is below Figure 6.7A. For $t = 1$, the proposed algorithm produces the bipartite graph in Figure 6.7C. A max weight matching of this graph selects the red and green colored edges, mapping OP_A to $FU2$, with edge weight 9, and OP_B to $FU1$, with edge weight 4. FU3 is unused during this clock because only 2 operations are executed. This produces a binding for clock 1 that injects $9 + 4 = 13$ errors for the typical input trace.

The described approach produces a binding for clock $t = 1$. This algorithm must be repeated for the remaining $s - 1$ clocks to produce a complete binding solution. Thus, we must generate and match a bipartite graph, B_t, for the remaining $t = 2..s$ clocks in the schedule. Notice that the considered operations change for each cycle (t), but the FUs in R do not. Also, the bipartite graph for each cycle (B_t) has no dependence on other cycles. Thus, binding decisions made in one cycle do not conflict with another cycle, allowing each clock cycle to be bound independently and in any order (separability).

By matching each set of concurrent operations to allocated resources, we bind each operation to maximize the number of locked inputs applied to locked FUs during the typical input trace/application. This maximizes the application errors caused by the locking configuration for the characteristic workload, as proved in Thm. 6.2.

6.3.2.3 Analysis of Obfuscation-Aware Binding Algorithm

To analyze the presented algorithm, we discuss 3 key properties.

1. *Runtime Complexity:* To bind an arbitrary scheduled DFG with s cycles, the proposed algorithm must generate and match s complete weighted bipartite graphs. Each graph has $|N_t|$ operations (sources) that must be matched to

one of the $|R|$ resources (destinations) with a maximum weight. A minimum weighted full match of an m-source and n-destination bipartite graph can be performed in $O(mnlog(n))$ [35]. By negating each edge weight ($w_{i,j}$) and assuming that $|N_m|$ is the maximum number of concurrent operations in the DFG, obfuscation-aware binding can be completed in $O(s|N_m||R|log(|R|))$. Thus, the algorithm runs in P-time.

2. *Validity and Completeness of Binding Solution:*

   Theorem 6.1. The proposed obfuscation-aware binding algorithm will always result in a valid and complete binding solution, if it exists.

   We omit a detailed proof of this claim for brevity. However, notice that during each clock cycle, bipartite matching ensures a valid matching between operations and FUs. By definition, this means that all operations in all clocks end up being bound to only one FU, with no more than one operation in a cycle being bound to a given FU. This ensures that the final solution is a valid and complete binding.

3. *Optimality of Binding Solution:*

   Theorem 6.2. The obfuscation-aware binding algorithm yields the maximum expected application errors for a locking configuration.

   Proof. To bind a DFG, a bipartite graph must be generated and fully matched for each cycle in the schedule ($t = 1..s$). Each graph has a source node for every operation $n \in N_t$ and a destination node for each resource in R. Every source-destination pair is connected by an edge of weight $w_{i,j}$, which is equal to the number of occurrences of each locked input for FU j during operation i. A bipartite graph defined in this way for cycle t $(1 \leq t \leq s)$ is necessarily independent of the bipartite graph for all other cycles. This implies that the full matching produced for each bipartite graph is independent. Therefore, the binding for each cycle in the schedule is separable.

Now, consider that each edge weight in the bipartite graph is equal to the number of occurrences of each locked input for FU j during operation i (i.e. expected error injections). By definition, a maximum weight full matching of this bipartite graph corresponds to the operation-FU mapping (binding) that causes the most expected error injections. Hence, the full matching for each bipartite graph is optimal for a given cycle in the DFG. Because each bipartite matching produces the maximum error injections for that cycle and the bipartite graph for each cycle is separable, the total binding solution yields the maximum expected error injections for the locking scheme. $\square$

Thus, the algorithm results in a binding with the highest possible application corruption. Remember that the locking scheme specified prior to binding ensured SAT resilience by limiting locked inputs to be sufficiently small in number. Therefore, our obfuscation-aware binding algorithm guarantees a locking scheme with the highest application corruption, while ensuring that SAT resilience is maintained.

6.3.3 Problem 2: Binding-Obfuscation Co-Design

The *binding-obfuscation co-design* problem relaxes our assumption that the identity of locked inputs are specified before binding. Instead, we assume only that the number of locked inputs are specified to ensure a SAT resilient locking solution. These locked inputs are to be chosen from some set of designer-specified candidate locked inputs to optimize application error. To formalize this, we assume that the allocation/scheduling phases of HLS have occurred and a SAT-resilient locking configuration has been specified, including 1) the number of resources locked, 2) the critical minterm locking scheme used, 3) the number of locked inputs, and 4) a list of candidate locked inputs. However, which specific inputs are locked from the candidate list is not known and needs to be chosen. We must map each operation to an FU *and* select locked inputs such that the application errors caused by locking are maximized via the cost function in Eqn. 6.1.

6.3.3.1 Binding-Informed Obfuscation Algorithm

Consider an arbitrary scheduled DFG, S, which spans s clock cycles. A set of R resources have been allocated to bind S. Once again, we assume without the loss of generality that all operations and resources are of the same type (e.g. add). This can be done by handling each set of dissimilar operations separately. Of these R resources, a subset (L) will be locked ($L \subseteq R$). Each $l \in L$ locks a set of inputs, M_l, which must be chosen from a list of candidates.

We assume that this list of candidate locked inputs, denoted C, is designer-specified. This set can be chosen by a variety of methods (e.g. randomly, most commonly occurring inputs in the DFG, etc.). We discuss strategies to choose C in Section 6.3.3.2, however, we largely consider this to be beyond the scope of the work. For each locked FU ($l \in L$), we must select the most commonly occurring candidate locked inputs among the operations bound to it to be in M_l. In this way, when we lock each input in M_l, we produce a locked FU with the maximum application error. While the exact input distribution for each operation varies among workloads, we have applied a "typical" input trace to our DFG to estimate the number of occurrences of each input i for operation j, denoted $K_{i,j}$ (Section 6.3.2.1).

