ABSTRACT

Title of dissertation: DESIGN TECHNIQUES FOR ENHANCING HARDWARE-ORIENTED SECURITY USING OBFUSCATION

Abhishek Chakraborty,
Doctor of Philosophy, 2021

Dissertation directed by: Professor Ankur Srivastava
Department of Electrical and
Computer Engineering

The increasing trend of outsourcing hardware designs to offshore foundries for fabrication cost reduction has raised several security concerns related to intellectual property (IP) piracy, reverse engineering, counterfeiting, etc. The exposure of chip designs to a potentially malicious offshore foundry is of major concern for both government and private organizations and hence, there has been extensive research on security and privacy issues of integrated circuit (IC) supply chain. In this dissertation, we study the effectiveness of hardware-oriented obfuscation approaches for enhancing security and trust at different levels of design abstractions.

At the circuit-level of design abstraction, we analyze the security offered by state-of-the-art technique called delay locking which uses a secret key for obfuscating the functionality as well as the timing profile of a circuit such that the critical design details are not exposed to an untrusted foundry. We propose a novel Boolean satisfiability (SAT) formulation based attack to defeat the delay locking

countermeasure by utilizing the detailed timing characterization of gates present in a circuit. Subsequently, we develop a new circuit-level obfuscation technique called stripped-functionality delay locking which is provably secure against all known attacks on logic locking. In addition, we also analyze the vulnerability of circuit-level obfuscation schemes to power side-channel analysis attacks.

Next, we study the limitations of circuit-level obfuscation approaches to provide reasonable security guarantees at the architecture-level of design abstraction. We demonstrate the applicability of an iterative SAT formulation based attack strategy against a many-core processor design (obfuscated using circuit-level techniques) to find an approximate key for running applications with almost no errors. Such an attack poses a major threat in the supply chain of processor designs as unlike earlier attack strategies, our proposed attack does not require any activated hardware for SAT formulation based analysis. Subsequently, we develop a couple of efficient architecture-level locking techniques which are highly resilient to SAT based attacks.

Finally, we develop a hardware-assisted obfuscation framework for protecting the IPs of neural network (NN) models, thus enhancing application-level security. The generation of production-level NN models is not a trivial task as it requires a long training time using high power computing resources along with the availability of massive amounts of labeled training data. Hence, the protection of IP rights of well-trained NN models has become a matter of major concern for the model owners. In this research direction, we demonstrate the utilization of a hardware root of trust

based obfuscation approach to safeguard the IPs of such NN models. Our proposed framework ensures that only authorized end-users who possess trusted edge devices will be able to run the intended applications with high accuracy.

DESIGN TECHNIQUES FOR ENHANCING HARDWARE-ORIENTED SECURITY USING OBFUSCATION

by

Abhishek Chakraborty

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Advisory Committee:
Professor Ankur Srivastava, Chair/Advisor
Professor Dana Dachman-Soled
Professor Manoj Franklin
Professor Gang Qu
Professor John Aloimonos

Dedication

To my parents, for their love and support.

Acknowledgments

I would like to express my sincere gratitude to all the people who made my Ph.D. journey at the University of Maryland, College Park a memorable experience. First and foremost, I would like to thank my advisor, Professor Ankur Srivastava, for his continued guidance, unswerving support, and patience throughout my Ph.D. studies. I am extremely grateful to him for providing me with the opportunity to work on several challenging problems over the past five years. His enthusiasm for research has been a constant source of inspiration that kept me motivated even during the difficult times. It is truly a pleasure to work with and learn from him.

I am grateful to Professor Dana Dachman-Soled, Professor Manoj Franklin, Professor Gang Qu, and Professor John Aloimonos for their time to serve on my Ph.D. dissertation committee and for their valuable feedback on this thesis.

I would like to extend my gratitude to all my colleagues in Professor Srivastava's research group, Chongxi Bao, Yang Xie, Zhiyuan Yang, Yuntao Liu, Ankit Mondal, Michael Zuzak, Daniel Xing, Nina Jacobsen, Priyadevi Mittu, and Isaac McDaniel, for the fruitful discussions on research topics, for the late nights we were working together before deadlines, and for all the good times we shared in our lab. I would also like to thank all my close friends for supporting me emotionally during the stressful times of my Ph.D. life.

I would like to acknowledge the financial support provided by the Graduate school (Ann G. Wylie Dissertation Fellowship), the Department of Electrical and Computer Engineering, and the Institute for Systems Research at the University of Maryland as well as the research funding provided by the Air Force Office of Scientific Research and Northrop Grumman Corporation.

Finally, I owe my deepest gratitude to my parents for the unconditional love that they have bestowed upon me for years. Their support, sacrifices, prayers, and encouragement really helped me to continue my research work even during the most challenging years of my life.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
CL	Cache Locking
CNN	Convolution Neural Network
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DIO	Distinguishing Input/Output
DL	Deep Learning
DNN	Deep Neural Network
DPA	Differential Power Analysis
DRM	Digital Rights Management
GPGPU	General Purpose GPU
GPR	General Purpose Register
GPU	Graphics Processing Unit
HD	Hamming Distance
HM	Host Memory
HPNN	Hardware Protected Neural Network
HSCOA	Hardware/Software Co-design based Accelerator Obfuscation
IC	Integrated Circuit
IOV	Instruction Output Value
IP	Intellectual Property
ISA	Instruction Set Architecture
LRU	Least Recently Used
LUT	Lookup Table
MAC	Multiply Accumulate
MLaaS	Machine Learning as a Service
MMU	Matrix Multiply Unit
NN	Neural Network
PI	Primary Input
PO	Primary Output
PTX	Parallel Thread Execution

RLL	Random Logic Locking
SASSIFI	SASSI-based Fault Injector
SAT	Boolean Satisfiability
SDK	Software Development Kit
SFLL	Stripped-functionality Logic Locking
SFU	Special Functional Unit
SIMT	Single Instruction Multiple Thread
SLL	Strong Logic Locking
SM	Streaming Multiprocessors
SP	Streaming Processors
TA	Template Analysis
TDB	Tunable Delay Buffer
TDK	Tunable Delay Key-gate
TPU	Tensor Processing Unit
XNOR	Exclusive NOR
XOR	Exclusive OR

Chapter 1: Introduction

The rise of ubiquitous computing introduces a new set of challenges in designing secure systems. Modern computing platforms are typically connected with several other systems (via the Internet) and are constantly interacting with their users (via user interfaces) as well as the environment (via sensors and actuators). These systems may contain a variety of sensitive information, e.g., medical records, financial data, classified documents, etc. which are targets of cyber attacks. Therefore, the security of such computing systems and the protection of information stored/processed by them is a matter of major concern for their users.

Traditional software-oriented security schemes assume that the underlying hardware systems are perfectly reliable and trustworthy. However, the emergence of various hardware-oriented attacks, including the recent Spectre [56] and Meltdown [63] attacks on Intel processors, have exposed severe security vulnerabilities in hardware designs. Hardware security threats may arise either due to malicious tampering performed during the chip design and fabrication phases (IC supply chain attacks) or due to unintended design flaws which could be exploited by attackers in the post-deployment phase. The realization of trustworthy hardware is of paramount importance as it forms the *root of trust* on which all security-critical operations of an electronic system depend [15].

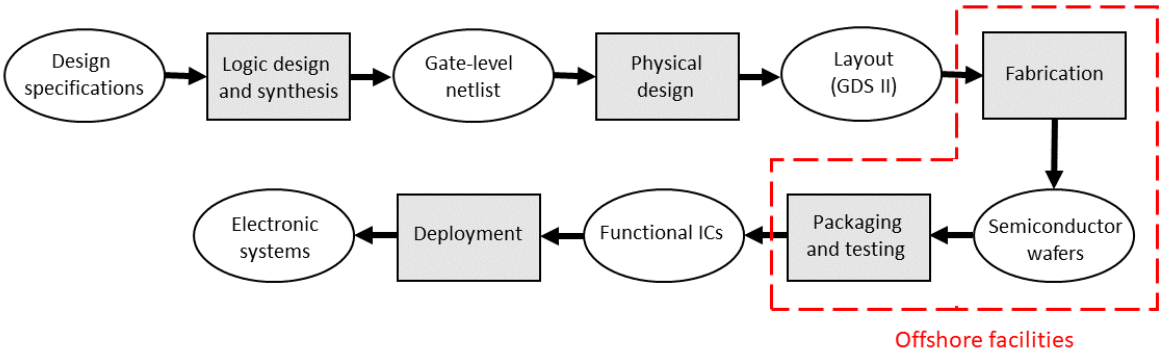


Figure 1.1: An overview of globalized IC design flow

1.1 Security Issues in IC Supply Chain

The life cycle of a modern integrated circuit (IC) is composed of several stages such as design, fabrication, testing, packaging, assembly, and finally deployment in electronic systems as shown in Fig. 1.1. Design companies are increasingly outsourcing their chip designs to offshore facilities due to the high cost of maintaining a semiconductor foundry as well as the evolving complexity of system designs, thus resulting in a globalized IC design flow. However, this trend has led to severe security vulnerability issues associated with intellectual property (IP) piracy, reverse engineering, counterfeiting, overproduction, etc. of hardware designs by potentially untrusted parties in the distributed IC supply chain [54]. It has been reported that such IC supply chain attacks lead to losses in magnitudes of billions of dollars per year [84]. Such attacks not only pose a threat to the business of private companies but also raise major security concerns for government agencies as

proprietary, mission-critical chip designs may be exposed to a malicious third-party foundry. Hence, research related to security and trust in the IC supply chain has gained a lot of focus over the past decade [107, 54, 79].

1.1.1 Design-for-trust Techniques

Several design-for-trust techniques have been proposed in literature including watermarking [51, 55], fingerprinting [24], camouflaging [73], split-manufacturing [48, 47], metering [18, 57] and logic locking [81, 72, 74, 114, 108] to enforce security and trust in IC supply chain. A brief overview of such design-for-trust schemes is presented as follows.

- **Watermarking and Fingerprinting:** These are *passive* techniques which help in IP piracy detection but cannot prevent the design from being stolen. In watermarking, a *secret* design constraint (designer's signature) is embedded into the hardware whereas in fingerprinting both the designer's as well as an end-user's signatures are embedded in hardware to track the source of piracy.

- **Camouflaging:** In this approach the designer substitutes selected gates in a design with their camouflaged counterparts (typically using dummy contacts or filler cells) to prevent reverse engineering attacks by end-users. Camouflaging schemes rely on a *trusted* foundry to fabricate such camouflaged gates.

- **Split-manufacturing:** In this technique the layout layers of a circuit design are split into two parts which are fabricated in separate foundries and finally stacked together. Split-manufacturing can prevent IP piracy of hardware designs by an untrusted foundry but not by an end-user.

- **Metering:** IC metering consists of a set of techniques and protocols by which a designer keeps track of individual chips post-fabrication using uniquely assigned IDs. While using *passive* metering techniques the designer can detect IC piracy, the use of *active* metering techniques allow her to control the chip behavior post-deployment.

- **Logic Locking:** Logic locking is a gate-level obfuscation approach to protect design IPs from both untrusted foundry and end-user. In this technique the designer inserts additional gates into the design and locks its functionality using a *secret* key.

From the above discussion, we can see that unlike techniques such as camouflaging or split-manufacturing which prevents design piracy by a single-entity (either untrusted end-user or malicious foundry), logic locking can ensure IP protection from *any* rogue entity in the entire IC supply chain outside the trusted design house. As this thesis focuses on developing obfuscation based security solutions, we discuss the concept of logic locking (or logic obfuscation) in more details in the next section.

1.1.2 IP Security through Logic Obfuscation

Logic locking is one of the most popular techniques adopted to defend against IP theft, overbuilding, counterfeiting and reverse engineering of chips at any outside design house stage in the IC supply chain [82, 25]. The primary idea of any standard combinational logic locking algorithm is to insert additional key-controlled logic gates called *key-gates* in a netlist for obfuscating its functionality [82, 23, 75, 76]. These key-gates are driven by key-inputs which are supplied from an on-chip *tamper-proof* memory. An IC locked in such a manner will exhibit correct functionality only when the *secret* key inputs are provided to *activate* the chip after fabrication.

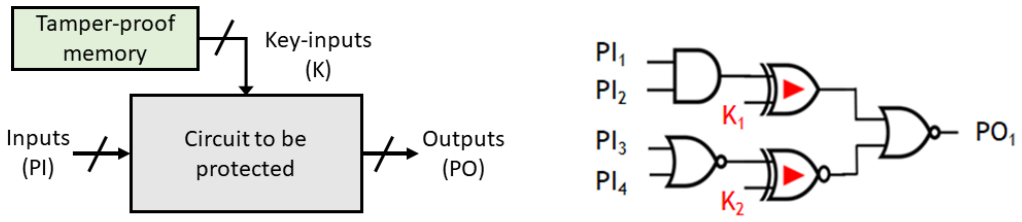


Figure 1.2: XOR/XNOR based logic locking

In Fig. 1.2, we illustrate the popular XOR/XNOR based logic locking scheme which aims to obfuscate a circuit using low area overhead. Note that based on the logic values of key-inputs K_1 and K_2 the XOR/XNOR gates (highlighted in red) behave either as a buffer or an inverter. Therefore, the functionality of such an obfuscated netlist is retrieved *only* when the correct key-inputs K_1 and K_2 are applied. For any incorrect key-inputs, however, errors will be introduced in the netlist output. In the next section we provide an overview of existing logic locking techniques.

