Research Thesis: Declaration of Authorship

Print name:	Yue Zhang
Title of thesis:	Advancing Logic Locking: A New Strategy Combining Dynamic and Zero-Knowledge Techniques

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as:

Y. Zhang, B. Halak and H. Wang, ZeKi: A Zero-Knowledge Dynamic Logic Locking Implementation with Resilience to Multiple Attacks, 2024 IEEE 37th International System-on-Chip Conference (SOCC), Dresden, Germany, 2024, pp. 1-6, doi: 10.1109/SOCC62300.2024.10737800.

Signature:

Date: 2026-02-02

UNIVERSITY OF SOUTHAMPTON

Advancing Logic Locking: A New Strategy Combining Dynamic and Zero-Knowledge Techniques

by

Zhang Yue ORCID: 0009-0002-9275-1989

Supervisor: Dr Basel Halak

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

January 2026

Abstract

The IC industry's growth has heightened attention to hardware security, particularly in the last decade. Major IC companies, aiming to cut costs, have begun outsourcing parts of their supply chain instead of managing it entirely in-house. This cost-saving strategy increases vulnerability to malicious attacks. To counter this, various defence mechanisms like logic locking have been proposed, offering protection against unauthorized IP access and supply chain threats with minimal design flow alterations.

Despite logic locking's many advantages in hardware defence, it confronts two major challenges. Firstly, significant efforts in the field have aimed at extracting secret keys from encrypted designs, particularly to clone IP. The SAT attack [1] notably undermines the efficacy of logic locking. To mitigate this, various techniques like point function-based logic locking (PFB) [2, 3, 4] have emerged, balancing SAT attack resilience with minimal product overhead. However, PFB's fixed mechanism inherently exposes specific properties, making it vulnerable to targeted attacks. [5, 6] Secondly, traditional logic locking presumes the design house's trustworthiness. However, internal malicious actors are a real threat in practice, against which current logic locking methods are ineffective.

This thesis introduces Dynamic Logic Locking (DLL), a new logic locking strategy to address the aforementioned security concerns. DLL's dynamic mechanism, unlike traditional methods, uses randomly generated locking blocks to counter SAT and other logic locking attacks. Its unique mechanism for each design eliminates structural vulnerabilities, unlike conventional fixed-mechanism blocks. Moreover, DLL's random block generation allows for zero-knowledge implementation by designers, who need not know the block's structure or secret key details. This makes DLL robust even against internal malicious attacks within the design house.

This thesis also introduces DLL-se, a sequential variant of DLL, expanding its application to sequential circuits for increased utility. Additionally, it presents a Python-based logic locking tool capable of integrating various logic locking types, including DLL, into Verilog netlists and simulating attacks on logic locking. With its clear GUI, the tool is accessible to designers and students and is suitable for hardware design and educational research.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to begin by expressing my deepest gratitude to my supervisor, Dr. Basel, whose guidance, understanding, and constant support were instrumental in the completion of this PhD project. His mentorship has been invaluable, and I truly believe he is one of the best mentors in the world.

I am also immensely grateful to my parents for their endless love, which has been my driving force throughout this journey. I sincerely hope they continue to enjoy good health and live strong and happy lives.

I would like to give special thanks to my dear friend and brother, Dr. Wang Haoyu, who generously helped me in rewriting parts of my dissertation. His assistance, encouragement, and belief in me were pivotal to my progress.

Lastly, I would like to take a moment to acknowledge myself. Despite the challenges, I made it to the finish line. Only I truly understand the significance of this achievement. I am proud of the journey and the person I have become. Keep pushing forward.

Chapter 1

Introduction

Since the beginning of the 21st century, the Integrated Circuit (IC) supply chain has undergone significant transformations. These changes are reflected in various aspects: the rising cost and complexity of IC manufacturing, increasing operational and troubleshooting expenses of chip factories, the market's growing demand for rapid response, acceleration of supply chain processes, the involvement of multiple third-party Intellectual Property (IP) providers, the introduction of cutting-edge technologies, and the influence of maintaining a leading position in the semiconductor market [11]. These factors have collectively driven the horizontal development of the IC supply chain, where different entities are responsible for the design, manufacturing, testing, packaging, and integration stages, forming a globalised supply chain.

In this context of a globalised supply chain, to cope with the complexity of designing key chip components, design teams increasingly acquire third-party IPs from numerous owners to expedite product launches. Furthermore, considering the total cost of manufacturing, wafer sorting, cutting, packaging, and package testing, along with the necessity of employing the latest technologies, many design companies opt to complete these stages in overseas facilities. The involvement of outsourcing and multiple stakeholders not only reduces the cost of chips but also shortens their time to market.

Since the beginning of the 21st century, the rapid expansion of the chip market has led to a significant growth of semiconductors industry, while also dramatically increasing the market value of major manufacturers such as TSMC, UMC, and SMIC [11]. The surge in demand has driven Original Equipment Manufacturers (OEMs) to continuously

advance their technologies in design, implementation, manufacturing, and testing in order to remain competitive. At the same time, global collaboration has deepened significantly. The acceleration of globalisation has made the supply chain more open and efficient, improving overall productivity.

While globalisation of the supply chain has brought benefits, it has also increased risks. The involvement of multiple entities, lack of trust, and insufficient monitoring have decreased the control of original manufacturers and IP owners/suppliers over the supply chain, leading to various hardware security threats such as IP piracy, overproduction of ICs, and counterfeiting[12, 13]. Hardware security issues have become increasingly serious, thereby attracting growing attention in this field. There is much work to be done in this field to ensure that IC products are not plagiarized or attacked by malicious entities.

To counter the upcoming hardware threats, the academic community has proposed various design trust countermeasures, ranging from passive to active strategies, such as watermarking, IC metering, IC camouflaging, and hardware obfuscation[12, 13]. Particularly, *logic locking*, as an active IP protection technique, has garnered widespread attention over the past two decades, with robust solutions developed at various levels of abstraction.

The structure of this thesis is organised as follows: Section 1.1 presents the problem statement of this research. Section 1.2 outlines the motivation behind the study. Section 1.3 presents the main research objectives. Section 1.4 summarises the key contributions of this work. Section 1.5 provides an overview of the overall thesis structure. Finally, Section 1.6 lists the papers published by the author during this doctoral research.

1.1 Problem Statement

1.1.1 Hardware Security Threats and Defence Techniques

In today's complex integrated circuit (IC) supply chains, a wide range of security threats have emerged, creating significant challenges for both industry stability and technological integrity. The rapid expansion of these supply chains has introduced numerous security vulnerabilities. Key threats include intellectual property (IP) piracy [14], overbuilding

[15], hardware Trojans [16, 17, 18, 19], reverse engineering [20, 21], and counterfeiting [22, 23], whose potential threat is labeled in the supply chain shown in figure 1.1.

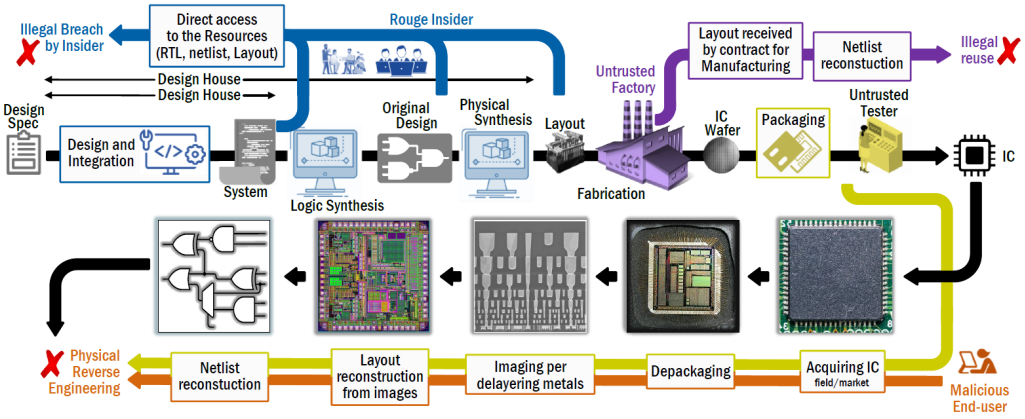

FIGURE 1.1: Different Threat Models at Different Stages of IC Design Flow [7].

In order to thwart emerging hardware security threats, various defence techniques have been developed, including watermarking and fingerprinting [24, 25], camouflaging [26], split manufacturing[27], and logic locking [9].

1.1.1.1 Watermarking and fingerprinting

Watermarking is a defence technique where the designer inserts a digital signature into the circuit; this signature could be a design constraint [24]. In fingerprinting, the user's digital signature is also embedded in the design, along with the designer's signature. Techniques like watermarking and fingerprinting are referred to as passive techniques, which can only detect malicious operations on the hardware but cannot protect the circuitry from malicious attacks [25]. Both techniques are employed at the logic design and physical design stages of the design flow.

1.1.1.2 Camouflaging

Camouflaging, as its name, in selected part of the circuitry, designer replaces the gates with their camouflaged counterparts. Compared to normal logic gates, camouflaged gates are much harder for the reverse engineering attackers to tell their function. With a trusted foundry, camouflage is able to protect the circuit from reverse engineering in certain level [26].

1.1.1.3 Split Manufacturing

In order to address the issue of an untrusted foundry, split manufacturing divides the manufacturing of the product into two stages: back-end-of-the-line (BEOL) and front-end-of-the-line (FEOL). The two parts of the product are then manufactured separately in two different foundries to ensure that no single foundry has full access to the design [28, 29, 30, 31, 32].

Split manufacturing and camouflaging are both layout-level techniques that are applied to the product after the physical design stage.

1.1.1.4 Logic Locking

Logic locking refers to the ability to introduce post-fabrication programmability through additional gates, known as key-gates, which are controlled by a secret key. These key-gates enable the locking of the circuit's functionality, ensuring that the circuit behaves correctly only when the correct key is provided. In other words, hardware designers of logic locking will implant a serious of logic gates [9] or other structures such as eFPGA [14, 33, 34], which will cause primary output of the protected circuit to distort. The only way for the protected circuit to perform functional correctness is to input a preset key sequence which only hardware designer has access to and is used to activate the product after fabrication. The correct key sequence is then stored in a tamper-proof memory in the system.

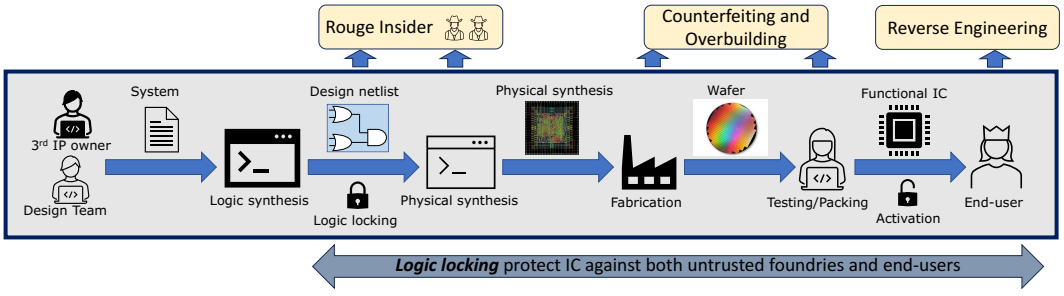

FIGURE 1.2: IC Supply Chain, Threat Modelling, and Logic Locking Engagement (Injecting in Netlist and Activating after Fabrication)

In the standard design flow, locking gates are implanted in the chip during the logic synthesis stage, as depicted in figure 1.2, and only the designer knows the correct key. This mechanism ensures that even if a malicious foundry obtains the design in the

subsequent stages, the attacker cannot make the chip function correctly without the right key. The chip is activated by the secret key after the design flow is completed, and the key is stored in tamper-proof memory, ensuring that malicious attackers cannot access it. This logic locking mechanism ensures the design's security throughout the design flow and at the end-user stage. Compared to other defence strategies, as shown in table 1.1, logic locking offers a broader range of protection across the supply chain.

TABLE 1.1: Hardware defence strategies comparison

DfTr technique	SoC integrator	Foundry	Test facility	End-user
Watermarking	N	N	N	N
Camouflaging	N	N	N	Y
Split manufacturing	N	Y	N	N
Logic Locking	Y	Y	Y	Y

One of the critical challenges in logic locking is enabling effective functional verification without disclosing the secret key to potentially untrusted parties involved in the design or testing process. To address this issue, several practical and secure strategies have been developed. The most widely adopted approach involves the use of **encrypted simulation modules** compliant with the *IEEE 1735* [35] standard for IP encryption. By encrypting the key-related modules and controlling access rights through license-based restrictions, third-party verification teams are able to simulate the design and verify functionality without accessing the actual RTL or the embedded key. In addition, commercial EDA tools such as Synopsys VCS, Cadence Xcelium, and Mentor Questa provide support for encrypted RTL simulation, enabling secure key encapsulation throughout the verification flow.

Another secure alternative is the use of **formal equivalence checking** tools such as *Cadence Conformal* or *Synopsys Formality*, which allow designers to verify the equivalence between the locked and unlocked versions of the circuit using symbolic representations of the key. In this setting, the key remains an abstract symbolic variable during verification, thereby preventing its exposure. These approaches collectively ensure that robust design verification can be conducted without compromising the confidentiality of the secret key—an essential requirement for secure hardware IP delivery and validation.

Hardware designers can implement logic locking at various levels of abstraction. Figure 1.3 illustrates a simple example of logic locking across different abstraction levels. For instance, at the layout level, as shown in Figure 1.3, the metal-insulator-metal

(MIM) structure, which connects two adjacent metal layers, can be used as a key-based programmable unit for routing-based locking [36]. Compared to other abstract levels, Gate-level logic locking technique is often the easiest to implement, and cause acceptable overhead.

Currently, most existing logic locking techniques are implemented at the gate-level, typically as a post-synthesis operation on the synthesized gate-level netlist in the supply chain. In this Phd project, all the work, including Zeki and Locklab, is focused on gate-level logic locking.

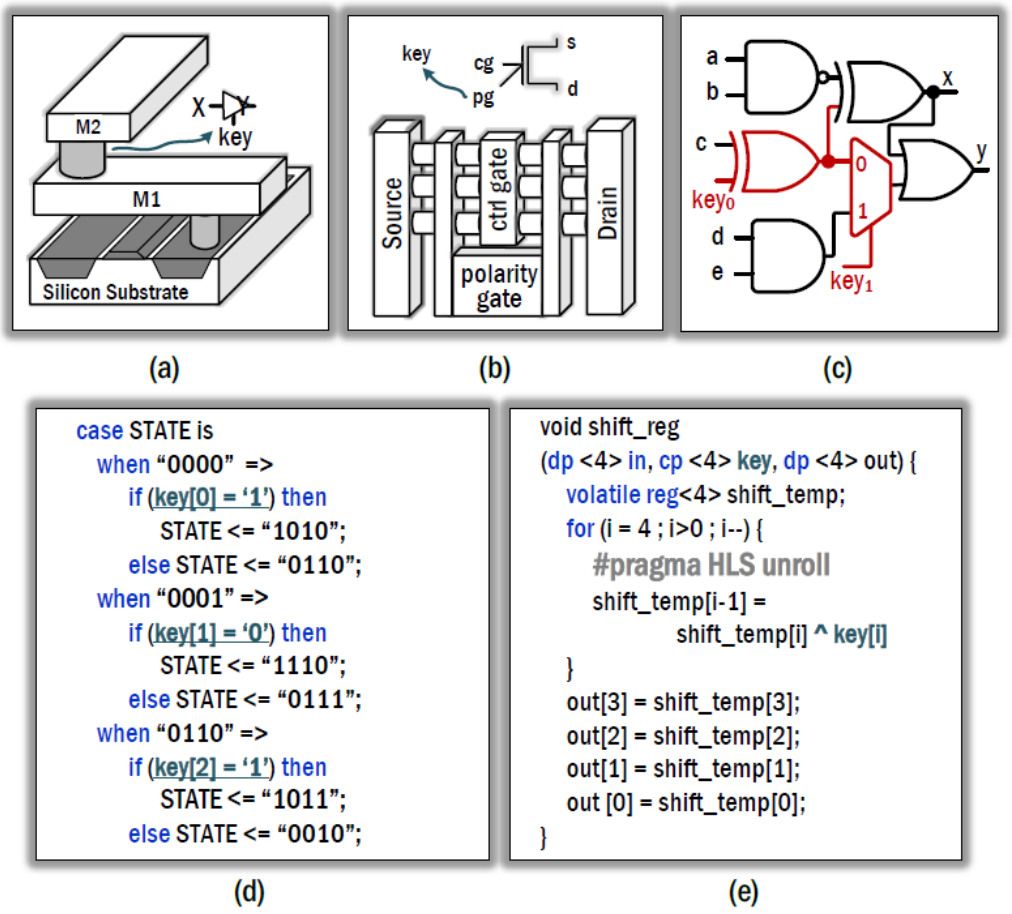

FIGURE 1.3: Logic Locking Examples at Different Levels of Abstraction: (a) Layout-level Key-based Routing, (b) Transistor-level Key-based Basic Gates, (c) Key-based Logic/Routing Gate-level, (d) RTL-level Keybased FSM, (e) HLS-level Key-based Shift Register.

1.1.2 Basic Working Principle of Logic Locking

Logic locking[9] stands out for its ability to safeguard the supply chain and end-users with minimal design flow modifications.

Logic locking protects a chip by inserting locking gates, typically XOR/XNOR gates. The chip with implanted locking gates requires the user to input a correct secure key to function properly; otherwise, it will produce incorrect outputs.

Figure 1.4 shows why this is the case.

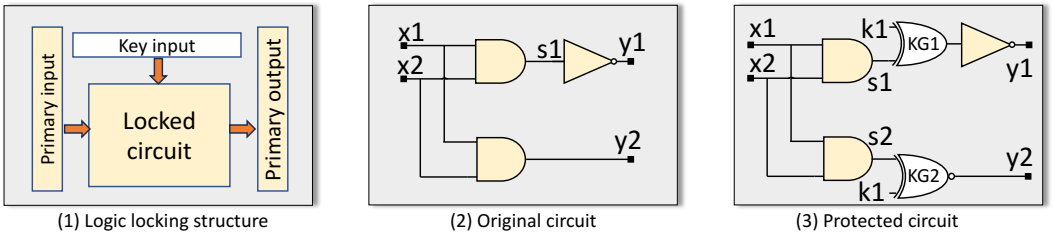

FIGURE 1.4: Basic working principle of logic locking.

1.1.3 Threat Model in Attacks on Logic Locking

The threat models for attacks on logic locking are usually divided into two categories: oracle-guided and oracle-less, where the oracle refers to a functional IC (or golden model) that provides correct input–output pairs. The work of this project is focused on oracle-guided threat model.

In the threat model for oracle-guided attacks, it is assumed that attackers have access to two critical entities:

1. **A functional IC**: Attackers can readily acquire a functional IC from the market. During the attack process, this functional IC provides valid input/output pairs as golden references;
2. **A gate-level netlist file of the locked circuit**: This file is obtained through reverse engineering or from an untrusted foundry.

1.1.4 Existing Challenges

1.1.4.1 Limitation of Single Locking Mechanism

Logic locking is a widely adopted hardware security technique due to its ease of implementation and relatively low overall hardware cost. It offers broad protection for integrated circuits (ICs) and other hardware products. However, its development has faced significant challenges, particularly from attacks designed specifically to bypass logic locking mechanisms or extract the secret key. One such attack, the SAT attack [1], involves converting a locked circuit's netlist into an SAT(satisfiability) problem and then using specialised SAT solvers to efficiently break the protection, making the attack process quick and effective.

In response to SAT attacks, numerous defence-oriented logic locking techniques have been proposed. Among them, Point-Function Based (PFB) logic locking [4, 2, 37, 3] provides relatively balanced performance: acceptable overhead and resilience against malicious attacks. However, since each PFB locking strategy relies on a single locking mechanism, it has inherent structural vulnerabilities. Once these vulnerabilities are published, they become exposed to attackers. If an attacker identifies these weaknesses, any circuit protected by that particular PFB strategy becomes susceptible to attacks.

1.1.4.2 Insider Threat

Compared to other hardware security techniques, logic locking provides extensive protection throughout the entire IC design flow. Typically, its protection spans from post-synthesis all the way to the product's market deployment. To enhance the security of the implementation, the secret key used to activate the product is typically introduced after the fabrication phase. However, this approach still carries potential risks. Insider threats may steal information about the locking block structure or the positioning of key-gates, as personnel must intervene during the insertion of locking gates or blocks.

If the locking mechanism of a locked product is leaked, malicious attackers may exploit the structural characteristics of the corresponding locking block to develop targeted attacks that compromise the circuit's protection. For example, sensitization attacks [5]

have been designed specifically against SFLL [3], while SPS attacks [6] target Anti-SAT [2]. Furthermore, the leakage of other information—such as the location of key gates or the arrangement of key bits—can significantly lower the difficulty for attackers attempting to break the locking scheme.

This necessity for human intervention introduces the possibility of leakage, creating a security vulnerability for the protected product. This risk is especially concerning given that nearly all logic locking strategies rely on a single locking mechanism. If the information regarding the locking mechanism leaks, the entire design flow becomes susceptible to threats, severely compromising the integrity of the protection.

1.1.4.3 Easy-to-use Simulation Tool

The rapid development of the IC industry in the 21st century has led to an increasing demand for hardware security technologies. However, while logic locking stands out as a key player in hardware security, its development has not kept pace with the industry's growth. One of the reasons for this slower progress is the lack of a simple, easy-to-use, and comprehensive simulation tool. This gap in tools hampers the broader adoption and refinement of logic locking techniques, limiting their effectiveness in meeting the growing security demands of the industry.

1.2 Motivation

To address the challenges outlined above, this project introduces a novel logic locking strategy, Zeki, and presents a related paper, Zeki: A Zero-Knowledge Dynamic Logic Locking. Compared to traditional logic locking techniques, Zeki introduces two key innovations. First, it employs a dynamic logic locking mechanism that randomly generates a locking block for each product based on different locking mechanisms. This approach avoids the inherent structural vulnerabilities present in single-locking mechanisms, which attackers could exploit to break the circuit's defences.

Second, unlike single-locking mechanisms, Zeki ensures that even if one product is compromised, all other products using the same locking strategy remain secure. Although the same locking mechanism is used for different chips within the same product line,

Zeki generates unique locking blocks for different products. This ensures that an attacker who successfully cracks one product's defence cannot apply the same method to other products protected by Zeki.

Additionally, the dynamic locking mechanism prevents insider involvement in the logic locking implementation, enabling the realization of zero-knowledge locking. Zeki is the first locking strategy to achieve zero-knowledge locking, significantly reducing internal participation and thus mitigating the risk of insider attacks or leaks.

In response to the current lack of effective automatic simulation tools in the logic locking field, I developed LockLab, an automated tool for simulating logic locking and associated attacks. LockLab provides a convenient and efficient platform for researchers and learners in the logic locking domain, greatly enhancing research productivity by enabling fast simulations of various locking strategies and corresponding attacks.

1.3 Research Objectives

The primary objective of this research is to enhance the security and robustness of logic locking strategies in the face of increasingly sophisticated hardware attacks. In light of the limitations of existing single-mechanism logic locking approaches and the practical risks posed by insider threats and structural vulnerabilities, this project aims to address these gaps through both theoretical development and practical tooling. The specific objectives are as follows:

- To design and implement a novel logic locking strategy, that leverages dynamic and randomized locking mechanisms to eliminate structural vulnerability caused by single mechanism and enable zero-knowledge protection.
- To develop a secure framework in which the locking process is resistant to insider threats by minimizing designer involvement during the locking phase.
- To ensure that the proposed method provides strong resilience against a broad range of attacks, including SAT-based, sensitization-based, and AI-based attacks.
- To develop an easy-to-use, automated simulation platform, which is capable of modelling various logic locking strategies and attacks, thereby accelerating research and enabling comparative evaluation.

- To evaluate the effectiveness, scalability, and robustness of the logic locking strategy and simulation platform developed in this work through extensive experiments using industry-standard benchmarks.

1.4 Contributions of This Project

The contributions of this project are listed below:

1. The introduction of the first zero-knowledge locking strategy significantly reduces the involvement of internal designers in the logic locking implementation, thereby minimizing the risk of internal attacks and information leaks. This reduction in insider participation enhances the overall security of the product by preventing potential malicious exploitation and safeguarding sensitive product information.
2. The proposed locking strategy, Zeki, employs a dynamic locking mechanism. Unlike traditional single-locking mechanisms, dynamic locking inherently avoids the structural vulnerabilities present in static locking blocks. Additionally, it prevents the issue where, once a product is compromised, all other products protected by the same mechanism can be attacked using the same exploited vulnerability.
3. Extensive simulations were conducted to evaluate Zeki's resilience against various types of attacks. The simulation results demonstrate that Zeki provides strong defence capabilities against a wide range of attacks targeting logic locking.
4. Furthermore, the LockLab tool was developed as an automated simulation platform for logic locking strategies and related attack simulations. LockLab is user-friendly and significantly enhances the efficiency of researchers and learners in the logic locking field. It supports simulations for multiple locking strategies, including RLL, FLL, SLL, Zeki, SFLL, Anti-SAT, and SARLock, as well as simulations for various attacks such as SAT, SPS, sensitization, and AI-based attacks. These simulations validate LockLab's effectiveness in practical applications.

1.5 General Structure of the Thesis

The structure of the remaining chapters in this thesis is as follows:

- **Chapter 2: Literature Review**

  This chapter provides a detailed examination of various logic locking strategies and the associated attack techniques targeting logic locking. Special emphasis will be placed on SAT attacks and the different types of PFB (Point-Function Based) logic locking, as these techniques have been extensively studied in the field of logic locking. Moreover, they are fundamental to understanding the core principles of this project, particularly in relation to Zeki's operational framework. Other logic locking strategies and corresponding attacks will also be discussed, providing a comprehensive overview of the current landscape in this area of research.

- **Chapter 3: Zeki**

  This chapter focuses on the Zeki technique, detailing the motivation behind its design and its contributions to the field. A thorough explanation of Zeki's mechanism will be presented, followed by a comparison with other PFB logic locking techniques that rely on a single locking mechanism. Additionally, simulations using various benchmarks will be conducted to demonstrate Zeki's resilience against different attacks, providing empirical evidence of its effectiveness. Furthermore, this chapter will explore Zeki's role as the first zero-knowledge logic locking technique, highlighting its advantages in defending against insider threats compared to other locking mechanisms.

- **Chapter 4: Zeki for Sequential Circuits**

  This chapter introduces the sequential version of Zeki, addressing the fact that, in the IC market, sequential circuits are more prevalent than combinational ones. While Zeki was originally designed for combinational circuits, its adaptability to sequential circuits will be discussed in this chapter. The implementation process of applying Zeki to sequential circuits will be explained, demonstrating how the technique can be used in a broader range of IC designs.

- **Chapter 5: LockLab and Lockit**

  This chapter presents an in-depth discussion of LockLab and Lockit, two automation tools for implementing logic locking. These tools significantly simplify the workflow for logic locking implementation, allowing designers to more easily integrate logic locking into their work or conduct experiments with various gate-level locking strategies. Additionally, LockLab proves to be an excellent educational

tool, making it a valuable resource for teaching hardware security and logic locking concepts.

- **Chapter 6: Conclusion**

  The final chapter will summarize all the work presented in this thesis, including the key contributions and findings. It will also discuss the limitations of the current research and outline potential future directions for further exploration in the field of logic locking and hardware security.

1.6 Publication during PhD

Zhang Y, Halak B, Wang H. ZeKi: A Zero-Knowledge Dynamic Logic Locking Implementation with Resilience to Multiple Attacks[C]//2024 IEEE 37th International System-on-Chip Conference (SOCC). IEEE, 2024: 1-6.

Kajtez N, Zhang Y, Halak B. Lockit: A Logic Locking Automation Software[J]. Electronics, 2021, 10(22): 2817.

Chapter 2

Literature Review

2.1 Introduction

Logic locking has evolved into one of the most prominent hardware security techniques for protecting integrated circuits (ICs) against a wide spectrum of threats, including IP piracy, overproduction, counterfeiting, and reverse engineering. Over the past decade, the research community has proposed numerous locking schemes and corresponding attacks, leading to a highly dynamic arms race between defenders and adversaries. In particular, the emergence of the SAT attack has fundamentally reshaped the design goals of logic locking, shifting the focus from simple output corruption towards rigorous SAT-resilient constructions.

This chapter provides a structured review of existing logic locking strategies and their associated attack methodologies. It begins with pre-SAT (section 2.2) schemes such as Random Logic Locking (RLL), Fault-analysis-based Logic Locking (FLL), and Strong Logic Locking (SLL), together with early attacks like the sensitisation attack. It then introduces SAT-based attacks and explains how they exploit distinguishing input patterns to recover the secret key efficiently in section 2.3. Subsequently, Point-Function based (PFB) locking schemes, including SARLock, Anti-SAT, ANDTree, TTLock, and SFLL are discussed as representative post-SAT countermeasures in section 2.4. The section will also introduce advanced approaches such as FSM/sequential locking, cyclic-based and routing-based locking, scan-chain locking, and higher-level or timing-based techniques,

as well as eFPGA-based IP-level locking. After that, recent machine-learning-based attacks are reviewed in section 2.5. In section 2.6 merits and drawbacks of different logic locking approaches are given and compared to show that Point-function based logic locking provides the most balanced performance. This literature review establishes the technical context and motivates the need for the new locking strategy and automation framework proposed in this thesis.

This chapter reviews the main developments in logic locking and related attack techniques. It first discusses pre-SAT logic locking schemes and early attacks, then explains the working principle of the SAT attack as a key turning point in the field. Afterwards, it surveys post-SAT logic locking strategies, including point-function-based locking, FSM/sequential locking, cyclic-based and routing-based approaches, scan chain locking, and other advanced techniques. Finally, recent machine-learning-based attacks are reviewed. The chapter concludes with a comparative discussion of the merits and limitations of these strategies, highlighting why point-function-based logic locking offers a relatively well-balanced solution and thereby motivating the choice of this family of techniques as the basis for the new strategy proposed in this thesis.

2.2 Pre-SAT Logic Locking and sensitisation Attack

Before the advent of SAT-based attacks, logic locking techniques were mainly designed to introduce functional corruption under incorrect key values. Early schemes such as Random Logic Locking (RLL) and Fault-analysis-based Logic Locking (FLL) focused on inserting XOR/XNOR key-gates to maximise output corruption, while Strong Logic Locking (SLL) was later proposed to resist sensitisation attacks. This section reviews these pre-SAT techniques and the first generation of attack against them, providing the historical background for the later shift towards SAT-resilient designs.

2.2.1 Random Logic Locking(RLL)

The original version of logic locking was introduced by [9] to protect IC products from threats such as overproduction, counterfeiting, and other malicious attacks. The concept of basic logic locking is illustrated in Figure 2.1. In this scheme, XOR/XNOR gates are

TABLE 2.1: Output of circuit in Figure 2.1(3) with different key-bit when the input bit x1=0, x2=1 cccc

gray!50 [x1,x2]	k1	k2	y1	y2
01	0	0	1	1
gray!50 01	0	1	1	0
01	1	0	0	1
01	1	1	0	0

embedded into the circuit to obscure its functionality. For example, when an XOR key-gate (KG) is inserted into the protected circuit (see Figure 2.1(3)), if the K1 bit is 0, KG1 functions as a simple wire. If K1 is set to 1, KG1 behaves as an inverter. In the case of XNOR gates, the logic is reversed.

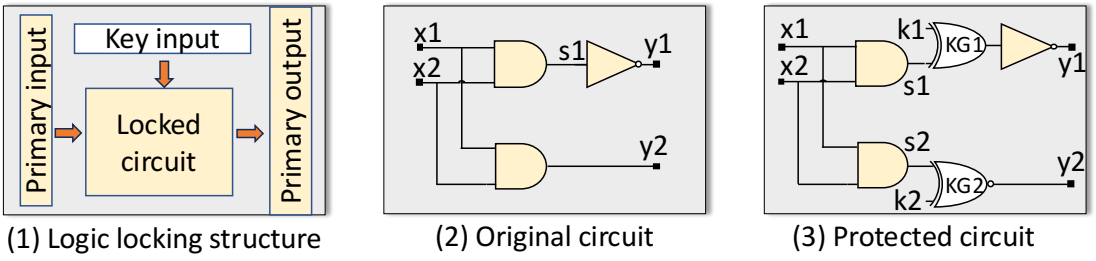

FIGURE 2.1: Basic working principle of logic locking

This mechanism enables the integration of N key-gates, comprising a mix of XOR and XNOR gates, to effectively generate a N-bit security key. The circuit will only function correctly when the correct key sequence is applied.

Table 2.1 presents an example of the input/output combinations for the circuit depicted in Figure 2.1, where the primary input pattern [x1, x2] is set to '01'. The correct key for this circuit is '01', and the corresponding output is '10', which is highlighted in gray in the table. As shown, injecting an incorrect key leads to output distortion, emphasizing the security provided by the locking mechanism.

In this original logic locking approach, the locking gates (XOR and XNOR) are inserted randomly into the circuit, a technique known as Random Logic Locking (RLL).

2.2.2 Fault-analysis based logic locking(FLL)

RLL inserts key gates randomly into the circuit; as a result, the inserted key gates' impact on the output has significant uncertainty. Ideally, 50% of the output bits should exhibit output corruption when an incorrect key is applied to the circuit. This metric can

be expressed as the Hamming distance (the bitwise difference) between the output bits produced with the correct and incorrect keys. A 100% output corruption is effectively equivalent to 0% at some level, since each bit can only be either 1 or 0. Based on this principle, 75% output corruption provides similar protection to 25%, while 50% offers the highest possible level of protection. For RLL, the output corruption rate is uncontrollable, and in the case of small circuits, this value may be relatively low.

Rajendran *et al.* suggest that the impact of an incorrect key is similar to a stuck-at fault, where a signal is stuck at '0' or '1'. FLL (Fault-analysis-based Logic Locking) aims to increase the output corruption caused by an incorrect key by inserting key gates at locations with the largest *fault impact* [38, 39]. A concept of a *fault impact metric* was proposed in [38, 39] to determine the optimal key gate insertion locations. Here, fault impact refers to the degree to which a selected location can propagate a detectable fault from the input to the output.

$$\begin{aligned} FaultImpact = (No.of.TestPatterns_{s-a-0} * No.of.Outputs_{s-a-0}) \\ +(No.of.TestPatterns_{s-a-1} * No.of.Outputs_{s-a-1}) \end{aligned} \tag{2.1}$$

According to the fault impact metric defined in [38], the fault impact of a certain location in the circuit is the sum of the products of the number of test patterns that detect a stuck-at fault and their corresponding output bit numbers. In the process of FLL insertion, the hardware designer computes the fault impact of the logic gates in the circuit to identify the insertion spots with the highest fault impact. The key gates are then inserted at those selected positions.