Given a list of candidate locked inputs (C), we must find the binding/locked input specification that produces the most expected application errors for the DFG. Section 6.3.2.2 defines an obfuscation-aware binding algorithm that, given a specified set of locked inputs (M_l), returns the binding with the most expected application errors. If we enumerate all combinations of candidate locked inputs from C of size $|M_l|$ for each locked FU ($l \in L$) and apply this obfuscation-informed binding algorithm to each combination, we generate the optimal binding for each enumerated set of locked inputs. By comparing the total expected application errors for each enumerated set (i.e. the total cost of each binding solution), we can determine the optimal binding/locked input specification for the DFG.

This approach iterates over all locked input combinations, an exponential number. However, many of these combinations are unlikely to yield an optimal solution. For example, consider a set of FUs locking a set of inputs that produce minimal application error. It is unlikely that FUs locking these inputs can be combined into a high error binding solution. However, this algorithm still evaluates them. A good heuristic would focus on locked input combinations causing substantial error for each FU, regardless of how other FUs are locked, to ensure that the total error is very high. This can be achieved by evaluating each FU sequentially. If we assume that the number of candidate locked inputs (C) is upper-bounded by a predefined constant (x), we can define a P-time heuristic:

1. Choose a single FU, $l_i \in L$. Assume all other FUs are unlocked.
2. Enumerate all locked input combinations $\binom{|C|}{|M_{l_i}|}$ for l_i.
3. Apply obfuscation-informed binding on each combination. Use the max cost binding solution to fix the locked inputs for l_i, M_{l_i}.
4. Consider a new locked FU, $l_i \in L$ for which M_{l_i} has not been fixed. With all prior M_l fixed, repeat steps 2-3 to specify M_{l_i}.
5. Repeat step 4 until M_l is chosen for each $l \in L$. Run obfuscation-informed binding once more for the final binding solution.

6.3.3.2 Analysis of Binding-Obfuscation Co-Design

To analyze the proposed algorithms, we discuss 4 key properties.

1. *Candidate Locked Input Selection:* We consider the nuances of candidate locked input selection to be out of scope. However, we briefly note some possible ways to choose the members of C for context. The most obvious

relies on the "typical" input trace to select the most common inputs in the DFG (i.e. the top 'x' most common inputs). However, if the attacker has input distribution knowledge, such an approach could leak candidate locked inputs, making it disadvantageous. In this case, less common inputs, or even a random set can be used. Regardless of the members of C, our approach still maximizes locking-induced application errors. Thus, binding-locking co-design will still increase application error over conventional locking approaches considering the same locked inputs.

2. *Runtime Complexity:* The optimal algorithm iterates over all locked input combinations for each locked FU. Given $|L|$ locked FUs that secure at most $|M|$ inputs from a set of size $|C|$, there are $\binom{|C|}{|M|}^{|L|}$ combinations. This results in a non-polynomial runtime. However, consider the proposed heuristic, which sequentially iterates over locked inputs combinations for one FU at a time. Because $|C|$ is upper-bounded by a predefined constant 'x' for this heuristic, there are $\binom{x}{|M|}$ combinations per FU. This is upper-bounded by $x^{x/2}$. x is a constant, so this upper bound is a constant, allowing it to be discarded from the time complexity. If we use the P-time algorithm from Section 6.3.2.2 to evaluate each locked input combination for $|L|$ FUs, our heuristic runs in $O(s|L||N_m||R|log(|R|))$, a P-time solution.

3. *Optimality of Binding Solution:* The resulting binding/locking will yield the maximum possible application errors, as quantified by the cost function in Eqn. 6.1. To prove this, remember Thm. 6.2, which proves that our binding

algorithm maximizes errors when locked inputs are specified. Because we iterate over all possible locked input combinations in C, the resulting solution must cause the maximum application errors possible for the DFG given the locking parameters. On the other hand, while our P-time heuristic will still yield a locking solution with substantial error, it no longer operates on every locked input combination, so it may not be highest possible error.

4. *Impact on Security:* Consider the impact of co-design on the application errors and SAT resilience of locking. Both our optimal and heuristic algorithm configure binding/locking to optimize locking-induced application errors for a fixed number of locked inputs. Because the locked input count dictates the SAT resilience of locking, our proposed approach produces a design that maximizes corruption for an attacker without any compromise in SAT resilience.

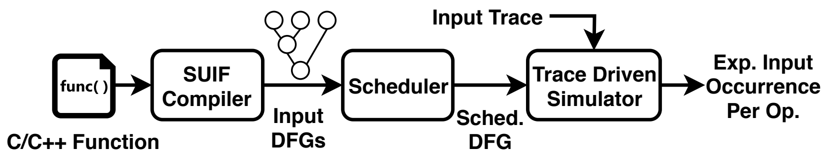


Figure 6.8: Experimental flow to generate benchmarks.

6.3.3.3 Binding-Time Logic Locking Design Methodology

Consider a designer that has set a target application error rate and a minimum SAT runtime permissible for a secure locking configuration in their custom IC. With only minor modifications, the proposed binding-locking co-design approach can be used to design a locking configuration meeting both goals. Essentially, by using our co-design approach to incrementally tune the number of locked inputs in each FU, a locking configuration can be designed that achieves a sufficient application error rate with the minimum number of locked inputs, hence, the maximum SAT resilience. If the SAT resilience of this locking configuration is insufficient, exponential SAT iteration runtime locking schemes can be used alongside the binding-obfuscation co-designed locking to increase SAT runtime to a sufficient level. Exponential SAT iteration runtime schemes generally incur too much design overhead to be used on their own. For example, a 384-bit Full-Lock [33] scheme implemented in the b14 netlist of the ISCAS'85 suite incurred a 192% increase in power and 61% increase in area, while requiring < 10 minutes to unlock with a SAT attack. This overhead is infeasible. However, our co-design approach uses critical minterm locking, which incurs minimal overhead compared to these techniques. So, by using low-overhead critical minterm locking to achieve as much SAT resilience as possible, the design overhead concerns associated with exponential SAT runtime schemes can be minimized, while still meeting design goals.