1.1.3 Evolution of Logic Obfuscation Techniques

Several combinational logic locking techniques have been developed over the years to thwart design piracy by untrusted entities in the IC supply chain. The earliest approach called random locking locking (RLL) was proposed in [82] which locks a design by introducing additional XOR key-gates at randomly selected locations in the netlist as illustrated using Fig. 1.2. The weakness of RLL technique to an automatic test pattern generation based attack led to the development of a strong locking locking (SLL) technique which inserts key-gates in a netlist such that sensitization of individual key bits to output is computationally intensive [75]. The security of logic locking schemes such as RLL and SLL have been threatened by the introduction of a Boolean satisfiability based attack (SAT attack) which iteratively solves SAT formulas to eliminate subsets of wrong keys and finally converges to find a *correct* key within few hours [97]. The efficiency of SAT attack relies on the small number of iterations required to decipher the *correct* key even for a reasonably large circuit.

To counter SAT attack, *point-function* schemes like Anti-SAT [108] and SARLock [115] have been proposed. The number of SAT attack iterations required to retrieve the correct key of a *point-function* obfuscated netlist is an exponential function in terms of key size. However, new attacks have been proposed which try to circumvent the aforementioned countermeasures: (i) the signal probability skew attack [40] identifies the location of Anti-SAT block in the netlist, thus making it prone to be removed by the attacker (ii) the Double DIP attack [91] utilizes an

extended SAT formulation to successfully retrieve a netlist encrypted using SARLock technique (iii) the AppSAT attack [89] exploits the fact that output corruptibility of above *point-function* schemes is very low for a wrong key. AppSAT attack also iteratively solves SAT formulations (similar to SAT attack) to find an *approximate key* which reconstructs the functionality (almost correct) of a locked netlist.

In [118], a technique called stripped-functionality logic locking (SFLL) has been proposed which provides a quantifiable trade-off among resiliency to different types of attacks, including SAT formulation based attacks, removal attack [119] and bypass attack [112]. There exists two versions of SFLL scheme called SFLL-HD and SFLL-$flex$, each of which *strips* some portion of design functionality and hides it in the form of protected patterns (*secret* key). In recent literature some security vulnerabilities have been identified in SFLL technique: (i) an attack approach detailed in [113] exploits structural traces left behind in a locked netlist due to functionality stripping and subsequently identifies some of the protected patterns in SFLL-HD (ii) the FALL attack [93] uses structural/functional analyses combined with SAT-based analyses to defeat SFLL-HD countermeasure (iii) the SURF attack [32] utilizes machine learning based approach to guess a likely key. However, even though SURF attack is highly scalable compared to SAT attack, it doesn't provide any formal guarantees of correctness of the key found.

In [52], another technique called Full-lock has been outlined which inserts routing blocks to increase per iteration time of SAT solving but incurs significantly higher implementation cost compared to SFLL. Recently, a lightweight locking technique called CAS-Lock [88] has been proposed which aims to simultaneously

resist SAT and bypass attacks while maintaining non-trivial output corruptibility. However, a recent attack [85] has identified security vulnerability in CAS-Lock scheme which neither requires a reverse-engineered netlist nor it requires access to an activated chip. A newly proposed countermeasure called *delay locking* [110] has been claimed to successfully thwart all existing attacks on logic locking schemes and hence, offers state-of-the-art IP protection of hardware designs in an untrusted IC supply chain.

1.2 Security Issues in Post-deployment Phase

Hardware security threats may also arise after ICs are deployed in parts of electronic systems. For example, end-users can use sophisticated *reverse engineering* techniques [101] to gain valuable insights on the design details of a proprietary IC and then, leverage such knowledge to perform malicious activities such as production of counterfeited chips. However, it is to be noted that the task of reverse engineering is much more challenging for an end-user compared to an attacker in an untrusted foundry setting. This is because an end-user needs to perform additional error-prone steps such as decapsulation, delayering, and imaging to obtain a complete view of the layout-level information of a chip which is readily available to an untrusted foundry in the form of a GDSII file.

Attackers may also analyze the *side-channel* information of a chip to decipher the secret data being processed by a system [96]. In a typical side-channel analysis attack, the unintended leakage of information from the physical characteristics of

a hardware system (e.g., power consumption profiles, timing variation patterns, electromagnetic emanation, etc.) is exploited to learn about the targeted secrets. For example, in the recent Spectre [56] and Meltdown [63] attacks on Intel processors, a malicious program utilizes timing side-channel information to gain access to secrets stored in the memory of other running programs.

1.3 Contributions and Thesis Organization

In this dissertation, we investigate the security offered by various hardware-oriented obfuscation approaches against IP theft attempts by an untrusted foundry as well as end-users. We formulate new attack strategies and also develop effective countermeasures for enhancing the strength of hardware-oriented obfuscation techniques at different levels of design abstractions. The major contributions of the thesis are as follows.

1.3.1 Circuit-level Obfuscation

In this research direction, we first propose a novel SAT formulation based attack strategy called TimingSAT to retrieve the functionality of design netlists obfuscated using state-of-the-art delay locking countermeasure [110]. A designer applies delay locking technique on a circuit in order to ensure that a *secret* key not only controls the functionality of a circuit but also its timing profile. A delay-locked circuit exhibits its original functionality only when both correct *functional* and *delay* keys are provided. Our proposed TimingSAT attack approach against delay locking

scheme applies a pre-processing step to model the timing characteristics of various types of gates present in an obfuscated design as Boolean functions. Such a circuit pre-processing helps us to build a timing profile embedded SAT formulation in terms of targeted key-inputs. Then, we apply a two step attack strategy: in the first stage a correct *functional* key is found using conventional SAT attack approach and in the second stage a correct *delay* key is determined using the aforementioned timing profile embedded SAT formulation of the circuit. In both stages of the TimingSAT attack, wrong keys are iteratively eliminated till a key belonging to the correct equivalence class is obtained.

Next, we study the vulnerability of logic locking schemes to side-channel analysis attacks. In particular, we propose a template analysis based profiling side-channel attack which utilizes the power consumption traces from a chip to decipher the key-inputs of a locked netlist. The proposed attack strategy is based on the observation that various key-gates in a netlist (locked using standard logic obfuscation algorithms) are located at different *logic depths*, which in turn enables a side-channel adversary to unlock the circuit functionality in a level-by-level manner.

1.3.2 Architecture-level Obfuscation

Till date, the security analyses of logic locking schemes have mostly been confined to only circuit-level of design abstraction. To the best of our knowledge, there has been no study which analyzes the security guarantees provided by such approaches to ensure IP protection of hardware implementations at the architecture-

level of design abstraction. In this research direction, we first outline a SAT formulation based attack strategy against an obfuscated Graphics Processing Unit (GPU) architecture to highlight the limitations of circuit-level locking approaches for ensuring architecture-level IP protection. Our proposed attack first translates the multi-cycle GPU core netlist (locked using Anti-SAT block [109]) to a functionally-equivalent single-cycle netlist and subsequently, retrieves an *approximate key* to unlock the overall GPU hardware design. Unlike earlier attack strategies, our proposed attack does not require any activated hardware for SAT formulation, rather it utilizes information from the publicly available GPU instruction set architecture (ISA) for necessary analysis. Therefore, such an attack poses a major threat in the supply chain of processor designs.

Next, we propose a couple of SAT attack resistant architecture-level countermeasures for protecting the IPs of hardware accelerator designs. The first technique called cache locking aims to significantly degrade the performance of high-level application programs running on an approximately unlocked GPU for a wrong *cache key*, thus making such a hardware ineffective for high-performance computing purposes. The second technique is based on a novel hardware-software co-design based obfuscation approach to provably secure the IPs of hardware accelerator architectures from untrusted parties. The resiliency of this scheme against SAT attack is manifested by using a sequence of keys to obfuscate the instruction encoding for an application program.

1.3.3 Hardware-assisted Obfuscation of Neural Networks

The protection of IPs of well-trained deep learning (DL) models has become a matter of major concern, especially with the growing trend of deployment of Machine Learning as a Service (MLaaS). In this research direction, we demonstrate the utilization of a hardware root of trust to safeguard the IPs of such DL models which potential attackers have access to. We propose an obfuscation framework called Hardware Protected Neural Network (HPNN) in which a neural network is first trained as a function of a secret key and then, the obfuscated DL model is hosted on a public model sharing platform. Such an obfuscation framework ensures that only authorized end-users who possess trustworthy hardware devices (with the secret key embedded on-chip) will be able to run intended DL applications using the published model.

In addition, we also develop a novel watermarking technique called DynaMarks to protect the IPs of DL models against *model extraction* attacks performed by authorized end-users. Unlike existing approaches, DynaMarks does not alter the training process of an original model but rather embeds watermark into a *surrogate model* by dynamically changing the output responses of the original model's prediction API based on certain secret parameters at inference runtime. The integration of DynaMarks scheme with the HPNN framework allows a DL model owner to reliably prove model ownership even under a strong attack scenario.

Organization of the thesis: The rest of the dissertation is organized as follows. In Chapter 2, we study the weaknesses of existing circuit-level obfuscation approaches and develop a new technique called stripped-functionality delay locking which offers state-of-the-art security against all known attacks on logic locking. These results have been published in [26, 27, 25]. In Chapter 3, we investigate the vulnerability of circuit-level obfuscation schemes to side-channel analysis attacks and the results from this work has been published in [30]. In Chapter 4, we demonstrate the limitations of circuit-level obfuscation techniques to provide adequate security guarantees at the architecture-level of design abstraction and subsequently, we propose a couple of efficient architecture-level obfuscation techniques to protect the IPs of hardware accelerator designs. These results have been published in [31, 29, 25]. In Chapter 5, we develop a hardware-assisted obfuscation framework to protect the IPs of well-trained neural network models, thus enhancing the security of neural network applications. A part of the results in this chapter has been published in [28]. Finally, Chapter 6 concludes this dissertation and discusses future research directions.

Chapter 2: Circuit-level Obfuscation

2.1 Introduction

In Section 1.1.2, we introduced the concept of logic locking which obfuscates the original design with a secret key such that only upon the application of the correct key the locked design becomes functionally equivalent to the original design. In [110], a new technique called *delay locking* has been proposed to enhance the security of existing logic obfuscation schemes against emerging attacks. The key idea of this approach is to incorporate delay dependence on key values for an obfuscated netlist in addition to traditional logic locking schemes. The outcome of delay locking is that a key not only controls the functionality of a circuit but also its timing profile. Therefore, in order to defeat delay locking, the adversary needs to devise an attack strategy to find a key which recovers not only the correct functionality of a circuit but also its correct timing profile which satisfies pre-defined timing constraints.

In this chapter, we evaluate the security offered by such delay locking based circuit obfuscation scheme. First, we formulate a SAT-based attack strategy to defeat delay locking countermeasure and subsequently develop an effective locking mechanism called stripped-functionality delay locking to protect IPs of hardware designs from an untrusted foundry. The main contributions of the chapter can be summarized as follows:

- We propose a novel SAT formulation based attack strategy called TimingSAT on a delay locked netlist which considers the timing information contained in individual logic gates. TimingSAT attack works in two stages: in the first stage an attacker mounts conventional SAT attack [97] to deduce a correct *functional* key, while in the second stage the attacker embeds the timing profile of a netlist to formulate an iterative SAT attack approach for retrieving a correct *delay* key. In order to carry out the second stage of TimingSAT attack, we first consider a circuit unrolling technique to transform a *timed* combinational netlist to an *untimed* combinational netlist which preserves the delay information of the gates present in the original *timed* netlist. Then, we utilize this timing information embedded netlist to formulate a SAT attack against delay locking countermeasure. We perform extensive experimental evaluations to demonstrate the effectiveness of TimingSAT attack to break delay-locked benchmark circuits within a few hours.

- In order to thwart TimingSAT attack we also develop a technique called stripped-functionality delay locking (SFDL) by combining the concepts of delay locking and stripped-functionality based locking. The key advantage offered by this new scheme is that unlike existing logic locking approaches SFDL simultaneously maintains strong SAT attack resiliency as well as high output corruptibility for wrong keys.

2.2 Background

In this section, we outline the threat model considered and the formulation of conventional iterative SAT attack approach to deobfuscate functionalities of ICs. Such a SAT formulation forms the crux of our proposed attack against delay locking technique as well.

2.2.1 Attack model

We assume an *untrusted foundry* adversarial model as considered in several previous works [97, 82, 108, 76, 110]. To mount a SAT attack on a locked design the foundry has access to the following components:

- An *activated* chip bought from the open market with the *secret* key embeddded in an on-chip tamper-proof memory. This chip is used to obtain *correct* output responses corresponding to the input patterns applied.
- The gate-level netlist reverse engineered from a chip's layout level information available in its GDSII file.

Also, it is to be noted that the foundry provides the designers with libraries containing necessary timing information. Thus, the adversary in an untrusted foundry is aware of setup time T_{setup} and hold time T_{hold} of latches as well as detailed clock tree network related information to determine clock skews feeding into different flip-flops in a design.