Compared to RLL, fault-analysis-based logic locking achieves a higher level of output corruption for incorrect key values. The Hamming distance (the bitwise difference) between the correct and corrupted outputs is approximately 50% when using the FLL algorithm.

2.2.3 sensitisation attack

sensitisation attack is the first powerful attack against logic locking. The threat model for sensitisation attack is also used for simulation of other attacks strategies against logic locking. [40, 41, 42]

1. The hardware designer is trusted.
2. End-user and the foundry is no trusted.
3. The attacker has access to a functional product, which can be bought from the market.
4. The attacker has access to the netlist of locked circuit of the product.

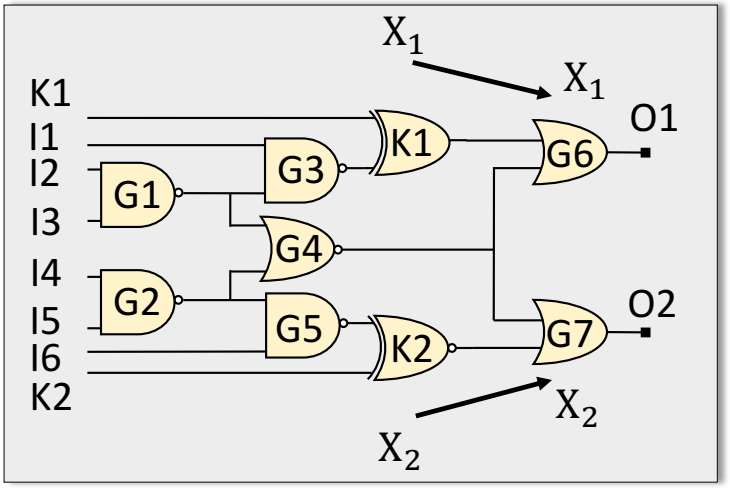

FIGURE 2.2: sensitisation Attack

Instead of applying brutal force decryption, the attackers in a sensitisation attack tries to sensitise the key bit to output, in another word, the key bit will be propagated to the output with no corruption.

Take the circuit in Figure 2.2 above as an example, when the input pattern of I1, I2, and I3 is 110, the output of G1, G3 is 1, 0, and the XOR key gate works as wire, hence the output of key gate 1 is key bit K1. Also, the output of G4 is 0, so the G6's output is key bit K1. Such attack is achievable since the attacker has access of functional IC and locked netlist of the product (able to deduce the input pattern needed for sensitise attack).

2.2.4 Strong Logic Locking

In response to the sensitisation attack, Strong Logic Locking (SLL) was developed. When a circuit is secured using the SLL algorithm, an attacker cannot sensitise a single key bit to the output without accessing other key bits [40, 41].

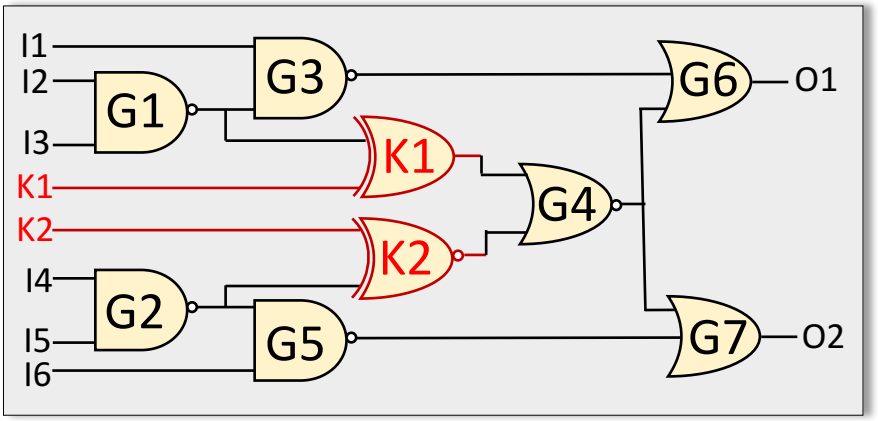

FIGURE 2.3: Strong logic locking

In SLL, the sensitisation of a key gate is obstructed by the presence of other key gates, a mechanism known as pairwise security. If the inserted key gates are pairwise secured, an attacker attempting a sensitisation attack cannot propagate a key bit to the primary output simply by controlling the primary input. They would also need to control the output of the pairwise-secured key gate, which in turn also requires access to its corresponding secured key.

For example, in Figure 2.3, to sensitise key bit K1, the attacker must ensure that the output of key gate 2 is '0'. However, without access to key bit K2, which is securely stored in tamper-proof memory, the attacker cannot manipulate the output of key gate 2. Similarly, other key bits cannot be sensitised to the primary output. In this way, the inserted key gates not only protect the circuit but also reinforce one another, making malicious attacks significantly more difficult.

In summary, prior to the development of SAT attacks, logic locking techniques predominantly focused on inserting individual key gates into the protected circuit. Various insertion strategies were explored to maximize output corruption or to enhance resilience against sensitisation attacks.

2.3 SAT Attack

This section provides a brief introduction to the SAT attack. The SAT attack is a game changer in logic locking, where a SAT-solver, a specialised software tool, is used for extracting the correct key bits efficiently. In an SAT attack, the SAT-solver is employed to iteratively refine the key search space, which requires access to a functional product. As such, the attack is referred to as an *oracle-guided* attack, with the term *oracle* representing the functional IC sample that the attacker obtains from the market. All logic locking strategies developed before the emergence of the SAT attack are easily broken by it, and SAT resilience has become the primary objective for subsequent countermeasures [1].

2.3.1 SAT Attack Background Knowledge

2.3.1.1 Boolean Satisfiability (SAT) problem

Boolean satisfiability (SAT) problem determines the satisfiability of a Boolean formula, in another word, whether there is any assignment to the Boolean formula variables which make it equal to 1 [43]. For example assignment (a, b, c) = (0, 0, 1) makes Boolean formula ((a + !b) & c) equal to 1.

2.3.1.2 Miter Circuit

In SAT attack, attackers transfer the problem into circuit equivalent checking, which is accomplished by a Miter circuit. The Miter circuit contains the two circuits whose equivalence is checked, and xors all the output bits of the two circuits. The outputs of all the XOR gates will be presented as input to an OR gate, and inputs of both circuits are the same [44]. If all the output bits of both circuits are the same, OR gate will produce 0, otherwise 1.

2.3.2 Working Principle of SAT Attack

During an SAT attack, the process is carried out on a Miter circuit, as illustrated in the figure 2.4. The Miter circuit operates by applying identical input patterns to two

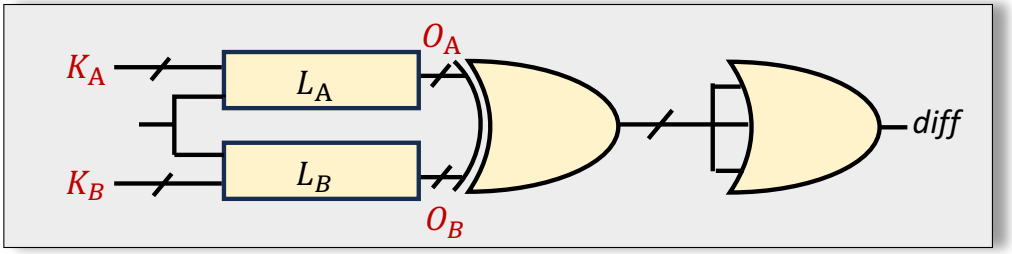

FIGURE 2.4: Miter Circuit

circuits while using different key values. If the outputs of the two circuits match, the 'diff' signal outputs a 0; otherwise, it outputs a 1.

In the SAT attack procedure, the netlist file of the Miter circuit undergoes a Tseitin transformation, converting it into CNF form, which represents a SAT problem. Once in CNF form, the 'diff' output is used as a constraint, where a value of 1 indicates a mismatch between the circuits' outputs.

If the SAT solver identifies an input pattern that satisfies the constraint (i.e., where the 'diff' output is 1), the input pattern is classified as a distinguishing input pattern (DIP). Distinguishing input patterns (DIPs) are the core of SAT attack. DIP refers to an input pattern, with whom there is at least one pair of different key values cause the circuit to produce different output. Take figure 2.4 as an example, with input pattern I1 and two different key values Ka and Kb applied to the Miter circuit, if the circuit produces 1 as output, I1 is a DIP of the circuit.

2.3.2.1 Attack Algorithm

SAT attack is an iterative process, the procedure of which is shown in figure 2.5 and explained below:

- Attacker feeds the CNF of Miter circuit to SAT solver to get DIP which will later be fed to the functional IC to get the correct corresponding output.
- And then DIP and the corresponding output is served as a constraint of locked circuit CNF.
- The SAT solver will then eliminate wrong key values that does not produce correct output of this DIP.

The eliminated incorrect key value will be added as a constraint for further SAT solver execution. This procedure will be repeated until no more DIP can be found which means that all the wrong keys have been ruled out.

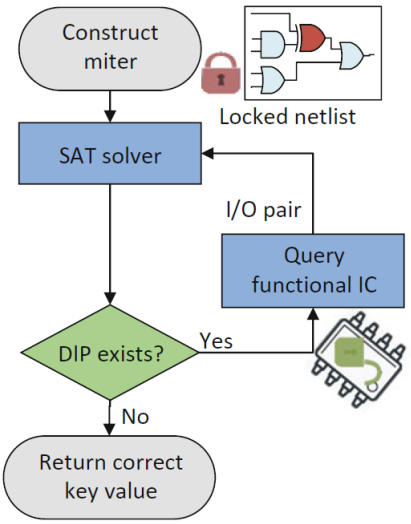

FIGURE 2.5: SAT attack flow chart [8]

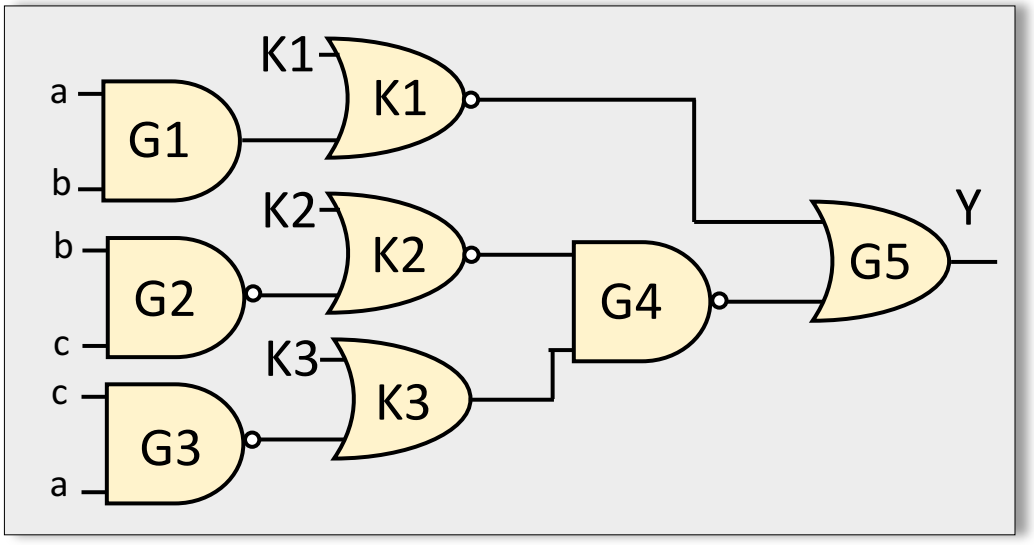

FIGURE 2.6: Circuit Sample for SAT Attack

TABLE 2.2: SAT attack example (K0-K7 represent all the key bit combination (000, ... , 111) of the three key-bits, K1-K2-K3, combination)

abc	Y	K0	K1	K2	K3	K4	K5	K6	K7	Incorrect keys identified
000	0	1	1	1	1	1	1	0	1	
001	0	1	1	1	1	1	1	0	1	
010	0	1	1	1	1	1	1	0	1	iter3: other wrong keys
011	1	1	1	1	1	0	1	1	1	
100	0	1	1	1	1	1	1	0	1	
101	1	1	1	1	1	1	1	1	0	
110	1	1	0	1	1	1	1	1	1	iter1: k2
111	1	0	1	1	1	1	1	1	1	iter2: k1

2.3.2.2 An Simple Example of SAT Attack

The circuit in figure 2.6 is used as the attacked circuit in the example SAT attack. Table 2.2 represents the attack procedure, K0 to K7 represent all the key bit combination (000, ... , 111) of the three key inputs, column Y stands for output bit of the circuit and abc stands for the input pattern.

In the first iteration DIP 110 is applied to the circuit, and the correct output bit produced by functional IC is 1. Among all the key values only K2(010) produces incorrect output, hence K2(010) is eliminated in this iteration. In iteration 2, DIP 111 is applied to the circuit, and K1(001) is eliminated. In the third iteration 010 is served as DIP and ruled out all the remaining incorrect keys. As there is no more DIPs (only 1 key value left), the SAT attack is successfully achieved.

2.3.2.3 SAT Attack Discussion

After the introduction of SAT attacks, nearly all pre-SAT logic locking techniques (RLL, FLL, SLL) became vulnerable. SAT attacks effectively transform the brute-force search for the secret key into a solvable mathematical problem for SAT solvers, which can solve it with high efficiency. During a SAT attack, each distinguishing input pattern (DIP) can eliminate a large number of incorrect keys, reducing the computational effort needed to extract the correct key. The relationship between the key length and the time required for decryption changes from exponential to linear. For example, extracting a 10-bit key no longer requires 1000 attempts but only a few dozen or fewer. This advantage becomes more pronounced as the key length increases, making longer keys easier to break using SAT attacks.

Because SAT problem in SAT attack is definitely solvable (at least one correct answer exists, the correct key), the only way to thwart SAT attack is to increase the execution time. The time consumption for an SAT attack is the sum of execution time of each iteration (DIP).

1. Increase the number of DIP needed to accomplish an SAT attack. This approach is used by many post-SAT logic locking strategies, which will be introduced later.

2. Increasing the execution time required for each individual DIP decryption, which is commonly referred to as an *iteration*, is the other approach. Some strategies modify the circuit structure of the design to make the netlist *SAT-hard*, making it more difficult for SAT solvers to resolve.

After SAT attack was developed, different countermeasures against SAT attack are proposed to protect hardware. In the following part some major Post-SAT logic locking strategies will be introduced.

2.4 Post-SAT Logic Locking Strategies

Following the introduction of the SAT attack, a new generation of logic locking strategies was developed with explicit SAT resilience as a primary design goal. These post-SAT schemes adopt different mechanisms to either increase the number of iterations required by the SAT solver or make each iteration computationally expensive. Among them, point-function-based locking, FSM/sequential locking, cyclic-based locking, routing-based locking, scan chain locking, and several other advanced approaches have been proposed. This section surveys these strategies, with particular emphasis on point-function-based locking, which forms the basis for the new technique introduced later in this thesis.

2.4.1 Point-Function Based Logic Locking

Following the advent of SAT attacks, many logic locking strategies have been developed to defend against them. Among these, Point-Function-Based (PFB) logic locking [4, 2,

37, 3] strikes the best balance between ease of applying locking mechanisms to a design and strong resistance to SAT attacks.

Point-function-based logic locking (PFB) derives its name from its functional behaviour, which mimics that of a point function: it produces the correct output only for one or a small amount of specific combination of inputs and key. For all other combinations, the circuit yields incorrect outputs, thereby increasing resistance to SAT attack.

As discussed in the last section, designer need to increase either number of DIP needed to accomplish an SAT attack or execution time needed for single iteration. Of the two primary approaches to defending against SAT attacks, PFB logic locking adopts the second approach, which focuses on increasing the number of iterations required to break the locking mechanism. PFB significantly reduces the number of input patterns that trigger erroneous outputs in the presence of incorrect keys. In other words, when a key is incorrect, the circuit will only generate faulty outputs for a highly limited set of input patterns. As a result, PFB locking restores the relationship between the time required to execute a SAT attack and the number of key bits to an exponential scale, making a successful SAT attack impractical.

2.4.1.1 SARLock

SARLock (SAT Attack Resistant Logic Locking) is the first logic locking strategy specifically designed to protect against SAT attacks [4].

In contrast to previous logic locking techniques, where the locking unit is integrated within the circuit, SARLock places the locking unit externally, as represented by the blue blocks in Figure 2.7. This external unit compares the key input with both the primary input of the circuit and the correct hardcoded key bits. If the key input matches the circuit's primary input but does not match the correct key bits, a flip signal is triggered, resulting in corrupted circuit output. For example, if the correct key is `'000'` and the primary input is `'111'`, the circuit will produce output corruption only when the key input is `'111'`. Other incorrect key patterns do not cause output distortion. This mechanism ensures that the primary output is affected only when the key input is incorrect and coincides with the circuit's primary input, effectively preventing unauthorised access while maintaining correct functionality when the proper key is applied.

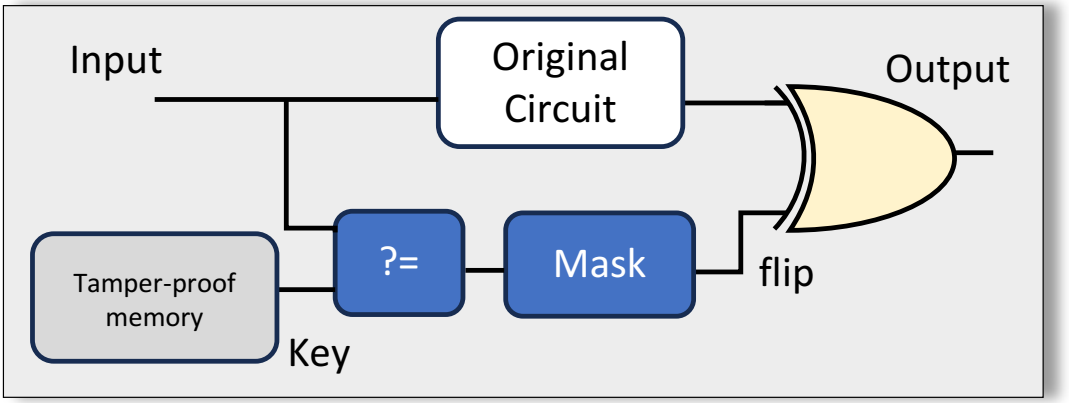

FIGURE 2.7: SARLock locking strategy

In a SARLocked circuit, each input pattern and its corresponding output value (DIP) is only able to eliminate one correct key. For a circuit with n-bit locking key, SAT attack needs to apply $2^n - 1$ iterations to find correct key value.

Vulnerability of SARLock

The security of SARLock is compromised by the AppSAT attack [45], which is designed to byoass its logic locking protection. To defeat SARLock, AppSAT introduces random incorrect inputs to the circuit, with each DIP restricted to at most one incorrect input. Since SARLock produces output corruption only when the input pattern matches the incorrect key sequence, the attacker adds an additional logic block to invert the corrupted outputs, thereby restoring the correct functionality of the circuit.

2.4.1.2 Anti-SAT

Anti-SAT leverages the complementary properties of circuits to enhance security. In an Anti-SAT locked circuit, identical input bits are fed into two complementary blocks. The key bits are split into two equal parts, which are then directed to an AND gate and a NAND gate, with their outputs subsequently fed into another AND gate. When the correct key is applied, the two key sequences are identical, causing the outputs of the AND and NAND gates to be complementary. As a result, the Anti-SAT block remains 1, and the circuit functionality remains unaffected. However, when incorrect key bits

are used, the outputs of the Anti-SAT blocks no longer remain complementary, leading to output corruption [2].

While Anti-SAT produces a stronger Distinguishing Input Pattern (DIP) than SARLock, it is still significantly weaker compared to pre-SAT locking strategies. The number of iterations required to break the logic locking in Anti-SAT is shown in the formula below. In the formula below, $2k$ represents the total number of key bits, as shown in Figure 2.8. AntiSAT produces the correct output when the two sets of key bits match each other, so the number of incorrect keys is $2^{2k} - 2^k$. The variable p refers to the number of incorrect keys that would induce output corruption for a single input pattern. This value can be adjusted between 1 and $2^k - 1$ by modifying circuit $g(x, k_{l1})$.

$$\frac{2^{2k} - 2^k}{p \times (2^k - p)} \tag{2.2}$$

Vulnerability of Anti-SAT

The Anti-SAT locking mechanism has been found to be vulnerable to the Signal Probability Skew (SPS) attack [6]. The SPS attack measures the probability skew of a gate outputting a logic '1'. The Anti-SAT locking mechanism consists of a multiple input AND gate and a multiple input NAND gate. As a consequence of this design, the absolute difference of the probability skew (ADS) of a flipping gate, denoted as gate G in Figure 2.8, is much higher than all the other gates in the circuit. By locating and attacking gate G, the attacker is able to prevent the circuit from producing output corruption when an incorrect key is applied.

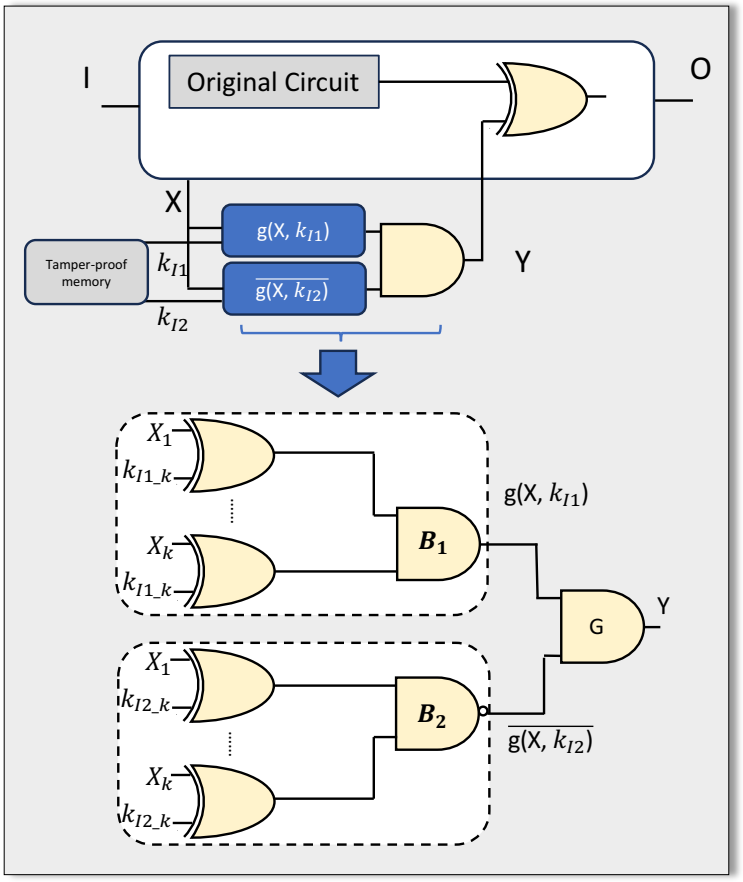

FIGURE 2.8: Original circuit with Anti-SAT block and An instance of Anti-SAT circuit

2.4.1.3 ANDTree

Similar to SARLock and Anti-SAT, ANDTree aims to enhance SAT resilience by increasing the number of iterations required for a successful attack. However, unlike these strategies, ANDTree adopts a different approach by inserting key gates—similar to pre-SAT logic locking methods—instead of incorporating an additional comparison block [37].

ANDTree utilizes a non-decomposable tree structure that outputs either `'1'` or `'0'` for a specific input pattern. Typically, an AND tree produces a `'1'` output only when all input bits are `'1'`. However, ANDTree logic locking does not necessarily require an AND tree; other non-decomposable structures, such as an OR tree, can also be employed. An OR tree operates similarly, outputting `'0'` only when all input bits are `'0'`. Importantly, all input signals in a non-decomposable tree must have a single fan-out, as shown in

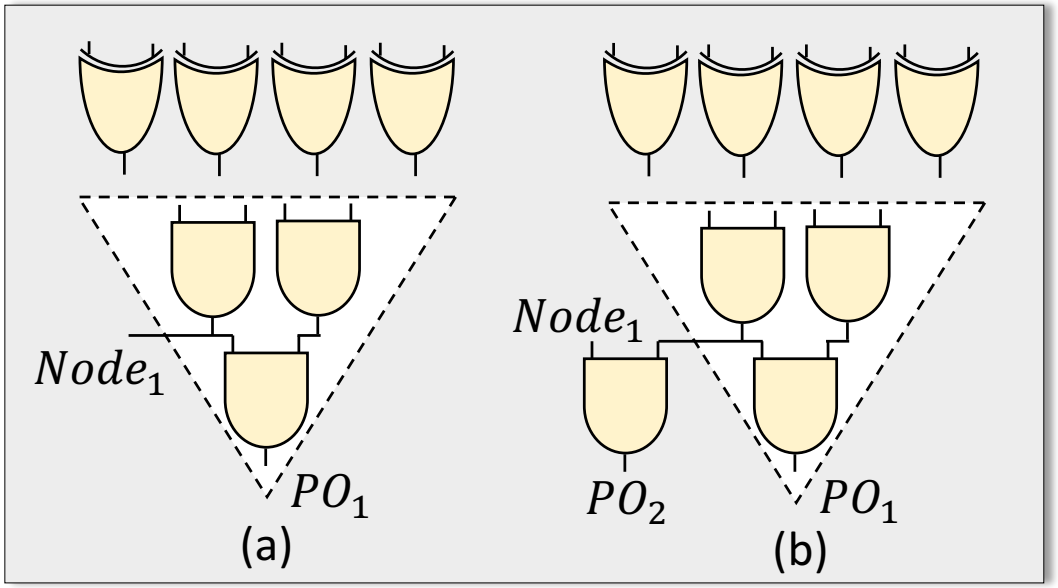

FIGURE 2.9: (a) non-decomposable ANDTree (b) decomposable ANDTree

Figure 2.9.a. Additional fan-outs compromise the security of the system, as they may allow attackers to extract more information and reduce the SAT resilience of ANDTree.

During the ANDTree locking process, designers must identify the largest non-decomposable tree in the circuit and insert key gates at all of its inputs. When a SAT attack is launched, the attacker can eliminate multiple incorrect keys only if a specific input pattern is used as a DIP, such as `'1111...1'` for the AND tree. When other input patterns are applied, eliminating incorrect keys becomes more challenging, as the non-decomposable tree may produce the same output for multiple patterns. This characteristic provides ANDTree with high SAT resilience but results in lower output corruption.

Vulnerability of ANDTree

Compared to other post-SAT logic locking strategies, ANDTree exhibits significant advantages, including lower overhead and strong SAT attack resilience. However, there are two main disadvantages associated with ANDTree. First, the locking strategy requires the circuit to contain a sufficiently large ANDTree structure to achieve adequate SAT resilience. Second, the ANDTree structure is easily identifiable, which enables attackers to locate and remove the inserted key gates once the corresponding structure is detected in the circuit. This compromises the overall security of the locking strategy.

2.4.1.4 TTLock

TTLock (Tenacious and Traceless Logic Locking) adopts a similar approach to SAR-Lock in enhancing SAT resilience. It compares the key input with the primary input of the circuit, introducing output distortion only when the two bit sequences match. Furthermore, in TTLock, slight modifications are made to the original circuit to ensure that output distortion occurs only under specific input patterns [46].

For example, in the logic cone shown in Figure 2.10, gate G1 is modified from an OR gate to an XOR gate, causing the output Y to be distorted only when the input pattern is `'110'`. This selective output corruption strengthens the defence while preserving the circuit's correct functionality under normal conditions.

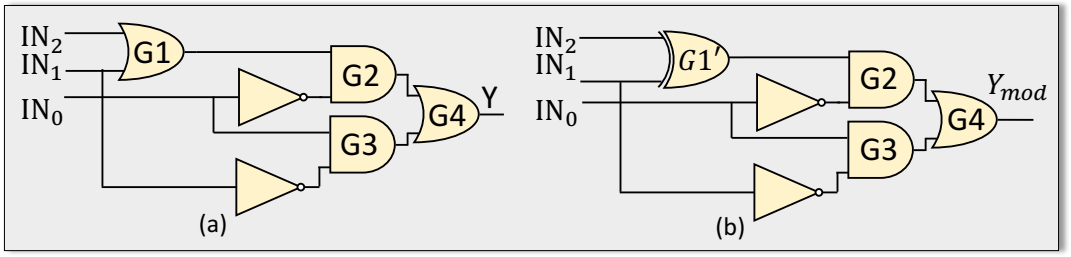

FIGURE 2.10: (a) Original logic cone. (b) Modified logic cone

When the input pattern is `'110'`, a flip signal is activated to cancel the distortion, but only when the correct key is applied to the circuit. For all other input patterns, the circuit produces distorted output only if the key sequence matches the input sequence.

The modification of the original circuit prevents attackers from recovering the correct functionality, even if they succeed in removing the restore logic.

TABLE 2.3: TTlock truth table

Y_{mod}	IN	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
✓	0	X	✓	✓	✓	✓	✓	✓	✓
✓	1	✓	X	✓	✓	✓	✓	✓	✓
✓	2	✓	✓	X	✓	✓	✓	✓	✓
✓	3	✓	✓	✓	X	✓	✓	✓	✓
✓	4	✓	✓	✓	✓	X	✓	✓	✓
✓	5	✓	✓	✓	✓	✓	X	✓	✓
X	6	X	X	X	X	X	X	✓	X
✓	7	✓	✓	✓	✓	✓	✓	✓	X

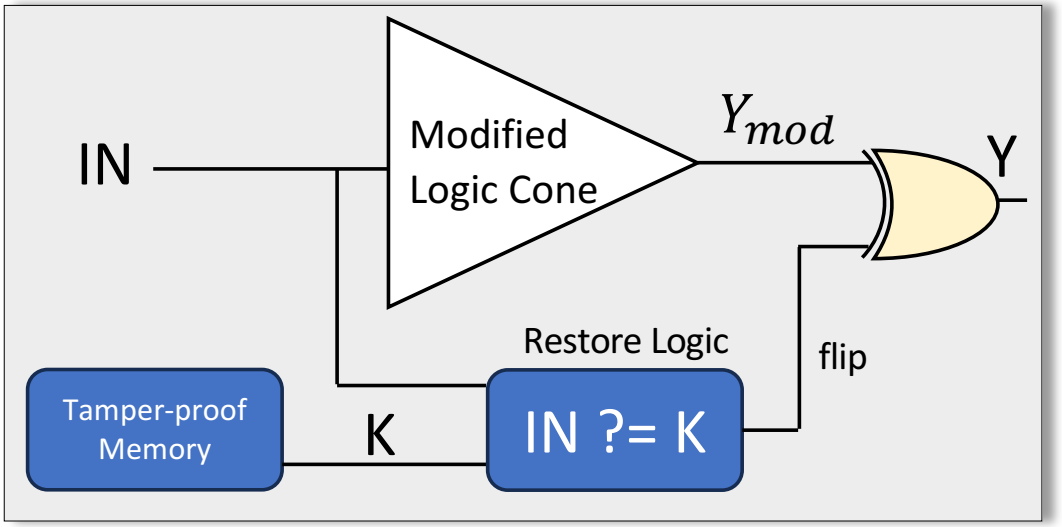

FIGURE 2.11: TTLock architecture and corresponding truth table

2.4.1.5 SFLL

SFLL stands for Stripped-functionality logic locking. The strategy strip some part of circuit functionality when wrong key is applied [3, 47, 48]. The basic version of SFLL is the same as TTlock, where specific input pattern is protected and every other input pattern produce output distortion only for one incorrect key.

SFLL stands for Stripped-Functionality Logic Locking. This strategy removes a portion of the circuit's functionality when an incorrect key is applied [3, 47, 48]. The basic version of SFLL operates similarly to TTLock, in which a specific input pattern is protected, and all other input patterns produce output distortion only for one incorrect key.

There are two variants of SFLL: SFLL-HD and SFLL-flex.

HD stands for Hamming Distance and is applied to both the protected input pattern and the incorrect keys. Unlike TTLock, which protects only a single input pattern, SFLL-HD protects an input cube whose Hamming distance to a specific input pattern is h. SFLL-HD0 functions similarly to TTLock, while SFLL-HD1 is illustrated in Figure 2.12. Input patterns `'2'`, `'4'`, and `'7'`, which are at a Hamming distance of 1 from `'6'`, are protected. For each of these inputs, incorrect keys with a 1-bit Hamming distance generate output distortion.

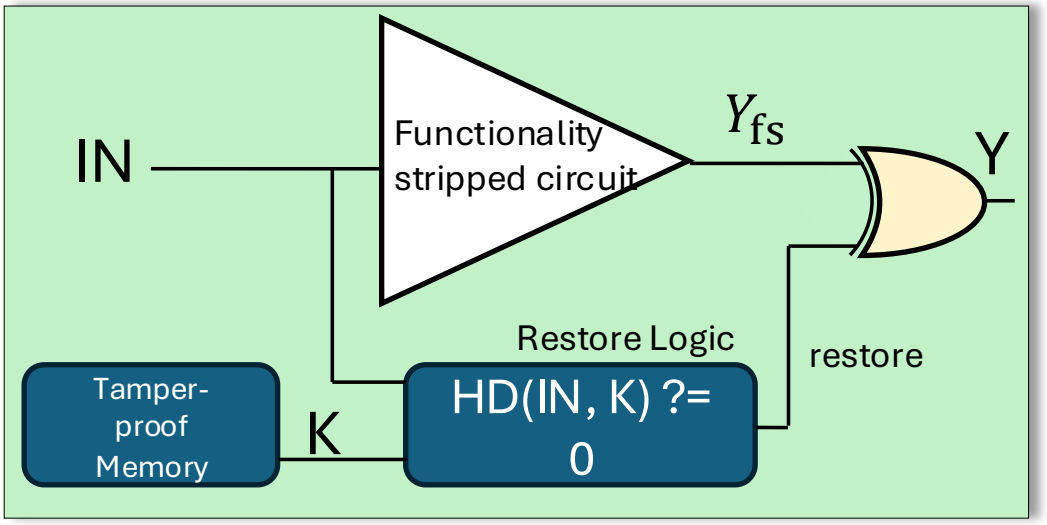

FIGURE 2.12: SFLL architecture

SFLL-flex allows users to select specific input patterns to protect, which is useful for IPs with critical inputs such as specific addresses, instructions, or data [49, 50, 51]. The protected input patterns are compressed and stored in LUTs. Compared to SFLL-HD, SFLL-flex incurs higher overhead but achieves greater output corruption.

Vulnerability of SFLL

The SFLL technique aims to prevent removal attacks by stripping a part of the function. However, this approach leaves a single protected input pattern in one input cube that produces output corruption for all incorrect keys. Sensitivity attacks uses input pattern sensitivity to break the SFLL. If the number of all the bit sequences of Hamming distance 1 from specific input pattern I_s is k_0, and among them there are k_1 bit sequences produce different output from I_s, the value k_1/k_0 is defined as sensitivity of input pattern I_s [52]. In SFLL-protected circuits, the protected input pattern exhibits much higher sensitivity than others. Attackers can use sensitivity quantification circui to detect the protected input pattern based on this characteristic and use it to remove the SFLL protection.