6.3.4 Experimental Evaluation of Proposed Algorithms

To evaluate each proposed algorithm, we applied them to bind the adders and multipliers in 11 benchmarks. Each benchmark was made by isolating a C function from 1 of 8 MediaBench benchmarks [38] and extracting the corresponding DFG with SUIF. Each DFG was scheduled to be executed on up to 3 FUs using a path-based scheduler [48]. The resulting DFGs contained an average of 18.6 add and 10.6 multiply operations spanning 13.5 cycles. To serve as the “typical” input trace/application for each benchmark, we used the MediaBench-provided sample workloads. For this input trace, each DFG was simulated and expected occurrences of input minterms for internal operations were computed. This information, along with the scheduled DFG, was used as input for each proposed algorithm. An overview of the process to generate each benchmark is in Figure 6.8.

To evaluate each algorithm we compared them to other common binding algorithms with identical locking configurations. Specifically, we used an area-aware approach [30], which minimizes register count, and a power-aware approach [16], which minimizes switching frequency, for comparison. For each benchmark, we enumerated all combinations of {1,2,3} locked FUs locking {1,2,3} inputs each. We then aggregated a list of the 10 most common inputs for each DFG to serve as the candidate locked inputs. For the 9 possible locking configurations (i.e. {1,2,3} locked FUs locking {1,2,3} inputs), we created a bound/locked circuit securing each combination of the 10 candidate inputs for each locked FU. We used 1) obfuscation-aware, 2) binding-obfuscation co-design (optimal and P-time heuristic), 3) area-aware, and 4) power-aware binding algorithms to generate each locked circuit. We then calculated the ratio between the number of application errors caused by each security-aware approach compared to each area/power-aware approach with the same locking configuration. The results were averaged over every locked FU count, locked input count, and locked input combination for Figure 6.9.

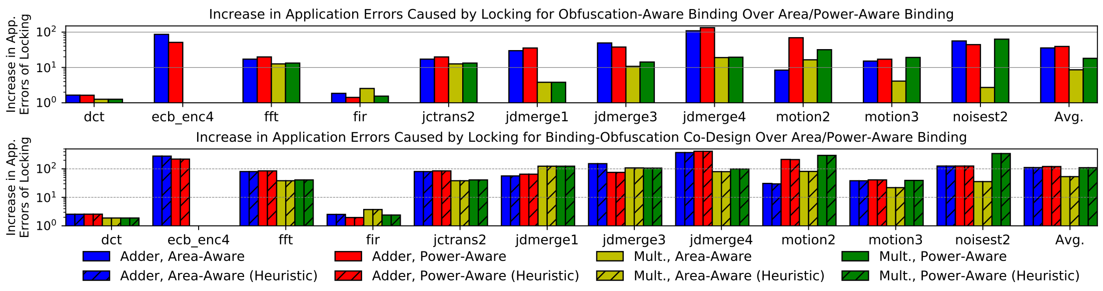


Figure 6.9: Impact of security-aware binding on the application errors caused by locking during a typical workload compared to area-aware [30] and power-aware [16] binding. Adder/multiplier binding were considered separately. No multipliers were present in the ecb_enc4 benchmark.

In this way, we compared each circuit created with a security-aware algorithm for each enumerated locking/locked input configuration to the same circuit incorporating an *identical* locking configuration created with an area/power-aware algorithm. This directly quantifies any increase in application errors due to our security-aware algorithms across a variety of circuits and locking configurations.

6.3.4.1 Experimental Analysis

As shown by Figure 6.9, obfuscation-aware binding increased the application errors caused by the locking construction by 22x and 29x compared to area and power aware binding, respectively. The optimal binding-obfuscation co-design algorithm increased application errors by 82x and 115x. Our P-time heuristic for this algorithm resulted less than a 0.5% solution degradation, again increasing application errors by 82x and 115x. This confirms the efficacy of our heuristic. As

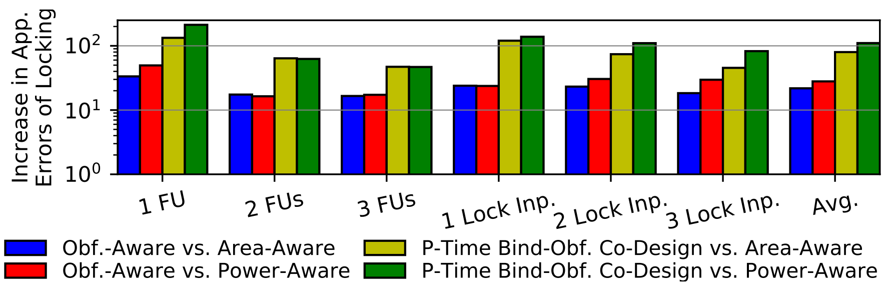


Figure 6.10: Impact of locking configuration on errors caused by security-aware binding. All results are normalized to the errors caused by the same locking configuration applied after area/power-aware binding.

a result, we rely on this P-time heuristic for the remainder of binding-obfuscation co-design analysis. Each algorithm caused sizable increases in application errors, without sacrificing SAT resilience.