2.2.2 An overview of SAT attack

The key idea of SAT attack [97] is to retrieve the *correct* key using a small number of chosen inputs and their corresponding outputs as obtained from an activated chip. These special input/output pairs are referred to as *distinguishing input/output (I/O) pairs*. Each distinguishing I/O pair can identify a subset of *wrong key combinations* and all together they guarantee that only a correct key can be consistent with the correct I/O pairs. The crux of SAT attack approach is to find this set of distinguishing I/O pairs by solving a sequence of SAT formulas. For convenience, we define the following:

Notation 1: Primary inputs/outputs: In this work, we assume full-scan chain access on all the flip-flops in the design so that the attacker can read/write values to all flip-flops as considered in previous works [97, 75, 108, 115]. We refer to the inputs and outputs of any combinational network between two sets of flip-flops (sets S_1 and S_2) as primary inputs (PIs) and primary outputs (POs) respectively, where outputs of S_1 provides PIs and inputs of S_2 are provided by POs. It is to be noted that the primary inputs and outputs of the entire chip is a subset of PIs and POs respectively.

Definition 1: Wrong key (WK): Let us consider the logic function $\vec{PO} = f_l(\vec{PI}, \vec{K})$ and its CNF SAT formula $C(\vec{PI}, \vec{K}, \vec{PO})$. Let $(\vec{PI}, \vec{PO}) = (\vec{PI}_i, \vec{PO}_i)$, where $(\vec{PI}_i, \vec{PO}_i)$ is a correct I/O pair. The set of key combinations WK_i which

result in an *incorrect* output of the logic circuit (i.e. $\vec{PO}_i \neq f_l(\vec{PI}_i, \vec{K}), \forall \vec{K} \in WK_i$) is called the set of wrong key combinations identified by the I/O pair $(\vec{PI}_i, \vec{PO}_i)$. In terms of SAT formula, it can be expressed as $C(\vec{PI}_i, \vec{K}, \vec{PO}_i) = False, \forall \vec{K} \in WK_i$.

Definition 2: Distinguishing input/output (DIO) pair: The SAT attack solves a set of SAT formulas in an iterative manner. In each iteration, it finds a *correct* I/O pair to eliminate a subset of wrong key combinations until none such wrong key is left. An I/O pair at i^{th} iteration is defined as a distinguishing I/O pair $(\vec{PI}_i^d, \vec{PO}_i^d)$, if it can identify a unique subset of wrong keys which have not been identified by the prior $(i-1)$ distinguishing I/O pairs, i.e. $WK_i \not\subset (\cup_{j=1}^{j=i-1} WK_j)$, where WK_i represents the set of wrong keys identified by the DIO pair at i^{th} iteration.

The SAT attack algorithm [97] relies on finding DIO pairs iteratively to identify *unique* wrong key combinations until no further such incorrect keys can be identified. At this final stage, the set of all distinguishing I/O pairs in conjunction identifies all possible wrong key combinations thereby unlocking the netlist. The *correct* key satisfies the following SAT formula G:

$$G := \bigwedge_{i=1}^{\lambda} C(\vec{PI}_i^d, \vec{K}, \vec{PO}_i^d) \tag{2.1}$$

where, $(\vec{PI}_i^d, \vec{PO}_i^d)$ is the distinguishing I/O pair from i^{th} SAT iteration and λ is the total number of SAT iterations. Finally, a key $\vec{K_C}$ is identified which results in correct functionality for all the identified DIO pairs. $\vec{K_C}$ is guaranteed to belong to the *correct equivalence class of keys* since no other DIO pairs exist.

2.3 Security Evaluation of Delay-locked Circuits

Several latest research works on logic locking have mostly focused on developing techniques to counter SAT formulation based attacks [25, 118, 52, 88, 110, 77, 78, 90]. In [110], a new countermeasure called delay locking has been proposed to thwart SAT as well as other existing attacks on logic locking. In this work, we evaluate the security offered by such delay-locked netlists to ensure design IP protection in an untrusted IC supply chain.

2.3.1 Delay Locking

In delay locking approach, *delay keys* are inserted into circuits to make the delay profiles of synthesized netlists dependent on these key values. If the *delay keys* are incorrect, then there may arise timing violations leading a circuit to malfunction [110].

The basic locking infrastructure used in this approach is called tunable delay key-gate (TDK). As shown in Fig. 2.1, each TDK takes 2 bits of key inputs: the first key bit k_1 is the *functional key* and the second key bit k_2 is the *delay key*. The functional key determines the logical functionality (*i.e.,* buffer or inverter) of this gate while the delay key determines the delay of this gate. The delay key is basically the control signal of a tunable delay buffer (TDB) where the tunability of delay is achieved by a capacitive load. The delay key controls whether the capacitive load is connected to the circuit. If it is connected, the TDK will have longer delay. Let

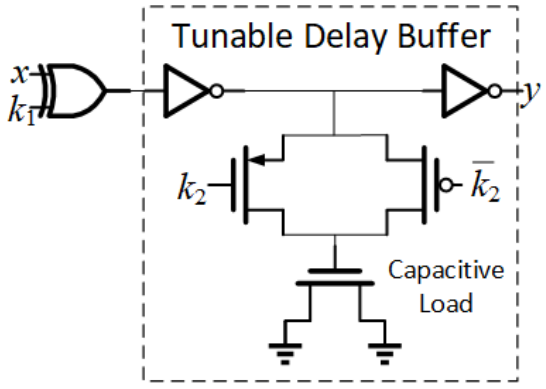


Figure 2.1: Structure of a tunable delay key-gate (TDK)

us define *delay ratio* as

$$dr = d_1/d_0 \tag{2.2}$$

where, d_1 and d_0 are the delay values when the capacitive load is connected or disconnected, respectively. The delay ratio of a netlist can be tuned at design time.

In a sequential circuit, the combinational path between two flip-flops (FFs) need to satisfy the timing constraints in order for the entire circuit to function correctly. For the combinational circuit between FFs i and j, constraint for longest path delay D_{ij}^{long} is expressed as

$$D_{ij}^{long} + T_{set}^{j} \leq T_{clk} + T_j - T_i, \quad \forall i, j \tag{2.3}$$

where, T_{set}^{j} is the setup time of FF j, T_{clk} is the clock period, T_i and T_j are the arrival time of clock signals at FFs i and j, respectively. $T_j - T_i$ gives the clock skew. Equation (2.3) specifies that the output of the the combinational circuit needs to be *ready* for the setup time T_{set}^{j} before the next clock cycle in order to propagate the correct output to the next stage. Besides, the combinational circuit

also needs to comply with a shortest path constraint expressed as

$$D_{ij}^{short} \geq T_{hold}^{j} + T_j - T_i, \quad \forall i, j \tag{2.4}$$

where, D_{ij}^{short} is the shortest path between FFs i and j, and T_{hold}^{j} is the hold time for FF j. This shortest path constraint specifies that the output of the combinational circuit must hold for at least T_{hold}^{j} after the clock signal arrives at FF j. A *correct* delay key will ensure that the timing profile of a circuit satisfies designer specified constraints.

2.3.2 TimingSAT Attack

The primary assumption behind delay locking scheme is that conventional SAT attack [97] is capable of deciphering only the *functional key*, but fails to deduce a correct *delay key* which meets the pre-defined timing constraints of a design. In this section, we propose TimingSAT attack which utilizes timing profiles of all the gates in a circuit to determine a correct *delay key* of the delay logic locked netlist. First, we outline a circuit unrolling approach which helps us to capture the timing information in the form of Boolean functions, followed by an iterative TimingSAT formulation which guarantees to find correct *delay key* for a delay locked netlist.

2.3.2.1 Circuit Unrolling

We use a circuit unrolling approach to transform *timed* combinational netlist to a larger *untimed* zero-delay combinational netlist which embeds within the new netlist necessary delay profiles of all the gates present in original *timed* netlist. We

utilized the data-dependent delay information of each gate to unroll the entire netlist as proposed in [100]. The main advantage of such an approach is fairly accurate evaluation of gate output timing profiles as it considers *input pattern-dependent* effects such as data-dependent gate delays and multiple-inputs switching activities. In the context of modeling timing profiles of integrated circuits, this technique is more *realistic* as it considers a much more complex *inertial-delay model* as compared to prior works like [35] which consider relatively simple *constant-delay model*.

The input patterns applied to a gate has significant impact on the delay values. We used TSMC 180nm CMOS model in Cadence to obtain data dependent delay characterization of all the gates present in our library. To highlight the *wide range* of delay values depending on input patterns applied, we report the delay figures at the output X of an XNOR gate for various transitions of inputs A and B in Table 2.1. It is quite apparent that the delay values (column 2 of the table) span over a wide range (from as low as 70ps to as high as 128 ps) depending on the nature of input transitions. Similar variations were observed in delay profiles for other gate types as well. Like [100], in this work the gate-delay characterization is considered as a *preprocessing step* where any suitable state-of-the-art timing characterization technique can be used. Our proposed TimingSAT attack on delay locking utilizes this gate characterization to construct its SAT formulations which embeds necessary timing information of a netlist. It is to be noted that such detailed gate-delay characterization is performed by the foundry (also the attacker as per our threat model) who provides such information to designers using liberty file (*.lib* format) for circuit optimization. Even though different designs may contain

Table 2.1: Input dependent delay profiling of XNOR gate

Input transition pattern	delay (ps)	quantized delay	X=A⊙B
$A\uparrow$,B_0	70	7	0
A_0,$B\uparrow$	80	8	0
A_1,$B\downarrow$	123	12	0
$A\downarrow$,B_1	128	13	0
$A\downarrow$,B_0	65	6	1
A_0,$B\downarrow$	72	7	1
A_1,$B\uparrow$	94	9	1
$A\uparrow$,B_1	100	10	1

different types of gates (having different delay profiles),the attacker can utilize details from the same *.lib* file to analyze any delay locked circuit realized using the same standard cell library. We would like to also highlight that our proposed formulations are independent of the characterization approach used to model the gate delay values and will work with any other suitable state-of-the-art timing characterization technique as well.

The *inertial delay model* [100] considered in this work rejects any input pulse whose width is shorter than the delay of the gate being driven by it. Such an assumption leads to the development of an accurate data-dependent delay model which can be utilized in the circuit unrolling process as outlined next. Let us consider a 2-input XNOR gate whose delay values have been *quantized* using unrolling time step granularity g of $10ps$ as presented in column 3 of Table 2.1 for various types of input transitions. The XNOR gate output X will rise at time t if the following

equation is evaluated to logic 1:

$$\begin{aligned} X(t)\uparrow \;=\; & A(t-7)\;\overline{A(t-6){:}A(t-1)}\;\overline{B(t-6){:}B(t-1)} \\ & +\; \overline{A(t-7){:}A(t-1)}\;B(t-8)\;\overline{B(t-7){:}B(t-1)} \\ & +\; A(t-9){:}A(t-1)\;\overline{B(t-10)}\;B(t-9){:}B(t-1) \\ & +\overline{A(t-11)}A(t-10){:}A(t-1)\;B(t-10){:}B(t-1) \end{aligned}$$

Intuitively, each line of the above equation corresponds to one of the four rising input cases as shown in Table 2.1 (rows 5 to 8): the first line shows output X rising at t as a result of A falling at $(t-6)$, i.e., $A(t-7)\overline{A(t-6)}$, and B staying constant low, i.e., $\overline{B(t-6)}$. Also, it is further required that both A and B remain low subsequently without glitching till time t, i.e., $\overline{A(t-5){:}A(t-1)}$ and $\overline{B(t-5){:}B(t-1)}$, as otherwise any input glitches might alter the output X logic value. Such glitch constraints will vary depending not only on the gate type but also on the nature of input transitions as detailed in [100]. In a similar manner, the remaining lines of $X(t)\uparrow$ are formulated to capture the effect of input transitions leading to evaluation of $X=1$ at time t. On the other hand, the output X will fall at time t if the following equation is evaluated to logic 0 (rows 1 to 4 of Table 2.1):

$$\begin{aligned} X(t)\downarrow \;=\; & \overline{A(t-8)}A(t-7){:}A(t-1)\;\overline{B(t-8){:}B(t-1)} \\ & +\; \overline{A(t-9){:}A(t-1)}\;\overline{B(t-9)}\;B(t-8){:}B(t-1) \\ & +A(t-13){:}A(t-1)\;B(t-13)\;\overline{B(t-12){:}B(t-1)} \\ & +A(t-14)\;\overline{A(t-13){:}A(t-1)}\;B(t-14){:}B(t-1) \end{aligned}$$

Finally, the logic value of node X will be high at time t if either (i) X rises at time t or (ii) X was high at time $t-1$ and does not fall at time t, else the value of X will be low. Combining these conditions, the logic value of node X at time t can be expressed as follows:

$$X(t) = X(t) \uparrow + X(t-1)\overline{X(t) \downarrow} \tag{2.5}$$

Thus, the *timed* gate output X is described as an *untimed* combinational function of gate inputs A and B. In this work, we used the above modeling approach to capture data-dependent delay for all types of gates present in our library, namely AND, NAND, OR, NOR, XOR, XNOR, BUF, and NOT. Subsequently, we **unrolled the entire circuit by unrolling individual nodes** in the design to construct a *timing profile embedded netlist*. We utilize such a timing information embedded netlist to formulate an iterative SAT attack technique on delay-locked circuits. It is to be noted that our proposed TimingSAT attack methodology against delay-locked circuits is *independent* of the gate delay characterization approach used.