2.4.2 FSM/Sequential Logic Locking

2.4.2.1 Working Principle

Most early logic locking techniques focus on protecting combinational logic. In contrast, FSM or sequential logic locking aims to protect the entire circuit by modifying its state

behavior. Instead of locking a single logic stage, FSM locking operates on the state transition level and affects the sequential execution of the design.

Existing FSM-based logic locking techniques modify the original state transition graph in several different ways. One common approach is to add extra states, such as locking or authentication states, which must be correctly traversed before normal operation is enabled [53]. Another approach introduces trap states that force the circuit into incorrect behavior when an invalid sequence is applied [54]. Fake states can also be added to confuse attackers and hide the real functional states [55]. Some techniques modify critical state paths, which increases timing complexity and makes analysis more difficult [56]. In more recent work, key-controlled transitions are introduced so that the connection between states depends on secret key bits, combining FSM locking with traditional key-based logic locking [57].

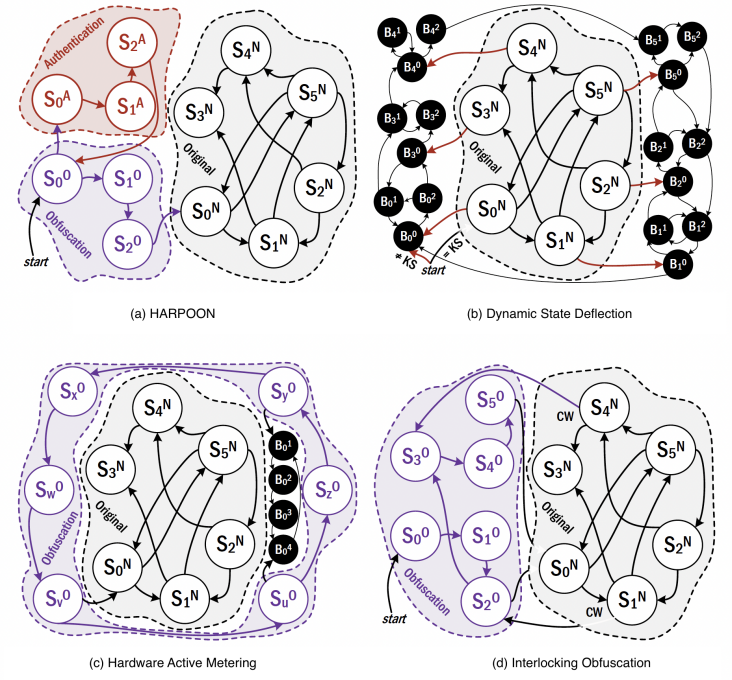

FIGURE 2.13: Examples of FSM-based logic locking strategies [7]

2.4.2.2 Advantages

FSM logic locking can reduce the need for explicit key storage, since the locking information is embedded in the state behavior of the circuit. This helps lower hardware overhead and reduces the risk of key exposure through dedicated key inputs.

2.4.2.3 Disadvantages

The main drawback of FSM logic locking is the significant overhead introduced by additional states and transitions. This increases circuit complexity and places extra pressure on synthesis and verification tools. To address these issues, several recent approaches [57, 58, 59] combine FSM-based locking with key-based logic locking in order to balance security and implementation cost.

2.4.2.4 Corresponding Attacks

Several attacks have been proposed to break FSM-based logic locking. One of the earliest is the two-stage attack [60], which analyzes the structure of the locked FSM and reconstructs the original design using its behavioral information. Other attacks, including RANE [61], Fun-SAT [62], and ORACALL [63], as well as more recent methods [64, 65, 66, 67, 68], specifically target the trap states introduced by FSM locking.

2.4.2.5 Summary

In summary, FSM logic locking provides strong protection against SAT-based and other key-recovery attacks by exploiting the sequential nature of the circuit. However, this security comes at the cost of high overhead. Although dedicated attacks against FSM locking exist, their impact is mainly limited to sequential locking schemes and does not significantly affect conventional key-based logic locking.

2.4.3 Cyclic-based Logic Locking

2.4.3.1 Working Principle

Cyclic-based logic locking protects a circuit by deliberately introducing feedback loops into the logic design [69, 70, 71, 72, 73, 74, 75, 76, 77, 78]. These cycles are controlled by key-dependent gates, so that correct functionality is only achieved when the correct key is applied. Although most commercial CAD tools are designed for acyclic logic and do not support combinational cycles, such structures can still be created through manual intervention during the design process.

2.4.3.2 Advantages

Cyclic-based logic locking is typically applied at the gate level, which makes it relatively straightforward to integrate into existing designs. It has shown strong resistance to SAT-based attacks and produces severe output corruption when the circuit is activated with an incorrect key.

2.4.3.3 Disadvantages

A major limitation of this technique is the lack of support for cyclic logic in most EDA tools. Even when cycles are inserted manually, they may lead to synthesis, timing, or verification issues, increasing the risk of design errors.

2.4.3.4 Corresponding Attacks

From an attack perspective, cyclic structures complicate the application of SAT attacks because standard CNF conversion assumes a directed acyclic graph representation. To address this, CycSAT [79] introduces cycle-avoidance clauses that restrict the SAT solver from generating cyclic assignments. More recent attacks further monitor the solver runtime to prevent it from being trapped in repeated cyclic explorations [80, 81].

2.4.3.5 Summary

Cyclic-based logic locking provides strong protection by exploiting the difficulty of analysing cyclic structures, resulting in high output corruption and good resistance to SAT-based attacks. However, practical adoption is limited by poor EDA tool support and the increased likelihood of design complexity and implementation errors. As a result, despite its security benefits, cyclic-based locking remains challenging for large-scale industrial deployment.

2.4.4 Routing-based logic locking

2.4.4.1 Working Principle

Routing or LUT-based logic locking leverages Look-Up Tables (LUTs) as key gates for circuit protection. Theoretically, an N-input LUT can implement up to 2^{2^N} possible functions. In this form of logic locking, certain gates in the original circuit are replaced by LUTs with the same number of inputs, as shown in Figure 2.14. The correct configuration for the LUT, which ensures it performs the same function as the original gate(s), serves as the correct key. Various approaches have been developed to enhance the attack resilience of LUT-based locking. [82]

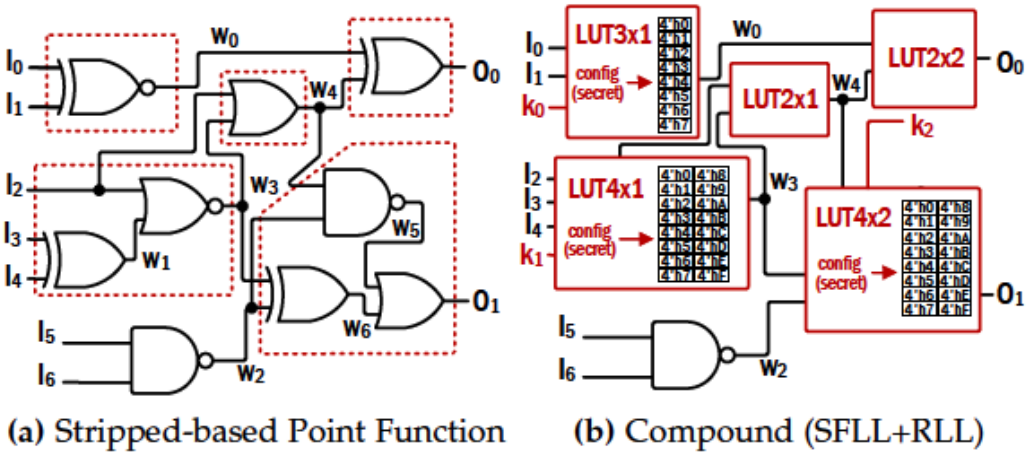

FIGURE 2.14: LUT-based logic locking [7]

Increasing the size of the LUT significantly escalates the complexity of the circuit. In LUT-based locking, larger LUTs (with additional inputs acting as key bits) are introduced to make the protected circuit more difficult for attackers to break. Naturally, adding more LUTs increases overall resilience. The specific placement of these LUTs within the circuit also greatly impacts its robustness. Recent research has explored the use of alternative gates, such as multiplexers (MUX) and FPGAs, within routing-based logic locking to further improve protection.

2.4.4.2 Advantages

LUT/routing-based logic locking strategies can be applied across different abstraction layers of the design. Additionally, this method provides substantial protection and high

output corruption rates when incorrect keys are used.

2.4.4.3 Disadvantages

However, one major drawback of LUT/routing-based logic locking is the significant overhead introduced by the LUTs. To enhance protection, more complex and larger LUTs must be incorporated, leading to substantially higher overhead compared to XOR/XNOR-based locking techniques.

2.4.4.4 Corresponding Attacks

Due to its unique structure, routing-based logic locking is difficult to solve when converted into Conjunctive Normal Form (CNF). To bypass its protection, attackers have developed methods such as the Bounded Variable Addition (BVA) approach, which reduces the complexity of the resulting SAT problem, making the time required for solving more manageable.

2.4.4.5 Summary

In summary, while routing-based logic locking offers strong resilience against attacks and ideal output corruption, techniques like BVA have made it vulnerable to SAT attacks once again. Moreover, the considerable overhead introduced by LUTs diminishes its appeal compared to other logic locking strategies.

2.4.5 Scan Chain Logic Locking

Working Principle

In IC industry, scan chains are extensively used to facilitate testing and debugging by providing full controllability and observability of internal registers and states, which, however, poses significant security risks. Scan Chain Logic Locking addresses this issue by securing the scan chain architecture itself [83, 84, 85, 86, 87, 88]. This technique modifies the scan chain to restrict unauthorised access to scan pins such as scan-in (`SI`), scan-enable (`SE`), and particularly scan-out (`SO`) [36, 89, 90, 91, 92]. By limiting access to

these pins, attackers are prevented from isolating and analysing smaller combinational logic blocks, forcing them to confront the complexity of the entire sequential circuit. Figure 2.15 illustrates a typical scan chain logic locking implementation.

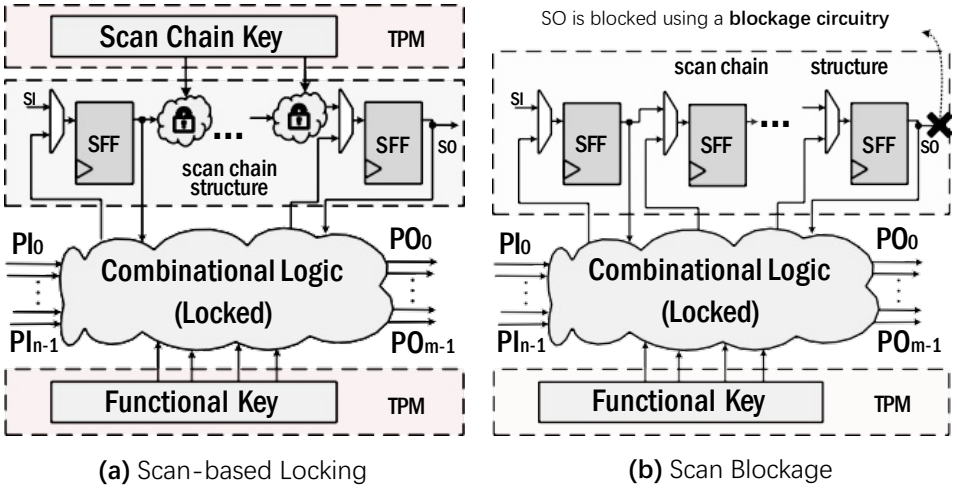

FIGURE 2.15: An illustration of Scan Chain Logic Locking

Scan chain logic locking techniques can operate in either static or dynamic modes. Static methods maintain a fixed locking configuration, while dynamic methods utilize components like Linear Feedback Shift Registers (LFSRs) or Pseudo-Random Number Generators (PRNGs) to change the locking configuration at runtime [83, 85, 93, 94]. Dynamic approaches increase the difficulty for attackers by invalidating any previously gathered information during the attack process.

Advantages

Scan chain logic locking is orthogonal to functional logic locking techniques and can be combined with them to enhance overall security. This combination significantly increases the complexity of potential attacks, particularly those based on satisfiability (SAT) solvers, by expanding the problem space from small combinational sub-circuits to the entire sequential circuit.

Disadvantages

One of the main drawbacks of scan chain logic locking is the potential negative impact on testability metrics such as test coverage. Modifying the scan chain can lead to reduced fault coverage and increased test time and complexity. Additionally, implementing scan

chain locking after Design-for-Testability (DFT) synthesis may introduce area and performance overheads. In some cases, after scan chain locking is applied, extra test pins are added to maintain high test coverage[36, 89, 90, 91], but this can significantly increase the die size of the chip.

Corresponding Attacks

Recent studies have revealed vulnerabilities in scan chain logic locking techniques. The ScanSAT attack[95] models locked scan chains as combinational circuits, allowing SAT-based methods to recover the key even in dynamic schemes like DOS [83]. DynUnlock[96] targets dynamic scan locking mechanisms such as EFF-Dyn [85] by reverse-engineering the PRNG to find its seed, thereby defeating dynamic key updates. Moreover, leakage-based attacks exploit flaws in scan blockage architectures like R-DFS[91]; attacks like shift-and-leak[89] and glitch-based shift-and-leak[90] can extract the key through primary outputs, even when shift operations are disabled.

Summary

In summary, scan chain logic locking enhances the security of ICs by protecting the scan chain architecture from unauthorised access. When combined with functional logic locking techniques, it provides a robust defence against various attacks. However, designers must carefully consider the trade-offs between security and testability, ensuring that the implementation does not significantly degrade test coverage or introduce new security flaws.

2.4.6 Other Logic Locking Techniques

Besides the locking strategies mentioned, other approaches like Behavioural Timing-based Locking, High-level Logic Locking, and eFPGA-based IP-level Locking have emerged also offer unique mechanisms to enhance circuit security against unauthorised access and tampering.

2.4.6.1 Working Principle

Behavioural Timing-based Locking

Behavioural Timing-based Locking secures both the functional and timing aspects of a circuit. Techniques such as Delay Logic Locking [97] introduce tunable delay elements whose delays depend on key values, creating ambiguity for attackers using standard CAD tools. Recent methods manipulate circuit timing through key-controlled clock gating and asynchronous latch-based designs [98, 99, 100, 101, 102, 103]. For example, latch-based approaches use key-controlled elements to asynchronously manage data storage timing in flip-flops. Without the correct key, data flow timing is disrupted, leading to functional corruption or system halts.

High-level Logic Locking

High-level Logic Locking secures design semantics by applying locking at higher abstraction levels, such as Register Transfer Level (RTL) or High-Level Synthesis (HLS) [104, 105, 106, 107, 108, 109, 110, 111, 112]. Methods include:

- **Locking before Synthesis:** Applying locking directly to high-level code (e.g., C/C++) before HLS [108].
- **HLS Extension:** Integrating locking mechanisms into HLS intermediate steps [105, 107, 112].
- **RTL Locking:** Applying locking to RTL code post-HLS or directly on designer-written RTL [106, 110, 111].
- **Compound Locking:** Combining high-level and gate-level locking techniques for enhanced security [109].

These methods target higher-order elements like constants, arithmetic operations, control flow, and memory access, providing comprehensive protection.

eFPGA-based IP-level Locking

eFPGA-based IP-level Locking employs embedded Field Programmable Gate Arrays (eFPGAs) within a System on Chip (SoC) to protect intellectual property modules [113, 114, 115, 116]. Selected modules are replaced with reconfigurable eFPGAs programmed by a secret bitstream. Attackers must recover the complete bitstream to replicate functionality, significantly enhancing security.

Advantages

The behavioural timing-based Locking obscures data capture timing[1], complicating circuit analysis while allowing the scan chain to remain accessible for testing and debugging. The approach of high-level logic locking protects against a broader range of threats, including insider attacks within the design house. Early-stage locking leverages synthesis optimisations to intertwine the locking mechanisms with the original design, increasing difficulty for attackers. As for eFPGA-based locking, it makes structural attacks are difficult due to the uniform architecture of eFPGAs. The large bitstream size and full configurability increase resilience against I/O query-based attacks, making it an SAT-hard problem.

Disadvanges

Implementation of behavioural timing-based Locking challenges arise due to limited EDA tool support for asynchronous designs. Replacing flip-flops with latches increases design complexity, making it less practical for large-scale SoCs. Challenges of high-level locking include tool immaturity and increased design complexity. Integrating locking at higher abstraction levels may impact performance and require adjustments to the entire design workflows, causing unacceptable financial burden. And the primary drawback of e-FPGA approach is substantial overhead in area, power, and performance compared to traditional methods.

Summary

Behavioural Timing-based Locking, High-level Logic Locking, and eFPGA-based IP-level Locking represent advanced strategies that extend protection beyond traditional methods. Behavioural Timing-based Locking secures timing properties, introducing ambiguities that challenge adversaries but faces practical implementation issues due to limited EDA tool support. High-level Logic Locking protects designs at higher abstraction levels, targeting semantic elements to enhance security against a wider range of threats, though it may complicate design workflows. eFPGA-based IP-level Locking provides robust protection by replacing critical modules with reconfigurable eFPGAs, making attacks exceedingly difficult but incurring significant overhead. Collectively, these techniques offer enhanced security features but require careful consideration of trade-offs in implementation complexity, design overhead, and practical feasibility.

2.5 ML-Based Attacks

In recent years, machine learning (ML)-driven attack and defence strategies have advanced rapidly, with ML-based attacks mainly targeting the recovery of the correct key or the restoration of a circuit's original gate-level structure by removing inserted locking gates.

SAIL [117, 118] is the first ML-based attack introduced against logic locking, aiming to remove locking gates and restore the circuit to its original unlocked state. SAIL only requires the locked gate-level netlist and does not necessitate access to a functional IC product. During the logic locking process, after locking gates are inserted into the circuit, it undergoes resynthesis to disguise the locked gates. While different CAD tools use varying synthesis rules in this process, the creators of SAIL discovered that only a small subset of these rules is commonly used, adhering to the 80/20 rule . This predictable behaviour can be learned and predicted by AI models. In an SAIL attack, the locked circuit serves as training data, closely resembling the original unlocked circuit. The attacker inserts additional locking gates to train a neural network capable of removing these gates, effectively restoring the original circuit. In benchmark tests, SAIL achieved an accuracy rate of up to 95%.

However, SAIL has two main limitations. First, it is effective only against XOR/XNOR-based logic locking techniques and fails to work with other locking methods, such as MUX-based locking or logic locking using locking blocks. Second, the attacker must be familiar with the specific synthesis rules of the CAD tool used by the designer, as this knowledge is crucial for successfully training the neural network. Subsequent attacks, such as Snapshot [119] and GNNUlock [120], overcame these limitations. Snapshot eliminates the need for CAD tool synthesis rules and directly targets the secret key, broadening its scope to include MUX-based logic locking. In contrast, GNNUlock employs Graph Neural Networks (GNNs) to reconstruct the original circuit's netlist connections and functionality, making it effective not only against traditional gate-based locking methods but also against more advanced techniques like SFLL and AntiSAT, which use locking blocks.

An alternative ML-based attack targets routing-based logic locking [121, 122, 123]. Unlike structural ML-based attacks that focus on recovering hidden gate structures, this

method uses GNNs to predict connections between concealed locking gates, such as those used in D-MUX-based locking [124].

2.6 Summary

Various state-of-the-art logic locking strategies and corresponding attacks are introduced in this chapter. Among all the logic-locking-targeted attacks, the SAT attack has played a pivotal role in shaping the development of logic locking techniques. Since its introduction, nearly all newly developed locking strategies have prioritized resilience against SAT-based attacks. However, each of these techniques comes with certain limitations. Some impose significant overhead, others introduce errors during the testing phase, and many lack full compatibility with existing Electronic Design Automation (EDA) tools. Additionally, certain methods may disrupt the overall design flow or lead to a considerable increase in die size. A comparative analysis of post-SAT locking strategies is provided in Table 3.1, highlighting their respective advantages and drawbacks.

2.6.1 Logic Locking Strategies Comparison

Term	Merits	Drawbacks
PFB	Low overhead; easy to implement	Vulnerable to Bypass attack
FSM Locking	No memory needed for key-storage	Increased complexity of circuit; high overhead caused by flip-flops added
Cyclic-based Locking	Easy to implement on gate-level; high output corruption	Not supported by EDA tools; manual implementation might cause errors
Routing-based Locking	Can be implemented at different abstract levels of design; high output corruption	Bring significant overhead

Scan-chain Locking	Work well with other locking techniques; significantly increase the complexity of design for attackers to break	Potential negative impact on testability by reducing fault coverage, increasing test time and complexity; high overhead; introduce extra test pins
Behaviour Time-based Locking	High resilience against attack while not affecting scan chain	Limited EDA tool support; increased circuit complexity
High-level Locking	Wider range of protection in supply chain	Might cause problem to the entire design flow, leading to unacceptable financial burden
eFPGA-based Locking	High resilience against gate-targeted attacks	High overhead caused by FPGA insertion; harder to implement

Table 2.4: Merits and Drawbacks of Locking Strategies

Among these approaches, Point-Function-Based (PFB) logic locking stands out as one of the most balanced techniques. Although it may not offer the highest resilience against every form of attack, it demonstrates strong defence capabilities against SAT and related attacks. PFB logic locking is relatively straightforward to implement and has minimal impact on the overall system performance, making it a practical and well-rounded solution in modern hardware security.

Chapter 3

ZeKi: A Zero-Knowledge Dynamic Logic Locking Implementation with Resilience to Multiple Attacks

Although logic locking emerged as a solution to safeguard the vast majority of general ICs, traditional logic locking methods, including some state-of-the-art approaches, are vulnerable to Satisfiability (SAT) attacks [1, 125] which iteratively deduce the correct keys to break the logic locking. While the latest Point Function-Based (PFB) logic locking techniques [2, 3, 4] can defend against SAT attacks, they are susceptible to various structural attacks [6, 5] due to its own defects. A structural attack targets the specific vulnerabilities of a given logic locking strategy. These vulnerabilities typically arise from the unique structure of the locking block introduced by logic locking schemes that rely on a single locking mechanism.

Additionally, there is a potential threat from internal IC designers who are well-versed in logic locking.

To address these threats by dynamic way with less overhead, I propose a comprehensive logic locking design and implementing framework called ZeKi: Dynamic Logic Locking (DLL) by 'Zero-Knowledge' implementation, an adaptive approach that customises the

implementation of diverse locking mechanisms with the lower overhead for each unique design. The implemented DLL provides defence against all attacks mentioned above. To our best knowledge, ZeKi is the first logic locking strategy to implement 'zero-knowledge' feature which effectively safeguards the design against potential malicious attacks from the design team. I demonstrate the effectiveness of DLL and its resilience against different attacks through experimental evaluations.

ZeKi's DLL matches SFLL-HD0 in SAT attack resistance, with logarithmically increasing DIPs; and a superior average Hamming difference of 89.5 compared to zero in other schemes on structural attack resistance. It also offers five times the resistance of RLL against sensitisation attacks. Zeki's DLL incurs lowest average power and area overheads of 4.27% and 4.48%, respectively.

This chapter introduces the implementation of Zeki in combinational circuits. The following chapter will address its application to sequential circuits. The structure of this chapter is organized as follows: Section 3.1 presents the motivation behind the proposed Zeki logic locking strategy. Section 3.2 outlines the key contributions of Zeki. Section 3.3 explains the working principles of the Zeki strategy. Section 3.4 summarizes the workflow for implementing Zeki in a protected circuit. Section 3.5 presents the experimental setup and results for Zeki-locked circuits. Finally, Section 3.6 provides a summary of the chapter.

Parts of this chapter are based on our previously published work [126]. Some text and figures are reused with permission.

3.1 Motivation

In this section the motivation of Zeki (Zero-Knowledge Dynamic Logic Locking) will be introduced including DLL (dynamic logic locking) block, which is proposed to thwart the threat from various present and future attacks and zero-knowledge locking which prevents attack from malicious insiders.

3.1.1 Logic Locking Strategies Comparison

As introduced in section 2.4, different logic locking techniques are proposed as hardware security strategies. As shown in table 3.1, various logic locking techniques have been developed to counter SAT attacks and other threats targeting logic locking. However, each of these techniques has its drawbacks. Some introduce significant overhead, others may cause errors during the testing phase, and many are not fully supported by existing EDA tools. Additionally, some techniques can negatively impact the overall design flow or substantially increase the die size.

Term	Merits	Drawbacks
PFB	Low overhead; easy to implement	Vulnerable to Bypass attack
FSM Locking	No memory needed for key-storage	Increased complexity of circuit; high overhead caused by flip-flops added
Cyclic-based Locking	Easy to implement on gate-level; high output corruption	Not supported by EDA tools; manual implementation might cause errors
Routing-based Locking	Can be implemented at different abstract levels of design; high output corruption	Bring significant overhead
Scan-chain Locking	Work well with other locking techniques; significantly increase the complexity of design for attackers to break	Potential negative impact on testability by reducing fault coverage, increasing test time and complexity; high overhead; introduce extra test pins
Behavioural Time-based Locking	High resilience against attack while not affecting scan chain	Limited EDA tool support; increased circuit complexity
High-level Locking	Wider range of protection in supply chain	Might cause problem to the entire design flow, leading to unacceptable financial burden

eFPGA-based Locking	High resilience against gate-targeted attacks	High overhead caused by FPGA insertion; harder to implement

Table 3.1: Merits and Drawbacks of Locking Strategies

Among these approaches, Point-Function Based (PFB) logic locking stands out as the most balanced technique. Although it may not offer the strongest resilience against all forms of attacks, it effectively defends against SAT attacks and other threats. PFB logic locking is relatively easy to implement and has a minimal impact on the design's overall performance, making it one of the most practical and well-rounded options available.

3.1.1.1 Vulnerability of PFB

However, Point-Function Based (PFB) logic locking also has its limitations.

There are two countermeasures for mitigating SAT attacks: increasing the number of iterations or extending the duration of each iteration. PFB logic locking adopts the first strategy, aiming to increase the overall computing time required for the SAT attack. Functionally, PFB logic locking incorporates a judgment module into ICs. For each input pattern, only one or a small number of incorrect keys will cause the circuit to malfunction; in other cases, incorrect keys do not result in erroneous outputs. Since each iteration of a SAT attack can only eliminate the keys that cause output errors for a specific input pattern, each iteration discards only a small number of incorrect keys. Consequently, this significantly prolongs the time required for a SAT attack to succeed, thereby enhancing the ICs' security.

The locking mechanism of three PFB logic locking strategies: SARlock, Anti-SAT and SFLL are shown in Figure 3.1. The figure clearly shows that the structures of the three presented Point-Function Based (PFB) logic locking techniques are highly similar. In each case, the functionality of the logic locking is provided by a locking block, represented by the light blue box in the diagram. This locking block receives both the circuit's input bits and the key bits as inputs.

Within the locking block, a validation mechanism, which can be temporarily referred to as "mechanism V," is applied. The purpose of this mechanism is to ensure that for each input pattern, only one or a very limited number of incorrect keys will result in erroneous outputs. In other words, "mechanism V" is designed to maintain the security of the circuit by minimizing the number of input patterns that can identify and eliminate multiple incorrect keys during SAT attacks.

By focusing on restricting the effectiveness of incorrect keys to as few input patterns as possible, PFB logic locking enhances its resilience against attacks. This approach is central to PFB's effectiveness, as it complicates the attacker's ability to identify correct keys through systematic testing of inputs. Thus, the core principle behind these techniques is to create a mechanism that restricts the propagation of errors, making it extremely difficult for attackers to use distinguishing input patterns (DIPs) to break the locking.

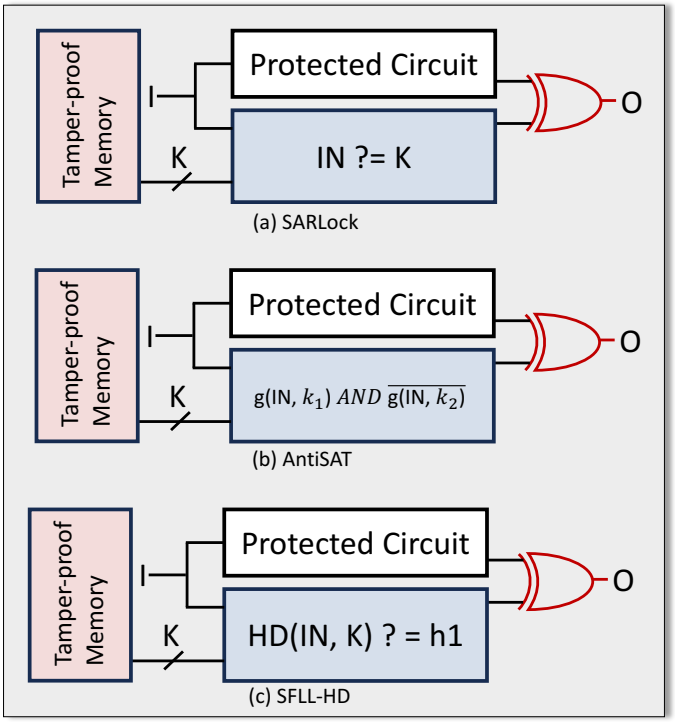

FIGURE 3.1: PFB locking with Different Validation Mechanisms

As illustrated in Figure 3.1, the defensive strength of PFB locking relies on its locking block, each operating based on a specific mechanism. These mechanisms confer inherent structural or parameter characteristics upon the locking block, which can easily be

exploited by attackers to develop targeted attacks. For instance, SARLock, which generates function distortion based on the equality of input and key, cannot resist approximate attacks [45]. Similarly, Anti-SAT's flip gate, which produces a high ABS value, becomes vulnerable to SPS attacks [6]. Likewise, the high sensitivity value generated by the protected input cube in SFLL renders it susceptible to sensitive attacks, potentially leading to the leakage of the secure key [5].

However, these attacks are only effective against their respective PFB locking techniques, as they exploit the properties specific to each locking mechanism. This raises the question: **Could a PFB technique employing a variable or dynamic locking mechanism, rather than a fixed one, circumvent these types of attacks?**

Optimistically, the attacks mentioned earlier specifically target individual PFB (SARLock vs bypass attack; Anti-SAT vs SPS attack; SFLL vs sensitivity attack) locking strategies rather than the entire class of PFB techniques. They exploit structural weaknesses unique to each PFB implementation. Therefore, in theory, if a new PFB locking strategy is developed based on an entirely novel validation mechanism, existing attacks that target specific PFB vulnerabilities should become ineffective against it. However, can this truly be considered a definitive solution?

Unfortunately, it would be highly unlikely. Introducing a new validation mechanism may simply initiate a familiar cycle: a new locking strategy is developed, research papers are published, and attackers then analyse the specific structure and validation mechanism, eventually finding its weaknesses within weeks, thereby breaking its defences. This cycle is almost inevitable with PFB locking strategies that rely on a single validation mechanism. Such a mechanism often reveals specific structural vulnerabilities that attackers can exploit. Once these weaknesses are identified, all products that employ this particular locking strategy risk losing their protection.

This brings us to an important question: **if relying on a single validation mechanism inherently poses security risks, what about PFB locking strategies that use multiple validation mechanisms?** By incorporating diverse mechanisms, these strategies may be able to distribute the locking validation across different structural components, potentially making it more challenging for attackers to identify and exploit

a single point of weakness. A multi-mechanism approach could introduce a level of unpredictability and complexity, thereby enhancing resilience and reducing the likelihood of universal vulnerabilities that affect all implementations of the locking strategy.

Untrusted Insider Additionally, one of the advantages of logic locking is its broad protection scope, extending from the logic synthesis stage to the final end-user. However, the initial design phase remains uncovered, as designers are responsible for integrating logic locking into the design. Expanding the scope of protection to include the design phase would significantly enhance the overall security of the system.

Traditional logic locking approaches presume the trustworthiness of the design house. However, in practical scenarios, the risk of rouge insider within the design house itself cannot be disregarded. Currently, none of the logic locking techniques sufficiently address this type of insider threat. Due to this threat, the victim IC could be compromised from the very beginning of the supply chain, potentially affecting every stage from IC design to manufacture.

3.2 Contribution

In this thesis, I introduced **Dynamic Logic Locking (DLL)** with a **Zero-Knowledge Implementation** (ZeKi) as a novel approach to address the security concerns mentioned above. ZeKi utilizes a dynamic locking mechanism that diverges from conventional techniques, such as PFB logic locking, by employing randomly generated locking blocks to counter SAT attacks and other attacks targeting logic locking. Unlike traditional methods that rely on locking blocks based on singularly specific mechanisms, DLL generates locking blocks using a unique mechanism for each design, thereby eliminating specific structural vulnerabilities targeted by attackers. Additionally, the dynamic generation of locking blocks enables hardware designers to achieve zero-knowledge locking implementation, as they are not required to access the knowledge of the locking block's structure or secret key bits. ZeKi can effectively resist attacks, even if they originate from within the design house. This further enhances the confidentiality and security of the protected circuit. This chapter will concentrate on the work of Zeki in the combinational circuit.

The contributions of ZeKi are listed below.

1. I propose Zero-Knowledge Implementation of Dynamic Logic Locking (ZeKi, DLL), which effectively safeguards IP from multiple attacks, including sensitisation attacks, SAT attacks, and structural attacks, ML-based attack, bypass attack and approximate attack. ZeKi also enables hardware designers to achieve zero-knowledge logic locking implementation, further enhancing the security of logic locking protection.

2. I present successful implementations of DLL in various benchmarks, demonstrating the feasibility and practicality of integrating generated logic locking into digital designs.

3. I also provide a comprehensive evaluation of the resistance capabilities of ZeKi's DLL against various attacks, as well as the power and area overhead incurred by DLL in protected designs.

3.3 Working Principle of Zeki

Zeki mainly consists of three components: the insertion of SLL locking gates, which ensures that the circuit has a sufficiently high Output Error Rate (OER); the Key Verification Unit, which guarantees that the circuit will generate the correct output when the correct key is input; and the DLL block, which provides resilience against SAT attacks and other types of attacks. In the subsequent part of this section, I will elaborate on the working principles of these three components.