We have aggregated the impact of locking configuration on the efficacy of each proposed algorithm in Figure 6.10. To generate Figure 6.10, we fixed a single locking parameter, listed on the x-axis, and averaged our results over all other locking parameters (e.g. the "1 FU" bars average over locking with {1,2,3} locked inputs). In this way, we isolated the impact of each locking parameter on the efficacy of each security-aware algorithm. Based on Figure 6.10, increases in application error remained consistently high in all cases. Remember, all increases were normalized to the application errors caused by area/power-aware binding for the same locking configuration (i.e. locked FU count, locked input count, and locked input identity). Thus, Figure 6.10 suggests that security-aware binding will consistently produce a 10-150x increase in errors, no matter the underlying locking construction.

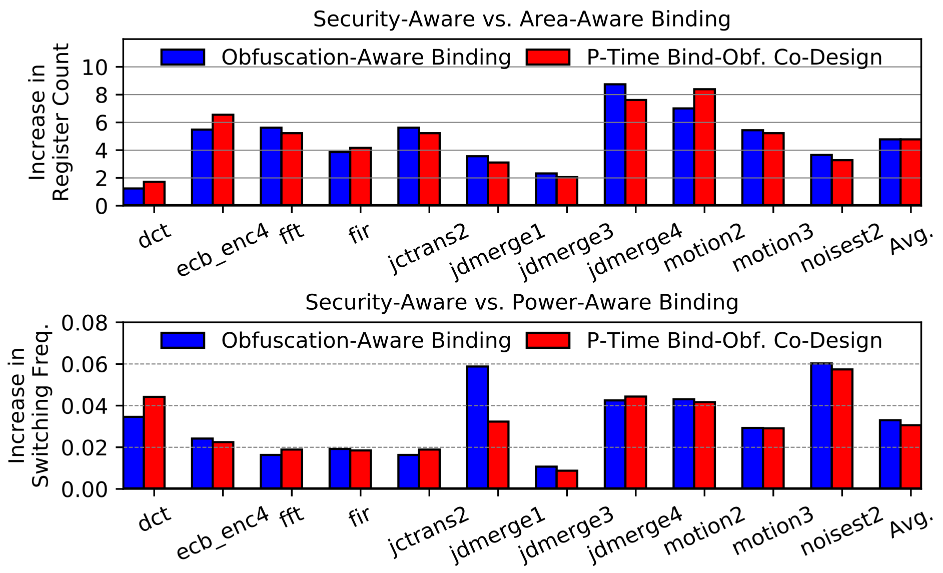


Figure 6.11: Design overhead of proposed security-aware binding algorithms compared to area-aware and power-aware binding algorithms.

Finally, we compared the design overhead of each proposed algorithm to 1) area-aware binding [30], which minimizes register count, and 2) power-aware binding [16], which minimizes switching frequency. We show the corresponding increases incurred by our security-aware algorithms on the register count and the switching rate in Figure 6.11. On average, our proposed algorithms performed similarly, requiring ~4.7 more register count than area-aware binding and incurring a 0.03 higher switching rate than power-aware binding. This confirms the low overhead nature of each security-aware algorithm.

6.3.5 Conclusion

In this section, we explored security-aware binding for HLS with 2 problem formulations, one where locked inputs were chosen prior to binding and one where the binding algorithm could inform locked input selection. To solve them, we developed an objective cost function that quantified the application errors injected by a locking configuration for a fixed binding. We then proposed a security-aware algorithm for both problems to maximize this cost function without degrading SAT resilience. To evaluate each algorithm, we applied them to 11 MediaBench benchmarks and their sample workloads. Our proposed obfuscation-aware binding (binding-obfuscation co-design) algorithm caused a 26x (99x) increase in locking-induced application errors over alternative binding schemes with no reduction in SAT resilience and only minimal degradation in area/power overhead. Thus, each approach can ensure that locking achieves the highest application corruption (error severity) without sacrificing SAT resilience. Doing so allows strong security guarantees to be achieved in ICs that transcend gate-level boundaries using conventional obfuscation techniques.

Chapter 7: Conclusions and Future Research Directions

In this dissertation, we explored the efficacy of logic obfuscation techniques when obfuscated circuits are viewed as a small part of a much larger and more complex IC. This forces us to consider hardware-oriented security beyond gate-level boundaries. In Chapter 4, we derived a fundamental mathematical trade-off between error severity and attack resilience, the 2 primary goals underlying all logic obfuscation. We then designed and utilized the GEM5-based ObfusGEM simulator to assess the ramifications of this trade-off on logic obfuscation through an architectural lens. Our experiments identified severe limitations underlying logic obfuscation techniques and implementation methodologies when considered beyond traditional gate-level boundaries due to the error resilience and input space non-uniformity of modern architectures and applications. In the remaining chapters, we explored 2 possible methods to address these limitations.

In Chapter 5, we explored methodologies to address the limitations identified in the previous chapter through the design of non-conventional obfuscation techniques. First, we proposed Trace Logic Locking, a novel enhancement of module-level logic obfuscation which enables existing art to secure arbitrary length sequences of input minterms, referred to as *traces*. We then proved that this trace-based approach to obfuscation injects an additional degree of freedom into the parametric space of locking, enabling obfuscation techniques to overcome the limitations of our derived trade-off. Second, we proposed Memory Locking, an automatable logic obfuscation technique that obfuscates the delicate timing balance maintained by on-chip SRAM circuitry. By obfuscating the analog effects governing SRAM timing, Memory Locking is resistant to many proposed attack methodologies, such as SAT-based attacks. This bypasses the trade-off between error severity and SAT resilience that was shown to limit the efficacy of obfuscation when viewed in the context of an IC as a whole in Chapter 4, allowing both to be achieved simultaneously. Finally, we proposed High Error Rate Keys to secure probabilistic circuits from SAT-style attacks. High Error Rate Keys concentrate obfuscation in high-error regions of a probabilistic circuit, thereby forcing the runtime of SAT-style attacks to scale exponentially in the number of key bits. Once again, HERKs serve as a non-conventional approach capable of easing the trade-off between error severity and SAT resilience, allowing security guarantees to be achieved in probabilistic applications beyond gate-level boundaries.