2.3.2.2 TimingSAT formulation

As highlighted earlier, once a netlist has been unrolled, we have a purely combinational *untimed* Boolean network which can be analyzed using state-of-the-art SAT solvers as outlined next. In order to retrieve all the key bits of the locked design the adversary performs the following two attack stages:

• **Stage 1: Finding *functional key.*** In this stage of the attack, the target is to retrieve only the *functional key* $\vec{K^F}$ used to *functionally* obfuscate the netlist. To perform this step, the attacker basically formulates an iterative SAT attack as outlined in section 2.2.2 by considering a modified netlist which logically replaces the TDKs simply with XOR key-gates driven by *functional* key inputs. It is to be noted that such a modification has no impact on the *functional* evaluation of the gate-level netlist under consideration. Thus, the attacker can utilize the input-output responses of the activated chip to model a SAT formulation as outlined in equation 2.1 to find correct *functional key* $\vec{K_C^F}$.

• **Stage 2: Finding *delay key.*** In this stage of TimingSAT attack, the adversary targets to retrieve the *delay key* $\vec{K^D}$ used to obfuscate the timing characteristics of the design. The initial netlist (before unrolling) considered in this phase is driven by primary inputs and *delay key* inputs to TDBs, while the XOR gates of the TDKs are fed with the *functional key* $\vec{K_C^F}$ as determined in Stage 1 of the attack. Thus, in the current stage $\vec{K_C^F}$ is a fixed *known* value and the attacker focuses to retrieve only the *delay key*. Let us assume that the primary input PI and *delay key* input $\vec{K^D}$ are applied to the combinational network and in response to this excitation the primary output updates from its initial state $\vec{PO}^{init}$ to its new state $\vec{PO}$. The attacker needs to ensure that $\vec{K^D}$ satisfies (i) the longest path timing constraints and (ii) the shortest path timing constraints for the proper functioning of the design. In order to do so, we utilize the circuit unrolling approach as outlined in section 2.3.2.1 to construct timing information embedded *unrolled* netlist. Also, like before,

the attacker has access to *correct* input-output responses of activated chip (correct in terms of both functionality and timing characteristics) which can be utilized to develop SAT formulation incorporating longest and shortest path delay constraints on the unrolled netlist as follows: Let N_g denote the total number of gates present in the reverse-engineered netlist and g_i denote the i^{th} logic gate, $i \in [N_g]$. We unroll every gate g_i present in the design using the technique outlined in section 2.3.2.1 for ld_{g_i} times, where ld_{g_i} denote the *longest path delay* corresponding to gate g_i across **all the paths** leading from the output of g_i to any of its destination flip-flops. The outcome of this unrolling process leads to the following chain of equations corresponding to the output X^{g_i} for gate g_i across $(ld_{g_i}+1)$ time steps, ranging from n to $(n-ld_{g_i})$, with the granularity of unrolling considered being 1 time step:

$$\begin{aligned}
X^{g_i}(n) &= X^{g_i}(n)\uparrow + X^{g_i}(n-1)\overline{X^{g_i}(n)\downarrow} \\
X^{g_i}(n-1) &= X^{g_i}(n-1)\uparrow + X^{g_i}(n-2)\overline{X^{g_i}(n-1)\downarrow} \\
&\vdots \\
X^{g_i}(n-ld_{g_i}) &= X^{g_i}(n-ld_{g_i})\uparrow \\
&\quad + X^{g_i}(n-ld_{g_i}-1)\overline{X^{g_i}(n-ld_{g_i})\downarrow}
\end{aligned} \tag{2.6}$$

where, n denotes the quantized time instant at which the output X^{g_i} is evaluated (see Table 2.1 for gate delay quantization). It is to be noted that there is no affect of $X^{g_i}(n-ld_{g_i}-1)$ on the primary output value at time n. This is because ld_{g_i} being the *longest path delay* across all possible paths from the output of gate g_i

to any of its destination flip-flops guarantee that $X^{g_i}(n - ld_{g_i} - 1)$ is a function of *only* prior input PI^{init}. This in turn implies that any gate (including primary output gates) lying in a path originating from $X^{g_i}(n - ld_{g_i} - 1)$ in the unrolled circuit is also a function of $\vec{PI^{init}}$. However, for any subsequent time step m between $n - ld_{g_i}$ and n, the logic value evaluated at X^{g_i} due to application of $\vec{PI}$ will propagate to destination flip-flops for some paths having path delays shorter than the difference in output observation instant n and the time instant m of evaluating X^{g_i}. On the other hand, for paths having longer path delays from the output of gate g_i to destination flip-flops, the contribution of X^{g_i} is due to its value computed with initial $\vec{PI^{init}}$. We utilize this path delay dependent nature of destination flip-flop updates (or $\vec{PO}$ updates as per Notation 1) to incorporate timing constraints in an iterative SAT formulation for a combinational network.

SAT formulation with longest path constraint:

The longest path timing constraint of a circuit signifies that the clock period should be sufficiently large to allow the data to propagate through **all** the possible combinational paths and to be set up at inputs of destination flip-flops before the arrival of next triggering clock signal. Let us denote the clock period of the activated chip as T_{clk}, the setup time for flip-flop j as T^j_{setup}, and the clock arrival times in flip-flops i and j with variables T_i and T_j. Since the attacker in an untrusted foundry setting is aware of the layout level details of the chip, she can ascertain clock skews affecting the arrival times of clock signals to various flip-flops in the design. Also, T_{clk} is known from design specifications. In order to ensure that there is no longest

path violation in the evaluation of primary outputs $\vec{PO}$, i.e., **all** primary outputs of the circuit are functions of current primary input $\vec{PI}$ and *delay key* $\vec{K^D}$, we unroll the circuit for T_{high} time steps, where

$$T_{high} = max\{T_{clk} + T_j - T_i - T_{set}^j\}/g, \quad \forall i, j \tag{2.7}$$

In the above equation, g represents the granularity of unrolling time steps based on which delay values have been quantized. If T_{high} number of unrolling time steps are performed, then at time step n of function evaluation, all the primary outputs are updated to $\vec{PO}$. Again, as evaluation of $\vec{PO}$ is dependent on all intermediate gates in their fan-in cones, it is guaranteed that for any such gate g_i, $X^{g_i}(n)$ is function of $\vec{PI}$ and *delay key* $\vec{K^D}$ and is not dependent on $\vec{PI^{init}}$. As an outcome of this unrolling process the original *timed* netlist is transformed to a larger *untimed* netlist which captures necessary timing profiles of all the gates. Let us denote the corresponding CNF SAT formula of the unrolled circuit as $C_h(\vec{PI}, \vec{K^D}, \vec{PO})$. Thus, we can write an iterative SAT formulation of the unrolled circuit incorporating longest path constraints as follows:

$$\begin{aligned} I_i = {} & C_h(\vec{PI}, \vec{K_1^D}, \vec{PO_1}) \wedge C_h(\vec{PI}, \vec{K_2^D}, \vec{PO_2}) \wedge (\vec{PO_1} \neq \vec{PO_2}) \\ & (\bigwedge_{j=1}^{j=i-1} C_h(\vec{PI_{j,d}}, \vec{K_1^D}, \vec{PO_{j,d}})) \wedge (\bigwedge_{j=1}^{j=i-1} C_h(\vec{PI_{j,d}}, \vec{K_2^D}, \vec{PO_{j,d}})) \end{aligned} \tag{2.8}$$

where, I_i denotes the i^{th} SAT iteration formulation, and $(\vec{PI}_{\{1\ldots i-1\},d}, \vec{PO}_{\{1\ldots i-1\},d})$ are the DIO pairs which are found in previous $(i-1)$ iterations using activated chip.

SAT formulation with shortest path constraint:

To realize the correct functionality of a delay locked design, the shortest path timing constraints should also be satisfied in addition to the longest path timing constraint. Let us denote the hold time of destination flip-flop j as T^j_{hold} (known to the attacker in an untrusted foundry setting). The hold time for **any** flip-flop in the design should be shorter than the shortest path delay through the combinational network. In order to ensure that there is no shortest path violation in the evaluation of primary outputs $\vec{PO}^{\tau}$ at time τ, τ being any time instant lesser than the shortest path delay of the circuit, we unroll the circuit for T_{low} time steps, where,

$$T_{low} = min\{T^j_{hold} + T_j - T_i\}/g, \quad \forall i, j \tag{2.9}$$

In the above equation, g represents the granularity of unrolling time steps based on which delay values have been quantized. The intuition behind unrolling the circuit T_{low} time steps is to ascertain that **none** of the primary outputs of the design are functions of current primary input $\vec{PI}$, but rather depends on the initial input $\vec{PI^{init}}$, resulting in no shortest path violation. We can write an iterative SAT formulation of the unrolled circuit incorporating shortest path constraints as follows:

$$\begin{aligned} J_i = & C_l(\vec{PI}, \vec{K^D_1}, \vec{PO^{init}_1}) \wedge C_l(\vec{PI}, \vec{K^D_2}, \vec{PO^{init}_2}) \wedge (\vec{PO^{init}_1} \neq \vec{PO^{init}_2}) \\ & (\bigwedge_{j=1}^{j=i-1} C_l(\vec{PI}_{j,d}, \vec{K^D_1}, \vec{PO^{init}_{j,d}})) \wedge (\bigwedge_{j=1}^{j=i-1} C_l(\vec{PI}_{j,d}, \vec{K^D_2}, \vec{PO^{init}_{j,d}})) \end{aligned} \tag{2.10}$$

where, J_i denotes the i^{th} SAT iteration formulation, $C_l(\vec{PI}, \vec{K^D}, \vec{PO^{init}})$ is CNF SAT for the unrolled circuit, and $(\vec{PI}_{\{1...i-1\},d}, \vec{PO}^{init}_{\{1...i-1\},d})$ are the DIO pairs which are found in previous $(i-1)$ iterations using activated chip.

Combined SAT formulation:

If a *delay key* $\vec{K^D}$ satisfies both longest and shortest path constraints across all the possible paths in a combinational network, then it belongs to the *equivalence class of correct delay keys* used to obfuscate the netlist. In the second stage of the TimingSAT attack, the objective of finding such a correct *delay key* can be formulated as an iterative SAT approach, with F_i denoting the i^{th} SAT iteration, as follows:

$$
\begin{aligned}
F_i \;&=\; (U \vee V) \wedge (Y_i \wedge Z_i), \qquad where, \\
U&=C_h(\vec{PI}, \vec{K_1^D}, \vec{PO_1}) \wedge C_h(\vec{PI}, \vec{K_2^D}, \vec{PO_2}) \wedge (\vec{PO_1} \neq \vec{PO_2}) \\
V&=C_l(\vec{PI}, \vec{K_1^D}, \vec{PO_1^{init}}) \wedge C_l(\vec{PI}, \vec{K_2^D}, \vec{PO_2^{init}}) \wedge (\vec{PO_1^{init}} \neq \vec{PO_2^{init}}) \\
Y_i&=(\bigwedge_{j=1}^{j=i-1} C_h(\vec{PI}_{j,d}, \vec{K_1^D}, \vec{PO}_{j,d})) \wedge (\bigwedge_{j=1}^{j=i-1} C_h(\vec{PI}_{j,d}, \vec{K_2^D}, \vec{PO}_{j,d})) \\
Z_i&=(\bigwedge_{j=1}^{j=i-1} C_l(\vec{PI}_{j,d}, \vec{K_1^D}, \vec{PO^{init}_{j,d}})) \wedge (\bigwedge_{j=1}^{j=i-1} C_l(\vec{PI}_{j,d}, \vec{K_2^D}, \vec{PO^{init}_{j,d}}))
\end{aligned}
\tag{2.11}
$$

The above iterative SAT formula basically combines the SAT formulations incorporating longest and shortest path delay constraints as outlined in equations 2.8 and 2.10 respectively. If the SAT formula is satisfied, then assignments for variables $\vec{PI}$, $\vec{K_1^D}$, $\vec{K_2^D}$, $\vec{PO}_1$, $\vec{PO}_2$, $\vec{PO_1^{init}}$, $\vec{PO_2^{init}}$ will be generated. The expression $(U \vee V)$ in

equation 2.11 above evaluates the circuit functionality for a specific input $\vec{PI} = \vec{PI}_{j,d}$ for two different *delay keys* $\vec{K_1^D}$ and $\vec{K_2^D}$ such that primary output is different **either** due to netlist unrolling for T_{high} time steps ($\vec{PO}_1 \neq \vec{PO}_2$) **or** due to netlist unrolling for T_{low} time steps ($\vec{PO_1^{init}} \neq \vec{PO_2^{init}}$). This guarantees that the input $\vec{PI} = \vec{PI}_{j,d}$ is capable of identifying two keys $\vec{K_1^D}$ and $\vec{K_2^D}$ which produce different outputs either due to longest path violation or due to shortest path violation or both. Hence, *at least* one of the two keys must be wrong. Again, the distinguishing input $\vec{PI}_{j,d}$ identified in the previous j^{th} iteration results in corresponding *correct* output $\vec{PO}_{j,d}$ with the netlist unrolled for T_{high} time steps **and** *correct* output $\vec{PO_{j,d}^{init}}$ with the netlist unrolled for T_{low} time steps. These correct outputs $\vec{PO}$ and $\vec{PO^{init}}$ are *known* from the activated chip response ($ac\_response$) available to the attacker. The clauses in the expression ($Y_i \wedge Z_i$) guarantee that the keys $\vec{K_1^D}$ and $\vec{K_2^D}$ which produce *different* outputs in the current SAT iteration (either different $\vec{PO}$ or different $\vec{PO^{init}}$ or both different), have produced the *correct* outputs for all the previous DIO pairs. This implies that in the current iteration at least one incorrect *delay key* has been identified which was not detected in any of the prior iterations.