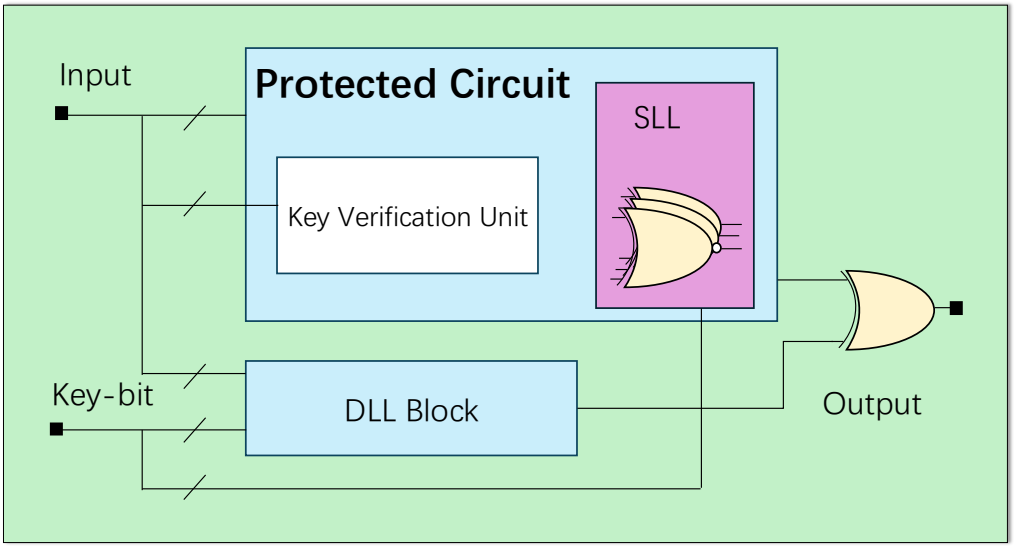

FIGURE 3.2: Main Architecture of Zeki

3.3.1 Dynamic Logic Locking

Dynamic Logic Locking (DLL) is proposed as a solution to overcome the limitations of PFB while providing enhanced resistance against SAT and other attacks. The locking block, which serves as the core component of PFB, can be regarded as a function. This function should ensure that each input pattern leads to output corruption for a small number of incorrect keys. Theoretically, as long as these criteria are met without causing excessive overhead or adverse effects on the design, the locking block does not need to adhere to a specific structure. DLL is precisely designed based on these principles.

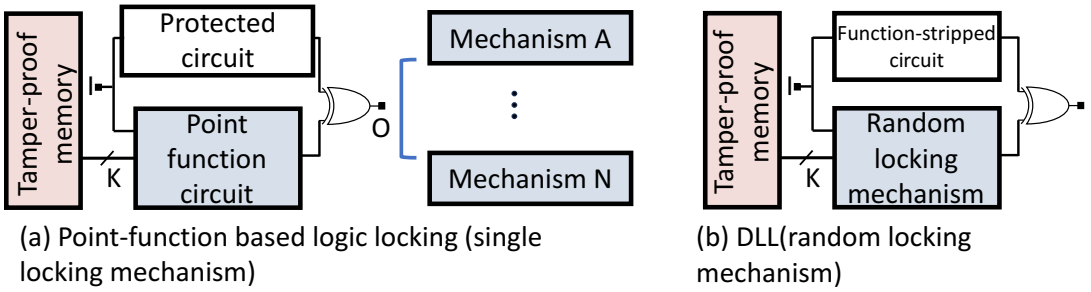

FIGURE 3.3: Conventional PFB locking(a) vs DLL(b)

Unlike applying a locking block of the same structure to all designs, DLL incorporates a randomly generated locking block for each unique design, as shown in Figure 3.3. Hardware designers only need to specify the length of the key bits and the desired level

of output corruption to ensure that the randomly generated locking block meets their security requirements. Due to the utilisation of a dynamic locking block, DLL exhibits two distinctive characteristics:

1. **Absence of specific structural vulnerabilities:** The lack of a fixed structure eliminates specific structural vulnerabilities that may arise in the locking block of Point function-based logic locking techniques. This enhances the overall security of DLL by making it impossible for attackers to break the protection by finding vulnerability of single locking mechanism.

2. **Protection against cross-design attacks:** Even if an attacker identifies weaknesses in a protected design, they cannot exploit these vulnerabilities to attack other designs. This is because the locking block for each protected design is unique and differs from one another. Consequently, the security of different designs remains intact even if one design is compromised.

3. **Zero-knowledge locking:** Due to randomly generated locking mechanism, the IP designers do not need to know its specifics, making ZeKi the first known logic locking technique that operates without locking knowledge. This extends the protection of the design flow to stages before logic synthesis, enhancing overall security.

In the subsequent sections of this chapter, I will demonstrate the working principle of DLL and its resilience against attacks. DLL primarily comprises three key components: inserted key gates, a locking block (DLL block in figure 3.2), and a mask.

The core component of ZeKi that enhances logic locking security is the Dynamic Logic Locking (DLL) technique. The locking block (DLL block in figure 3.2) in DLL consists of two primary components. The first is the mask, which connects to the key-bit input. Its primary function is to ensure that the circuit operates correctly when the correct key is inputted. The second part, referred to as the locking block for simplicity, is designed to ensure that the circuit exhibits a malfunction when an incorrect key is inputted. This component is crucial for DLL's resilience against SAT attacks, sensitisation attacks, and structural attacks.

PFB logic locking employs locking blocks to validate a predefined relationship between the input and key-bits, determining whether the circuit should produce an erroneous

output. This 'validation operation' checks for the uniqueness or near-uniqueness of the key concerning the input, ensuring that the protected circuit generates incorrect outputs for only one or a few incorrect keys for each input pattern, thereby enhancing resistance to SAT attacks. Take SARLock as an example, where the flip bit indicates whether functional corruption will be generated:

$$flip = (input[0:8] == key[0:8]) \wedge (key! = correctkey)$$

In this case, SARLock has a 9-bit input and a 9-bit key, with the predefined relationship being bitwise equality between the two. The incorrect key disrupts the output only when this condition is met. This mechanism can be equivalently represented as:

$$\begin{aligned} flip = (input[0:2] = key[0:2]) \wedge \\ (input[3:5] = key[3:5]) \wedge \\ (input[6:8] = key[6:8]) \wedge (key! = correctkey) \end{aligned}$$

In the formula above, the input/key sequence is divided into three groups of 3-bit chunks. Each block performs a bitwise equality check, and the overall validation is successful only when all three groups satisfy this condition.

Both operations are equivalent. Thus, it can be concluded that the validation function for a set of input/key can be broken down into a sum of validation operations of shorter input/key sequences. If these subdivided bit sequences simultaneously meet the validation operation, then the original input/key satisfies the validation operation.

Upon further consideration, it becomes apparent that the validation operations for each group in a logic locking mechanism do not need to be identical. The critical requirement is maintaining the uniqueness of the input/key relationship. Additionally, the number of bits allocated to each group does not need to be equal; it is sufficient if their total equals the length of the original input/key sequence. See the following example:

$$\begin{aligned} flip = (input[0:1] = key[0:1]) \wedge (input[2]! = key[2]) \\ \wedge (input[3:5] = key[4:5][3])(shift - 1 - bit) \wedge \\ (input[6:8] + key[6:8] == 5) \wedge (key! = correctkey) \end{aligned}$$

This principle underpins the workings of DLL, which randomly generates its locking mechanism. DLL divides the key/input sequence into several smaller chunks of 2-bit or 3-bit, shown in Figure 3.4. These chunks are then subjected to distinct random validation operations. The circuit only malfunctions when all these validation operations are simultaneously satisfied. This mechanism allows a fixed-length key to generate countless variations of validation operations, achieving dynamic logic locking.

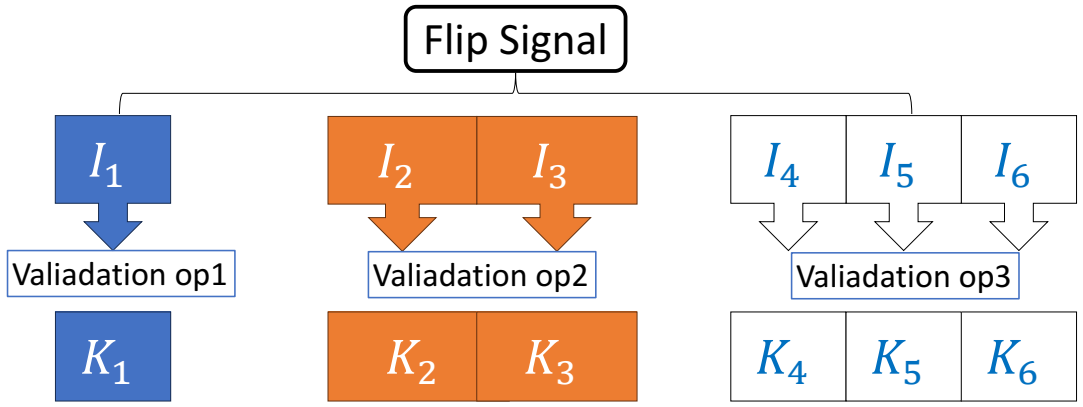

FIGURE 3.4: DLL breaks input/key sequence into chunks and assigns different validation operations to them.

DLL generates locking blocks that exhibit significant randomness in both functionally and structurally. Functionally, even for a 2-bit input/key chunk, there are 24 different validation operations that can ensure the uniqueness of the input/key relationship. For a 3-bit chunk, the possibilities increase even further. Structurally, even with identical validation operations, they can be implemented using various combinations of logic gates, as shown in Figure 3.5.

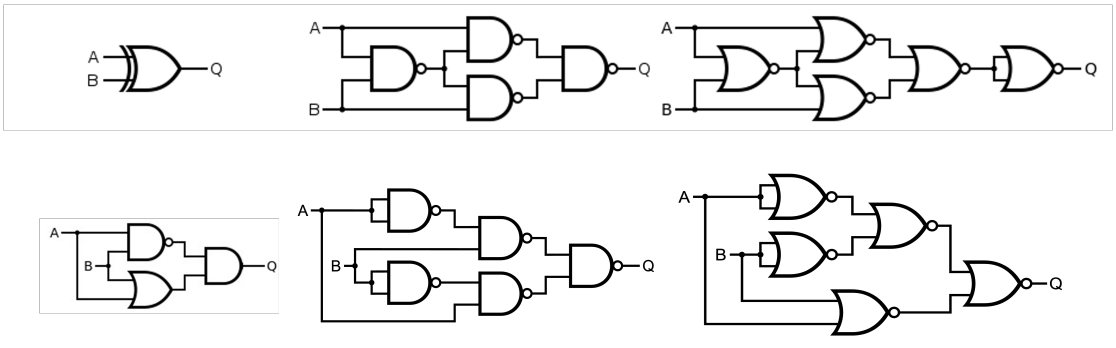

FIGURE 3.5: Different implementation for XOR operation

Thanks to the locking blocks mechanism, DLL can not only perform high resilience to SAT attack as PFB but also avoid suffering from structural attacks targeting its inherent structure or parameter characteristics. In addition, the circuit is also modified to ensure

that the primary output is influenced by a few randomly selected input bits and the flip signal, which prevents attackers from removing the locking block by setting the flip signal to a constant value. Ultimately, the flip signal generated by the locking block is connected to the design output. When an incorrect key and its corresponding input are applied to the circuit, the flip signal contaminates the final output, indicating the presence of output corruption.

3.3.2 Strong Logic Locking (SLL) Insertion

In this section, the insertion of SLL (strong logic locking) will be explained. Before the explanation of SLL insertion, however, the concept of OER will be introduced. **Output Error Rate (OER)** is closely related to the **Output Corruption (OC)** metric. While OC measures the average Hamming distance between the reference and observed outputs, OER quantifies the proportion of input patterns that result in incorrect outputs. Unlike OC, which accounts for the number of incorrect output bits, OER simply considers whether any of the M output bits are incorrect.

In the context of logic locking, the output error rate (OER) is a critical parameter. If the OER is too low, it means that even if an incorrect key is input into the circuit, the primary output is unlikely to exhibit errors. This significantly reduces the effectiveness of logic locking and allows malicious attackers, who may not have high accuracy requirements in their attacks, to exploit ICs they have illegally obtained. To defend against SAT (Satisfiability) attacks, many PFB (Point-function based) logic locking techniques adopt a strategy of increasing the number of iterations. This approach primarily focuses on reducing the effectiveness of DIP (Distinguishing input pattern), meaning that each DIP only generates output distortion for a single incorrect key. However, this strategy can result in the OER of a locked circuit becoming too low.

To prevent this issue and improve the overall OER of the locked circuit, Zeki integrates Strong Logic Locking (SLL) key-gates into the circuit. The number of key-bits dedicated to SLL is determined by user-defined parameters, allowing flexibility in the design.

$$L_{SLL} = L_{IN} * R_{SLL}$$

During the implementation of logic locking, Zeki randomly selects L_{SLL} of the key-bits from all available key-bits to serve as the SLL key-bits. These selected key-bits are used simultaneously in the Dynamic Logic Locking (DLL) block and as the key for the SLL gates.

By combining DLL and SLL, Zeki ensures that the protected circuit not only achieves high SAT resilience, but also maintains a sufficiently high OER, thereby preventing the issues associated with low output error rates. This dual approach strengthens the security of the circuit while preserving the integrity of its functionality.

3.3.3 Key Verification Unit

The incorporation of SLL (Strong Logic Locking) ensures that the protected circuit attains a sufficiently high OER (Output Error Rate). Meanwhile, the DLL (Dynamic Logic Locking) endows the circuit with defences against various attacks. Nevertheless, it should not be overlooked that it is of paramount importance to guarantee that the circuit can generate the correct output when the correct key is input. In Zeki, the Key Verification Unit (KVU) undertakes this crucial task.

To further enhance the security of the overall defence mechanism, the KVU solely utilizes the circuit's primary inputs rather than key-bits as its inputs. This approach effectively thwarts malicious external attackers from pinpointing the location of the KVU via key-bits, thereby precluding attacks such as removal attacks.

Based on the description of the working principle of the DLL in section 3.3.1, in order to withstand SAT attacks, for each input pattern, only one incorrect key will trigger an erroneous output. Similarly, for a correct key, there exists a corresponding input pattern that will prompt the DLL block to output a flip signal, resulting in output distortion of the circuit. In light of this fact, when the input is the input pattern I_1 corresponding to the correct key, the KVU intervenes to prevent the DLL block from generating the flip signal, thus ensuring that the circuit does not produce output distortion when the correct key is input.

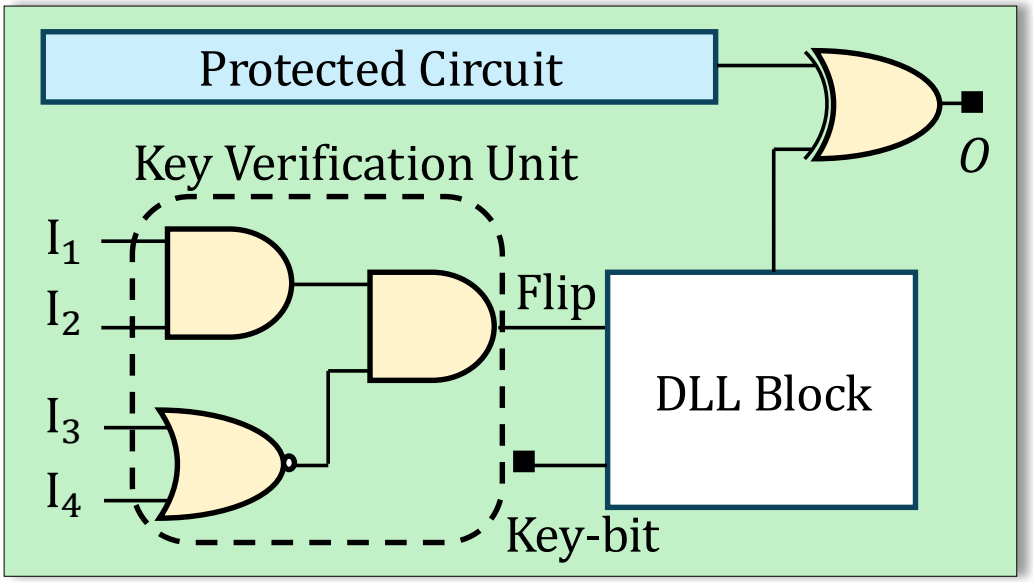

FIGURE 3.6: Key Verification Unit

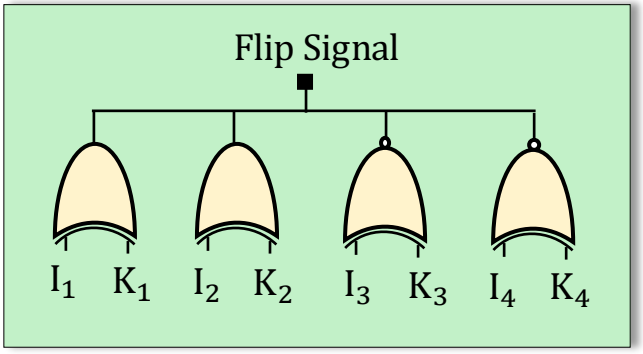

FIGURE 3.7: DLL Block of Key Verification Unit

Taking the Figure 3.6 and Figure 3.7 as an example, in this Zeki, according to the structure of the DLL block in Figure 3.7, if the correct key is '0000', the input pattern that generates the flip signal is '1100'. As depicted in the Figure 3.6, the KVU of the circuit will generate a flip signal when the input pattern is '1100' to prevent the DLL block from corrupting the circuit output.

3.3.4 Zero-Knowledge Locking Strategy

ZeKi is the first technique to achieve 'zero-knowledge locking' in logic locking. In previous logic locking techniques, although participants in the design flow post-logic synthesis were unaware of the locking information, personnel within the design house were informed. Moreover, once a logic locking technique was published, attackers could maliciously analyse its structural weaknesses for exploitation. However, ZeKi eliminates these risks. Since its locking blocks are randomly generated for each individual product, even personnel within the design house lack pertinent information, making it impossible for attackers to identify structural weaknesses in DLL based on existing knowledge. Therefore, the random locking block generation mechanism effectively realises 'zero-knowledge locking'. This approach fosters a higher level of trust and security throughout the product's lifecycle. Figure 3.8 illustrates the 'zero-knowledge' concept.

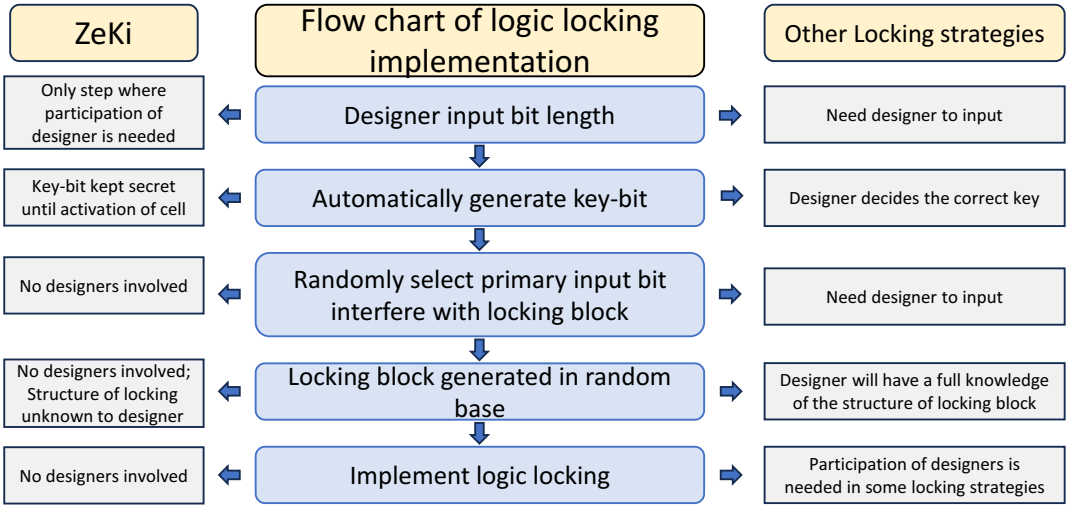

FIGURE 3.8: Implementation flow: ZeKi versus Other Logic Locking

As shown in figure 3.8, in the process of ZeKi's DLL, the only input required from the designer is the key-bit length for the lock. All other procedures will be carried out automatically, which means the potential rouge insider has no access to the detail of locking block structure, primary input selected for locking or the correct key value, which effectively proves the implementation of 'zero-knowledge locking'.

As discussed in Section 1.1.1.4, hardware designers can perform verification without revealing the secret key by using **encrypted simulation modules** compliant with the *IEEE 1735* [35] standard for IP encryption, or by employing **formal equivalence**

checking tools such as *Cadence Conformal* or *Synopsys Formality*. These approaches enable secure verification within the supply chain while maintaining zero-knowledge locking.

3.4 Implementation of Zeki

3.4.1 Parameter Input

Before applying Zeki to the product, the designer must first input some basic parameters.

1. **Key-bit length**: This refers to the number of key bits included in Zeki. Typically, 64-bit or 128-bit key lengths are the standard choices. However, if the circuit is small or if higher security is required, the designer may adjust the key-bit length accordingly, either by decreasing or increasing it.

2. **c**: To ensure the output corruption rate of the locked circuit, that is, the rate at which sufficient primary output bits produce erroneous results when an incorrect key is provided, the circuit will use a combination of SLL (strong logic locking) and Zeki blocks. A portion of the key bits will be used as the key input for the SLL. The designer can adjust the proportion of these key bits relative to the total key-bit length, with the default value set at 25

3.4.2 Implementation Flow of Zeki

1. **Generate a Key Sequence**:

   The first step is to randomly generate a key sequence that matches the specified key-bit length.

2. **Generate SLL Locking Gates**:

   Based on the SLL key-bit rate (the rate of SLL gates to be inserted) and the key-bit length, the number of SLL gates to be inserted, denoted as S_N, is calculated. From this, S_N key-bits are randomly selected to serve as inputs for the SLL gates. According to the selected key values, and following the principles of Strong Logic Locking (SLL), XOR or XNOR locking gates are inserted into the circuit.

3. **Randomly Group Key-bits**:

   The key-bits in the generated key sequence are randomly grouped, either into pairs or triples.

4. **Randomly Select Input Bits and Group Them**:

   Next, input bits, which are primary inputs of the circuit, are randomly selected, and these input bits are grouped in the same manner as the key-bits (into pairs or triples).

5. **Map Key-bit Groups to Input-bit Groups**:

   The randomly grouped key-bits and input bits are then mapped one-to-one, creating corresponding pairs or triplets for further processing.

6. **Define Logic Relationships for Validation**:

   Each pair or triplet of key-bit and input-bit is assigned a corresponding logical relationship, such as relationship a and relationship b. These validation logic relationships are predefined in the Zeki validation logic library, which is part of the tool's functionality. The library provides a variety of logic relationships, which are randomly chosen during the generation of the Dynamic Logic Locking (DLL) block.

7. **Generate Correct Key and DLL Block**:

   Using the validation logic calculated from the generated key-bit and input-bit combinations, the correct key is derived. Based on this, the DLL block is generated and connected to the fan-in of the protected circuit's output.

8. **Create Key Verification Unit**:

   Using the correct key derived in the previous step, a key verification unit is generated using the same random logic method as the DLL generation. This verification unit is connected to the flip signal of the DLL output, ensuring that when the correct key is input, the circuit will not produce incorrect results.

3.5 Experiment Results and Discussion

3.5.1 Experimental Setup

The experiments were carried out on an 8 core M1 Mac processor with 8GB of RAM. Combinational benchmarks from EPFL combinational suite [127] and ITC99 suite [128] and the combinational benchmarks from parts of sequential benchmark circuits from the ISCAS89 [129] were locked in the experiments. To evaluate Zeki on a larger benchmark, a new benchmark named sha256x10 was created in this work by instantiating ten parallel SHA-256 modules. This composite design contains over 500,000 logic gates. Table 3.2 presents data on circuits; the number of gates in the benchmarks ranges from 3,448 to 510,190, demonstrating a diverse array of circuit sizes. MiniSAT [130] was utilized as an SAT solver to implement SAT attacks. The Synopsys Design Compiler was used in this project to determine the area and power overhead introduced by DLL, and the Global Foundries 65nm LPe library is used in the simulation. I implemented ZeKi using Python scripts to automatically lock benchmark circuits via the DLL approach, which required between 1.3 and 161.7 seconds to process and lock each circuit across all benchmarks, as shown in table 3.2. The subsequent subsections provide an evaluation of attack resilience and DLL overhead.

TABLE 3.2: Benchmarks and corresponding time consumption to implement Zeki

Benchmark	Input Number	Output Number	Gate Number	ZeKi Running Time (s)
b21	522	512	20,027	6.7
b18	3,357	3,343	111,421	32.2
s38354x	38	304	11,448	4.2
s38417x	28	106	8,709	3.2
s15850x	77	150	3,448	1.3
sha256	678	258	51,019	15.2
sha256x10	6780	2580	510,190	161.7

3.5.2 Resilience Against Multiple Attacks

Before delving into further details, I describe the terminology used in the remainder of the paper.

3.5.2.1 SAT attack resilience

As discussed in Chapter 2, there are two ways to mitigate the threat from the SAT attack: increasing the average execution time for each iteration and increasing the number of iterations, and Zeki takes the latter approach to achieve SAT resilience. The same approach is also taken by PFB logic locking to achieve maximum iteration number for SAT attack.

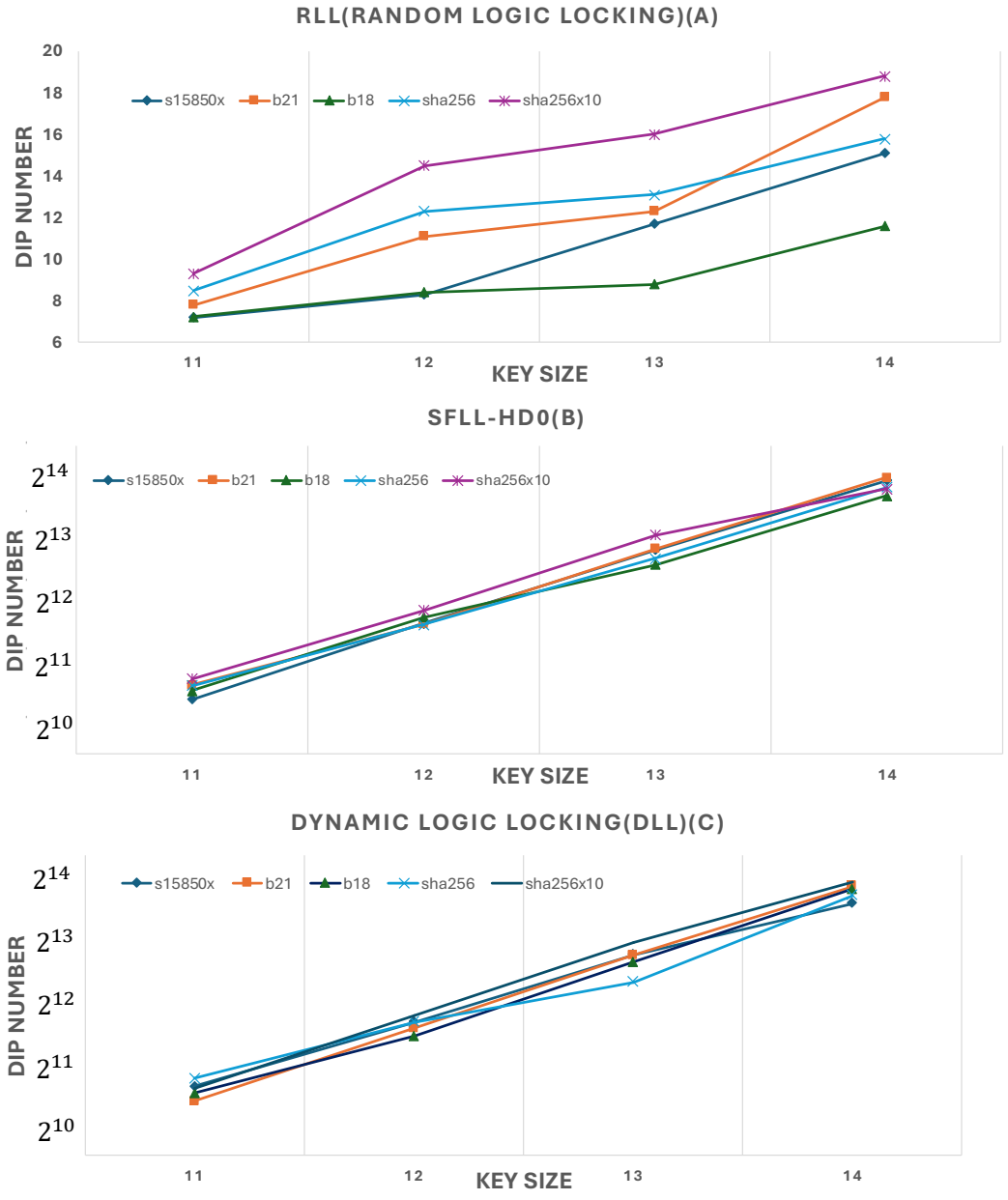

FIGURE 3.9: SAT attack simulation on design locked with (a) RLL[9], (b) SFLL[3], (c)ZeKi's DLL.

Definition 1 If a technique, denoted as L, requires at least $\frac{a*k+b}{2^k}$ iterations of a SAT attack to be compromised when using a k-bit key, indicating that the difficulty of breaking it is exponentially related to the key length, then L is considered k-secure relative to SAT attacks.

The resilience analysis of the SAT attack is shown in Figure 3.9, which compares SAT attacks between DLL and other logic locking strategies. This experiment selected Random Logic Locking (RLL) and SFLL-HD0 [3] as reference techniques for DLL, as they exhibit the highest resistance to SAT attacks.

I sample protected designs with key lengths of 11, 12, 13, and 14 bits, albeit shorter than typical real-world key sizes, usually 128 or 64 bits. This limitation was due to computational and time constraints. However, it should be noted that the underlying principles for SAT attack resilience comparison remain consistent across varying key lengths.

Figure 3.9 illustrates that conventional logic locking, such as RLL in Figure 7.a, has low SAT attack resilience, showing limited improvement even with increased key lengths. Conversely, SAT resilient methods of PFB logic locking like SFLL-HD0 (Figure 7.b) demonstrate exponential growth in the required iterations of SAT attacks as key lengths increase. Similarly, DLL, as seen in Figure 7.c, also shows an exponential increase in SAT resilience with increasing key lengths. A successful SAT attack on DLL necessitates 2^k Deterministic Independent Pairs (DIPs), where k is the key length, thereby confirming its k-secure status against SAT attacks.

TABLE 3.3: Benchmarks and corresponding SAT-solver Running Time

Benchmark	**Gate number**	**SAT-solver(MiniSAT) Running Time (s)**
b21	20027	2.12
b18	111421	10.4
s38354x	11448	0.6
s38417x	8709	0.22
s15850x	2448	0.13
sha256	51019	6.78
sha256x10	510190	77.3

In this project, the time consumption of the SAT solver (MniSAT) to complete a single iteration is also simulated. In the experiment, the Verilog netlist files of various benchmarks used in this study are duplicated to form a Miter circuit in CNF format, which

is then used in the SAT attack. The Miter circuit is subsequently loaded into the SAT solver for evaluation. As demonstrated in Table 3.3, circuits with higher complexity (i.e., a greater number of gates) result in longer solving times. For instance, consider s15850x, the least time-consuming circuit. The average cracking time for this circuit is approximately 0.13 seconds per attack. Based on this, an attacker attempting to break a 32-bit key (noting that 64- or 128-bit keys are more common in practice) would require approximately 17.7 years to crack the secret key using the SAT attack. For circuits with higher complexity, the required time would be considerably longer. Thus, it can be concluded that DLL effectively achieves SAT-resilience.

3.5.2.2 Sensitisation attack resilience

The Sensitisation attack, initially proposed in [40], involves sensitising key bits to the output of the circuit to deduce their values. However, if the relationship between a key bit and the circuit's output is influenced by other key bits, preventing it from being easily sensitised, it is termed "pair-wise" as discussed in section 2.2.3. This designation effectively thwarts the sensitisation attack.

Definition 2 In a locked circuit, if $k\%$ key gates are pairwise secure, meaning the circuit is k-secure against sensitisation attacks, with the ideal scenario being 100-secure. When a circuit is 100-secure against sensitisation attack, all of its key-bits will not able to be propagated to the primary output bit by controlling input of the circuit.

In DLL, the gates within the locking block can be considered key-gates. Due to the interdependence of the outputs of all key-gates in DLL, attackers cannot sensitise key bits by controlling the input and observing the output. Consequently, all gates in the DLL locking block are effectively 'pair-wise' related to each other, rendering the system k-secure against sensitisation attacks.

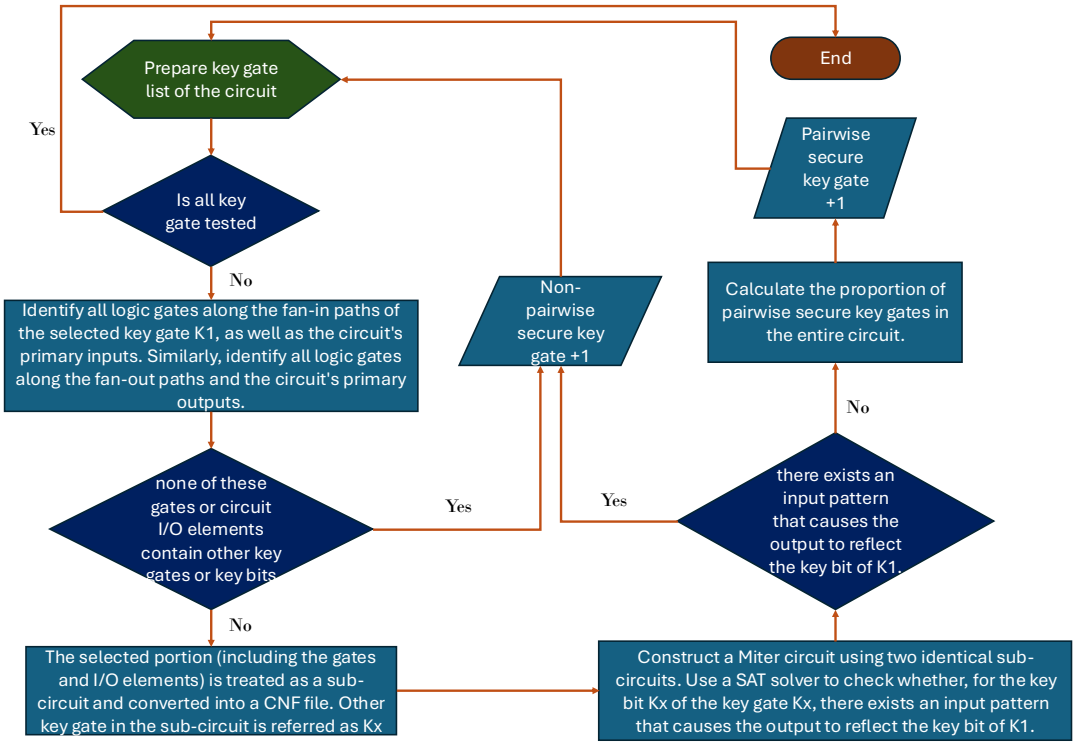

FIGURE 3.10: Process diagram of sensitisation attack resilience simulation

In order to prove this, an experiment is designed and implemented in this project, the process of which is shown in Figure 3.10. This simulation is implemented on circuit locked by DLL and RLL, the result shows different sensitisation attack resilience of the two strategies. Figure 3.11 illustrates the percentage of 'pair-wise' gates in DLL, achieving 100-secure against sensitisation attacks. Conversely, Random Logic Locking (RLL) exhibits only a limited number of pair-wise key gates against sensitisation attacks. On average, DLL performs six times better than RLL.