In Chapter 6, we explored how architectural context can be used to enhance the hardware-oriented security guarantees of existing obfuscation techniques. First, we explored the notion of security-aware architectural design practices. To this end, we proposed and evaluated a quantitative, tool-driven design approach for both on-chip memory and data path architectures to enhance hardware security guarantees through logic obfuscation. Second, we considered leveraging the architectural context available during the resource binding phase of high-level synthesis (HLS) to co-design architectures and locking configurations capable of high corruption (error severity) *and* SAT resilience simultaneously. To do so, we developed 2 security-focused binding/locking algorithms, which were capable of achieving 26x and 99x increase in the application errors of a fixed locking configuration while maintaining SAT resilience and incurring minimal overhead compared to other binding schemes.

7.1 Future Work

Hardware security is a rapidly expanding field of research. As such, there are a wide array of new research directions that are direct extensions from the issues studied in this dissertation. We outline several possible paths for future work in the remainder of this chapter.

7.1.1 Security-Aware Logic Synthesis

Nearly every hardware system is developed in a high-level language, such as Verilog, and converted into a gate-level netlist via a process known as *logic synthesis*. This provides two benefits: 1) Designers can focus primarily on algorithm development, automating many challenges caused by the scale of modern ICs. 2) Synthesis tools automate a wide array of optimizations to trade off design goals (power, area, timing, etc.) and ensure that all requirements are met. However, most security research obfuscates the gate-level netlists produced following logic synthesis, forgoing synthesis optimizations entirely. The work highlighted in this thesis argues that obfuscation applied in this way is sub-optimal in both security and traditional design goals. For example, Section 6.3 identified a link between obfuscation effectiveness and the resource binding phase of high-level synthesis. Through obfuscation-aware resource binding algorithms developed to target this link, a 98.5x improvement in the effectiveness of obfuscation could be achieved. Such a security-aware approach could be considered for general logic synthesis as well. This offers the possibility of research developing security-aware synthesis algorithms that use the design context and plasticity present during synthesis to optimize for hardware security and ensure that security goals are met. In particular, I foresee two key research thrusts. First, work investigating the impact of a design's topology and function on the hardware security achieved by logic obfuscation, including quantifiable metrics to evaluate these security goals. Second, work developing security-aware synthesis algorithms that optimize circuit topology to ensure that any security metrics are met.

7.1.2 Hardware Security in the Physical Design Realm

Recent research has demonstrated that contactless probing, a common IC test and evaluation technique, can be used to extract hardware secret keys from a circuit even when tamper-proof memory elements are used [63, 62, 90]. This form of physical attack represents a fundamental shift in the hardware security threat model, which previously considered them to be largely out-of-scope. Now, any keys stored in hardware can be compromised with sufficient reverse engineering.

Hence, any security scheme that uses on-chip keys, such as logic obfuscation, is compromised. These physical attacks are unique because they exploit the previously ignored physical design (i.e. layout, placement, routing, etc.) of a circuit to compromise security. In order to protect our critical infrastructure and military technology from these novel security threats, we must fundamentally shift our view of hardware security to consider an IC's physical design. This opens the door to a new generation of research to rethink the nature of hardware security through the lens of physical design. Possible future work in this area spans topics ranging from developing mathematical models for a physical attacker, to exploring novel attack strategies that exploit physical leakage, to developing security-aware physical design automation algorithms.

7.1.3 Hardware Security and Artificial Intelligence

Artificial intelligence and machine learning have exploded in popularity, becoming indispensable in applications from medicine to autonomous vehicles. While these technologies undoubtedly improve our daily lives, they also open the door to an entirely new set of security concerns. These range from machine-learning-based attacks on existing security schemes, to verifying the integrity of trained applications, to hardening the security of machine-learning-focused hardware (e.g. Google's Tensor Processing Unit). For each of these threat vectors, hardware security is a vital component. Machine learning applications and hardware present unique security challenges due to their limited mathematical verifiability [44] and error resilience properties [115, 46]. These challenges present significant barriers to securing these specialized hardware systems with existing technologies and security schemes. It is vital that we develop methods to overcome these challenges to protect and secure the wide range of critical infrastructure that is reliant on artificial intelligence and machine learning.

Bibliography

[1] Bilge ES Akgul, Lakshmi N Chakrapani, Pinar Korkmaz, and Krishna V Palem. Probabilistic cmos technology: A survey and future directions. In *International Conference on Very Large Scale Integration*, pages 1–6. IEEE, 2006.

[2] Yousra Alkabani and Farinaz Koushanfar. Active hardware metering for intellectual property protection and security. In *USENIX security symposium*, 2007.

[3] Alexandru Amaricai, Sergiu Nimara, Oana Boncalo, Jiaoyan Chen, and Emanuel Popovici. Probabilistic gate level fault modeling for near and sub-threshold cmos circuits. In *2014 17th Euromicro Conference on Digital System Design*, pages 473–479. IEEE, 2014.

[4] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *Transactions on Cryptographic Hardware and Embedded Systems*, 2019.

[5] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. Preventing ic piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers*, pages 66–75, 2010.

[6] Kirti Bhanushali and W Rhett Davis. Freepdk15: An open-source predictive process design kit for 15nm finfet technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, pages 165–170, 2015.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[9] Alberto Bosio and Giorgio Di Natale. Lifting: A flexible open-source fault simulator. In *2008 17th Asian Test Symposium*, pages 35–40. IEEE, 2008.

[10] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *IEEE international symposium on circuits and systems*, 1989.