The Stage 2 of TimingSAT attack algorithm is shown in Algorithm 2.1. It starts by first solving the for the clause ($U \vee V$), and in any subsequent i^{th} iteration it adds clauses of the form ($Y_i \wedge Z_i$) as highlighted in equation 2.11. The algorithm terminates when the SAT formula F_i is unsatisfiable and thus implying that there is no more DIO pairs. Finally, a *delay key* belonging to the *correct equivalence class* is obtained by finding $\vec{K^D}$ that satisfies all the correct input-output pairs when the netlist is unrolled for T_{high} time steps as well as for T_{low} time steps, thus pertaining

Algorithm 2.1: TimingSAT Attack Algorithm (Stage 2)

Input: C_h, C_l, $\vec{PI}^{init}$, and *ac_response*

Output: *correct delay key* $\vec{K}_C^D$

$i := 1$;

$G_i := True$;

$U = C_h(\vec{PI}, \vec{K}_1^D, P\vec{O}_1) \wedge C_h(\vec{PI}, \vec{K}_2^D, P\vec{O}_2) \wedge (P\vec{O}_1 \neq P\vec{O}_2)$;

$V = C_l(\vec{PI}, \vec{K}_1^D, \vec{PO}_1^{init}) \wedge C_l(\vec{PI}, \vec{K}_2^D, \vec{PO}_2^{init}) \wedge (\vec{PO}_1^{init} \neq \vec{PO}_2^{init})$;

$F_i = (U \vee V)$;

$\vec{PO}_{i,d}^{init} := ac\_response(\vec{PI}^{init})$;

while *sat[*F_i*]* **do**

  $\vec{PI}_{i,d} :=$sat_assignment$_{\vec{PI}}[F_i]$;

  $P\vec{O}_{i,d} := ac\_response(\vec{PI}_{i,d})$;

  $G_{i+1} := G_i \wedge C_h(\vec{PI}_{i,d}, \vec{K}^D, P\vec{O}_{i,d}) \wedge C_l(\vec{PI}_{i,d}, \vec{K}^D, \vec{PO}_{i,d}^{init})$;

  $F_{i+1} := F_i \wedge C_h(\vec{PI}_{i,d}, \vec{K}_1^D, P\vec{O}_{i,d}) \wedge C_h(\vec{PI}_{i,d}, \vec{K}_2^D, P\vec{O}_{i,d})$

    $\wedge\, C_l(\vec{PI}_{i,d}, \vec{K}_1^D, \vec{PO}_{i,d}^{init}) \wedge C_l(\vec{PI}_{i,d}, \vec{K}_2^D, \vec{PO}_{i,d}^{init})$;

  $i := i + 1$;

end

$\vec{K}_C^D :=$ sat_assignment$_{\vec{K^D}}(G_i)$;

to the longest and shortest path delay constraints of the design. This combined formulation corresponding to Stage 2 of TimingSAT attack is **guaranteed** to find a *correct delay key* $\vec{K_C^D}$.

2.3.3 Experimental Results

In this section, we report the effectiveness of our proposed TimingSAT attack to deobfuscate delay-locked benchmark circuits. We first state the configuration used to model the capacitive load of a tunable delay key-gate (TDK) as presented in section 2.3.1. Then, we highlight the preprocessing step used to construct input transition dependent delay models for various types of gates in our library. Finally, we report the runtimes of TimingSAT attack to recover correct *delay keys* for the benchmark circuits.

2.3.3.1 Modeling TDB and gate delays

In order to model the tunable delay buffer (TDB) into the TimingSAT formulation, we use a multiplexer to choose between paths having different signal propagation delays as shown in Fig. 2.2. The i^{th} bit of the *delay key* input K_i^D acts as the select line for the corresponding multiplexer to select one of the two paths having n_1 and n_2 delay buffers. By adjusting n_1 and n_2, different delay ratios $dr = n_1/n_2$ can be achieved. To model delay of gates present in our library, we measure the *data-dependent* delay values of each gate as follows: Different types of logic gate in our library (namely AND, NAND, OR, NOR, XOR, XNOR, BUF,

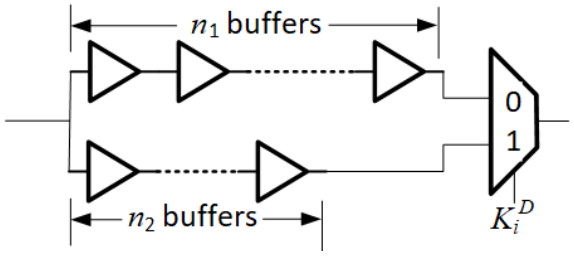


Figure 2.2: Equivalent representation of TDB

Table 2.2: Specifications of evaluated ISCAS circuits

benchmark	netlist	$\#PI$	$\#PO$	$\#TDK\ gates$	$\#gates$
ISCAS85	$apex2$	39	3	31	643
	$c880$	60	26	19	1344
	$i4$	192	6	17	370
ISCAS89	$s838$	67	34	20	488
	$s1488$	14	25	33	872
	$s1494$	14	25	32	868

and NOT) were synthesized using TSMC 180nm standard cell library in Cadence to capture their timing characteristics. For each gate type, we characterized the delay profiles corresponding to all the input patterns leading to transitions at the gate output. The actual delay values were then quantized into unrolling time steps with a granularity g of 10 ps (as shown in Table 2.1 corresponding to different input patterns for an XNOR gate).

2.3.3.2 TimingSAT Attack Results

We used 6 netlists from ISCAS benchmark as specified in Table 2.2 for evaluation of our proposed TimingSAT attack. We inserted TDKs in the netlists with 5% key-gate overhead as done in previous works [97, 82, 108, 76, 110]. For running SAT formulation based experiments, we used an Intel Xeon E3-1245 CPU with 32 GB RAM. The SAT solving times required to find the *functional keys* of targeted netlists, i.e., Stage 1 of TimingSAT attack have already been reported in [97]. Therefore, we only report the SAT solving time to find the *delay key* $\vec{K^D}$ of size $\|\vec{K^D}\|$ using our proposed Algorithm 2.1. For each benchmark circuit, we simulated Stage 2 of TimingSAT attack for different *delay ratio* (dr) values, leading to the overall evaluation of 24 benchmark designs (6 different netlists, each locked with 4 different *delay ratio* values $dr = 2, 3, 4, 5$). For all our experiments, we set $T_{high} = 200$ and $T_{low} = 10$ to unroll the netlists imposing the longest path and shortest path delay constraints respectively. In Table 2.3, we report the number of SAT iterations and CPU time for SAT solving (in mins) as required to find correct *delay key* $\vec{K_C^D}$ for different delay locked netlists with $dr = 5$. It was observed that all the evaluated benchmark circuits were deobfuscated within a few hours, thus highlighting the effectiveness of our proposed TimingSAT attack approach. In Fig. 2.3, we outline the SAT solving time vs. *delay ratio* for different netlists. From the plots, it is apparent that for various dr values considered ($dr = 2, 3, 4, 5$), TimingSAT attack successfully retrieves the *correct* keys of all the locked netlists within a practical time.

Table 2.3: TimingSAT attack stage 2: Finding *delay key* ($dr = 5$)

netlist	$\|\vec{K^D}\|$	#iterations	CPU time (mins)
$apex2$	31	9	29.68
$c880$	19	1	22.39
$i4$	17	1	4.79
$s838$	20	4	9.39
$s1488$	33	1	142.83
$s1494$	32	1	153.67

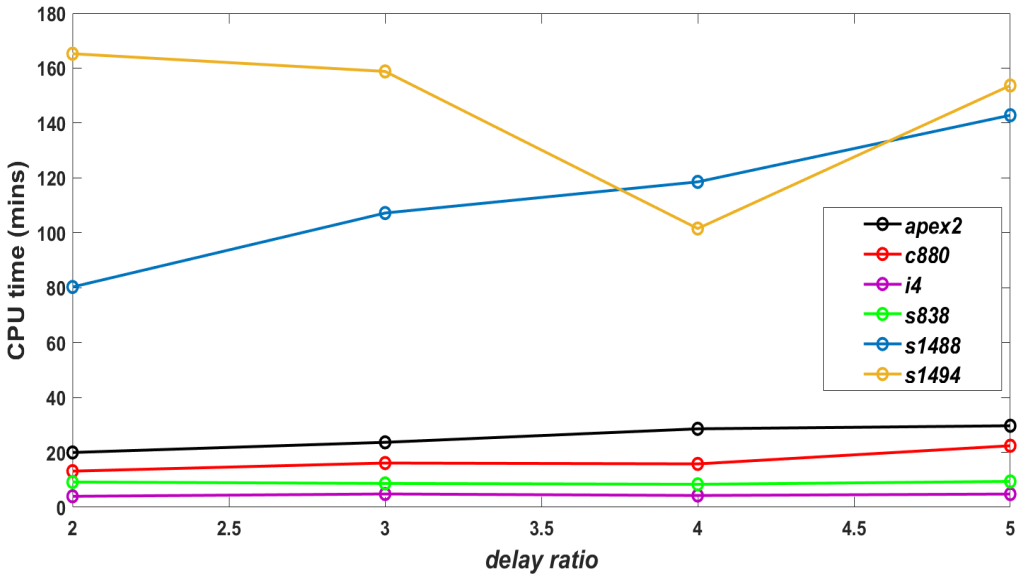


Figure 2.3: CPU time (mins) for TimingSAT attack Stage2 vs *delay ratio*

2.3.3.3 Improving Scalability of TimingSAT Attack

The complexity of TimingSAT attack is dependent on the size of the unrolled circuit. The circuit unrolling strategy outlined in section 2.3.2.1 is mainly determined by the following three factors (i) the topology of the netlist including number and types of gates present (ii) the granularity g of unrolling time steps based on which delay values are quantized and (iii) the number of unrolling time steps which is obtained by dividing the worst-case circuit delay by g. The parameter g is set by the attacker based on the precision required to avoid any glitches to gate inputs as required for the *correct* unrolled Boolean formulation of a circuit [100].

There exists a *trade-off* between accuracy of delay modeling and scalability of corresponding circuit timing analysis. For example, on one hand, if we consider a simple constant-delay model as assumed in [21] then SAT solver based timing analysis can be easily scaled to large benchmark netlists. On the other hand, if we consider a complex inertial-delay model (as assumed in this work) then the timing analysis will not be as scalable as the former approach due to increased computational burden. However, it is to be noted that the latter approach is much more realistic as it accounts for input pattern dependent effects (such as data-dependent gate delays and multiple-inputs switching activities) and hence *guarantees the correctness of the delay key* thus obtained. In this work, we have limited our experiments to moderate size benchmark circuits which could be analyzed by setting g=10ps in a system with 32 GB of RAM.

Randomized algorithm approach: In order to enhance the scalability of Stage 2 of TimingSAT attack without compromising the accuracy of circuit timing characterization, the attacker can use a randomized algorithm based approach to find *delay key* $\vec{K^D}$. In such an approach the attacker will first unroll the reverse-engineered netlist using inertial-delay model based gate timing characterization. Then, she will randomly fix a chosen fraction of bits of $\vec{K^D}$ to either 0 or 1 and then run TimingSAT attack algorithm (Algorithm 2.1) to figure out the remaining portion of the key. If the run of the algorithm terminates successfully within a pre-determined time limit, then the attacker is still guaranteed to have found correct $\vec{K^D}$. On the other hand, if the algorithm times out then the attacker needs to repeat the process till it converges to find a correct *delay key*, each time randomly fixing a portion of $\vec{K^D}$ to some constant value. Such a randomized algorithm based attack approach basically reduces the computational complexity of a SAT solver (thus improving TimingSAT attack scalability) by decreasing the number of unknown variables to be ascertained. The success of this attack relies on the fact that for a delay locked benchmark circuit there will be multiple $\vec{K^D}$ values (large key space) which satisfy both shortest and longest path timing constraints. Therefore, by fixing a fraction of $\vec{K^D}$, the SAT solver will be able to efficiently determine a combination of the other key bits which doesn't lead to any circuit timing violations. The development of other techniques to enhance the scalability of TimingSAT attack will be considered as a future research direction.

2.4 Stripped-Functionality Logic Locking

In [118], a provably secure logic locking scheme called stripped-functionality logic locking (SFLL) has been proposed which provides a quantifiable trade-off between SAT attack resiliency and removal/ bypass types of attacks. SFLL *strips* a portion of the design functionality to create a functionality stripped circuit (FSC) which differs from the original design. This FSC can be conceived to have a controllable *built-in error* which is canceled by using a restore unit. There are two versions of SFLL called SFLL-HD and SFLL-$flex$, each of which restores the stripped functionality using protected input patterns. Our proposed SFDL approach (details in next subsection) utilizes the SFLL-$flex^{cxk}$ version which allows the designer to select c number of IP-critical input cubes[1] with each cube have k specified bits. In SFLL-$flex$, the restore unit stores the protected input cubes (which is the secret *restore key* $\vec{K}^R$) in a look-up table (LUT). In addition, the LUT also stores a *flip vector* associated with each such input cube to ascertain which netlist wires are to be flipped to recover the stripped functionality. SFLL-$flex^{cxk}$ exhibits a trade-off between resilience to oracle-guided (SAT based) and removal attacks which can be expressed as follows:

- $(k - \lceil log_2\ c \rceil)$ -security against SAT attack
- $c \cdot 2^{(n-k)}$ -security against removal attack

[1]Input cube refers to a partly-specified input pattern $\vec{PI}$. An n-bit input cube with k specified (care) bits corresponds to 2^{n-k} input patterns. Note that the bits of an input pattern are either *logic*-0/*logic*-1 or don't cares (x's).

where, the notion of λ-security and related mathematical derivations can be found in [118]. The above set of equations imply that higher the resiliency of a SFLL-$flex$ locked circuit against SAT attack, lower is its output corruptibility which in turn makes it susceptible to removal/bypass types of attacks.