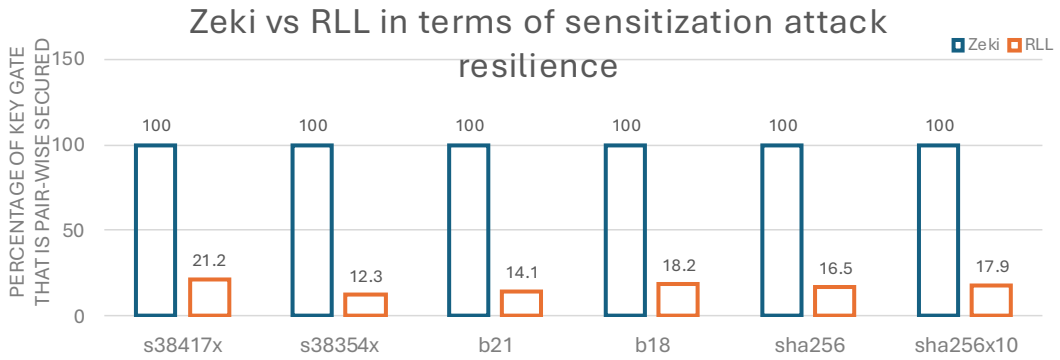

FIGURE 3.11: ZeKi sensitisation attack resilience vs RLL[9]

3.5.2.3 Structural attack resilience.

Structural attacks exploit inherent weaknesses in certain logic locking techniques, particularly those using PFB. The SAT resilience of PFB is achieved through locking blocks with a consistent topological structure, which remains unchanged even as the number of key-bits and the block's size increases. This uniformity allows attackers to exploit vulnerabilities in the locking block's structure, rendering all products using that technique susceptible. For example, Anti-SAT's locking block is vulnerable to SPS attacks [6] due to its output gate's excessive signal skew. Similarly, SFLL [3] is susceptible to sensitive attacks that detect key-bits through sensitive input pattern values [5]. The issue can be addressed by DLL, as it generates a random locking mechanism each time. This results in significant variations among the locking blocks of different products.

Definition 3 From this inference, it follows that the greater the variability in the locking blocks of a logic locking technique, the less likely they are to exhibit a common structural weakness, thereby harder to break. In the structural attacks resistance testing, I first locked a selected benchmark with a locking block named *LB0*. I then executed 100 separate lockings on the benchmark, generating additional locking blocks labeled *LB1-LB100*. When the same input is fed to the locking block, *LB0*'s output is compared with the outputs of *LB1-LBN*, and the average Hamming distance (*HDM*) is calculated. The value of x is determined by calculating *100*HDM/OutputLength %*, which defines the technique's security level against structural attacks as x-secure, where the ideal value is 100-secure. In this experiment, Anti-SAT and SFLL were used as references. The

average secure value of DLL-generated locking blocks exceeded 80, while both Anti-SAT and SFLL had secure values of 0.

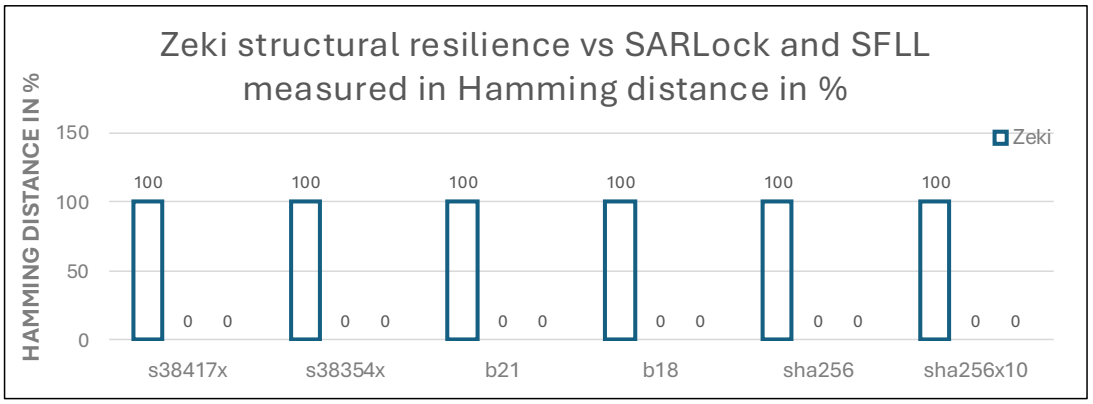

FIGURE 3.12: ZeKi structural attack resilience comparison with SARLock[4] and SFLL[3]

In this thesis, a comparative analysis was conducted on the resilience of DLL against structural attacks in Figure 3.12. DLL utilises random generation for its locking blocks, creating significant variability in the locking blocks for each design. This approach has shown to be highly effective in simulations, with DLL achieving over 80-secure resilience in the simulation results against structural attacks across different benchmarks. This level of randomness of the generated locking block makes it extremely difficult to find common inherent structural or parameter characteristics of the DLL locking block, making it highly resilient to structural attacks.

3.5.3 Power, Delay, and Area overhead

Results from Figure 4.4 indicate that a 128-bit key significantly increases overhead in smaller benchmarks as S38417x (11.1% power, 12.2% area overheads) but is less in larger benchmarks, e.g., b21 benchmark with 2.1% power, and 1.7% area overheads; sha256x10 benchmark with 0.13% power, and 0.15% area overheads. In larger benchmarks, power and area overheads remain below 10% for 128-bit key. From the overhead data it can be found that DLL's overhead is linearly related to the number of key bits. Overall, the average power and area overhead is 2.4% and 2.46% respectively for a 64-bit key; and 5.1% and 5.45% for 128-bit key.

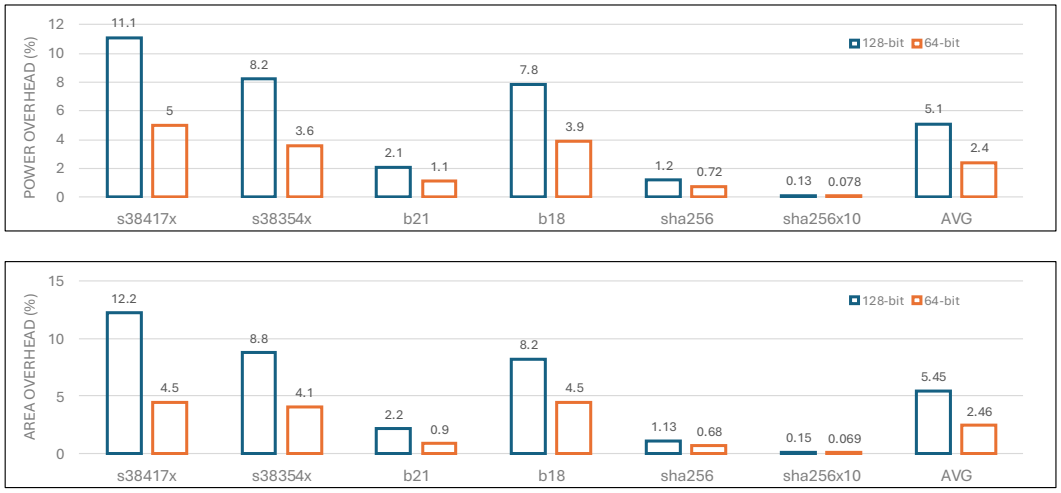

FIGURE 3.13: Power, and Area overhead of DLL protected circuit with 64-bit and 128-bit key

3.6 Summary

In summary, this chapter introduced a ZeKi which generate Dynamic Logic Locking (DLL) security circuit by 'Zero-Knowledge' Implementation. ZeKi's DLL demonstrated resilience against SAT attacks and other mainstream attacks targeting logic locking while maintaining lowest power and area overhead. The security performance of ZeKi has achieved 1 to 89 times than the state-of-the-art logic locking techniques depending on different attacks and locking structures. By employing a dynamic locking block, ZeKi ensures that each circuit is protected with a randomly generated and unique locking mechanism, thereby enhancing overall security and reducing the risk of attacks exploiting specific vulnerabilities. It enables hardware designers to achieve zero-knowledge locking for their designs, further strengthening the confidentiality. Overall, the proposed ZeKi logic locking tool contributes to advancing logic locking techniques and effectively protects integrated circuits.

Attacks on logic locking can be categorized into two types based on the attacker's access to a functional IC: oracle-based and oracle-less attacks. Current research, including this thesis, predominantly focuses on oracle-based attacks. In the threat model for oracle-based attacks, it is assumed that attackers have access to two critical entities: **A functional IC**: Attackers can readily acquire a functional IC from the market. During the attack process, this functional IC provides valid input/output pairs as golden references; **A gate-level netlist file of the locked circuit**: This file is obtained through

reverse engineering or from an untrusted foundry.

Chapter 4

ZeKi: The Sequential version

In the previous chapter, we introduced the fundamental principles and implementation flow of Zeki, focusing primarily on its application to combinational circuits. While the previous chapter concentrated on combinational logic, it is important to note that in the IC industry, the majority of products utilise sequential circuits, ranging from microprocessors to memory units and communication devices. (While combinational circuits are those where the output is purely dependent on the current inputs, sequential circuits involve memory elements (such as flip-flops or latches) that store state information, making their outputs dependent not only on the current inputs but also on the circuit's past history.)

In fact, Zeki can also be applied to sequential circuits. The ability to apply Zeki to sequential circuits expands its utility, allowing designers to implement security features across a wider range of applications. In this chapter, we will explore the operational principles and implementation flow of Zeki when applied to sequential circuits.

Two approaches of Zeki implementation will be introduced in this chapter ZekiA and ZekiB, the first approach is relatively simple by placing the entire locking block in the last stage of the sequential circuit and placing the entire Key verification unit in the first stage. In the second approach, locking block and Key verification unit will be randomly put in different stages in the sequential circuit to make it harder for malicious attackers to trace the inserted gates of Zeki.

In this chapter the power and area overhead of Zeki will also be given.

Parts of this chapter are based on our previously published work [126]. Some text and figures are reused with permission.

4.1 Implementation of SLL

This chapter will introduce two approaches of Zeki sequential implementation, however, in both strategies, SLL (Strong Logic Locking) is inserted in the same manner. In the insertion process of SLL, corresponding key-bits are first selected, and then logic gate in the circuit are randomly selected. And then SLL gates are inserted in the circuit. The working flow of this implementation is listed as follows:

1. Randomly select K gates from all the gates, where K corresponds to the key-bit length.
2. For each selected gate, determine the position K_n where the output key-gate will be inserted.
3. Rename the output of each key-gate to match the name of key-gate input 1 (e.g., KGi_1).
4. Based on the value of the key-bit, insert the corresponding key-gate at the selected position K_n. If the key-bit value is 0, insert an XOR gate; if the key-bit value is 1, insert an XNOR gate. One input of the key-gate will be the node renamed in the previous step, the other input will be the key-bit, and the output will correspond to the original node before renaming. The output will have the original name of the node at position K_n.
5. Add the new node to the wire list.
6. Add the new key-bit to the input list.

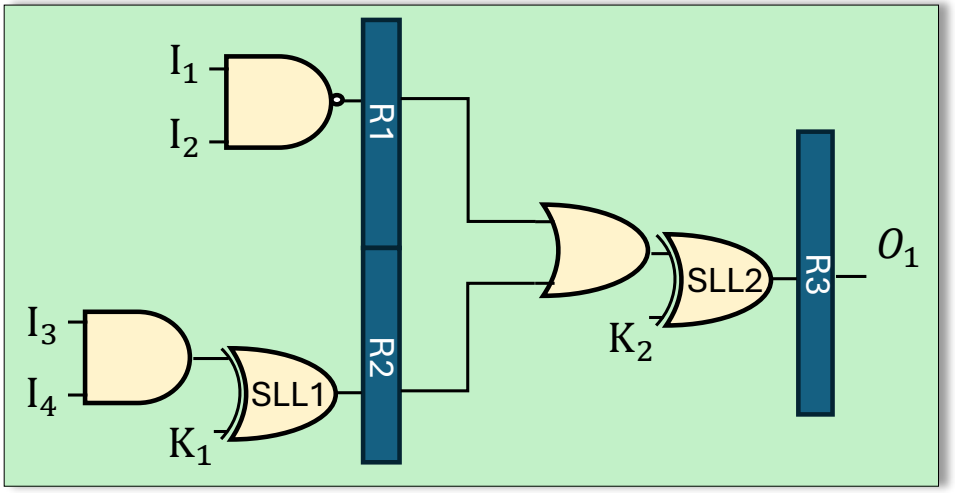

FIGURE 4.1: Add SLL(strong logic locking) to Sequential Circuit

4.2 ZekiA: Implement Zeki in Sequential Circuit in Single Stage

ZekiA is a foundational version of Zeki implemented on sequential circuits. In ZekiA, in addition to the previously mentioned SLL (Strong Logic Locking), which is distributed across all stages, the DLL (Dynamic Logic Locking) block and the key verification unit are placed within a single stage. Both the DLL block and the key verification unit are positioned in the final stage, the stage closest to the primary output, to ensure that the flip signal can directly impact the output. The circuit's primary and key-bit signals are fed directly into these two blocks.

The detailed implementation flow of ZekiA is as follows:

1. **Parsing Netlist**

   The first step is to parse the netlist of sequential circuit, and the process of this is similar to combinational circuit but with extra information the flip-flop list and the stage attribution. All the gate and wire variables will be labeled which stage they belong to.

2. **Generate a Key Sequence**:

   Randomly generate a key sequence that matches the specified key-bit length.

3. **Generate SLL Locking Gates**:

   Based on the SLL key-bit rate (the rate of SLL gates to be inserted) and the key-bit length, the number of SLL gates to be inserted, denoted as S_N, is calculated. From this, S_N key-bits are randomly selected to serve as inputs for the SLL gates. According to the selected key values, and following the principles of Strong Logic Locking (SLL), XOR or XNOR locking gates are inserted into the circuit. In this step the program also need to make sure inserted SLL key-gates belong to the same stage to avoid sequential problem.

4. **Randomly Group Key-bits**:

   The key-bits in the generated key sequence are randomly grouped, either into pairs or triples.

5. **Randomly Select Input Bits and Group Them**:

   Next, input bits, which are primary inputs of the circuit, are randomly selected, and these input bits are grouped in the same manner as the key-bits (into pairs or triples).

6. **Map Key-bit Groups to Input-bit Groups**:

   The randomly grouped key-bits and input bits are then mapped one-to-one, creating corresponding pairs or triplets for further processing.

7. **Define Logic Relationships for Validation**:

   Each pair or triplet of key-bit and input-bit is assigned a corresponding logical relationship, such as relationship a and relationship b. These validation logic relationships are predefined in the Zeki validation logic library, which is part of the tool's functionality. The library provides a variety of logic relationships, which are randomly chosen during the generation of the Dynamic Logic Locking (DLL) block.

8. **Generate Correct Key and DLL Block**:

   Using the validation logic calculated from the generated key-bit and input-bit combinations, the correct key is derived. Based on this, the DLL block is generated

and connected to the fan-in of the protected circuit's output. In sequential circuit, the output of DLL block is connected to the input of primary output flip-flop, and the DLL block is put in the final stage to decrease the overall number of flip-flops.

9. **Create Key Verification Unit**:

   Using the correct key derived in the previous step, a key verification unit is generated using the same random logic method as the DLL generation. This verification unit is connected to the flip signal of the DLL output, ensuring that when the correct key is input, the circuit will not produce incorrect results. In sequential circuit, the key verification unit is put in the final stage, which is the stage closest to the primary output.

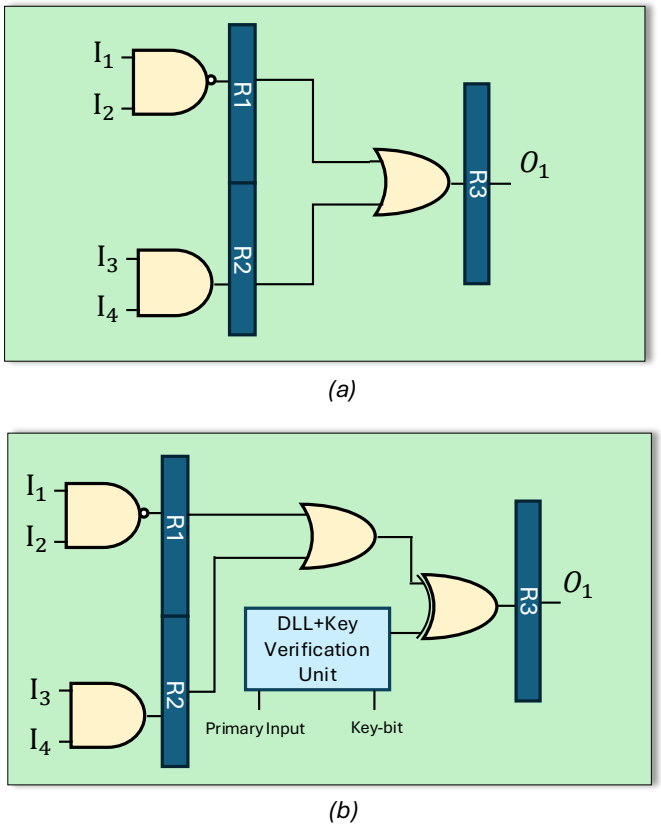

FIGURE 4.2: (a) Original Circuit; (b) Circuit Locked with ZekiA

4.3 ZekiB: Implement Zeki in Sequential Circuit in Multiple Stages

Implementing Zeki in a single stage of a sequential circuit is straightforward and minimizes the risk of errors. However, this approach has its drawbacks. Since both the DLL (Dynamic Logic Locking) block and the key verification unit are placed within a single stage, it becomes easier for attackers to identify and target these locking blocks. This centralized placement of the blocks also reduces randomness, which contradicts the dynamic nature of Zeki, thereby undermining its intended security purpose.

To address this issue, an alternative approach, ZekiB, allows users to distribute the DLL block and key verification unit across multiple stages of the circuit. The initial steps of ZekiB's implementation mirror those of ZekiA, beginning with the insertion of SLL and the generation of DLL blocks and the key verification unit using a random validation formula. Once these blocks are created, the system traverses the gates starting from the block's output and moving towards its input. The system then randomly assigns whether the gate, along with its fan-out, should belong to the same stage. This process ensures that the DLL block and key verification unit are distributed randomly across various stages of the circuit.

The implementation flow of ZekiB is as follows:

1. **Parsing Netlist**

   The first step is to parse the netlist of sequential circuit, and the process of this is similar to combinational circuit but with extra information the flip-flop list and the stage attribution. All the gate and wire variables will be labeled which stage they belong to.

2. **Generate a Key Sequence**:

   Randomly generate a key sequence that matches the specified key-bit length.

3. **Generate SLL Locking Gates**:

   Based on the SLL key-bit rate (the rate of SLL gates to be inserted) and the key-bit length, the number of SLL gates to be inserted, denoted as S_N, is calculated. From this, S_N key-bits are randomly selected to serve as inputs for the SLL gates.

According to the selected key values, and following the principles of Strong Logic Locking (SLL), XOR or XNOR locking gates are inserted into the circuit. In this step the program also need to make sure inserted SLL key-gates belong to the same stage to avoid sequential problem.

4. **Randomly Group Key-bits**:

   The key-bits in the generated key sequence are randomly grouped, either into pairs or triples.

5. **Randomly Select Input Bits and Group Them**:

   Next, input bits, which are primary inputs of the circuit, are randomly selected, and these input bits are grouped in the same manner as the key-bits (into pairs or triples).

6. **Map Key-bit Groups to Input-bit Groups**:

   The randomly grouped key-bits and input bits are then mapped one-to-one, creating corresponding pairs or triplets for further processing.

7. **Define Logic Relationships for Validation**:

   Each pair or triplet of key-bit and input-bit is assigned a corresponding logical relationship, such as relationship a and relationship b. These validation logic relationships are predefined in the Zeki validation logic library, which is part of the tool's functionality. The library provides a variety of logic relationships, which are randomly chosen during the generation of the Dynamic Logic Locking (DLL) block.

8. **Generate Correct Key and DLL Block**:

   Using the validation logic calculated from the generated key-bit and input-bit combinations, the correct key is derived. Based on this, the DLL block is generated.

9. **Create Key Verification Unit**:

   Using the correct key derived in the previous step, a key verification unit is generated using the same random logic method as the DLL generation. This verification unit is connected to the flip signal of the DLL output, ensuring that when the correct key is input, the circuit will not produce incorrect results.

10. **Locate DLL block and Key Verification Unit in Stages**

    Select all the output gates in DLL block and Key Verification Unit and locate them to the last stage of the circuit, which is the closest stage to the primary output, and name this group of gates G1. Use random number generator planted in software to decided whether the gates whose fan-out gates are included in G1, and name this group of gates G2. Repeat this procedure until all the gates in DLL block and key verification unit are located to their stages. Add DLL block and key verification Unit to the circuit according to the stage location and connect the output flip signal to the primary output of circuit.

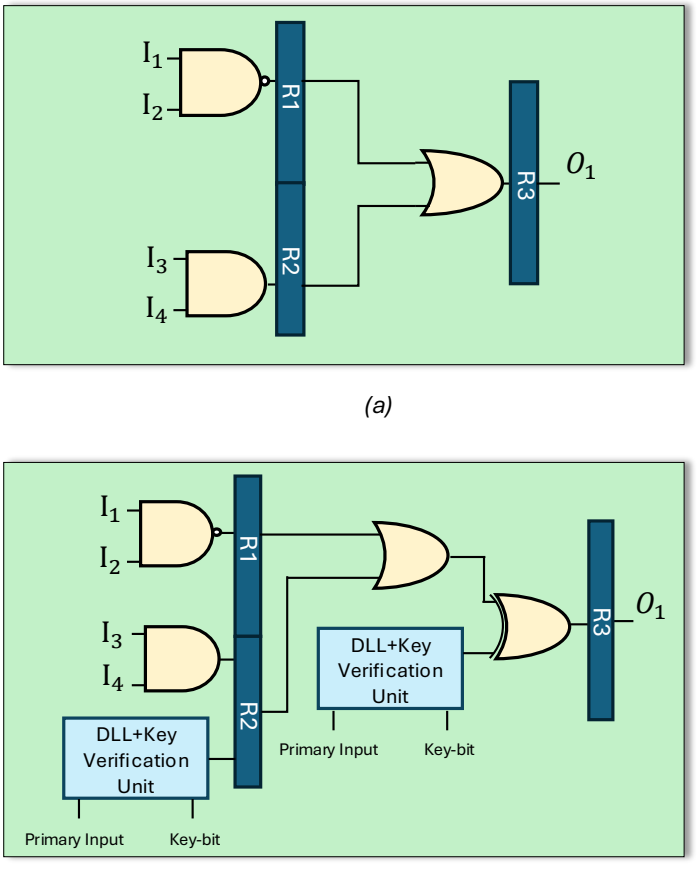

FIGURE 4.3: (a) Original Circuit; (b) Circuit Locked with ZekiB

Compared to ZekiA, ZekiB exhibits higher uncertainty, which in turn enhances overall security. The increased unpredictability makes it more challenging for attackers to

trace the specific characteristics of the locking block. However, ZekiB also introduces a greater number of flip-flops, leading to additional overhead. This trade-off requires designers to carefully select between the different solutions based on their specific needs and application requirements.

In the context of logic locking and integrated circuit (IC) design, ZekiB's higher uncertainty is advantageous in countering attacks such as reverse engineering and hardware trojans. However, the additional flip-flops introduce complexity and may impact performance, particularly in terms of power consumption and circuit area. As a result, designers must consider the balance between security and performance, choosing the appropriate locking scheme based on the security threat level and the constraints of the target system.

4.4 Power, Delay, and Area overhead

Figure 4.4 shows the overhead caused by Zeki in both combinational benchmarks (b21, b18) and sequential benchmarks(s15850x, s38354x...), where ZekiA is used as implementation approach for sequential benchmarks in Figure (a) and (b), and ZekiB is used in (c).

Results from Figure 4.4 indicate that a 128-bit key significantly increases overhead in smaller benchmarks as S15850x (18.5% power, 17.2% area overheads) but is less in larger benchmarks, e.g., b21 benchmark with 2.1% power, and 1.7% area overheads. In larger benchmarks, power and area overheads remain below 10% for 128-bit key. Sequential circuits show reduced time overhead due to distributed locking block and mask components, with time overheads under 3% for all 128-bit key configurations. DLL's overhead is linearly related to the number of key bits. Zeki in benchmarks in Figure 4.4 (a) and (b) is implemented by ZekiB for sequential circuit, while in Figure 4.4 (c), ZekiA is used. It is obvious ZekiA introduces much lower overhead in both power and area for all the sequential benchmarks compared to its counterpart. Overall, the average power and area overhead is 4.27% and 4.48% respectively for a 64-bit key; and 9.2% and 9.28% for 128-bit key. Because LUTs and flip-flops are not used in the implementation of

ZeKi's locking blocks, the area overhead is significantly lower compared to LUT/FPGA-based locking strategies or those with high overhead, as shown in Table 4.1. It achieved the lowest area and power overhead compared to the state-of-the-art locks.

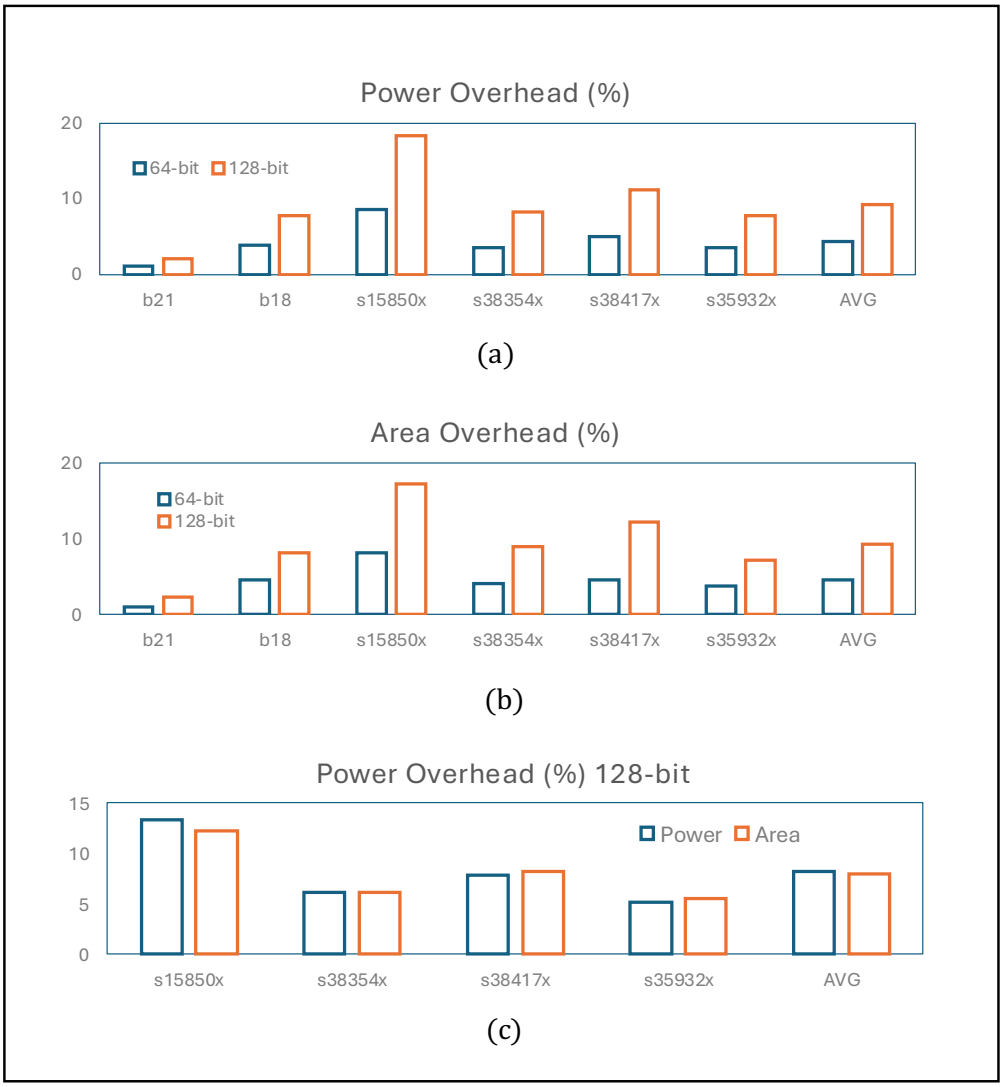

FIGURE 4.4: Power, and Area overhead of DLL protected circuit with 64-bit and 128-bit key: (a) Power Overhead; (b) Area Overhead; (c) Power and Area Overhead of ZekiB Implemented Benchmarks

TABLE 4.1: Area and Power Overhead

Locking Strategy	Cross-lock [131]	Logic locking with Camouflage [132]	RTLock[133]	**ZeKi (64-Bit Key)**
Area	18.6%	31.78%	19.88%	**4.27%**
Power	14.42%	26.52%	2.2%	**4.48%**

4.5 Summary

This chapter extends the ZeKi logic locking strategy from combinational circuits to sequential circuits, addressing a broader range of practical applications in integrated circuit (IC) design. Two implementation approaches are proposed: ZeKiA, which places the Dynamic Logic Locking (DLL) block and key verification unit within a single stage of the sequential circuit, and ZeKiB, which distributes these components across multiple stages to enhance resistance against structural attacks and improve security through increased randomness.

Both implementations incorporate Strong Logic Locking (SLL) to insert key-gates throughout the design. ZeKiA offers simplicity and lower overhead, while ZeKiB introduces higher security at the cost of additional flip-flops and increased complexity. The chapter outlines the detailed implementation flows for both approaches and discusses the trade-offs between security and design overhead.

Experimental results demonstrate that ZeKi maintains low area and power overhead across a variety of combinational and sequential benchmarks. Specifically, ZeKi achieves average overheads of 4.27% (area) and 4.48% (power) with a 64-bit key, significantly outperforming existing state-of-the-art locking strategies, as shown in table 4.1. Overheads remain acceptable even with 128-bit keys, especially for large-scale benchmarks, showcasing ZeKi's scalability and practicality for real-world sequential circuit protection.

Because the gate number of locking block is regardless of the gate number of circuit to be locked, the gate number Zeki brings can be considered as constant on some level. However, as Zeki applies dynamic logic locking mechanism, the locking block is generated randomly, the gate number also changes with different implementation, but at the same order of magnitude.

Chapter 5

LockLab

Despite its potential, the adoption of logic locking has been slow, partly due to the lack of integration with standard IC design flows and the complexity of existing algorithms. Automating the logic locking process and integrating it into existing design tools are crucial steps toward wider adoption.

This chapter presents two software tools developed to automate logic locking: *Lockit* and *LockLab*. *Lockit* is a Python-based tool that implements the Stripped Functionality Logic Locking with Hamming Distance (SFLL-HD) algorithm [3] to perform locking on gate-level netlists. *LockLab* extends this by supporting multiple locking strategies and simulating various attack methodologies, providing a comprehensive platform for designers to implement and evaluate logic locking techniques.

In the course of this PhD project I designed LockLab, an automated logic locking simulation tool. LockLab facilitates the seamless application of various logic locking techniques to circuit netlists, as well as the simulation of different attacks on locked circuits. This tool offers significant convenience for designers employing logic locking in their work, as well as for students learning about logic locking concepts.

LockLab was inspired by Lockit, an automation tool designed for SFLL (Strong Fault Logic Locking) insertion, for which I served as the second author. The introduction of Lockit greatly simplified processes related to the implementation and testing of SFLL, including integrating SFLL into netlist files and performing subsequent security tests. With Lockit, users could easily insert SFLL into circuits that required protection or testing, significantly improving the efficiency of these tasks.

During my exploration of various logic locking techniques, I encountered a recurring issue: many of the logic locking methods that needed testing either lacked available software tools or, if tools existed, they were no longer maintained or usable due to various reasons. Furthermore, no simple, user-friendly automated simulation tools were available that could handle multiple logic locking schemes and simulate corresponding targeted attacks. I developed LockLab to fill this gap. LockLab aims to provide an easy-to-use, flexible, and comprehensive solution for logic locking simulation and attack testing. Besides automatic SFLL implementation (the main function of Lockit), Locklab can also achieve multiple logic locking strategies implementation and attack simulation which are listed below:

1. Logic Locking Insertion: LockLab can automatically insert logic locking into Verilog-format netlist files, with the output files preserving the same format as the original. Users can choose from a range of logic locking techniques, including RLL (Reverse Engineering Logic Locking), FLL (Functional Logic Locking), SLL (Strong Logic Locking), SARLock, AntiSAT, Andtree, SFLL, and Zeki. The resulting files can be used for subsequent design iterations or for functional testing in related fields.

2. Attack Simulation: Once logic locking is applied to a circuit, LockLab can simulate various types of attacks on the locked circuit, including SAT (Satisfiability) attacks, SPS (Single Point of Secret) attacks, sensitization attacks, brute-force attacks, and an assessment of AI attack feasibility. LockLab provides detailed results, including the total time required for the attack and whether the key has been successfully cracked. Additionally, it verifies whether the attacked key or the modified circuit maintains the original circuit's functionality.

Parts of this chapter are based on our previously published work [10, 126]. Some text and figures are reused with permission.

5.1 Contributions

LockLab, a novel software tool introduced by this work specifically is developed to address the limitations in the current landscape of logic locking implementation. The

source code of LockLab is shown in Appendix A. The contributions of this software are outlined as follows:

1. Multi-Strategy Logic Locking Implementation: The tool provides the capability to implement a wide variety of gate-level logic locking strategies: RLL, FLL, SLL, Zeki, AntiSAT, SARlock, and SFLL. This allows hardware designers to evaluate different locking techniques on their circuit designs without the need for extensive manual intervention.

2. Comprehensive Attack Simulations: The software is designed to simulate multiple attacks including SAT attack, SPS attack, Sensitization attack and evaluate AI-based attack availability on the locked circuits, enabling users to test the resilience of their designs against various threat models. By simulating real-world attack scenarios, the tool allows for a more robust evaluation of the circuit's security.