[11] Abhishek Chakraborty, Nithyashankari Gummidipoondi Jayasankaran, Yuntao Liu, Jeyavijayan Rajendran, Ozgur Sinanoglu, Ankur Srivastava, Yang Xie, Muhammad Yasin, and Michael Zuzak. Keynote: A disquisition on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[12] Abhishek Chakraborty, Yuntao Liu, and Ankur Srivastava. Timingsat: Timing profile embedded sat attack. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–6, 2018.

[13] Abhishek Chakraborty, Yang Xie, and Ankur Srivastava. Gpu obfuscation: attack and defense strategies. In *Design Automation Conference*, 2018.

[14] Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. Sail: Machine learning guided structural analysis attack on hardware obfuscation. In *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 56–61. IEEE, 2018.

[15] Rajat Subhra Chakraborty and Swarup Bhunia. Harpoon: an obfuscation-based soc design methodology for hardware protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1493–1502, 2009.

[16] Jui-Ming Chang and Massoud Pedram. Register allocation and binding for low power. In *ACM/IEEE Design Automation Conference (DAC)*, 1995.

[17] Jianqi Chen, Monir Zaman, Yiorgos Makris, RD Shawn Blanton, Subhasish Mitra, and Benjamin Carrion Schafer. Decoy: Deflection-driven hls-based computation partitioning for obfuscating intellectual property. In *Design Automation Conference (DAC)*. IEEE, 2020.

[18] Clayton M Christensen, Steven King, Matt Verlinden, and Woodward Yang. The new economics of semiconductor manufacturing. *iEEE SpEctrum*, 45(5):24–29, 2008.

[19] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design & Test of computers*, 17(3):44–53, 2000.

[20] Avinash R Desai, Michael S Hsiao, Chao Wang, Leyla Nazhandali, and Simin Hall. Interlocking obfuscation for anti-tamper hardware. In *Proceedings of the eighth annual cyber security and information intelligence research workshop*, pages 1–4, 2013.

[21] Jaya Dofe and Qiaoyan Yu. Novel dynamic state-deflection method for gate-level design obfuscation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):273–285, 2017.

[22] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*. IEEE, 2014.

[23] Mohamed El Massad, Siddharth Garg, and Mahesh Tripunitara. Reverse engineering camouflaged sequential circuits without scan access. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 33–40. IEEE, 2017.

[24] Mohamed El Massad, Siddharth Garg, and Mahesh V Tripunitara. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS*, pages 1–14, 2015.

[25] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2014.

[26] Marc Fyrbiak, Sebastian Wallat, Jonathan Déchelotte, Nils Albartus, Sinan Böcker, Russell Tessier, and Christof Paar. On the difficulty of fsm-based hardware obfuscation. *Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[27] Jason George, Bo Marr, Bilge ES Akgul, and Krishna V Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 158–168, 2006.

[28] Matthew R Guthaus, James E Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2016.

[29] Ku He, Andreas Gerstlauer, and Michael Orshansky. Controlled timing-error acceptance for low energy idct design. In *Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[30] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data path allocation based on bipartite weighted matching. In *Design Automation Conference*, 1991.

[31] Sheikh Ariful Islam, Love Kumar Sah, and Srinivas Katkoori. High-level synthesis of key-obfuscated rtl ip with design lockout and camouflaging. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(1):1–35, 2020.

[32] Nithyashankari Gummidipoondi Jayasankaran, Adriana Sanabria Borbon, Edgar Sanchez-Sinencio, Jiang Hu, and Jeyavijayan Rajendran. Towards provably-secure analog and mixed-signal locking against overproduction. In *Proceedings of the International Conference on Computer-Aided Design*, page 7, 2018.

[33] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks. In *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[34] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Interlock: An intercorrelated logic and routing locking. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[35] Richard M Karp. An algorithm to solve the m$\times$ n assignment problem in expected time o (mn log n). *Networks*, 1980.

[36] Pinar Korkmaz, Bilge ES Akgul, and Krishna V Palem. Energy, performance, and probability tradeoffs for energy-efficient probabilistic cmos circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 55(8):2249–2262, 2008.

[37] Leonidas Lavdas, M Tanjidur Rahman, Mark Tehranipoor, and Navid Asadizanjani. On optical attacks making logic obfuscation fragile. In *2020 IEEE International Test Conference in Asia (ITC-Asia)*, pages 71–76. IEEE, 2020.

[38] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*. IEEE, 1997.

[39] Julian Leonhard, Muhammad Yasin, Shadi Turk, Mohammed Thari Nabeel, Marie-Minerve Louërat, Roselyne Chotin-Avot, Hassan Aboushady, Ozgur Sinanoglu, and Haralampos-G Stratigopoulos. Mixlock: Securing mixed-signal circuits via logic locking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 84–89. IEEE, 2019.

[40] Meng Li, Kaveh Shamsi, Yier Jin, and David Z Pan. Timingsat: Decamouflaging timing-based logic obfuscation. In *2018 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2018.

[41] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[42] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *2007 IEEE 13th International symposium on high performance computer architecture*, pages 181–192. IEEE, 2007.

[43] Jinghang Liang, Jie Han, and Fabrizio Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on computers*, 62(9):1760–1771, 2012.

[44] Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing, and Ankur Srivastava. A survey on neural trojans. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 33–39. IEEE, 2020.

[45] Yuntao Liu, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava. Strong anti-sat: Secure and effective logic locking. In *International Symposium on Quality Electronic Design (ISQED)*, 2020.

[46] Yuntao Liu, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava. Robust and attack resilient logic locking with a high application-level impact. *ACM Journal on Emerging Technologies in Computing Systems*, 2021.

[47] Mohamed El Massad, Jun Zhang, Siddharth Garg, and Mahesh V Tripunitara. Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism. *arXiv preprint arXiv:1703.10187*, 2017.

[48] S Ogrenci Memik, Elaheh Bozorgzadeh, Ryan Kastner, and Majid Sarrafzadeh. A super-scheduler for embedded reconfigurable systems. In *International Conference on Computer Aided Design*, 2001.