2.4.1 Combining Delay Locking and SFLL-flex

We propose a new obfuscation technique called stripped-functionality delay locking (SFDL) by combining the concepts of delay locking and functionality stripping of circuits such that a locked design is robust against SAT attack and at the same time exhibits high output corruptibility. The construction of our proposed SFDL obfuscation scheme is illustrated using Fig. 2.4. The designer first locks a circuit by applying SFLL-$flex$ to obtain an FSC. Then, the designer applies delay locking on this FSC using two types of tunable delay buffers (TDBs) as follows:

- **Timing-driven TDBs (T-TDBs):** Such TDBs are driven by *delay key* $\vec{K}_T^D$ as done in conventional delay locking.

- **LUT-driven TDBs (L-TDBs):** This new type of TDBs are driven by a key $\vec{K}_L^D$ which is derived as a function of the LUT entries in the restore unit.

The bits of conventional *delay key* $\vec{K}_T^D$ (shown in green) are chosen by the designer to drive the T-TDBs (inserted at randomly selected netlist wires) such that the desired shortest and longest path timing constraints are met when the functionality of FSC is restored. Note that the contents of restore unit's LUT (*restore key* $\vec{K}^R$) and the key-bits of $\vec{K}_T^D$ reside in a tamper-proof memory. The key

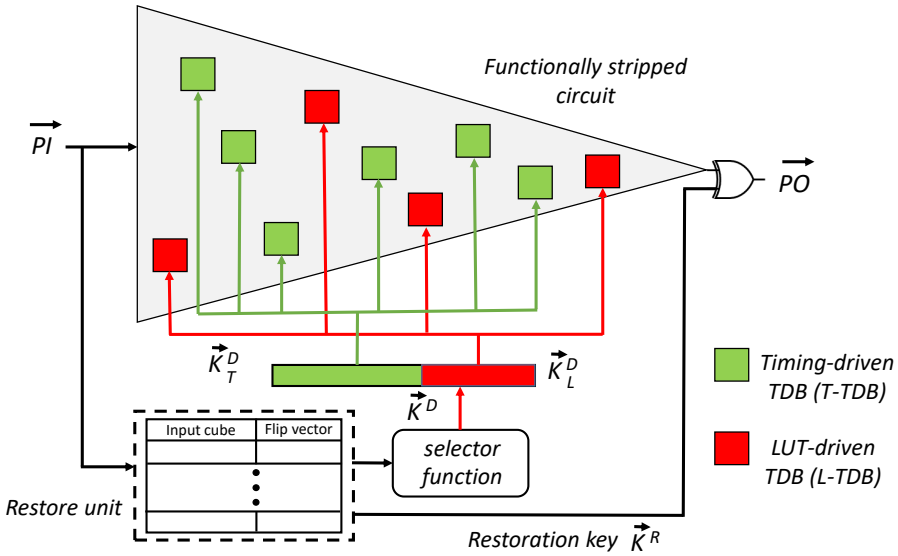


Figure 2.4: Stripped-functionality delay locking technique

$\vec{K}_L^D$ (shown in red) is derived from key $\vec{K}^R$ by using *any* designer-specified *selector function* as shown in Fig. 2.4. For simplicity in our experimental evaluations (as reported later in section 2.4.2) we used a c-to-1 MUX which selects a row (out of c entries) of the LUT as $\vec{K}_L^D$ to drive the L-TDBs.

The original behavior of an SFDL-locked circuit will be retrieved when (i) the restore unit's LUT is loaded with the key $\vec{K}^R$ to retrieve the correct functionality and (ii) the *delay key* $\vec{K}_T^D$ is loaded to reconstruct the proper timing profile of the restored circuit. Note that loading the LUT with correct $\vec{K}^R$ implies that $\vec{K}_L^D$ is also derived correctly and thus, the entire SFDL *delay key* $\vec{K}^D$=$\{\vec{K}_T^D, \vec{K}_L^D\}$ ensures that there are no timing errors (due to violations of shortest or longest path constraints) in the unlocked circuit. On the other hand, if the LUT is initialized incorrectly or a wrong $\vec{K}_T^D$ is applied, then the SFDL construction ensures that the original circuit behavior is not correctly restored due to functional as well as timing errors.

2.4.2 Security Evaluations of SFDL Scheme

In this section, we present outcomes of experimental evaluations to highlight the effectiveness of our proposed SFDL technique to secure design IPs from an attacker in untrusted foundry. The main objective of these evaluations is to demonstrate the security-corruptibility trade-off resiliency achieved by SFDL which distinguishes it from existing schemes.

2.4.2.1 Experimental Setup

To implement the SFLL-$flex$ portion of SFDL, we performed *stuck-at-fault* based circuit functionality stripping as considered in prior works [86, 87]. First, we randomly selected a wire in the netlist to inject either a *stuck-at-0* or a *stuck-at-1* fault and then used ATALANTA-M ATPG tool to generate a list of input patterns (called *failing patterns*) which detected the injected fault. These failing patterns represent the protected input cubes for the modified circuit. In other words, they constitute the *restore key* $\vec{K}^R$ to be used to populate the LUT for restoring the FSC. As SAT-resiliency is governed by the number of care-bits in this LUT [87], we set the number of care-bits in $\vec{K}^R$ (secret) equal to 5% of the total number of gates in the original netlist and hard-coded the remaining care-bits to their correct logic values (*logic*-0 or *logic*-1). The objective of the attacker is to use a SAT solver to determine these *unknown* care-bits of $\vec{K}^R$.

2.4.2.2 TimingSAT Attack Resiliency

Our proposed SFDL scheme first applies SFLL-$flex$ on a netlist to obtain an FSC, followed by delay locking of the FSC using additional delay key bits. It is to be noted that now the attacker will not succeed to find the *restore key* $\vec{K}^R$ by ignoring delay key-bits (Stage 1 of TimingSAT attack, see section 2.3.2.2) within a practical time limit as determining $\vec{K}^R$ has been demonstrated to be highly resistant against SAT attack [86, 118]. Therefore, the security guarantees provided by SFDL against SAT formulation based attacks is expected to be *at least* as strong as that provided by baseline SFLL-$flex$ as now the attacker not only needs to determine $\vec{K}^R$ but also the *delay key* $\vec{K}^D_T$. In order to assess the robustness of such delay profile based circuit obfuscation scheme against TimingSAT attack, we perform a comparative study between conventional delay locking and SFDL. The outcome of this experimental evaluation across different benchmark netlists is reported in Table 2.4. Note that for conventional delay locking as the number of *functional* key-bits equals the number *delay* key-bits, the total number of unknown variables ($\#vars$) to be determined by the SAT solver is twice the *delay key* size $\|\vec{K}^D\|$, i.e., $\#vars$=2$\|\vec{K}^D\|$, see sub-columns 2 and 3 of Table 2.4. In our experimental setup of SFDL, we considered a c-to-1 MUX based implementation of the *selector function* to obtain $\vec{K}^D_L$ portion of *delay key* from $\vec{K}^R$. In such a setting, $\#vars$ to be determined by a SAT solver is equal to $\|\vec{K}^D\|+\|\vec{K}^R\|-\|\vec{K}^D_L\|$ as $\vec{K}^D_L$ is same as a row entry of LUT which constitutes $\vec{K}^R$. Also, setting $\|\vec{K}^D\|=\|\vec{K}^R\|$, $\#vars$ to be determined for such a locked netlist is lesser than its delay locking counterpart, i.e., $\#vars<2\|\vec{K}^D\|$.

Table 2.4: Delay Locking vs. SFDL: Resiliency against *TimingSAT* attack

netlist	$\|\vec{K}^D\|$	Delay Locking		SFDL		
		#vars	time (mins)	$\|\vec{K}_L^D\|$	#vars	time
*apex*2	31	62	29.68	10	52	*TO*
*c*880	19	38	22.39	10	28	*TO*
*i*4	17	34	4.79	8	26	*TO*
*s*838	20	40	9.39	7	33	*TO*
*s*1488	33	66	142.83	4	62	*TO*
*s*1494	32	64	153.67	4	60	*TO*

However, even then we observed that though conventional delay locked netlists were deobfuscated within a few hours, TimingSAT attack timed-out for netlists locked using SFDL (time-out limit TO=10 hours as [97, 108]), see sub-columns 4 and 7 of Table 2.4. The reasons behind this anomaly are as follows:

(i) In order to mount TimingSAT attack against a netlist locked using conventional delay locking strategy, the attacker utilizes a two-step approach: In Stage 1 the *functional* key-bits are determined using conventional SAT attack. Then, in Stage 2 the attacker fixes the retrieved *functional key*, followed by unrolling the netlist and finally launches TimingSAT attack on the unrolled netlist to ascertain the *delay key* $\vec{K}^D$. It is to be noted that in each stage, the attacker needs to find $\#vars/2$ number of unknown key-bits. Also, we observed that the first stage of this attack can be accomplished in a short time.

Table 2.5: Delay Locking vs. SFDL: Area of tamper-proof memory

netlist	Delay Locking		SFDL		
	#bits	area (μm^2)	#bits	area (μm^2)	% decrease
$apex2$	62	90.52	52	75.92	16.13
$c880$	38	55.48	28	40.88	26.32
$i4$	34	49.64	26	37.96	23.53
$s838$	40	58.40	33	48.18	17.50
$s1488$	66	96.36	62	90.52	6.06
$s1494$	64	93.44	60	87.60	6.25

(ii) On the other hand, the attacker will not succeed to defeat our proposed SFDL scheme using such a two-step attack strategy. This is because determining just $\vec{K}^R$ (ignoring the *delay key*) has been shown to be provably secure against SAT attack [118]. In fact the number of SAT iterations required to find $\vec{K}^R$ is exponential in terms of the key-size and thus, the attacker will fail to complete the first step of the attack within a practical time limit. Moreover, in Stage 2, the attacker needs to apply SAT attack on a much larger unrolled netlist to account for any timing violations arising from a wrong $\vec{K}_T^D$. This will further increase the time required to mount TimingSAT attack on netlists locked using the SFDL scheme.

Overhead analysis: The area overhead introduced to obfuscate a design using SFLL-fault technique mainly due to the restore unit's LUT whose contents are stored in an on-chip tamper-proof memory [86]. To demonstrate the cost-effectiveness of

our proposed SFDL scheme over its delay locking counterpart, we report the area overhead of the tamper-proof memory for both the techniques in Table 2.5. For this comparative study, we consider a Global Foundries $65nm$ LPe technology based tamper-proof memory which occupies $1.46\mu m^2$ per bit [118, 86]. From the table, we can observe that the area overhead of SFDL scheme is significantly lower than conventional delay locking with area reduction ranging from 6.06% to 26.32% across different benchmark circuits. This is because SFDL utilizes a key of smaller length ($<2\|\vec{K}^D\|$) to obfuscate a netlist compared to conventional delay locking.

2.4.2.3 Resiliency against Other Attacks

The proposed SFDL approach also successfully thwarts other recent attacks on logic locking schemes. For example, SFDL technique is robust against SMT attack [21] which uses advanced theory solver-based algorithms to deobfuscate conventional delay locked netlists. This is because breaking a stripped-functionality based locking scheme (such as SFDL) is not only SAT-hard but also a SMT-hard problem [22].

SFDL technique is also secure against state-of-the-art Functional Analysis on Logic Locking (FALL) attack [94]. This is due to the implementation of SFLL-$flex$ portion of SFDL using stuck-at-fault based circuit functionality stripping (also known as SFLL-$fault$) as outlined in [86, 87]. Though FALL attack effectively defeats SFLL-HD scheme, the applicability of similar analyses to find the protected cubes in SFLL-$fault$ is still an open problem [94].

Table 2.6: SFLL-$flex$ vs. SFDL: Comparative study of corruptibility

netlist	SFLL-$flex$	SFDL		
	corruptibility(%)	$\|\vec{K}^D\|$	corruptibility(%)	% increase
$apex2$	0.00	31	2.67	**2.67**
$c880$	0.00	19	6.07	**6.07**
$i4$	0.00	17	16.30	**16.30**
$s838$	0.17	20	13.93	**13.76**
$s1488$	0.00	33	5.20	**5.20**
$s1494$	0.00	32	12.80	**12.80**

2.4.2.4 Output Corruptibility Evaluation

Existing logic locking schemes exhibit an inherent trade-off between robustness to SAT formulation based attacks and output corruptibility [122]. For example, if on one hand a locking approach demonstrates high resiliency to SAT attack (such as Anti-SAT [108]) then on the other hand it exhibits very low output corruptibility for wrong keys. This makes such locked netlists vulnerable to other types of attacks (such as AppSAT [89], removal [119], and bypass [112] attacks) which exploit the low error rates of wrong keys to reconstruct the original netlist. Our proposed SFDL technique overcomes this shortcoming to develop an effective obfuscation solution to protect the IPs of hardware designs. SFDL combines SFLL-$flex$ and delay locking approaches to *simultaneously* achieve strong SAT attack resiliency as well as high output corruptibility.