By addressing the challenges and limitations in the current state of logic locking tools, LockLab provides a comprehensive solution that enhances the security design process while remaining easy to use and versatile for both logic locking learners and researchers. Learners of logic locking or hardware security can generate netlist of circuit locked with different locking strategies by LockLab. They can compare the original circuit and locked circuit netlist to learn the difference between the two. In addition, they can run functional tests on the generated locked circuit. Power, area and other post-synthesis simulations can also be done to the generated netlist file to see the overhead introduced. For researchers, not only can they do the tests on locked circuits generated as introduced, but also can they test different attacks on locked circuit on LockLab to test their resilience against various attacks.

The remainder of this chapter will introduce and discuss LockLab's setup before simulation, basic operation, output formats, as well as the design principles and workflow of its different modules.

5.2 Introduction to LockLab

The GUI(Graphical User Interface) of LockLab is shown in figure 5.1, which is user-friendly.

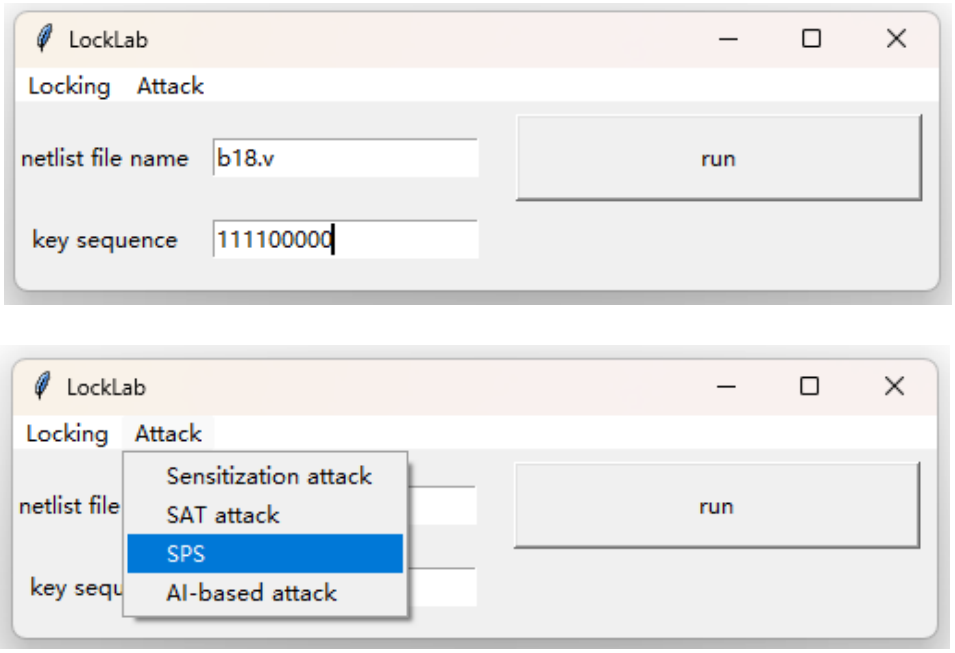

FIGURE 5.1: GUI of LockLab

For attack simulations specifically, users must save the names of the key-gates and key-bits in a CSV file and place it in the same directory as LockLab. This file facilitates the identification and tracking of key-related components during the simulation.

After each simulation, LockLab generates a corresponding output.log file that summarizes and compares the original file, the newly generated file, or the attack results. This log file serves as a detailed record of the simulation process and its outcomes, providing critical insights for evaluation and debugging.

Typically, the output.log includes the following information:

1. Name of the original file
2. Name of the newly generated file
3. Total number of logic gates in the original circuit
4. Total number of logic gates in the newly generated circuit

5. Locking strategy used
6. Type of attack performed
7. Key sequence
8. List of key-gates
9. Whether the attack was successful
10. The key derived from the attack (if successful)
11. Total runtime
12. Additional details relevant to the simulation

The inclusion of this comprehensive data ensures that users can analyze the effectiveness of the logic locking strategy and assess the resilience of the locked circuit against different types of attacks. A detailed example of the output log structure is provided below.

Example of Output Log File

```
file name: b18.v
new file name: b18locked.v
locking strategy: RLL
attack strategy:
original gate number: 111421
new gate number: 111433
key-gate number: 12
key-bit sequence: 000000111111
...
other:
```

5.3 Netlist Parsing

I build a parser for Locklab to understand verilog netlist file. Before each simulation start, the netlist file will be parsed before used by LockLab to organize its data. In terms of Parsing, LockLab takes similar approach with Lockit[10].

The process of netlist parsing, as illustrated in Figure 5.2, is designed to extract essential elements from a netlist—such as gates, registers, wires, inputs, and outputs—to

construct a graph-based representation of the circuit. This representation facilitates further analysis and manipulation.

The parsing process begins by dividing the netlist into segments using semicolons as delimiters. These segments typically fall into categories: those that contain comprehensive lists of inputs, outputs, or wires, and those that describe individual gates or registers.

Each segment is subsequently broken down into smaller tokens using delimiters such as spaces, commas, parentheses, and newlines. However, the parsing of gate inputs and outputs is handled with special care to ensure their relationships are accurately preserved. These tokens serve as the foundation for identifying the key attributes of the netlist.

The module name of the circuit to be locked is extracted from the segment that begins with the keyword module. The string following this keyword is assigned as the module's name, which also serves as the name of the graph. Segments starting with input, output, or wire provide lists of input, output, and wire names, respectively. Each name is represented as a node in the graph and is assigned a type attribute corresponding to its classification.

The remaining segments contain descriptions of individual gates and state elements. For each gate, the second token in the segment acts as a unique identifier and is used as the node's name in the graph. The first token identifies the gate or state element type, which is validated against a predefined list obtained from the technology file. This token is stored as the gate attribute, while a type attribute is assigned based on its classification. The subsequent tokens in the segment describe the connections between the gate and various wires, inputs, and outputs. These connections are used to define the edges in the graph, and the tokens collectively form a pinout attribute, detailing the gate's connectivity.

5.4 Verilog-CNF Transfermation

LockLab is able to transfer netlist file of a cricuit in Verilog into CNF file, which can be recognised and solved by SAT-solver. This approach not only plays a significant role in SAT attack simulation, as discussed in 2.3, but also enables LockLab to automatically

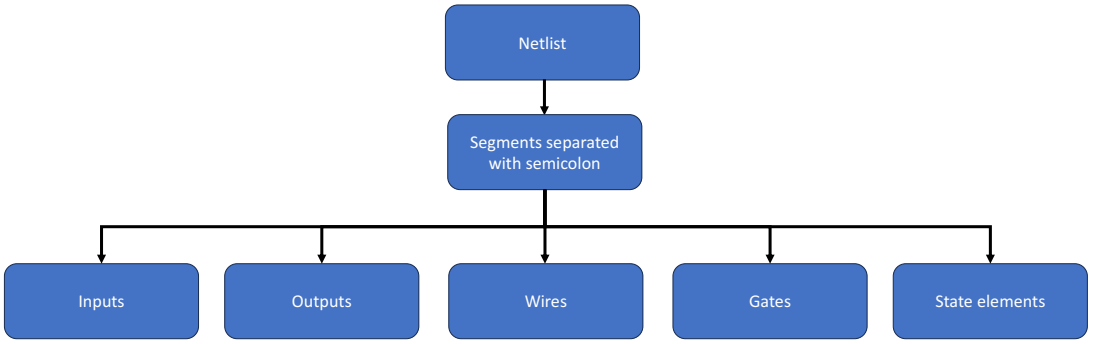

FIGURE 5.2: Parsing process of the netlist [10]

test the function correctness of generated locked circuit. The working principle of this operation will be explained in the following part of this section.

5.4.1 CNF transfer

In general, SAT solver takes Boolean formula in CNF (conjunctive normal form). In simple terms, a formula in CNF is a conjunction of disjunction of literals.[134]

- Conjunction: can be understood as AND.
- Disjunction: can be understood as OR.
- Literal: a variable or its negation.
- Clause: disjunction of literals. CNF is formed by conjunction of clauses.

The following is a example of CNF formula:

$$T = ((!a + b + !c)\&(a + !b + !c)\&c) \tag{5.1}$$

In order for a CNF form to be satisfied, all the disjunction clauses must be satisfied. In recent decades powerful SAT solvers have been developed. [135, 136] Before applying SAT solver to the circuit, the researcher must use Tseitin transformation to transform the netlist into CNF. [137]

TABLE 5.1: CNF formulas for AND, NAND, OR, NOR, INV, BUFFER gates generated using the Tseitin transformation [8]

Gate type	Gate function	φ_y
AND	$y = \text{AND}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i + \neg y)\right] \cdot \left(\sum_{i=1}^{j} \neg w_i + y\right)$
NAND	$y = \text{NAND}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(w_i + y)\right] \cdot \left(\sum_{i=1}^{j} \neg w_i + \neg y\right)$
OR	$y = \text{OR}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(\neg w_i + y)\right] \cdot \left(\sum_{i=1}^{j} w_i + \neg y\right)$
NOR	$y = \text{NOR}(w_1, \ldots, w_j)$	$\left[\prod_{i=1}^{j}(\neg w_i + \neg y)\right] \cdot \left(\sum_{i=1}^{j} w_i + y\right)$
NOT	$y = \text{NOT}(w_1)$	$(y + w_1) \cdot (\neg y + \neg w_1)$
BUFFER	$y = \text{BUFFER}(w_1)$	$(\neg y + w_1) \cdot (y + \neg w_1)$

5.4.2 Tseitin Transformation

For the SAT-solver to solve CircuitSAT problem, Tseitin transformation transfers circuit netlist to CNF where both input and output port serve as literals. Tseitin transformation achieves this by generating CNF for individual logic gates in the circuit. The formula below is an example of AND2 gate transformation.

$$\begin{aligned} z &\Leftrightarrow a\&b \\ &= (z \Rightarrow a\&b)\&(a\&b \Rightarrow z) \\ &= (!z + a\&b)\&(!(a\&b) + z) \\ &= (!z + a)\&(!z + b)\&(z+!a+!b) \\ &= (!z + a)(!z + b)(z+!a+!b) \end{aligned} \tag{5.2}$$

The Tseitin transformation then puts together the entire circuit as a conjunction of CNF clauses of all the individual logic gates.

During the conversion of the netlist file into a CNF file, each gate is transformed according to its corresponding Tseitin transformation formula, as illustrated in the table 5.1.

Once all logic gates have been transformed into CNF, they are placed alongside each other, as shown in figure 5.3, to serve as constraints for the SAT solver.

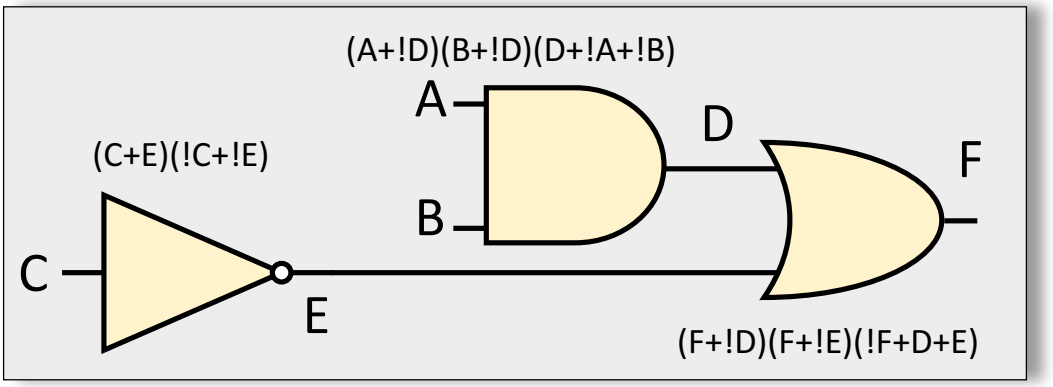

FIGURE 5.3: Application of the Tseitin transformation to a circuit with three gates

Within this framework, all circuit nodes, along with the input and output bits, are treated as literals. After these transformations are completed, the original Verilog gate-level netlist is converted into a '.cnf' file, as demonstrated in the code example below, where each number represents a literal and each line represets a single clause in CNF. This file is then ready for use in SAT-solver-based analysis.

```
p cnf 63 31
1 0
-34 2 0
-34 3 0
-2 -3 34 0
-35 4 0
-35 5 0
...
```

5.5 Locked circuit Self-test

When users apply a locking strategy to a netlist file using LockLab, the tool automatically performs a functionality test on the generated locked circuit netlist. This ensures that when the correct key is applied, the locked circuit produces outputs identical to the original circuit. The detailed procedure for this functionality test is as follows:

1. **Merge Netlists:** The netlist of the original circuit and the locked circuit are combined into a single file.

2. **Share Primary Inputs:** The inputs of both circuits are converted into wires, and a new input list is added to the combined netlist, allowing both circuits to share the same primary inputs.

3. **Compare Outputs:** The output bits of the two circuits are paired and connected to 2-input XOR gates. The outputs of these XOR gates are then fed into an N-input OR gate.

4. **Transform to CNF:** The new combined circuit is converted into a CNF file, and a constraint is added to ensure that the correct key sequence results in the output of the N-input OR gate being 0.

5. **Generate Input Patterns:** A set of 100 random input patterns is generated as constraints. The CNF file is then solved using a SAT solver. If solutions are found for all 100 patterns, the locked circuit is deemed functionally correct. Otherwise, it is considered faulty.

If the results indicate that the locked circuit is not functionally equivalent to the original circuit when the correct key is applied, the locking process is deemed unsuccessful. In such cases, the system automatically retries the locking process. If the process fails three consecutive times, the locking procedure terminates, and the results are logged in the output.log file.

This functionality significantly enhances the usability of LockLab by eliminating the need for external hardware simulation tools, such as ModelSim, for validating the netlist's functionality. Additionally, users are not required to manually write testbench files, simplifying the overall workflow.

1. **Selection of an Input Cone to Lock** SFLL-HD locks a single input cone, selecting the largest cone by default or one matching the user-specified key size. If the cone is larger than the key size, only a subset of inputs equal to the key size is protected. The cone must exceed the Hamming distance H; otherwise, users must adjust H. The algorithm identifies logic cone outputs recursively, grouping module outputs, state element inputs, and preceding wires, with recursion terminating at module inputs or state elements.

5.6 Implementation of logic locking strategies

5.6.1 Random Logic Locking (RLL)

is one of the most basic forms of logic locking. LockLab supports the insertion of RLL into a netlist, generating output files that are ideal for learning purposes or for use as benchmark references when comparing others, more advanced locking strategies. In RLL XOR and XNOR gates are inserted in the circuit based on the corresponding key-bit to randomly selected position in the protected circuit. The working flow of Random logic locking is shown in the flowchart:

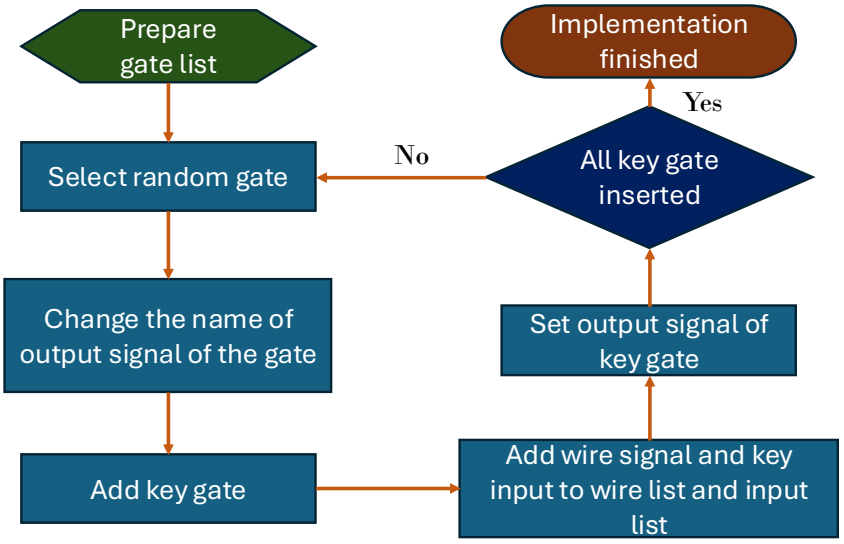

FIGURE 5.4: Working Flow of RLL in LockLab

The time consumption of RLL insertion on different benchmarks: b21(20027 gates), b18(111421 gates), s38417x(8709 gates), and s15850x(3448 gates) are shown in Figure 5.5, it can be seen that the overall running time increases as key-bit length and logic gate number in the circuit.

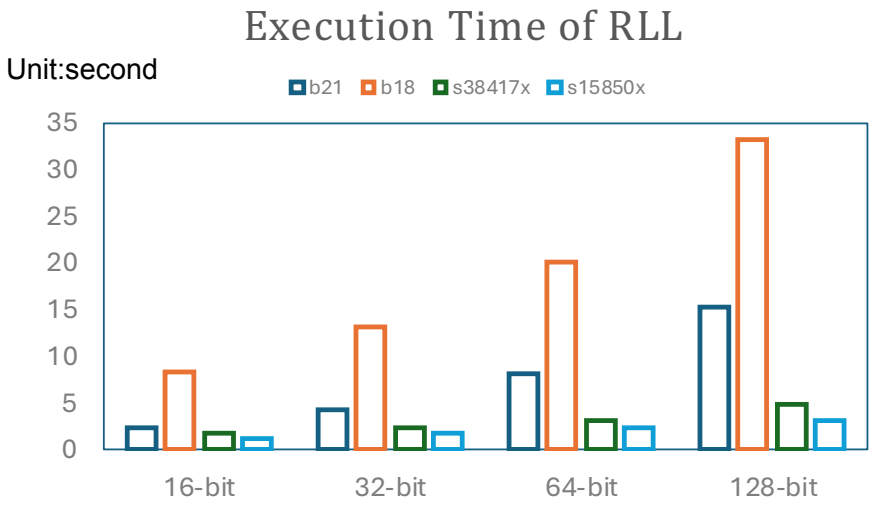

FIGURE 5.5: Execution Time of RLL

5.6.2 Fault-based Logic Locking (FLL)

In FLL (Fault-based Logic Locking), key-gates are strategically inserted at positions in the circuit that maximize the output corruption rate caused by an incorrect key. To achieve this, the fault impact is calculated for each gate, which is a numerical value derived from a combination of the gate's fan-in and fan-out. The fault impact serves as a metric for evaluating the effectiveness of a key-gate insertion position in terms of its potential to disrupt the circuit's functionality. By targeting gates with the highest fault impact, FLL increases the likelihood that an incorrect key will cause significant distortion in the circuit's output, thereby enhancing the circuit's security against unauthorized access or tampering. The working flow of Fault-based logic locking is shown in the flowchart:

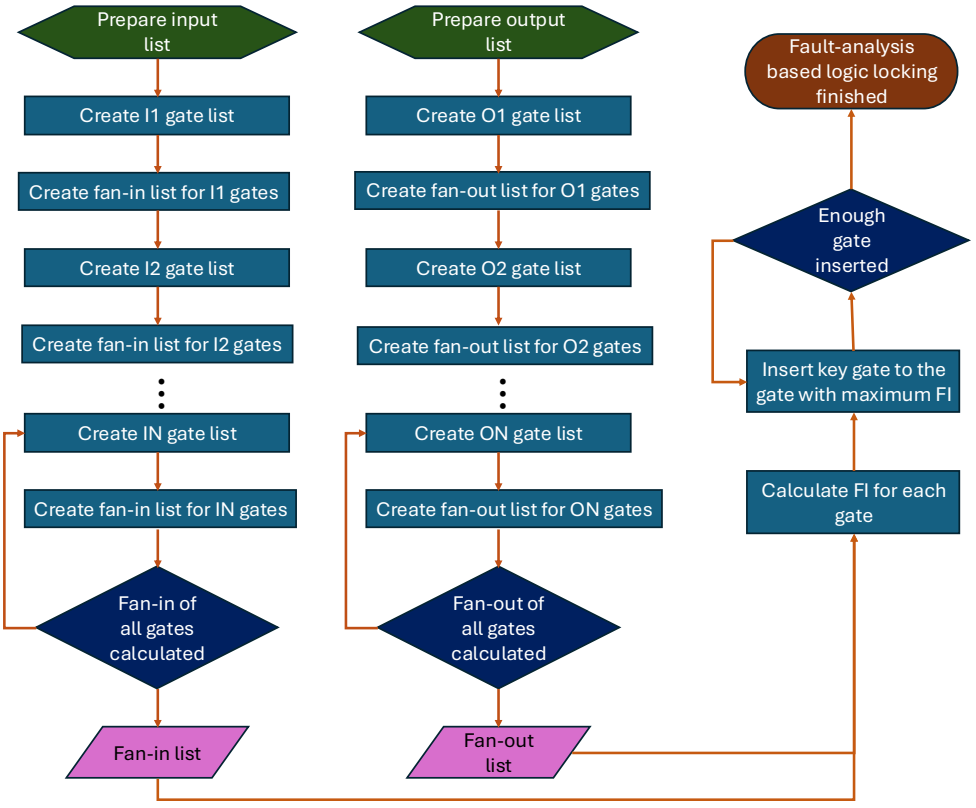

FIGURE 5.6: Working Flow of FLL in LockLab

The time consumption of FLL insertion on different benchmarks: b21(20027 gates), b18(111421 gates), s38417x(8709 gates), and s15850x(3448 gates) are shown in Figure 5.7, it can be seen that the overall running time increases as key-bit length and logic gate number in the circuit.

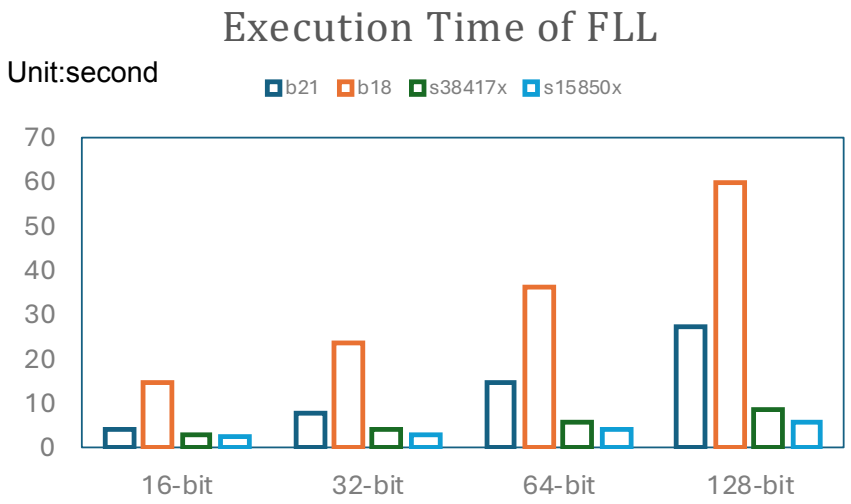

FIGURE 5.7: Execution Time of FLL

5.6.3 Strong Logic Locking (SLL)

SLL (Strong Logic Locking) is a locking strategy designed to defend against sensitization attacks. It achieves this by ensuring that all inserted key-gates are pairwise-secure, meaning that each key-gate is mutually protected, preventing one key-gate from being "muted" or bypassed by another. This pairwise security mechanism ensures that malicious attackers cannot exploit sensitization attacks to extract confidential information from the locked circuit. By maintaining the interdependence of the key-gates, SLL strengthens the integrity of the circuit, making it more resilient to attempts at reverse engineering and unauthorized access. The working flow of Strong logic locking is shown in the flowchart

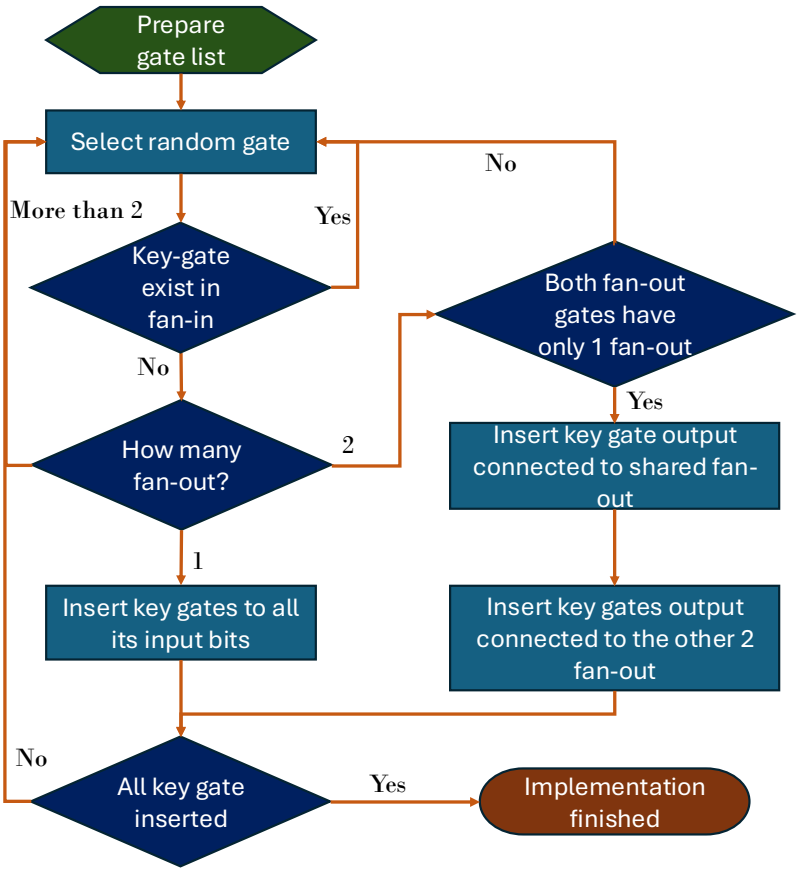

FIGURE 5.8: Working Flow of SLL in LockLab

The time consumption of SLL insertion on different benchmarks: b21(20027 gates), b18(111421 gates), s38417x(8709 gates), and s15850x(3448 gates) are shown in Figure 5.9, it can be seen that the overall running time increases as key-bit length and logic gate number in the circuit.

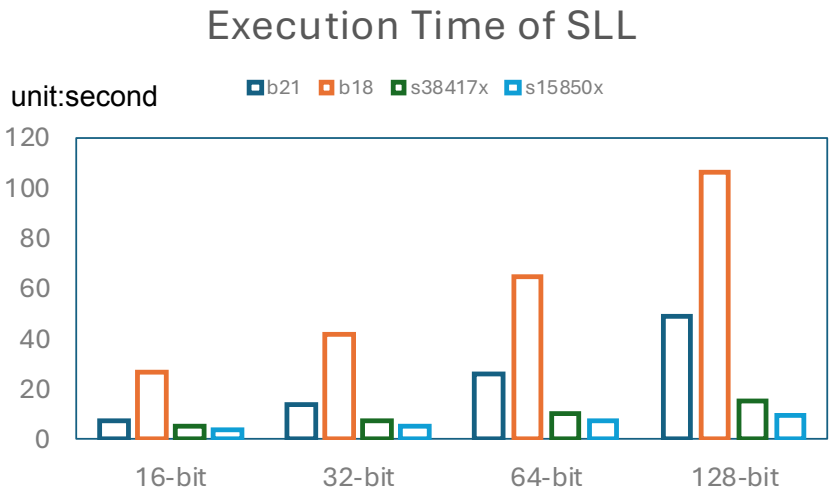

FIGURE 5.9: Execution Time of SLL

5.6.4 SARLock

SARLock is one of the earliest locking strategies proposed to counter SAT (Satisfiability) attacks. SARLock significantly increases the number of iterations required for a SAT attack by ensuring that each Distinguishing Input Pattern (DIP) can eliminate only a single incorrect key. This approach effectively raises the difficulty of cracking the protected circuit, thereby enhancing its resilience against SAT-based attacks. By limiting the impact of each input pattern on the key space, SARLock forces attackers to explore a much larger key space, making it more computationally expensive and time-consuming to break the lock. The working flow of LockLab for SARLock simulation is shown in the flowchart:

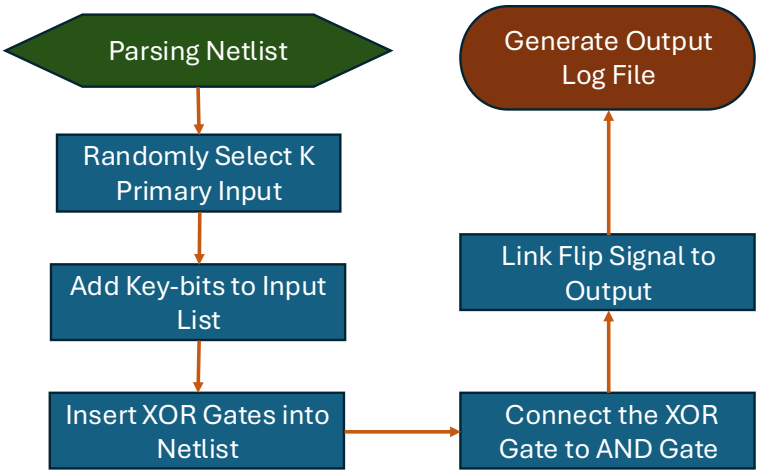

FIGURE 5.10: Working Flow of SARLock Simulation in LockLab

The execution time of SARlock simualtion is as Figure 5.11 shows, it can be seen that the overall execution time of SARlock increases as key-bit length grows. Another significant source of execution time is netlist parsing. In the simulation result benchmark b18 has total gate number of 111421, and s15850x has 3480. From the figure it can be seen that the total gate number of protected circuit does cause extra execution time, but not significant.

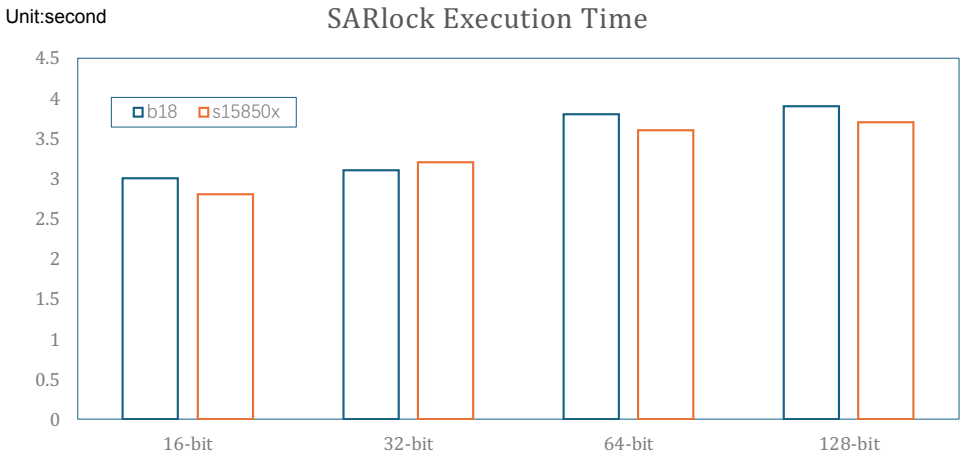

FIGURE 5.11: Execution Time of SARLock Simulation in LockLab

5.6.5 AntiSAT

Anti-SAT is another post-SAT locking strategy after the propose of SARLock, which performs higher output corruption rate than SARLock. The working flow of Anti-SAT attack is shown in the flowchart:

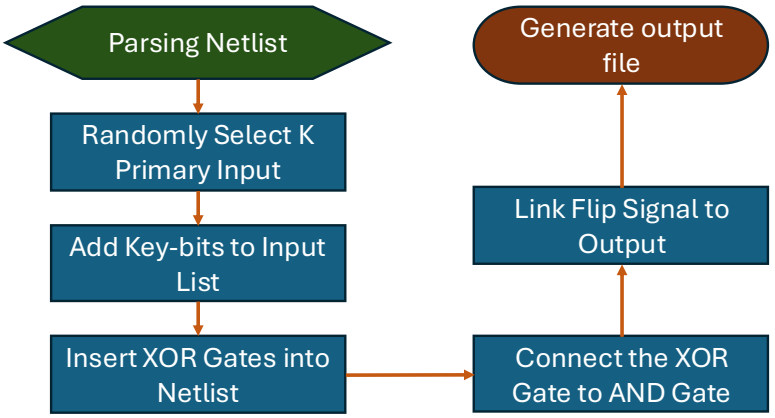

FIGURE 5.12: Working Flow of Anti-SAT Simulation in LockLab

The execution time of Anti-SAT simualtion is as Figure 5.13 shows, it can be seen that the overall execution time of SARlock increases as key-bit length and overall logic gate number in the circuit grow.

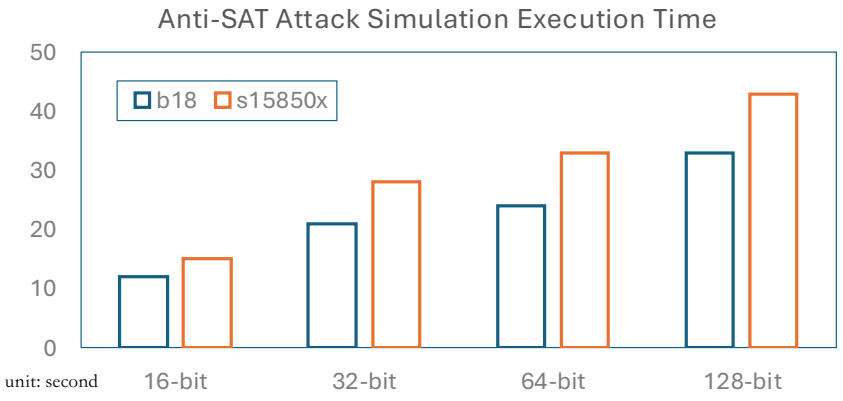

FIGURE 5.13: Execution Time of Anti-SAT Simulation in LockLab

5.6.6 SAT attack

LockLab is able to effectively simulate SAT attacks on a input protected circuit, as SAT is one of the most critical attacks against logic locking, understanding its impact is crucial. Users simply need to provide the netlist file of the circuit they wish to analyze. The software automatically detects the circuit's input and output bits, streamlining the simulation process. However, the key bits need to be manually inputted by the user. These key bits should be placed in a CSV file, which must be stored along with the Verilog netlist file in the 'datafile' folder located within the root directory of LockLab.

As highlighted in the previous section, the initial step in launching a SAT attack involves converting the gate-level netlist file into a CNF (Conjunctive Normal Form) file, which is then used by the SAT solver to initiate the attack simulation. This step ensures the attack framework can operate efficiently on the locked circuit.

5.6.6.1 SAT-solver and Output File

LockLab implement SAT attack using an open-source tool named 'MiniSAT', which is effecient and easy to use. After the attacking process is finished, an output file will be generated storing the output data. In the output file the following data are stored: whether the attack is successed, time consumption (in second), iteration number of the attack, average time consumption per iteration, all the key-bits and its corresponding key-bit in the original Verilog netlist file are listed.

5.6.6.2 Working Flow of SAT Attack

The working flow of SAT attak in LabLock is as follows:

1. **Parse the Netlist**

   The first step is to parse the netlist of the locked circuit. This involves extracting the structure and components of the circuit, such as gates, connections, and input/output pins, which are essential for performing the subsequent analysis and attack simulations.