[49] Debabrata Mohapatra, Georgios Karakonstantis, and Kaushik Roy. Significance driven computation: a voltage-scalable, variation-aware, quality-tuning motion estimator. In *ACM/IEEE international symposium on Low power electronics and design*, pages 195–200, 2009.

[50] Nasir Mohyuddin, Ehsan Pakbaznia, and Massoud Pedram. Probabilistic error propagation in a logic circuit using the boolean difference calculus. In *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 359–381. Springer, 2011.

[51] Ankit Mondal, Michael Zuzak, and Ankur Srivastava. Statsat: A boolean satisfiability based attack on logic-locked probabilistic circuits. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[52] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247. IEEE, 2005.

[53] Md Rafid Muttaki, Roshanak Mohammadivojdan, Mark Tehranipoor, and Farimah Farahmandi. Hlock: Locking ips at the high-level language. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 79–84. IEEE, 2021.

[54] Prashant J Nair, David A Roberts, and Moinuddin K Qureshi. Fault sim: A fast, configurable memory-reliability simulator for conventional and 3d-stacked systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[55] Satwik Patnaik, Nikhil Rangarajan, Johann Knechtel, Ozgur Sinanoglu, and Shaloo Rakheja. Advancing hardware security using polymorphic and stochastic spin-hall effect devices. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 97–102. IEEE, 2018.

[56] Satwik Patnaik, Nikhil Rangarajan, Johann Knechtel, Ozgur Sinanoglu, and Shaloo Rakheja. Spin-orbit torque devices for hardware security: From deterministic to probabilistic regime. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[57] Christian Pilato, Luca Collini, Luca Cassano, Donatella Sciuto, Siddharth Garg, and Ramesh Karri. On the optimization of behavioral logic locking for high-level synthesis. *arXiv preprint arXiv:2105.09666*, 2021.

[58] Christian Pilato, Francesco Regazzoni, Ramesh Karri, and Siddharth Garg. Tao: techniques for algorithm-level obfuscation during high-level synthesis. In *Proceedings of the 55th Annual Design Automation Conference*, page 155. ACM, 2018.

[59] Jan Rabaey. *Low power design essentials*. Springer Science & Business Media, 2009.

[60] M Sazadur Rahman, Adib Nahiyan, Sarah Amir, Fahim Rahman, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. Dynamically obfuscated scan chain to resist oracle-guided attacks on logic locked design. Cryptology ePrint Archive, Report 2019/946, 2019. `https://eprint.iacr.org/2019/946`.

[61] M Tanjidur Rahman, Nusrat Farzana Dipu, Dhwani Mehta, Shahin Tajik, Mark Tehranipoor, and Navid Asadizanjani. Concealing-gate: Optical contactless probing resilient design. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 17(3):1–25, 2021.

[62] M Tanjidur Rahman, Qihang Shi, Shahin Tajik, Haoting Shen, Damon L Woodard, Mark Tehranipoor, and Navid Asadizanjani. Physical inspection & attacks: New frontier in hardware security. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 93–102. IEEE, 2018.

[63] M Tanjidur Rahman, Shahin Tajik, M Sazadur Rahman, Mark Tehranipoor, and Navid Asadizanjani. The key is left under the mat: On the inappropriate security assumption of logic locking schemes. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 262–272. IEEE, 2020.

[64] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of Design Automation Conference*, 2012.

[65] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Transactions on computers*, 64(2):410–424, 2013.

[66] Amin Rezaei, You Li, Yuanqi Shen, Shuyu Kong, and Hai Zhou. Cycsat-unresolvable cyclic logic encryption using unreachable states. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 358–363. ACM, 2019.

[67] R Robache, J-F Boland, Claude Thibeault, and Yvon Savaria. A methodology for system-level fault injection based on gate-level faulty behavior. In *2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS)*, pages 1–4. IEEE, 2013.

[68] Shervin Roshanisefat, Hadi Mardani Kamali, and Avesta Sasan. Srclock: Sat-resistant cyclic logic locking for protecting the hardware. In *Great Lakes Symposium on VLSI*, 2018.

[69] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 2014.

[70] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Epic: Ending piracy of integrated circuits. In *Conference on Design, automation and test in Europe*, 2008.

[71] Akashdeep Saha, Sayandeep Saha, Siddhartha Chowdhury, Debdeep Mukhopadhyay, and Bhargab B Bhattacharya. Lopher: Sat-hardened logic embedding on block ciphers. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[72] Pia N Sanda, Jeffrey W Kellington, Prabhakar Kudva, Ronald Kalla, Ryan B McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R Jones. Soft-error resilience of the ibm power6 processor. *IBM Journal of Research and Development*, 2008.

[73] Abhrajit Sengupta, Mohammed Ashraf, Mohammed Nabeel, and Ozgur Sinanoglu. Customized locking of ip blocks on a multi-million-gate soc. In *International Conference on Computer-Aided Design*, 2018.

[74] Abhrajit Sengupta, Mohammed Nabeel, Nimisha Limaye, Mohammed Ashraf, and Ozgur Sinanoglu. Truly stripping functionality for logic locking: A fault-based perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[75] Abhrajit Sengupta, Mohammed Nabeel, Muhammad Yasin, and Ozgur Sinanoglu. Atpg-based cost-effective, secure logic locking. In *IEEE 36th VLSI Test Symposium (VTS)*. IEEE, 2018.

[76] Bicky Shakya, Xiaolin Xu, Mark Tehranipoor, and Domenic Forte. Cas-lock: A security-corruptibility trade-off resilient logic locking scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 175–202, 2020.

[77] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Appsat: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 95–100. IEEE, 2017.

[78] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. Cross-lock: Dense layout-level interconnect locking using cross-bar architectures. In *Great Lakes Symp. on VLSI*, 2018.