In case of conventional SFLL-$flex$ scheme (with low number of protected input cubes), an attacker can mount AppSAT attack [89] to find an approximate *functional key* which will retrieve an almost correct netlist (negligible corruptibility). However, it is to be noted that such an approximate attack strategy will not be effective against our proposed SFDL technique as the timing graph of a circuit *differs* due to the application of correct and approximate *functional keys*. This is because of the construction of SFDL scheme where a portion of the *delay key* $\vec{K}_L^D$ is derived from the *functional key* $\vec{K}^R$ using a selector function (see Fig. 2.4). Therefore, an approximate *functional key* $\vec{K}_{app}^R$ will lead to a different $\vec{K}_L^D$ (and hence, a different *delay key*), causing the overall circuit to violate timing constraints. We cannot leverage our proposed two step TimingSAT attack strategy (as described in section 2.3.2.2) to retrieve an approximate netlist due to such dependence of circuit delay profile on *functional key* (not only on *delay key*) in SFDL technique.

We performed an experimental analysis to study the comparative output corruptibility between SFLL-$flex$ and SFDL schemes. In our experiments, we first applied AppSAT attack to find $\vec{K}_{app}^R$ and then used it to initialize the restoration unit's LUT for both the locking configurations. In case of SFDL, the $\vec{K}_L^D$ portion of *delay key* was derived from $\vec{K}_{app}^R$ (using selector function) to drive the L-TDBs while the $\vec{K}_T^D$ portion was selected randomly (as TimingSAT attack is not applicable) to drive the T-TDBs present in the FSC portion. In our experiments, we set the number of TDBs equal to 5% of the total number of gates in a netlist and excited both types of locked netlists (locked using SFLL-$flex$ and SFDL schemes) using $3,000$ randomly generated PIs. As it can be seen from the experimental outcomes

reported in Table 2.6, the corruptibility exhibited by SFLL-$flex$ technique is zero (or almost zero) across different benchmark netlists. This is in accordance with theoretical expectations for such a locked netlist which offers high SAT attack resiliency but low error rates for wrong keys. On the other hand, we observed *significant* output corruptibility ranging from 2.67% to as high as 16.30% for circuits obfuscated using SFDL technique which provide the *same* level of SAT attack resiliency as their SFLL-$flex$ counterparts. This is due to the presence of incorrectly configured TDBs (wrong *delay key*) in the delay-locked FSC which leads to significant shortest/longest path timing violations, thus resulting in faulty circuit output responses.

2.5 Conclusion

In this chapter, we first proposed TimingSAT attack to defeat the security offered by state-of-the-art delay locking countermeasure [110]. TimingSAT attack operates in two stages: in the first stage the attacker finds a correct *functional key* and in the second stage the attacker utilizes timing information embedded unrolled netlist to formulate an iterative SAT attack which retrieves correct *delay key*. The experimental results on different benchmark circuits highlight the effectiveness of the proposed TimingSAT attack to deobfuscate delay locked netlists (for different *delay ratios*) within few hours. Subsequently, we developed a technique called stripped-functionality delay locking (SFDL) which resists not only TimingSAT attack but

also thwarts all known attacks against logic locking. Our proposed SFDL scheme combines the concept of stripped-functionality based logic locking with delay locking to develop an effective IP security solution for hardware designs.

Chapter 3: Side-channel Analysis of Circuit-level Obfuscation

3.1 Introduction

In Section 1.1.3, we presented an overview of several logic obfuscation techniques which have been proposed to defend against IP theft of outsourced ICs by untrusted foundries. Existing attacks on these techniques (e.g., SAT attack, removal/bypass attacks) exploit the weaknesses of logic locking schemes from a theoretical point of view; such attacks do not exploit the potential weaknesses arising from physical implementations of logic obfuscation schemes. It has been demonstrated that unintentional leakage of information in the form of power dissipation, electromagnetic emanation, timing characteristics, etc. from hardware implementations of designs can be analyzed to launch side-channel attacks [71]. These kinds of attacks can break a system using limited computational power and in a short amount of time even though its security robustness has been theoretically established. In this chapter, we explore how an adversary can exploit side-channel information from a functional chip to reverse engineer its internal design obfuscation key. The main contributions of the chapter are as follows:

- We propose a template analysis based side-channel attack which defeats standard logic obfuscation techniques. In this attack approach, the adversary utilizes a low number of side-channel traces to unlock the functionality of an obfuscated circuit in a level-by-level manner following an iterative approach.

- The proposed attack methodology is validated using simulated power traces which are more realistic as it considers the switching activities across all the logic depths of a netlist rather than calculating the Hamming weights only at the output nets as proposed in [117].

The outcomes of this work highlights the need to develop a side-channel attack secure logic encryption scheme which can thwart IP piracy, counterfeiting, and overbuilding of ICs by untrusted foundries.

3.2 Power Analysis Attack on Circuit-level Obfuscation

3.2.1 Side-channel Analysis Attacks

The vulnerability of hardware implementations to power side-channel attacks [71] has been well established in the literature. A side-channel adversary tries correlate the observable side-channel leakage to the secret key dependent internal state of an implementation. In [117], a Differential Power Analysis (DPA) against logic encryption circuits have been proposed. However, such a DPA attack fails to retrieve the correct key for a majority of the benchmark circuits encrypted using the Strong Logic Locking (SLL) scheme [75]. In this work, we explore the applicability

Table 3.1: Comparison of this work with previous attacks against logic locking schemes

Scheme	DAC'12 [75]	DFTS'15 [117]	HOST'15 [97]	HOST'17 [89]	This work
RLL [82]	√	√	√	√	√
SLL [75]	×	*partially*	√	√	√
Anti-SAT [108]	×	×	×	*partially*	√

of a template based side-channel analysis strategy to break various existing logic encryption schemes. Table 3.1 highlights the contribution of our work with previous attacks against standard logic locking techniques to retrieve the secret key.

Template Analysis (TA) attack is a powerful form of side-channel attack where the adversary can retrieve the secret key of a targeted design using a *limited* number of power traces [33]. A conventional TA attack approach against a cryptographic implementation, consists of an profiling or *offline* phase and an attack or *online* phase. The success rate of TA is determined by the manner in which the noise content in the collected power side-channel samples is handled. The TA attack methodology proposed in this work for deobfuscation of a locked netlist assumes an attack environment which is similar to the conventional approach. The power consumption of the chip has been modeled using the standard Hamming distance (HD) power model which considers the switching activities of every gate in the circuit. The proposed attack assumes that the adversary has access to the following:

- An activated chip bought from the market with the secret key in an on-chip tamper-proof memory used to observe correct input/output (I/O) patterns.

- A duplicate chip fabricated using reverse engineered locked netlist of the activated chip but without the tamper-proof memory. The key inputs to the key-gates in the duplicate chip are controlled by the adversary.

The details related to the applicability of the aforementioned attack model in the context of the proposed TA attack are presented in the next section. A key advantage of such a TA attack is that it doesn't need full scan chain access to the activated chip as required by most of the existing works on logic obfuscation. To the best of our knowledge, this is the first work to utilize template based side-channel analysis against logic locking schemes.

3.2.2 Proposed Template Attack Against Logic Locking

In this section, we propose a power side-channel based Template Analysis (TA) attack to deobfuscate netlists which are locked using standard logic encryption schemes. Our proposed attack technique is applicable against existing logic locking schemes such as Random Logic Locking (RLL), Strong Logic Locking (SLL), and *point-function* based logic locking approaches such as Anti-SAT [108]. The primary property of a targeted netlist exploited by the ensuing attack strategy is that various key-gates inserted by existing logic locking algorithms are located at different logic depths and even in the same logic depth, the arrival times of the key-gate inputs are different. In an actual attack setup, the attacker will have access to the detailed layout level information from GDSII file and hence, will be able to ascertain the arrival times of various key-gate inputs. But, in this work to illustrate the

attack methodology, we consider the logic depths of key-gates present in benchmark gate-level netlists as rough estimates of corresponding key-gate inputs' arrival times. To illustrate the steps of the proposed attack, let us consider the MCNC benchmark netlist *apex*2 with 5% key-gate overhead (*apex*2_*enc*05) locked using RLL scheme. The logic depths of key-gates in the aforementioned netlist along with the related key-gate output nets, key-gate operations, and key inputs are presented in Table 3.2. The logic depths of key-gates were obtained by first representing all gates in the netlist as nodes in a Directed Acyclic Graph (DAG), followed by topologically sorting of the nodes and then calculating the length of longest path to every node from any source node, i.e. any node driven by only PIs or key inputs. We follow the convention that the logic gates fed by PIs or key inputs only are at logic depth 1. From Table 3.2, it can be easily observed that the key-gates of the *apex*2_*enc*05 netlist are at varying logic depths, e.g. key-gate with output net *pi*17$*enc* is located at logic depth 1, key gates with output nets *n*72$*enc*, *pi*02$*enc*, and *pi*11$*enc* are all located at logic depth 2, and so on.

3.2.2.1 Attack Methodology

In this subsection, we present the steps of our proposed template based side-channel analysis against conventional logic locking algorithms. The requirements of the ensuing attack strategy is similar to the attack model assumed in [117]. The side-channel attacker can monitor power trace samples corresponding to individual logic depths in an activated chip. This is a practically valid assumption as the

adversary has access to the information related to arrival times of signals at each logic depth from the layout files of the chip. She can subsequently analyze such power samples to deobfuscate the functionality of the activated chip following the steps outlined in Algorithm 3.1. At first, the attacker initializes the sets of determined key bits $\mathbb{K}$ and undetermined key bits $\mathbb{U}$ to ϕ (no key bits have been deciphered) and $\{K_1, K_2, ...K_n\}$ respectively, where an n-bit key ($K_1K_2...K_n$) has been used to lock the netlist. The total number of key-gates/ key-inputs can be easily identified by studying the netlist and hence, the value of n is already known to the attacker. Next, the logic depth L_i, $\forall i \in \{1, 2, ..., n\}$, is ascertained for each of the n key-gates in the netlist. After this step, the adversary tries to unlock the functionality of the circuit level-by-level by concentrating only on key-gates located at a particular logic depth L_i, denoted by the set $\mathbb{T}$ and starting from key-gates at the lowest logic depth.

In order to mount a TA attack on the locked netlist, at first m power side-channel traces are collected by exciting the activated chip with m randomly generated primary inputs (PIs). Then, the adversary considers only the power sample instants which contains maximum information related to the switching activities of a targeted logic depth. We refer to this portion of a power trace time instants (consisting of the power samples of interest) as the *attack window*. For simplicity, we assume a zero-wire delay model and unit propagation delays of logic gates in our experiments. After collection of power samples in the *attack window*, the adversary performs *template matching* between the power samples of the activated chip and corresponding simulated power samples of the duplicate chip for different key guesses at the targeted logic depth using some suitable *thresholding metric*.

Table 3.2: Logic depths of $apex2\_enc05$ netlist key-gates locked using RLL scheme

key-gate output net	key-gate operation	key input	Logic Depth
$n72\$enc$	$xnor(keyinput0, n72)$	$keyinput0$	2
$n107\$enc$	$xnor(keyinput1, n107)$	$keyinput1$	6
$pi02\$enc$	$xnor(keyinput2, pi02\$inv)$	$keyinput2$	2
$pi17\$enc$	$xnor(keyinput3, pi17)$	$keyinput3$	1
$pi11\$enc$	$xor(keyinput4, pi11\$inv)$	$keyinput4$	2
$n113\$enc$	$xnor(keyinput5, n113)$	$keyinput5$	4
$n119\$enc$	$xor(keyinput6, n119)$	$keyinput6$	11
$n167\$enc$	$xor(keyinput7, n167)$	$keyinput7$	3
$n195\$enc$	$xnor(keyinput8, n195)$	$keyinput8$	11
$n216\$enc$	$xnor(keyinput9, n216)$	$keyinput9$	3
$n284\$enc$	$xnor(keyinput10, n284)$	$keyinput10$	9
$n293\$enc$	$xor(keyinput11, n293)$	$keyinput11$	30
$n296\$enc$	$xnor(keyinput12, n296)$	$keyinput12$	36
$n561\$enc$	$xor(keyinput13, n561)$	$keyinput13$	34
$n562\$enc$	$xor(keyinput14, n562)$	$keyinput14$	36
$n563\$enc$	$xor(keyinput15, n563)$	$keyinput15$	38
$n570\$enc$	$xor(keyinput16, n570)$	$keyinput16$	15
$n325\$enc$	$xor(keyinput17, n325)$	$keyinput17$	13
$n332\$enc$	$xor(keyinput18, n332)$	$keyinput18$	6
$n353\$enc$	$xor(keyinput19, n562)$	$keyinput19$	4
$n355\$enc$	$xnor(keyinput20, n355)$	$keyinput20$	8
$n364\$enc$	$xnor(keyinput21, n364)$	$keyinput21$	20
$n366\$enc$	$xor(keyinput22, n366)$	$keyinput22$	23
$n378\$enc$	$xnor(keyinput23, n378)$	$keyinput23$	4
$n423\$enc$	$xnor(keyinput24, n366)$	$keyinput24$	26
$n471\$enc$	$xnor(keyinput25, n471)$	$keyinput25$	3
$n554\$enc$	$xnor(keyinput26, n554)$	$keyinput26$	28
$n581\$enc$	$xor(keyinput27, n581)$	$keyinput27$	7
$n592\$enc$	$xnor(keyinput28, n592)$	$keyinput28$	28
$n605\$enc$	$xor(keyinput29, n605)$	$keyinput29$	5
$n646\$enc$	$xor(keyinput30, n646)$	$keyinput30$	32

For example, to perform TA attack at logic depth 1 of *apex2_enc05* netlist, the adversary first performs the aforementioned *template matching* for *guess* 0 and *guess* 1 of *keyinput3* feeding a key-gate at logic depth 1. After successful determination of *keyinput3*, the *template matching* is performed for the next *attack window* targeting the 3-tuple (*keyinput0*, *keyinput2*, *keyinput4*) and simulating power for all possible combinations of the key tuple (2^3 *guesses*) being targeted. Thus, after processing a particular logic depth the cardinality of the set $\mathbb{U}$ decreases, whereas the cardinality of the set $\mathbb{K}$ increases, till all the key inputs are determined for the key-gate(s) located at $max(L_i)$ logic depth. The TA attack methodology described relies on the *accurate* selection of power samples corresponding to an *attack window*.