2. **Input Key Input List**

Prior to conducting a SAT attack simulation, the user must provide the list of key inputs. This list contains the key bits that are used in the locked circuit, which are critical for generating the constraints needed for the SAT solver.

3. **Construct a Miter Circuit**

   The next step is to construct a Miter circuit by combining two identical locked circuits. These two circuits share the same primary input, and their outputs are compared. The outputs of the two circuits are XORed bit by bit, and all XOR gate outputs are then fed into an N-input OR gate, which generates a diff signal. This signal will indicate any discrepancies between the outputs of the two circuits, which is crucial for evaluating the success of the attack.

4. **Convert Verilog to CNF File**

   After generating the Miter circuit, it is converted into a Conjunctive Normal Form (CNF) file. CNF is a standard format used for encoding logical formulas, which is necessary for solving satisfiability problems using SAT solvers.

5. **Generate a Random Input Pattern as the DIP**

   A random DIP (Distinguishing Input Pattern) is generated, which is used to differentiate between the correct and incorrect keys. The DIP is then converted into a constraint that can be incorporated into the CNF file, adding complexity to the SAT attack.

6. **Apply Constraints to the CNF File**

   An additional constraint is applied to the Miter circuit's CNF file, which ensures that the keys used in the two protected circuits cannot be identical. This constraint is added to the CNF formula to make the attack more challenging by eliminating the possibility of the two circuits using the same key.

7. **Input the CNF File with Constraints into MiniSAT**

   The CNF file, now containing the constraints, is input into MiniSAT[130], a SAT solver. If a solution is found, proceed to Step 8. If no solution is found, proceed to Step 9.

8. **Add new constrain**

If a solution is found in Step 7, repeat Steps 1-6 to generate a new CNF file with additional constraints. The CNF file from the previous step is also incorporated into the new CNF file, applying the relevant input pattern and requiring the two key inputs to be identical as an additional constraint. Proceed to Step 7 to resolve the updated CNF file.

9. **Modify the Constraints**

   If no solution is found in Step 7, modify the constraints in the CNF file to require the two key inputs to be identical. Use MiniSAT to solve the modified CNF, and the output will provide the correct key.

10. **Output the Log File**

    Finally, output the log file, which contains the results of the SAT attack simulation, including whether the key was successfully found, the time taken for the attack, and any relevant details about the attack process. Additional Context:

5.6.7 SPS (Signal Probability Skew) Attack

LockLab can also allow users to simulate SPS attack, a kind of gate-level structure attack, which aims to find the gate with the largest ADS(Absolute difference of the probability skew). In PFB locking blocks such as AntiSAT [2, 138, 139], the logic gate with the highest ADS is often the locking gate that produces the flip signal. SPS attack helps the malicious attacker to locate that gate and mute the output flip signal, disabling the protection provided by the locking block.

The working flow of SPS attack is shown in the flow chart:

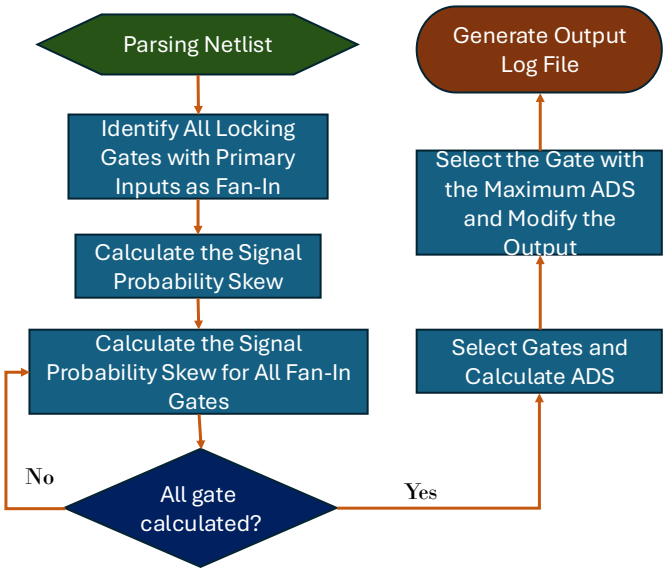

FIGURE 5.14: Working Flow of SPS Attack Simulation in LockLab

The execution time of SARlock simualtion is as Figure 5.11 shows, it can be seen that the overall execution time of SARlock increases as key-bit length grows. Another significant source of execution time is netlist parsing. In the simulation result benchmark b18 has total gate number of 111421, and s15850x has 3480. From the figure it can be seen that the total gate number of protected circuit does cause extra execution time, but not significant.

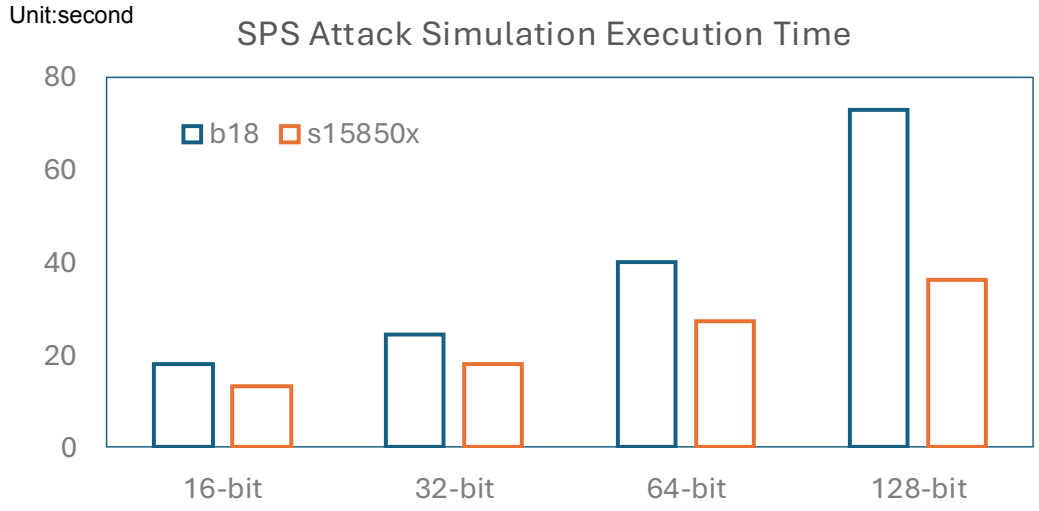

FIGURE 5.15: Execution Time of SPS Attack Simulation in LockLab

5.6.8 Sensitization Attack

The implementation is shown in the flow chart:

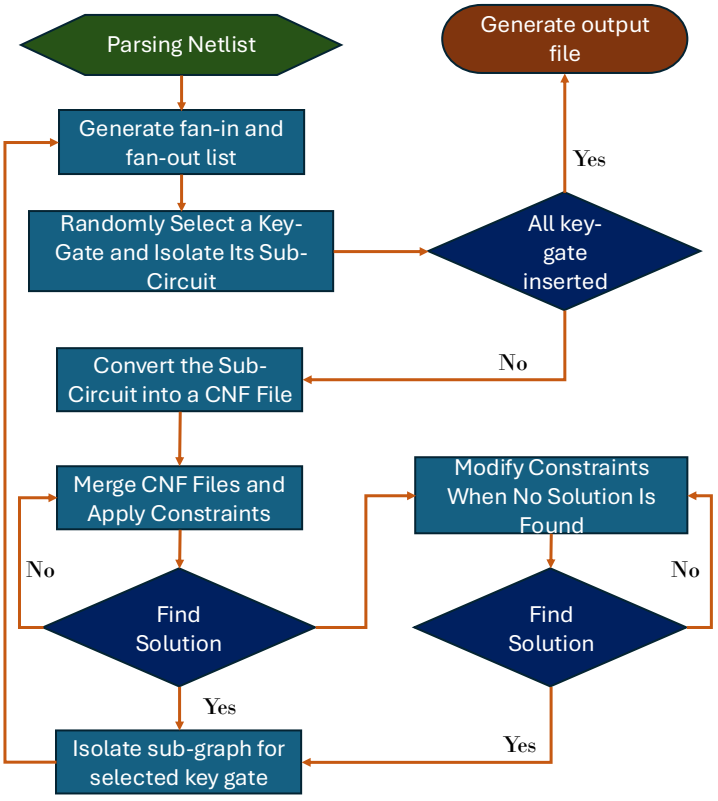

FIGURE 5.16: Working Flow of Sensitization Attack Simulation in LockLab

The execution time of Sensitization attack simualtion is as Figure 5.17 shows, it can be seen that the overall execution time of sensitization attack increases as key-bit length and overall logic gate number grow. In the simulation netlist of two benchmarks, b18 (111421 gates) and s15850x (3480 gates) are first locked with RLL and attack by sensitization attack.

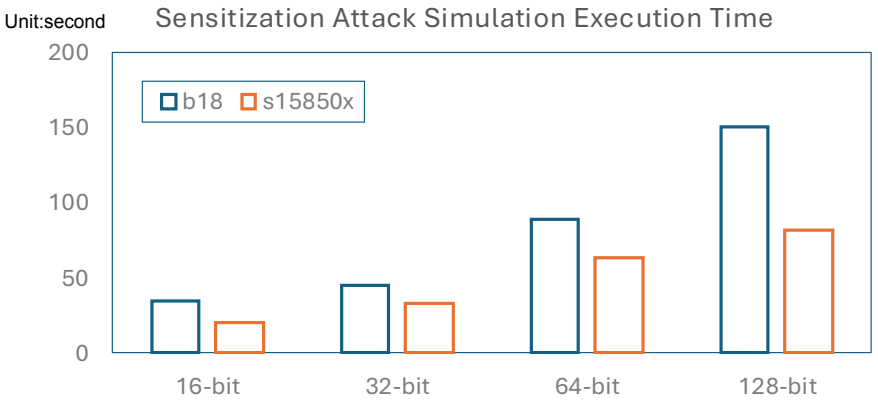

FIGURE 5.17: Execution Time of Sensitization Attack Simulation in LockLab

5.6.9 AI-based Attack Evaluation

AI-based attacks are among the fastest-growing threats to logic locking in recent years. Due to the rapid advancements in artificial intelligence (AI), AI-based attacks are expected to become a key research area in the future of hardware security. Currently, the primary AI-based attacks, such as Snapshot [119] and Sail [117, 118], target locking strategies like RLL (Random Logic Locking) and SLL (Strong Logic Locking), which rely on individual key-gates, rather than strategies like Anti-SAT and SFLL that are based on locking blocks. These AI-based attacks focus on inserting a single key-gate into a locked circuit, then learning the variations caused by the insertion of an unknown gate. By analyzing these changes, the AI model learns the synthesis tool's rules to infer the optimal placement of key-gates in the protected circuit.

Although LockLab does not directly simulate AI-based attacks, it can be used to evaluate a locked circuit's resistance to such attacks. This evaluation is achieved through assessing the positioning of the key-gates, thereby determining the overall resilience of the protected circuit to AI-based attacks.

The workflow for evaluating AI-based attack resistance is shown in the flow chart:

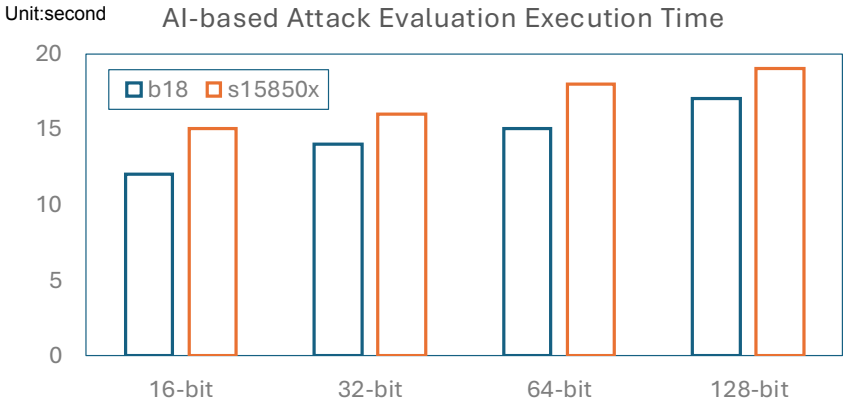

FIGURE 5.18: Execution Time of AI-based Attack Evaluation in LockLab

The execution time of AI-based Attack Evaluation in LockLab is as Figure 5.18 shows, it can be seen that the overall execution time increases as key-bit length and overall gate number grow. It can be seen that for the locking stratetgies that protect circuit with locking block, the key-gate often mixed with other inserted key-gates, which makes it harder for AI-based attack to distinguish the inserted key-gate. On the other hand, for approaches such as RLL or FLL which protect the circuit by single key-gates, AI-based attack has much higher threat.

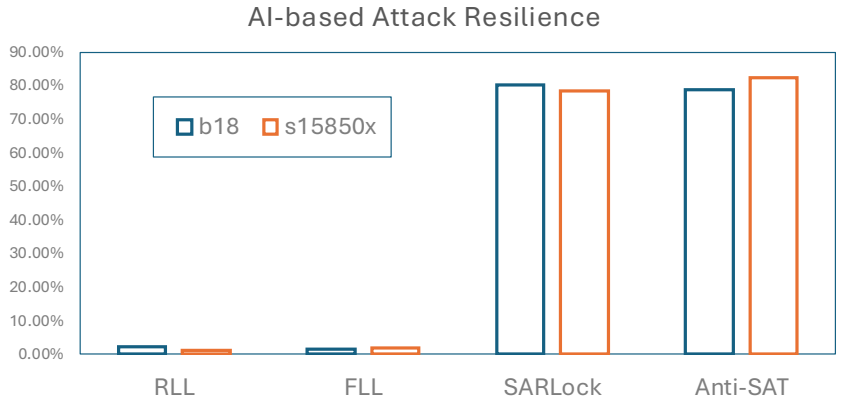

FIGURE 5.19: AI-based Attack reslience of RLL, FLL, SARLock, and Anti-SAT

5.6.10 Zeki implementation

LockLab can also implement Zeki on netlist file, the time consumption of different benchmarks with gate number ranging from 3k to 500k are shown in the table 5.2. And the detailed implementation working flow as explained in chapter 3.

TABLE 5.2: Benchmarks and corresponding time consumption to implement Zeki

Benchmark	Input Number	Output Number	Gate Number	ZeKi Running Time (s)
b21	522	512	20,027	6.7
b18	3,357	3,343	111,421	32.2
s38354x	38	304	11,448	4.2
s38417x	28	106	8,709	3.2
s15850x	77	150	3,448	1.3
sha256	678	258	51,019	15.2
sha256x10	6780	2580	510,190	161.7

5.7 Summary

In this chapter, I introduced LockLab, an automated tool designed to facilitate the implementation and evaluation of logic locking techniques for IC industry and can also be used for educational purpose. LockLab enables users to apply various locking strategies including RLL, FLL, SLL, Anti-SAT, SFLL, Zeki and SARLock, and offers an easy-to-use platform for testing the resistance of locked circuits against common attacks, including SAT-based, sensitization attack, SPS attack, and AI-based attacks evaluation.

The benchmarks used in the simulation has gate number ranging from 3k to 500k, and the simulation time consumption never exceed 200 second, so it is safe to say that Locklab is an effective tool.

LockLab provides users with a simple and intuitive automated tool for simulating a wide range of locking strategies and attacks. The tool features a clean and user-friendly interface (UI), making it easy to use. After each simulation, LockLab generates a detailed output log file, which allows users to review the results, including whether the newly inserted key-gates have been successfully applied or whether an attack was successful.

Upon inserting the locking mechanism into the circuit's netlist, LockLab also offers an automatic verification function. This function checks whether the circuit's functionality

remains intact when the correct key is applied, and whether the circuit produces output corruption when an incorrect key is provided. This process helps ensure that the logic locking mechanism is effectively securing the circuit while maintaining its intended operation under normal conditions.

The chapter outlined the detailed functionality of LockLab, including its ability to parse netlist files, insert key-gates, and perform functionality tests on the locked circuits to ensure that they maintain their intended output under correct key inputs.

Through these features, LockLab provides significant value for both academic and industrial users. For researchers, it serves as an educational tool for studying the fundamentals of logic locking and the impact of different strategies on circuit security. For engineers and designers, it offers a practical solution for testing and refining logic locking techniques to protect intellectual property (IP) from reverse engineering and unauthorized access.

Furthermore, LockLab's integration of automated testing and simulation reduces the need for external hardware simulators and manual testbench writing, significantly streamlining the verification process. By making the evaluation of logic locking more efficient and accessible, LockLab contributes to the ongoing effort to enhance hardware security in an era of increasing threats to integrated circuit design and hardware piracy.

As the IC industry continues to evolve with emerging attack methods and advanced reverse engineering techniques, tools like LockLab will play a crucial role in ensuring that hardware security remains a priority in the design and manufacturing processes. Future enhancements to LockLab, such as improved AI-based attack simulations and the incorporation of new locking strategies, will further strengthen its capabilities and help safeguard the integrity of semiconductor designs in a rapidly advancing technological landscape.

Chapter 6

Conclusions

6.1 Reflective Summary

This project focuses on two main objectives: the development of Dynamic Logic Locking and Zero-Knowledge Logic Locking through Zeki, and the creation of a simulation tool capable of supporting various gate-level locking strategies and attacks.

In contrast to traditional logic locking strategies, Zeki does not rely on a single locking mechanism. Instead, it employs a dynamic locking mechanism. For the circuit to be protected, Zeki randomly generates locking blocks based on different validation mechanisms, thus avoiding the structural vulnerabilities inherent in single-mechanism locking. Additionally, the dynamic nature of the locking mechanism further enhances isolation between different protected products. Even if an attacker successfully breaches one product's defence, they cannot use the same strategy to compromise other products, as each product's locking mechanism is uniquely generated.

Moreover, Zeki is the first logic locking strategy to implement zero-knowledge locking. Zero-knowledge logic locking takes security a step further by ensuring that no one, not even those with physical access to the locked circuit, can discern the key or the details of the locking mechanism. By minimizing the exposure of sensitive design details, this strategy strengthens defence against internal threats such as insider attacks, where trusted personnel might attempt to compromise the circuit's security. This approach, combined with the random generation of locking blocks, significantly reduces the threat from insider attacks, as no one knows the details of the locking blocks.

LockLab is a gate-level logic locking and attack simulation tool based on the concept behind Lockit. It allows users to insert various logic locking strategies into netlist files or simulate different types of attacks on locked circuits. The process is simple and user-friendly, making it highly convenient for researchers and students in the logic locking field. This tool greatly facilitates the study and exploration of logic locking techniques.

6.2 Main Contributions

6.2.1 Chapter 3

This chapter provides a comprehensive explanation of the motivation, operational principles, and implementation flow of Zeki. It compares the working principles of various PFB (Point-Function Based) locking validation formulas, from which the common characteristics of these strategies are derived. Using these characteristics, the chapter introduces the dynamic locking block, a randomly generated validation mechanism in Zeki. To further enhance the output corruption rate, SLL (Strong Logic Locking) is incorporated, along with a Key Verification Unit that also relies on a random mechanism to verify the correctness of the key.

Additionally, Zeki has been evaluated through various simulations, demonstrating its exceptional resilience against a range of attacks, including SAT attacks, SPS attacks, and sensitisation attacks.

6.2.2 Chapter 4

Given that the majority of products in the IC industry are based on sequential circuits rather than combinational circuits, Zeki has been adapted to include a version suitable for sequential circuits. This section discusses the implementation of Zeki in sequential circuits and evaluates the overhead incurred by Zeki in both combinational and sequential circuit benchmarks.

6.2.3 Chapter 5

The chapter also introduces LockLab and Lockit. Lockit, which I co-authored as the second author, is an SFLL (Strong Fault Logic Locking) automation tool described in my published paper. Building on this idea, I later developed LockLab. While Lockit primarily focuses on the SFLL locking strategy, LockLab offers users the flexibility to apply seven different logic locking strategies (RLL, FLL, SLL, Anti-SAT, SARLock, SFLL, and Zeki) to Verilog-format netlist files. Furthermore, LockLab enables the simulation of SAT attacks, sensitisation attacks, SPS attacks, and even AI-based attack evaluations. LockLab also features a user-friendly GUI and provides clear output logs after simulations, making it an accessible and valuable tool for researchers and learners in the field.

6.3 Limitations and Future Work

While the contributions of this thesis mark significant progress in the field, there are important limitations and directions for future exploration:

- **Overhead for small designs:** Zeki's area and logic overhead becomes disproportionately large for small-scale circuits (typically below 1,000 gates). The 64-bit and 128-bit locking blocks, while secure, can introduce more than 10% overhead in such designs, making them less practical for resource-constrained applications.
- **Benchmark limitations:** The current evaluation is based on established academic benchmarks (e.g., ITC'99, MCNC), which are relatively small and outdated. Future work should explore Zeki's scalability and performance on modern industrial-scale circuits, especially those derived from open-source SoC platforms or RISC-V cores.
- **Abstraction level:** LockLab currently operates at the gate-level. Integrating support for High-Level Synthesis (HLS) would align better with modern design practices and facilitate early-stage locking exploration.
- **Hybrid and advanced locking:** Future versions of Zeki may integrate with other advanced locking paradigms such as FSM-based locking, embedded FPGA

(eFPGA) security primitives, or runtime reconfigurable locking mechanisms, to provide broader protection coverage.

6.4 Achievement of Objectives

1. **To design a novel, dynamic logic locking mechanism.**

   This PhD project introduces *ZeKi*, a logic locking technique that generates Dynamic Logic Locking (DLL) security circuits through a *zero-knowledge* implementation. By employing a dynamic locking block, ZeKi ensures that each circuit is protected using a randomly generated and unique locking mechanism, thereby enhancing security and mitigating the risk of structural attacks. It enables hardware designers to implement logic locking without disclosing sensitive information, thus strengthening confidentiality. Overall, the proposed ZeKi framework contributes to the advancement of logic locking techniques and offers effective protection for integrated circuits.

2. **To reduce the risk of insider threats by minimizing key exposure.**

   The mechanism of ZeKi allows hardware designers to implement logic locking on protected circuits without revealing any locking information, including the secure key, locking block structure, key-bit arrangement, or locking mechanism—achieving true zero-knowledge locking. ZeKi's zero-knowledge property ensures that no party involved in the design or verification process gains access to the locking key or mechanism, significantly reducing the risk of insider threats.

3. **To develop a practical simulation platform for logic locking research.**

   LockLab, a novel software tool developed in this work, addresses current limitations in logic locking research by providing a practical simulation environment. The tool allows users to apply multiple logic locking strategies to gate-level netlist files, including RLL, FLL, SLL, ZeKi, Anti-SAT, SARLock, and SFLL. In addition, it supports the simulation of various attacks, such as SAT attacks, SPS attacks, sensitisation attacks, and the evaluation of AI-based attack methodologies.

4. **To ensure that the proposed locking strategy provides strong resilience against a broad range of attacks.**

This PhD work extensively evaluates the resilience of ZeKi against a range of attack types, including SAT attacks, sensitisation attacks, and structural attacks, using a diverse set of benchmark circuits. ZeKi demonstrates exponential resilience against SAT attacks, meaning that the number of DIPs (Distinguishing Input Patterns) required by an attacker grows exponentially with key length. In terms of sensitisation attacks, all logic gates introduced by ZeKi are 'pairwise-secured', a configuration defined in this thesis as '100-secure', ensuring that sensitisation attacks cannot compromise the circuit. For structural attacks, ZeKi uses randomly generated locking blocks, introducing significant variability across designs. Simulation results show that ZeKi achieves over 80% structural security across different benchmarks, validating its robustness.

5. **To evaluate the performance of the proposed mechanism.**

   ZeKi's performance has been evaluated using benchmark suites including ISCAS85, ITC'99, EPFL, and OpenCores, covering circuits with gate counts ranging from 3,000 to over 100,000. To test scalability on larger designs, a custom benchmark named *sha256x10* was created by parallelizing ten SHA-256 modules, resulting in a circuit with over 500,000 logic gates.

   Although the overhead introduced by ZeKi is more noticeable in smaller circuits (e.g., S15850x with 18.5% power and 17.2% area overhead), it becomes significantly lower in larger circuits. For example, in the b21 benchmark, ZeKi incurs only 2.1% power and 1.7% area overhead. Compared to LUT/FPGA-based logic locking methods and other high-overhead approaches, ZeKi achieves the lowest area and power overhead among state-of-the-art techniques. These results demonstrate not only ZeKi's efficiency but also its excellent scalability for larger designs.

6.5 Final Remarks and Broader Impact

This thesis contributes to the ongoing evolution of logic locking by introducing Zeki, a zero-knowledge, dynamic locking framework that prioritizes both structural unpredictability and minimal key exposure. Through the development of LockLab, it also provides the research community with a powerful and accessible platform for simulating diverse locking strategies and attacks.

Together, these contributions demonstrate that logic locking can be made both more secure and more adaptable to the complex needs of modern IC design. Although limitations remain, especially in overhead for small designs, the proposed methods represent a step toward scalable and trustworthy hardware protection. Future work can build upon this foundation by integrating Zeki with emerging paradigms such as reconfigurable logic, post-quantum security primitives, and high-level synthesis flows.

The broader implication of this work lies in reinforcing the security of the semiconductor supply chain in an era of increasing globalization and design outsourcing.

Appendix A

Source code of LockLab

```
    #!/user/bin/env python3
# -i- coding: UTF-8 -*-i-

import re
import random
import copy
import readverilog


def read_user(filename):
    user_in = [None] * 3
    with open(filename, 'r') as f:
        user_in[0] = (str(f.readline())).strip('\n')
        user_in[1] = int(f.readline())
        user_in[2] = list((str(f.readline()).strip('\n')))

    if len(user_in[2]) == user_in[1]:
        print('length fit')
    else:
        print('length not fit')
    return user_in


# output key input name list
def key_inputs(key_no):
    key_input_name = [None] * key_no
    for ki in range(key_no):
        key_input_name[ki] = str('keybit' + str(ki + 1))
    return key_input_name


# output key wire name list
```

```
def key_points(key_no):
    key_point_name = [None] * key_no
    for ki in range(key_no):
        key_point_name[ki] = str('keypoint' + str(ki))
    return key_point_name


###############
def modi_module(circuit1, filename, key_input_name):
    module_symbol = circuit1.inputlist + circuit1.outputlist

    with open(filename, 'a+') as f1:
        print('module output opened successfully')
        i2 = 1
        f1.write('module ' + circuit1.name + ' (')
        for l1 in module_symbol:
            if i2 == 1:
                f1.write(l1)
            elif (i2 > 9) and (i2 % 10 == 1):
                f1.write(',' + '\n' + l1)
            else:
                f1.write(',' + l1)
            i2 += 1
        f1.write(',' + '\n\n')

        i3 = 1
        for l2 in key_input_name:
            if i3 == 1:
                f1.write(l2)
            elif (i3 > 9) and (i3 % 10 == 1):
                f1.write(',' + '\n' + l2)
            else:
                f1.write(',' + l2)
            i3 += 1
        f1.write(');' + '\n\n\n')
        print('module list output finished')


# filename1 is the name of netlist file
# filename2 is the name of output file
# key_input_name is the list of key input name
# ln is the end line number after reading of module finished
def modi_input(input_symbol, filename1, key_input_name):
    with open(filename1, 'a+') as f2:
        print('input opened successfully')
        i1 = 1
        f2.write('input  ')
        for l in input_symbol:
```

```
            if (i1 > 9) and (i1 % 10 == 0):
                f2.write(l + ',' + '\n')
            else:
                f2.write(l + ',')
            i1 += 1
        f2.write('\n')

        i2 = 1
        for l1 in key_input_name:
            if i2 == 1:
                f2.write(l1)
            elif (i2 > 9) and (i2 % 10 == 1):
                f2.write(',' + '\n' + l1)
            else:
                f2.write(',' + l1)
            i2 += 1
        f2.write(';' + '\n\n\n')
        print('input list finished')


def modi_wire(wire_symbol, filename, key_gate_name):
    with open(filename, 'a+') as f2:
        print('wire opened successfully')
        i1 = 1
        f2.write('wire  ')
        for l in wire_symbol:
            if (i1 > 9) and (i1 % 10 == 0):
                f2.write(l + ',' + '\n')
            else:
                f2.write(l + ',')
            i1 += 1
        f2.write('\n')

        i2 = 1
        for l1 in key_gate_name:
            if i2 == 1:
                f2.write(l1)
            elif (i2 > 9) and (i2 % 10 == 1):
                f2.write(',' + '\n' + l1)
            else:
                f2.write(',' + l1)
            i2 += 1
        f2.write(';' + '\n\n\n')
        print('wire list finished')


def modi_output(output_symbol, filename):
    with open(filename, 'a+') as f2:
```

```
        print('output opened successfully')
        i2 = 1
        f2.write('output ')
        for l1 in output_symbol:
            if i2 == 1:
                f2.write(l1)
            elif (i2 > 9) and (i2 % 10 == 1):
                f2.write(',' + '\n' + l1)
            else:
                f2.write(',' + l1)
            i2 += 1
        f2.write(';' + '\n\n\n')
        print('output list finished')


def write_gate(filename, gate_list):
    with open(filename, 'a+') as f:
        for g1 in gate_list:
            f.write(g1)
        f.write('\n' + 'endmodule')


########################################################################


def modi_gate(userin, keynamelist, keywirelist, gate_symbol, gate_data, input_symbol):
    randnolist = []
    print(gate_data)

    for ir in range(userin[1]):
        while 1:
            temprand = random.randint(0, len(gate_data) - 1)
            if gate_data[temprand][3][0] not in input_symbol:
                randnolist.append(temprand)
                break
    ranlen = set(randnolist)
    if len(ranlen) != len(randnolist):
        return '', 1

    print(randnolist)
    keygatelist = []
    for gr in range(userin[1]):
        keyinput = copy.deepcopy(gate_data[randnolist[gr]][3][0])
        gate_data[randnolist[gr]][3][0] = keywirelist[gr]
        if userin[2][gr] == '0':
            gatetype = 'xor'
        elif userin[2][gr] == '1':
```

```
            gatetype = 'xnor'
        else:
            print('gatetype error')
            gatetype = 'xor'

        keygatelist.append([gatetype + ' ' + 'keygate' + str(gr) + ' (' + keywirelist[gr] +
                            ', ' + keynamelist[gr] + ', ' + keyinput + ');'])

    newlist = []
    for gg1 in gate_data:
        inlist = ''
        for il in gg1[3]:
            inlist += ', ' + il
        newlist.append([gg1[0] + ' ' + gg1[1] + ' ' + ' (' + gg1[2] + inlist + ');'])

    newlist += keygatelist
    # print(newlist[10])
    print(newlist[-1])

    newtxt = '\n'
    for nn1 in newlist:
        newtxt = newtxt + nn1[0] + '\n'

    return newtxt, 0


def unill(userfile, filein, fileout):
    userin = read_user(userfile)
    print(userin)

    keynamelist = key_inputs(userin[1])
    keywirelist = key_points(userin[1])
    readline1 = modi_module(filein, fileout, userin[0][:-2], keynamelist)

    readline2, input_symbol = modi_input(filein, fileout, keynamelist, readline1)
    readline3, output_symbol = modi_output(filein, fileout, readline2)
    readline4, wire_symbol = modi_wire(filein, fileout, keywirelist, readline3)
    gate_no, gate_symbol, gate_data = read_gate(filein, readline4)

    keygatelist, failflag = modi_gate(userin, keynamelist, keywirelist, gate_symbol, gate_data,

    if not failflag:
        write_gate(fileout, keygatelist)

    return input_symbol, output_symbol, wire_symbol, gate_symbol, failflag
```

```
import z3

import benchmarks
import circuit
import dip_finder
import sat_model
import oracle_runner

class SatAttack:
    """The main class for conducting the SAT attack."""

    def __init__(self, locked_filename, unlocked_filename):
        self.locked_filename = locked_filename
        self.unlocked_filename = unlocked_filename
        self.iterations = 0

    def run(self):
        """Run the SAT attack."""
        print("Reading in locked circuit...")
        self.nodes, self.output_names = benchmarks.read_nodes(self.locked_filename)

        print("Reading in unlocked circuit...")
        self.oracle_ckt = benchmarks.read_ckt(self.unlocked_filename)

        key_inputs = [node.name for node in self.nodes.values() if node.type == "Key Input"]
        primary_inputs = [node.name for node in self.nodes.values() if node.type == "Primary Input"]

        print("\n# Primary Inputs: %i" % (len(primary_inputs)))
        print("# Key Inputs: %i" % (len(key_inputs)))

        finder = dip_finder.DipFinder(self.nodes, self.output_names)
        runner = oracle_runner.OracleRunner(self.oracle_ckt)

        oracle_io_pairs = []
        while finder.can_find_dip():
            dip = finder.find_dip()
            oracle_output = runner.run(dip)
            finder.add_constraint(dip, oracle_output)

            oracle_io_pairs.append((dip, oracle_output))
            self.iterations += 1

        key = self._find_key(oracle_io_pairs, key_inputs)
        expected_key = benchmarks.get_expected_key(self.locked_filename)
```

```
        #print("\nExpected key: %s" % (self._key_string(expected_key)))
        #print("Found key:    %s" % (self._key_string(key)))

        print("\nChecking for circuit equivalence...\n")
        self._check_key(key)
        if self._check_key(key):
            print("Locked and unlocked circuits match")
        else:
            print("Key found does not match oracle")

    def _find_key(self, oracle_io_pairs, key_names):
        """
        Find a key that satisfies all DIPs found during the SAT attack.
        This key will be in the set of correct keys.

        oracle_io_pairs: array of dip/output pairs in the form of (dip, output)

        returns: key that satisfies all dip constraints
        """

        s = z3.Solver()

        for io_pair in oracle_io_pairs:
            dip = io_pair[0]
            output = io_pair[1]

            constraint_ckt = circuit.Circuit.specify_inputs(dip, self.nodes, self.output_names)
            output_constraints = [constraint_ckt.outputs()[name] == output[name] for name in out

            s.add(*output_constraints)

        s.check()
        model = s.model()
        key = sat_model.extract_from_model(model, key_names, completion=True)
        return key

    def _check_key(self, key):
        """
        Check that the key returned from the SAT attack is correct. It
        does this by creating a miter circuit with a locked version
        and an oracle. If the diff signal returned from the miter circuit
        cannot be True, then the circuits are equivalent.

        key: the key returned from the SAT attaack
        """

        locked_ckt = circuit.Circuit.specify_inputs(key, self.nodes, self.output_names)
        miter = circuit.Circuit.miter(locked_ckt, self.oracle_ckt)
```

```
        s = z3.Solver()
        s.add(miter.outputs()["diff"] == True)

        return s.check() == z3.unsat

    def _key_string(self, key):
        ordered_names = sorted(key.keys(), key=lambda name: str(name[8:]))
        key_string = ""

        for name in ordered_names:
            if key[name]:
                key_string += "1"
            else:
                key_string += "0"

        return key_string

def read_user(filename):
    user_in = [None] * 3
    with open(filename, 'r') as f:
        user_in[0] = (str(f.readline())).strip('\n')
        user_in[1] = int(f.readline())
        user_in[2] = list((str(f.readline()).strip('\n')))

    print('Locking key: '+str(user_in[2]))
```

```
from node import Node
from token_type import TokenType

class Parser():
    def parse(self, tokenizer):
        """
        Parses circuit nodes given a list of tokens from an input verilog file.

        tokenizer: Tokenizer object for the Verilog input file
        returns: the nodes of the circuit, the names of the output nodes
        """
        self.outputs = []
        self.nodes = {}

        while True:
            token_type = tokenizer.get_token_type()
```

```
            if token_type == TokenType.EOF:
                break
            elif token_type == TokenType.INPUT:
                self._parse_inputs(tokenizer)
            elif token_type == TokenType.OUTPUT:
                self._parse_outputs(tokenizer)
            elif token_type == TokenType.WIRE:
                self._parse_wires(tokenizer)
            elif token_type == TokenType.AND:
                self._parse_gate(tokenizer, "And")
            elif token_type == TokenType.XOR:
                self._parse_gate(tokenizer, "Xor")
            elif token_type == TokenType.OR:
                self._parse_gate(tokenizer, "Or")
            elif token_type == TokenType.NOT:
                self._parse_gate(tokenizer, "Not")
            elif token_type == TokenType.NAND:
                self._parse_gate(tokenizer, "Nand")
            elif token_type == TokenType.XNOR:
                self._parse_gate(tokenizer, "Xnor")
            elif token_type == TokenType.NOR:
                self._parse_gate(tokenizer, "Nor")
            else:
                tokenizer.skip_token()

        return self.nodes, self.outputs

    def _parse_inputs(self, tokenizer):
        """
        Parses input nodes, both key and primary inputs

        tokenizer: the Tokenizer object with the verilog input
        """
        tokenizer.skip_token() # input token

        while True:
            # This check is NOT robust and could be improved probably
            if "key" in tokenizer.id_value():
                self.nodes[tokenizer.id_value()] = Node(tokenizer.id_value(), [], "Key Input")
            else:
                self.nodes[tokenizer.id_value()] = Node(tokenizer.id_value(), [], "Primary Input

            tokenizer.skip_token()

            if tokenizer.get_token_type() == TokenType.SEMICOLON:
                tokenizer.skip_token()
                break
            else:
```

```
            tokenizer.skip_token()

    def _parse_outputs(self, tokenizer):
        """
        Parses output nodes

        tokenizer: the Tokenizer object with the verilog input
        """
        tokenizer.skip_token() # output token

        while True:
            self.outputs.append(tokenizer.id_value())
            self.nodes[tokenizer.id_value()] = Node(tokenizer.id_value(), [], "Output")
            tokenizer.skip_token()

            if tokenizer.get_token_type() == TokenType.SEMICOLON:
                tokenizer.skip_token()
                break
            else:
                tokenizer.skip_token()

    def _parse_wires(self, tokenizer):
        """
        Parses wire nodes

        tokenizer: the Tokenizer object with the verilog input
        """
        tokenizer.skip_token() # wire token

        while True:
            if tokenizer.get_token_type() == TokenType.SEMICOLON:
                tokenizer.skip_token() # semicolon
                break
            elif tokenizer.get_token_type() == TokenType.COMMA:
                tokenizer.skip_token() # comma
            elif tokenizer.get_token_type() == TokenType.LEFT_BRACKET:
                self._parse_bus(tokenizer)
            else:
                self._parse_single_wire(tokenizer)

    def _parse_bus(self, tokenizer):
        """
        Parses wire nodes that are buses

        tokenizer: the Tokenizer object with the verilog input
        """
        tokenizer.skip_token() # left bracket
        low_number = tokenizer.int_value()
```

```
        tokenizer.skip_token() # low number
        tokenizer.skip_token() # colon
        high_number = tokenizer.int_value()
        tokenizer.skip_token() # high value
        tokenizer.skip_token() # right bracket
        bus_name = tokenizer.id_value()
        tokenizer.skip_token() # bus name

        for i in range(high_number - low_number + 1):
            wire_name = bus_name + "__index" + str(i)
            self.nodes[wire_name] = Node(wire_name, [], "Wire")

    def _parse_single_wire(self, tokenizer):
        """
        Parses a single wire node

        tokenizer: the Tokenizer object with the verilog input
        """
        self.nodes[tokenizer.id_value()] = Node(tokenizer.id_value(), [], "Wire")
        tokenizer.skip_token()

    def _parse_gate(self, tokenizer, gate_type):
        """
        Parses a gate node

        tokenizer: the Tokenizer object with the verilog input
        """
        tokenizer.skip_token() # gate token
        tokenizer.skip_token() # gate identifier token
        tokenizer.skip_token() # left paren token

        output_name = self._parse_id(tokenizer)
        tokenizer.skip_token() # comma

        inputs = []
        while True:
            if tokenizer.get_token_type() == TokenType.RIGHT_PAREN:
                tokenizer.skip_token() # right paren
                break
            elif tokenizer.get_token_type() == TokenType.IDENTIFIER:
                input_name = self._parse_id(tokenizer)
                inputs.append(input_name)
            elif tokenizer.get_token_type() == TokenType.COMMA:
                tokenizer.skip_token() # comma
            else:
                print("Error: unexpected token type " + tokenizer.get_token_type())
                raise
```

```
        tokenizer.skip_token() # semicolon
        self.nodes[output_name].inputs = inputs
        self.nodes[output_name].type = gate_type

    def _parse_id(self, tokenizer):
        """
        Parses an identifier name

        tokenizer: the Tokenizer object with the Verilog input
        """
        id_name = tokenizer.id_value()
        tokenizer.skip_token() # id

        if tokenizer.get_token_type() == TokenType.LEFT_BRACKET:
            tokenizer.skip_token() # left bracket
            index = tokenizer.int_value()
            tokenizer.skip_token() # number
            tokenizer.skip_token() # right bracket

            return id_name + "__index" + str(index)
        else:
            return id_name
```

```
from z3 import *

class CircuitBuilder():
    def build_miter(self, ckt0, ckt1):
        """
        Builds a miter circuit z3 representation from two smaller circuits."

        ckt0: the first half of the miter circuit
        ckt1: the second half of the miter circuit
        returns: a miter circuit z3 representation
        """
        output_xors = [Xor(ckt0.outputs()[name], ckt1.outputs()[name]) for name in ckt0.outputs()]
        diff = Or(*output_xors)
        return {"diff": diff}

    def build(self, nodes, output_names, key_suffix = "", spec_inputs = None):
        """
        Builds a circuit z3 representation from a list of nodes in the circuit.

        nodes: the nodes in the circuit
        output_names: the names of the output nodes
        key_suffix: suffix to apply to all key names
```

```
        spec_inputs: inputs to be replace by a value
        returns: a z3 representation for the outputs of the circuit
                 corresponding to the nodes passed in
        """
        self.visited_nodes = []
        self.inputs = []
        self.specified_inputs = spec_inputs
        outputs = {}

        for name in output_names:
            outputs[name] = self._build_node(nodes, name, key_suffix)

        return outputs, self.inputs

    def _build_node(self, nodes, name, key_suffix):
        """
        Returns the z3 representation for a single node.

        nodes: a list of all nodes in the circuit
        name: the name of the node to build
        key_suffix: the suffix to apply to key names
        """
        node = nodes[name]

        if name in self.visited_nodes:
            return node.z3_repr

        self.visited_nodes.append(name)

        if node.type == "Key Input":
            self._build_key(node, name, key_suffix)
        elif node.type == "Primary Input":
            self._build_input(node, name)
        else:
            fanin = [self._build_node(nodes, child_name, key_suffix) for child_name in node.inpu
            self._build_gate(node, fanin)

        return node.z3_repr

    def _build_gate(self, node, fanin):
        """
        Sets the z3 representation for a logic gate node.

        node: the node to find the z3 representation for
        fanin: the input nodes the the node
        """
        if node.type == "And":
            node.z3_repr = And(*fanin)
```

```
        elif node.type == "Xor":
            node.z3_repr = Xor(*fanin)
        elif node.type == "Or":
            node.z3_repr = Or(*fanin)
        elif node.type == "Not":
            node.z3_repr = Not(*fanin)
        elif node.type == "Nand":
            node.z3_repr = Not(And(*fanin))
        elif node.type == "Xnor":
            node.z3_repr = Not(Xor(*fanin))
        elif node.type == "Nor":
            node.z3_repr = Not(Or(*fanin))
        else:
            print("Unknown node type " + str(node))
            raise

    def _build_key(self, node, name, key_suffix):
        """
        Sets the z3 representation for a key input node

        node: the node to find the z3 representation for
        name: the name of the key
        key_suffix: the suffix to apply to the key
        """
        key_name = name + key_suffix

        if self.specified_inputs is not None and name in self.specified_inputs:
            node.z3_repr = self.specified_inputs[key_name]
        else:
            self.inputs.append(key_name)
            node.z3_repr = Bool(key_name)

    def _build_input(self, node, name):
        """
        Sets the z3 representation for a primary input node

        node: the node to find the z3 representation for
        name: the name of the key
        """
        if self.specified_inputs is not None and name in self.specified_inputs:
            node.z3_repr = self.specified_inputs[name]
        else:
            self.inputs.append(name)
            node.z3_repr = Bool(name)
```