[79] Kaveh Shamsi, Meng Li, Kenneth Plaks, Saverio Fazzari, David Z Pan, and Yier Jin. Ip protection and supply chain security through logic obfuscation: A systematic overview. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2019.

[80] Kaveh Shamsi, Travis Meade, Meng Li, David Z Pan, and Yier Jin. On the approximation resiliency of logic locking and ic camouflaging schemes. *Trans. on Information Forensics and Security*, 2018.

[81] Kaveh Shamsi, David Z Pan, and Yier Jin. On the impossibility of approximation-resilient circuit locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–170. IEEE, 2019.

[82] Haoting Shen, Navid Asadizanjani, Mark Tehranipoor, and Domenic Forte. Nanopyramid: An optical scrambler against backside probing attacks. In *ISTFA 2018: Proceedings from the 44th International Symposium for Testing and Failure Analysis*, page 280. ASM International, 2018.

[83] Yuanqi Shen and Hai Zhou. Double dip: Re-evaluating security of logic encryption algorithms. In *Great Lakes Symposium on VLSI 2017*, 2017.

[84] Deepak Sirone and Pramod Subramanyan. Functional analysis attacks on logic locking. In *Design, Automation & Test in Europe Conference & Exhibition*, 2019.

[85] Deepak Sirone and Pramod Subramanyan. Functional analysis attacks on logic locking. *IEEE Transactions on Information Forensics and Security*, 15:2514–2527, 2020.

[86] Ansgar Stammermann, Domenik Helms, Milan Schulte, Arne Schulz, and Wolfgang Nebel. Binding allocation and floorplanning in low power high-level synthesis. In *International Conference on Computer Aided Design*. IEEE, 2003.

[87] Sanbao Su, Yi Wu, and Weikang Qian. Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis. In *Proceedings of the 55th Annual Design Automation Conference*, page 54. ACM, 2018.

[88] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143. IEEE, 2015.

[89] S. S. Technology. Why node shrinks are no longer offsetting equipment costs. 2012.

[90] Huanyu Wang, Domenic Forte, Mark M Tehranipoor, and Qihang Shi. Probing attacks on integrated circuits: Challenges and research opportunities. *IEEE Design & Test*, 34(5):63–71, 2017.

[91] I-Chyn Wey, You-Gang Chen, Chang-Hong Yu, An-Yeu Wu, and Jie Chen. Design and implementation of cost-effective probabilistic-based noise-tolerant vlsi circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(11):2411–2424, 2009.

[92] Yang Xie and Ankur Srivastava. Mitigating sat attack on logic locking. In *Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[93] Yang Xie and Ankur Srivastava. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 9. ACM, 2017.

[94] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2018.

[95] Xiaolin Xu, Bicky Shakya, Mark M Tehranipoor, and Domenic Forte. Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks. In *International conference on cryptographic hardware and embedded systems*, pages 189–210, 2017.

[96] Fangfei Yang, Ming Tang, and Ozgur Sinanoglu. Stripped functionality logic locking with hamming distance based restore unit (sfll-hd)–unlocked. *IEEE Transactions on Information Forensics and Security*, 2019.

[97] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *2016 IEEE symposium on security and privacy (SP)*, pages 18–37. IEEE, 2016.

[98] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In *Intl. Symposium on Hardware Oriented Security and Trust*, 2016.

[99] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Ttlock: Tenacious and traceless logic locking. In *Intl. Symposium on Hardware Oriented Security and Trust*, 2017.

[100] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Removal attacks on logic locking and camouflaging techniques. *Transactions on Emerging Topics in Computing*, 2017.

[101] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security analysis of anti-sat. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[102] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2016.

[103] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In *Conference on Computer and Communications Security*, 2017.

[104] Muhammad Yasin, Chongzhi Zhao, and Jeyavijayan JV Rajendran. Sfll-hls: Stripped-functionality logic locking meets high-level synthesis. In *Intl. Conf. on Computer-Aided Design*, 2019.

[105] Monir Zaman, Abhrajit Sengupta, Danqing Liu, Ozgur Sinanoglu, Yiorgos Makris, and Jeyavijayan JV Rajendran. Towards provably-secure performance locking. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1592–1597, 2018.

[106] Dongrong Zhang, Miao He, Xiaoxiao Wang, and Mark Tehranipoor. Dynamically obfuscated scan for protecting ips against scan-based attacks throughout supply chain. In *2017 IEEE 35th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2017.

[107] Hai Zhou. A humble theory and application for logic encryption. *IACR Cryptology ePrint Archive*, 2017:696, 2017.

[108] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. Cycsat: Sat-based attack on cyclic logic encryptions. In *IEEE/ACM International Conference on Computer-Aided Design*, 2017.

[109] Hai Zhou, Amin Rezaei, and Yuanqi Shen. Resolving the trilemma in logic encryption. In *International Conference on Computer-Aided Design (ICCAD)*, 2019.

[110] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo. Ultra low-power high-speed flexible probabilistic adder for error-tolerant applications. In *2011 International SoC Design Conference*, pages 393–396. IEEE, 2011.

[111] Michael Zuzak, Yuntao Liu, and Ankur Srivastava. Trace logic locking: Improving the parametric space of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[112] Michael Zuzak, Yuntao Liu, and Ankur Srivastava. A resource binding approach to logic obfuscation. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 235–240. IEEE, 2021.

[113] Michael Zuzak, Ankit Mondal, and Ankur Srivastava. Evaluating the security of logic-locked probabilistic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[114] Michael Zuzak and Ankur Srivastava. Memory locking: An automated approach to processor design obfuscation. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 541–546, 2019.

[115] Michael Zuzak and Ankur Srivastava. Obfusgem: Enhancing processor design obfuscation through security-aware on-chip memory and data path design. In *International Symposium on Memory Systems*. ACM, 2020.