Bibliography

[1] 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.

[2] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38:199–207, 2019.

[3] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1601–1618, 2017.

[4] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J V Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 236–241, 2016.

[5] Joseph Sweeney, Marijn J Heule, and Lawrence Pileggi. Sensitivity analysis of locked circuits. EPiC Series in Computing, 73, 2020.

[6] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Removal attacks on logic locking and camouflaging techniques. IEEE Transactions on Emerging Topics in Computing, 8:517–532, 2020.

[7] Hadi Mardani Kamali, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. Advances in logic locking: Past, present, and prospects. Cryptology ePrint Archive, 2022.

[8] Muhammad Yasin, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Trustworthy Hardware Design: Combinational Logic Locking Techniques. Springer International Publishing, 2020.

[9] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. Epic: Ending piracy of integrated circuits. In 2008 Design, Automation and Test in Europe, pages 1069–1074, 2008.

[10] Nemanja Kajtez, Yue Zhang, and Basel Halak. Lockit: A logic locking automation software. Electronics, 10:2817, 11 2021.

[11] Bicky Shakya, Mark M Tehranipoor, Swarup Bhunia, and Domenic Forte. Introduction to hardware obfuscation: Motivation, methods and evaluation. In Hardware Protection through Obfuscation, pages 3–32. Springer, 2017.

[12] Mohammad Tehranipoor and Cliff Wang. Introduction to hardware security and trust. Springer New York, NY, 2011.

[13] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. Proceedings of the IEEE, 102(8):1283–1295, 2014.

[14] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Larry Pileggi, and Ken Mai. Hardware redaction via designer directed fine grained efpga insertion. In 2021 Design, Automation and Test in Europe Conference and Exhibition, pages 1186–1191, Feb 2021.

[15] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. Proceedings of the IEEE, 102:1283–1295, 2014.

[16] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. Trustworthy hardware: Identifying and classifying hardware trojans. Computer, 43:39–46, 2010.

[17] Michael S. Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. Proceedings of the IEEE, 102:1229–1247, 2014.

[18] R Naveenkumar and NM Sivamangai. Hardware trojans detection and prevention techniques review. Wireless Personal Communications, 136(2):1147–1182, 2024.

[19] Fangzhou Wang, Qijing Wang, Lilas Alrahis, Bangqi Fu, Shui Jiang, Xiaopeng Zhang, Ozgur Sinanoglu, Tsung-Yi Ho, Evangeline FY Young, and Johann Knechtel. Trolloc: Logic locking and layout hardening for ic security closure against hardware trojans. arXiv preprint arXiv:2405.05590, 2024.

[20] Karen Goertzel. Integrated circuit security threats and hardware assurance countermeasures. CrossTalk, 26:33–38, 2013.

[21] Randy Torrance and Dick James. The state-of-the-art in semiconductor reverse engineering. In Proceedings of the 48th Design Automation Conference, pages 333–338, 2011.

[22] Ujjwal Guin, Daniel DiMase, and Mohammad Tehranipoor. Counterfeit integrated circuits: Detection, avoidance, and the challenges ahead. Journal of Electronic Testing, 30:9–23, 2 2014.

[23] Ujjwal Guin, Ke Huang, Daniel DiMase, John M. Carulli, Mohammad Tehranipoor, and Yiorgos Makris. Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain. Proceedings of the IEEE, 102:1207–1228, 8 2014.

[24] Andrew E. Caldwell, Hyun-Jin Choi, Andrew B. Kahng, Stefanus Mantik, Miodrag Potkonjak, Gang Qu, and Jennifer L. WongAuthors. Effective iterative techniques for fingerprinting design ip. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23:208–215, 2004.

[25] Andrew B Kahng, John Lach, William H Mangione-Smith, Stefanus Mantik, Igor L Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking techniques for intellectual property protection. In Proceedings of the 35th annual Design Automation Conference, pages 776–781, 1998.

[26] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of integrated circuit camouflaging. In Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security. ACM Press, 2013.

[27] Muhammad Liman Gambo and Ahmad Almulhem. Zero trust architecture: A systematic literature review. arXiv preprint arXiv:2503.11659, 2025.

[28] Richard Wayne Jarvis and Michael G McIntyre. Split manufacturing method for advanced semiconductor circuits, March 27 2007. US Patent 7,195,931.

[29] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh Tripunitara. Securing computer hardware using 3d integrated circuit technology and split manufacturing for obfuscation. In 22nd USENIX Security Symposium (USENIX Security 13), pages 495–510, 2013.

[30] Domenic Forte, Swarup Bhunia, and Mark M. Tehranipoor. Hardware Protection Through Obfuscation. Springer Series in Advanced Microelectronics. Springer Cham, Cham, Switzerland, 2017.

[31] Yujie Wang, Pu Chen, Jiang Hu, and Jeyavijayan JV Rajendran. Routing perturbation for enhanced security in split manufacturing. In 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), pages 605–510. IEEE, 2017.

[32] Kun Yang, Ulbert Botero, Haoting Shen, Domenic Forte, and Mark Tehranipoor. A split manufacturing approach for unclonable chipless rfids for pharmaceutical supply chain security. In 2017 Asian hardware oriented security and trust symposium (AsianHOST), pages 61–66. IEEE, 2017.

[33] Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, Benjamin Tan, Christian Pilato, Ganesh Gore, Xifan Tang, Scott Temple, Pierre-Emmanuel Gaillardon, and Ramesh Karri. Exploring efpga-based redaction for ip protection. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9, Nov 2021.

[34] Bo Hu, Jingxiang Tian, Mustafa Shihab, Gaurav Rajavendra Reddy, William Swartz, Yiorgos Makris, Benjamin Carrion Schaefer, and Carl Sechen. Functional obfuscation of hardware accelerators through selective partial design extraction onto an embedded fpga. In Proceedings of the 2019 on Great Lakes Symposium on VLSI, pages 171–176, 2019.

[35] IEEE Standards Association. IEEE Standard for Encrypted Electronic Design Intellectual Property (IP) – IEEE 1735-2014. https://standards.ieee.org/ieee/1735/6000/, 2014. DOI: https://doi.org/10.1109/IEEESTD.2014.6893771.

[36] Ujjwal Guin, Qihang Shi, Domenic Forte, and Mark M Tehranipoor. Fortis: a comprehensive solution for establishing forward trust for protecting ips and ics. ACM transactions on design automation of electronic systems (TODAES), 21(4):1–20, 2016.

[37] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. Provably secure camouflaging strategy for ic protection. IEEE transactions on computer-aided design of integrated circuits and systems, 38(8):1399–1412, 2017.

[38] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Logic encryption: A fault analysis perspective. In 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 953–958. IEEE, 2012.

[39] 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, 64:410–424, 2015.

[40] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. Proceedings - Design Automation Conference, pages 83–89, 2012.

[41] 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, 35:1411–1424, 2016. SLL.

[42] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Circuit obfuscation and oracle-guided attacks: Who can prevail? In Proceedings of the Great Lakes Symposium on VLSI 2017, pages 357–362, 2017.

[43] Whitesitt. J. Eldon. Boolean algebra and Its applications. DOVER publications, 2013.

[44] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In International Conference on Formal Methods in Computer-Aided Design, volume 1954, pages 354–371. Springer, 2000.

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

[46] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan Rajendran. What to lock? functional and parametric locking. In Proceedings of the Great Lakes Symposium on VLSI 2017, pages 351–356, 2017.

[47] Abhrajit Sengupta, Mohammed Nabeel, Muhammad Yasin, and Ozgur Sinanoglu. Atpg-based cost-effective, secure logic locking. Proceedings of the IEEE VLSI Test Symposium, 2018-April:1–6, 2018.

[48] 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, 39(12):4439–4452, 2020.

[49] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pages 167–178, 2017.

[50] James C. Candy and Gabor C. Temes. Multirate Filter Designs Using Comb Filters. IEEE, 2009.

[51] Jean-Philippe Diguet, Samuel Evain, Romain Vaslin, Guy Gogniat, and Emmanuel Juin. Noc-centric security of reconfigurable soc. In First International Symposium on Networks-on-Chip (NOCS'07), pages 223–232. IEEE, 2007.

[52] Nadia Creignou and Hervé Daudé. Sensitivity of boolean formulas. European Journal of Combinatorics, 34:793–805, 7 2013.

[53] 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, 28:1493–1502, 10 2009.

[54] 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.

[55] Travis Meade, Zheng Zhao, Shaojie Zhang, David Pan, and Yier Jin. Revisit sequential logic obfuscation: Attacks and defenses. In 2017 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–4. IEEE, 2017.

[56] 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.

[57] Shervin Roshanisefat, Hadi Mardani Kamali, Kimia Zamiri Azar, Sai Manoj Pudukotai Dinakarrao, Naghmeh Karimi, Houman Homayoun, and Avesta Sasan. Dfssd: Deep faults and shallow state duality, a provably strong obfuscation solution for circuits with restricted access to scan chain. In 2020 IEEE 38th VLSI Test Symposium (VTS), pages 1–6. IEEE, 2020.

[58] Leon Li, Shuyi Ni, and Alex Orailoglu. Janus: Boosting logic obfuscation scope through reconfigurable fsm synthesis. In 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 292–303. IEEE, 2021.

[59] Leon Li and Alex Orailoglu. Janus-hd: Exploiting fsm sequentiality and synthesis flexibility in logic obfuscation to thwart sat attack while offering strong corruption. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1323–1328. IEEE, 2022.

[60] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting fsms from flattened gate-level netlist. In Proceedings of 2010 IEEE international symposium on circuits and systems, pages 2610–2613. IEEE, 2010.

[61] Shervin Roshanisefat, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Rane: An open-source formal de-obfuscation attack for reverse engineering of logic encrypted circuits. In Proceedings of the 2021 on Great Lakes Symposium on VLSI, pages 221–228, 2021.

[62] Yinghua Hu, Yuke Zhang, Kaixin Yang, Dake Chen, Peter A Beerel, and Pierluigi Nuzzo. Fun-sat: Functional corruptibility-guided sat-based attack on sequential logic encryption. In 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 281–291. IEEE, 2021.

[63] Akashdeep Saha, Hrivu Banerjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. Oracall: An oracle-based attack on cellular automata guided logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 40(12):2445–2454, 2021.

[64] Rajit Karmakar, Suman Sekhar Jana, and Santanu Chattopadhyay. A cellular automata guided obfuscation strategy for finite-state-machine synthesis. In Proceedings of the 56th Annual Design Automation Conference 2019, pages 1–6, 2019.

[65] Robert Tarjan. Depth-first search and linear graph algorithms. SIAM journal on computing, 1(2):146–160, 1972.

[66] Travis Meade, Shaojie Zhang, and Yier Jin. Netlist reverse engineering for high-level functionality reconstruction. In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pages 655–660. IEEE, 2016.

[67] Niladri Bhattacharjee, Viktor Havel, Suruchi Kumari, Nima Kavand, Jorge Navarro Quijada, Akash Kumar, Thomas Mikolajick, and Jens Trommer. Dynamic reconfigurable security cells based on emerging devices integrable in fdsoi technology. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2024.

[68] Armin Darjani, Nima Kavand, and Akash Kumar. Flip-lock: A flip-flop-based logic locking technique for thwarting ml-based and algorithmic structural attacks. In Proceedings of the Great Lakes Symposium on VLSI 2024, pages 185–191, 2024.

[69] Marc D Riedel and Jehoshua Bruck. The synthesis of cyclic combinational circuits. In Proceedings of the 40th annual Design Automation Conference, pages 163–168, 2003.

[70] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Cyclic obfuscation for creating sat-unresolvable circuits. In Proceedings of the Great Lakes Symposium on VLSI 2017, pages 173–178, 2017.

[71] 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, 2019.

[72] Shervin Roshanisefat, Hadi Mardani Kamali, and Avesta Sasan. Srclock: Sat-resistant cyclic logic locking for protecting the hardware. In Proceedings of the 2018 on Great Lakes Symposium on VLSI, pages 153–158, 2018.

[73] Shervin Roshanisefat, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Sat-hard cyclic logic obfuscation for protecting the ip in the manufacturing supply chain. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 28(4):954–967, 2020.

[74] Amin Rezaei, Yuanqi Shen, Shuyu Kong, Jie Gu, and Hai Zhou. Cyclic locking and memristor-based obfuscation against cycsat and inside foundry attacks. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 85–90. IEEE, 2018.

[75] Pei-Pei Chen, Xiang-Min Yang, Yu-Cheng He, Yung-Chih Chen, Yi-Ting Li, and Chun-Yao Wang. Looplock 3.0: A robust cyclic logic locking approach. In 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 594–599. IEEE, 2024.

[76] Yeganeh Aghamohammadi and Amin Rezaei. Lipstick: Corruptibility-aware and explainable graph neural network-based oracle-less attack on logic locking. In 2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 606–611. IEEE, 2024.

[77] Xiang-Min Yang, Pei-Pei Chen, Hsiao-Yu Chiang, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang. Looplock 2.0: An enhanced cyclic logic locking approach. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(1):29–34, 2021.

[78] Hsiao-Yu Chiang, Yung-Chih Chen, De-Xuan Ji, Xiang-Min Yang, Chia-Chun Lin, and Chun-Yao Wang. Looplock: Logic optimization-based cyclic logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10):2178–2191, 2019.

[79] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. Cycsat: Sat-based attack on cyclic logic encryptions. In 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 49–56. IEEE, 2017.

[80] Yuanqi Shen, You Li, Amin Rezaei, Shuyu Kong, David Dlott, and Hai Zhou. Besat: Behavioral sat-based attack on cyclic logic encryption. In Proceedings of the 24th Asia and South Pacific Design Automation Conference, pages 657–662, 2019.

[81] Kaveh Shamsi, David Z Pan, and Yier Jin. Icysat: Improved sat-based attacks on cyclic locked circuits. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–7. IEEE, 2019.

[82] Kevin Lopez and Amin Rezaei. K-gate lock: Multi-key logic locking using input encoding against oracle-guided attacks. In Proceedings of the 30th Asia and South Pacific Design Automation Conference, pages 794–800, 2025.

[83] Wei Zeng, Azadeh Davoodi, and Rasit Onur Topaloglu. Obfusx: Routing obfuscation with explanatory analysis of a machine learning attack. In Proceedings of the 26th Asia and South Pacific Design Automation Conference, pages 548–554, 2021.

[84] 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.

[85] Rajit Karmakar, Santanu Chatopadhyay, and Rohit Kapur. Encrypt flip-flop: A novel logic encryption technique for sequential circuits. arXiv preprint, 2018.

[86] Rajit Karmakar, Harshit Kumar, and Santanu Chattopadhyay. Efficient key-gate placement and dynamic scan obfuscation towards robust logic encryption. IEEE Transactions on Emerging Topics in Computing, 9(4):2109–2124, 2019.

[87] Seetal Potluri, Aydin Aysu, and Akash Kumar. Seql: Secure scan-locking for ip protection. In 2020 21st International Symposium on Quality Electronic Design (ISQED), pages 7–13. IEEE, 2020.

[88] Levent Aksoy, Muhammad Yasin, and Samuel Pagliarini. Kratt: Qbf-assisted removal and structural analysis attack against logic locking. In 2024 Design,

Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2024.

[89] Nimisha Limaye, Abhrajit Sengupta, Mohammed Nabeel, and Ozgur Sinanoglu. Is robust design-for-security robust enough? attack on locked circuits with restricted scan chain access. In 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pages 1–8. IEEE, 2019.

[90] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. On designing secure and robust scan chain for protecting obfuscated logic. arXiv preprint, 2020.

[91] Ujjwal Guin, Ziqi Zhou, and Adit Singh. Robust design-for-security architecture for enabling trust in ic manufacturing and test. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 26(5):818–830, 2018.

[92] Nimisha Limaye, Emmanouil Kalligeros, Nikolaos Karousos, Irene G Karybali, and Ozgur Sinanoglu. Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 40(9):1740–1753, 2020.

[93] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Scramble: The state, connectivity and routing augmentation model for building logic encryption. In 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 153–159. IEEE, 2020.

[94] M Sazadur Rahman, Adib Nahiyan, Fahim Rahman, Saverio Fazzari, Kenneth Plaks, Farimah Farahmandi, Domenic Forte, and Mark Tehranipoor. Security assessment of dynamically obfuscated scan chain against oracle-guided attacks. ACM Transactions on Design Automation of Electronic Systems (TODAES), 26(4):1–27, 2021.

[95] Lilas Alrahis, Muhammad Yasin, Nimisha Limaye, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. Scansat: Unlocking static and dynamic scan obfuscation. IEEE Transactions on Emerging Topics in Computing, 9(4):1867–1882, 2019.

[96] Nimisha Limaye and Ozgur Sinanoglu. Dynunlock: Unlocking scan chains obfuscated using dynamic keys. In 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 270–273. IEEE, 2020.

[97] 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, pages 1–6, 2017.

[98] Grace Li Zhang, Bing Li, Bei Yu, David Z Pan, and Ulf Schlichtmann. Timing-camouflage: Improving circuit security against counterfeiting by unconventional timing. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 91–96. IEEE, 2018.

[99] Joseph Sweeney, V Mohammed Zackriya, Samuel Pagliarini, and Lawrence Pileggi. Latch-based logic locking. In 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pages 132–141. IEEE, 2020.

[100] Kimia Zamiri Azar, Hadi Mardani Kamali, Shervin Roshanisefat, Houman Homayoun, Christos P Sotiriou, and Avesta Sasan. Data flow obfuscation: A new paradigm for obfuscating circuits. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 29(4):643–656, 2021.

[101] Mark Sazadur Rahman, Rui Guo, Hadi M Kamali, Fahim Rahman, Farimah Farahmandi, Mohamed Abdel-Moneum, and Mark Tehranipoor. O'clock: lock the clock via clock-gating for soc ip protection. In Proceedings of the 59th ACM/IEEE Design Automation Conference, pages 775–780, 2022.

[102] Zhao Qi Jiang, Wen Jia Li, Zhi Xia Xu, Hua Wei Tang, Meng Wang, Jie Chang, Hui Feng Ma, Yu Xiang Li, Zheng Zhu, Chun Ying Guan, et al. Photonic spin-hall logic devices based on programmable spoof plasmonic metamaterial. Laser & Photonics Reviews, 18(8):2301371, 2024.

[103] Paul R Genssler, Lilas Alrahis, Ozgur Sinanoglu, and Hussam Amrouch. Hdcircuit: Brain-inspired hyperdimensional computing for circuit recognition. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–2. IEEE, 2024.

[104] Muhammad Yasin, Chongzhi Zhao, and Jeyavijayan JV Rajendran. Sfll-hls: Stripped-functionality logic locking meets high-level synthesis. In 2019 IEEE/ACM

International Conference on Computer-Aided Design (ICCAD), pages 1–4. IEEE, 2019.

[105] 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, pages 1–6, 2018.

[106] Christian Pilato, Animesh Basak Chowdhury, Donatella Sciuto, Siddharth Garg, and Ramesh Karri. Assure: Rtl locking against an untrusted foundry. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 29(7):1306–1318, 2021.

[107] 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.

[108] 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.

[109] Nimisha Limaye, Animesh B Chowdhury, Christian Pilato, Mohammed TM Nabeel, Ozgur Sinanoglu, Siddharth Garg, and Ramesh Karri. Fortifying rtl locking against oracle-less (untrusted foundry) and oracle-guided attacks. In 2021 58th ACM/IEEE Design Automation Conference (DAC), pages 91–96. IEEE, 2021.

[110] Chandan Karfa, TM Abdul Khader, Yom Nigam, Ramanuj Chouksey, and Ramesh Karri. Host: Hls obfuscations against smt attack. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 32–37. IEEE, 2021.

[111] Luca Collini, Ramesh Karri, and Christian Pilato. A composable design space exploration framework to optimize behavioral locking. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1359–1364. IEEE, 2022.

[112] Gourav Takhar, Ramesh Karri, Christian Pilato, and Subhajit Roy. Holl: Program synthesis for higher order logic locking. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 3–24. Springer, 2022.

[113] Bo Hu, Jingxiang Tian, Mustafa Shihab, Gaurav Rajavendra Reddy, William Swartz, Yiorgos Makris, Benjamin Carrion Schaefer, and Carl Sechen. Functional obfuscation of hardware accelerators through selective partial design extraction onto an embedded fpga. In Proceedings of the 2019 on Great Lakes Symposium on VLSI, pages 171–176, 2019.

[114] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Larry Pileggi, and Ken Mai. Hardware redaction via designer-directed fine-grained efpga insertion. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1186–1191. IEEE, 2021.

[115] Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, Benjamin Tan, Christian Pilato, Ganesh Gore, Xifan Tang, Scott Temple, Pierre-Emmanuel Gaillardon, and Ramesh Karri. Exploring efpga-based redaction for ip protection. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9. IEEE, 2021.

[116] Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, Benjamin Tan, Christian Pilato, Ganesh Gore, Xifan Tang, Scott Temple, Pierre-Emmanuel Gaillardon, and Ramesh Karri. Not all fabrics are created equal: Exploring efpga parameters for ip redaction. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2023.

[117] 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, 2018.

[118] Prabuddha Chakraborty, Jonathan Cruz, Abdulrahman Alaql, and Swarup Bhunia. Sail: Analyzing structural artifacts of logic locking using machine learning. IEEE Transactions on Information Forensics and Security, 16:3828–3842, 2021.

[119] Dominik Sisejkovic, Farhad Merchant, Lennart M Reimann, Harshit Srivastava, Ahmed Hallawa, and Rainer Leupers. Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach. ACM Journal on Emerging Technologies in Computing Systems (JETC), 17(3):1–26, 2021.

[120] Lilas Alrahis, Satwik Patnaik, Faiq Khalid, Muhammad Abdullah Hanif, Hani Saleh, Muhammad Shafique, and Ozgur Sinanoglu. Gnnunlock: Graph neural networks-based oracle-less unlocking scheme for provably secure logic locking. In 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 780–785. IEEE, 2021.

[121] Lilas Alrahis, Satwik Patnaik, Muhammad Abdullah Hanif, Muhammad Shafique, and Ozgur Sinanoglu. Untangle: Unlocking routing and logic obfuscation using graph neural networks-based link prediction. In 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pages 1–9. IEEE, 2021.

[122] Lilas Alrahis, Satwik Patnaik, Muhammad Shafique, and Ozgur Sinanoglu. Muxlink: Circumventing learning-resilient mux-locking using graph neural network-based link prediction. In 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 694–699. IEEE, 2022.

[123] Armin Darjani, Nima Kavand, Shubham Rai, and Akash Kumar. Thwarting gnn-based attacks against logic locking. IEEE Transactions on Information Forensics and Security, 2024.

[124] Dominik Sisejkovic, Farhad Merchant, Lennart M Reimann, and Rainer Leupers. Deceptive logic locking for hardware integrity protection against machine learning attacks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(6):1716–1729, 2021.

[125] Kimia Zamiri Azar, Hadi Mardani Kamali, Farimah Farahmandi, and Mark Tehranipoor. Understanding Logic Locking. Springer, 2024.

[126] Yue Zhang, Basel Halak, and Haoyu Wang. Zeki: A zero-knowledge dynamic logic locking implementation with resilience to multiple attacks. In 2024 IEEE 37th International System-on-Chip Conference (SOCC), pages 1–6, 2024.

[127] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. Hypotenuse, 256(128):214335, 2015.

[128] Computer Engineering Research Center (CERC), University of Texas at Austin. Itc'99 benchmarks documentation – introduction. https://www.cerc.utexas.edu/itc99-benchmarks/bendoc1.html, 1999.

[129] F. Brglez et al. Combinational profiles of sequential benchmark circuits. In IEEE ISCAS, page 1929–1934. IEEE, 1989.

[130] Niklas Eén and Niklas Sörensson. Minisat sat solver – introduction. http://minisat.se/, 2003.

[131] Kaveh Shamsi, Meng Li, David Z. Pan, and Yier Jin. Cross-lock dense layout-level interconnect locking using cross-bar architectures. In Proceedings of the 2018 Great Lakes Symposium on VLSI (GLSVLSI), pages 147–152. ACM, May 2018.

[132] Satwik Patnaik, Mohammed Ashraf, Ozgur Sinanoglu, and Johann Knechtel. Obfuscating the interconnects: Low-cost and resilient full-chip layout camouflaging. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(12):4466–4481, 2020.

[133] Md Rafid Muttaki, Shuvagata Saha, Hadi M Kamali, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. Rtlock: Ip protection using scan-aware logic locking at rtl. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2023.

[134] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In Holger H. Hoos and David G. Mitchell, editors, Theory and Applications of Satisfiability Testing, pages 183–198, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[135] Armin Biere. Lingeling, Plingeling, and Treengeling Entering the SAT Competition 2013. In Proceedings of SAT Competition 2013, pages 51–52, 2013.

[136] Niklas Sorensson and Niklas Een. MiniSat v1.13: A SAT solver with conflict-clause minimization. In Theory and Applications of Satisfiability Testing – SAT 2005, volume 3569 of Lecture Notes in Computer Science, pages 502–518. Springer, 2005.

[137] G S Tseitin. On the Complexity of Derivation in Propositional Calculus. Springer Berlin Heidelberg, 1983.

[138] Jingbo Zhou and Xinmiao Zhang. Generalized sat-attack-resistant logic locking. IEEE Transactions on Information Forensics and Security, 16:2581–2592, 2021.

[139] Yuntao Liu, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava. Strong anti-sat: Secure and effective logic locking. In 2020 21st International Symposium on Quality Electronic Design (ISQED), pages 199–205. IEEE, 2020.