3.2.2.2 Template Matching

The objective of the adversary is to determine whether the power dissipation of the activated chip during the *attack window* relates closely to the overall switching activity at that logic depth as determined by guess 0 or by guess 1 of a targeted key input (say *keyinput3*). However, a single power trace of the activated chip might not be sufficient to determine the correct key bit at the targeted logic depth as the noise content in power samples of the activated chip during the *attack window* might relate it closely to the power trace simulated for the wrong key guess, resulting in error. In order to successfully mount the proposed TA attack, we considered m different inputs to the activated chip to obtain a power trace matrix P of dimension mXl where l is the number of power samples in a single trace. For the experiments, we only considered the sample instant containing maximum information about the

Algorithm 3.1: TA attack against a logic encrypted netlist

Input: (i) Activated chip (ii) Reverse engineered netlist/ Duplicate chip

Output: n-bit key used for logic locking

$\mathbb{K}=\phi$ /*set of determined key bits*/

$\mathbb{U}=\{K_1, K_2, ...K_n\}$ /*set of undetermined key bits*/

Get logic depth L_i of i^{th} key gate, $\forall i \in \{1, 2, ..., n\}$

for $i = 1$ *to* $max(L_i)$ **do**

  $\mathbb{T} = \{$key gates at level $L_i\} \subseteq \mathbb{U}$

  $\mathbb{P} = \{$power traces of activated chip for m PIs$\}$

  Perform TA attack for $2^{|T|}$ key guesses using $\mathbb{P}$

  $\mathbb{K}=K \cup T$

  $\mathbb{U}=U \setminus T$

end

return $\mathbb{K}$

switching activities in an *attack window* to get a vector s of length m from the power matrix P. For each input, we also obtained the HD values corresponding to the *attack window* due to key-bit guesses 0 and 1, resulting in two more vectors of length m, denoted by σ_0 and σ_1 respectively. The adversary needs to apply a *thresholding operation* considering the vectors s, σ_0, and σ_1 to determine the which set of simulated power traces corresponds more closely to the power traces of the activated chip in the *attack window*. We consider ℓ_1-norm $\left(||\cdot||_1\right)$ as the metric to implement such a *thresholding operation* to determine the targeted key bit. First, we define two vectors δ^0 and δ^1, each of length m, corresponding to key bit guesses 0 and 1 respectively as follows:

$$\delta^0 = s - \sigma_0, \quad \delta^1 = s - \sigma_1 \tag{3.1}$$

A correct guess of the targeted key bit will have a lower ℓ_1-norm value, i.e., a lower magnitude of the sum of absolute values of the δ vector elements, compared to the ℓ_1-norm value for a wrong key bit guess. Therefore, the correct key bit can be retrieved using the following equation:

$$\textit{correct key bit} = \min\{||\delta^0||_1, ||\delta^1||_1\} \tag{3.2}$$

In scenarios where multiple key-gates are present at a targeted logic depth, then more than two δ vectors are necessary for the proposed TA attack based deobfuscation scheme. In fact, if k key bits are located at a particular logic depth, then 2^k number of δ vectors are required to determine all the key inputs in the *attack window*. For example, in RLL locked *apex*2_*enc*05 netlist, two key-gates are present at logic depth 6 corresponding to key inputs *keyinput*1 and *keyinput*18. For retrieving both the key bits, we consider four δ vectors $\delta^{00}_{depth6}, \delta^{01}_{depth6}, \delta^{10}_{depth6}$, and δ^{11}_{depth6} which

correspond to the key 2-tuple ($keyinput1$,$keyinput18$) guesses $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ respectively. Out of the four possible key 2-tuples, only one correspond to the correct key input pair in the *attack window* corresponding to logic depth 6. The application of the aforementioned *thresholding operation* on benchmark netlists locked using standard logic encryption schemes led to successful retrieval of the secret key.

3.2.3 Experimental Results

In this section, we report the experimental results of the proposed Template analysis (TA) attack on MCNC benchmark circuits encrypted using standard locking techniques. The power traces for the experiments were obtained using a similar approach as proposed in [117]. However, we considered the Hamming distance (HD) of the switching activities in the internal nets as well as in the output nets of the circuit and superimposed Additive white Gaussian noise (AWGN) with Signal-to-noise ratio 30 dB to generate more realistic power consumption signatures of the activated chip.

3.2.3.1 TA attack against Random Logic Locking

In Table 3.2, we reported the logic depths of different key-gates inserted in the $apex2\_enc05$ obfuscated using the RLL scheme. As outlined in section 3.2.2.1, we first target the key-gates at logic depth 1 and after successful retrieval of the targeted key bits, we attack the key-gates at higher logic depths. In Fig. 3.1, we

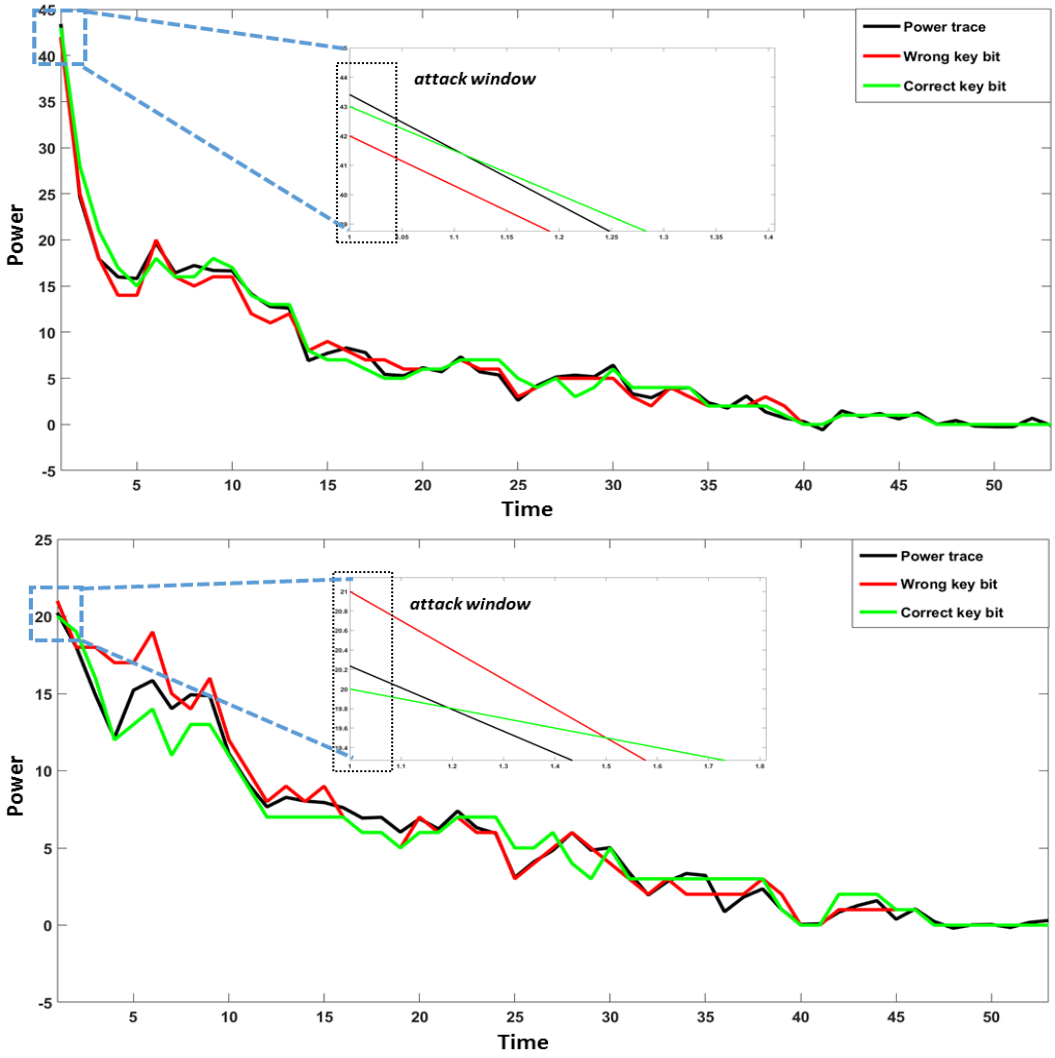


Figure 3.1: Power profiles for two random input vectors applied to $apex2\_enc05$ netlist targeting gate operation $xnor(keyinput3, pi17)$ at logic depth 1 to recover $keyinput3$.

plot the power traces of the activated chip (*black*), simulated power traces for correct key guess (*green*) and wrong key guess (*red*) at logic depth 1 due to applications of two random input vectors to the circuit. The power traces corresponding to both the correct and wrong key guesses were obtained using the standard Hamming distance (HD) power model considering the switching activities in all the nets of the circuit. The targeted key-gate operation is $xnor(keyinput3, pi17)$ and the *attack window* corresponds to the power dissipation of the activated chip due to switching activities of all gates located at logic depth 1.

For the $apex2\_enc05$ netlist, we applied 5 random input vectors to the circuit, i.e. $m = 5$, to obtain the following δ vectors (as outlined in previous section) for the *attack window* targeting logic depth 1:

$$\delta^0_{depth1} = [+0.02, +0.05, +0.21, -0.07, +0.16] \tag{3.3}$$

$$\delta^1_{depth1} = [+1.02, -0.95, -0.78, +0.92, -0.83] \tag{3.4}$$

The values of ℓ_1-norms of above δ vectors are as follows:

$$||\delta^0_{depth1}||_1 = 0.53, \quad ||\delta^1_{depth1}||_1 = 4.52 \tag{3.5}$$

The value of $||\delta^0_{depth1}||_1$ is 0.53 which corresponds to the correct key bit ($keyinput3 = 0$) in the activated chip, while the value of $||\delta^1_{depth1}||_1$ is 4.52 for the wrong key bit guess. Thus, the ℓ_1-norm is significantly lower for the correct key bit guess compared to the wrong bit guess. However, as stated previously, a single input vector might lead to erroneous determination of a targeted key bit due to the effect of noise in the power sample. Therefore, the adversary should consider larger values of m to

increase the confidence of successful key bit retrieval. For RLL locked $apex2\_enc05$ netlist, $m = 5$ was seen to be sufficient for various runs of circuit simulations with different randomly generated input vectors and targeting various logic depths.

Following a similar approach, we targeted $keyinput24$ (correct value in activated chip being 1) driving a key-gate at logic depth 26 of the RLL locked $apex2\_enc05$ netlist as shown in Table 3.2. In Fig. 3.2, we plot the power traces of the activated chip (*black*), simulated power traces for correct key guess (*green*) and wrong key guess (*red*) at logic depth 26 due to applications of two random input vectors to the circuit. It is to be noted that when the adversary targets key-gates in n^{th} logic depth, then all the key inputs to key-gates at lower logic depths have already been retrieved. In other words, the *attack window* marks the boundary between the determined and undetermined sets of key-gate inputs in the netlist. Therefore, while targeting $keyinput24$ at logic depth 26, we assume all the key inputs to key-gates at any prior logic depth have been successfully determined starting from the key-gates at the very first logic depth. In this case also, we set $m = 5$ to obtain the following δ vectors:

$$\delta^0_{depth26} = [+0.78, +0.81, +1.04, +1.17, +0.99] \tag{3.6}$$

$$\delta^1_{depth26} = [-0.22, -0.19, +0.05, +0.17, -0.01] \tag{3.7}$$

The ℓ_1-norms of above δ vectors are as follows:

$$||\delta^0_{depth26}||_1 = 4.80, \quad ||\delta^1_{depth26}||_1 = 0.64 \tag{3.8}$$

Like the previous case, the lower value of ℓ_1-norm (0.64) corresponds to the correct key bit ($keyinput24 = 1$) of the activated chip, while higher value of ℓ_1-norm (4.80) corresponds to the wrong key bit guess.

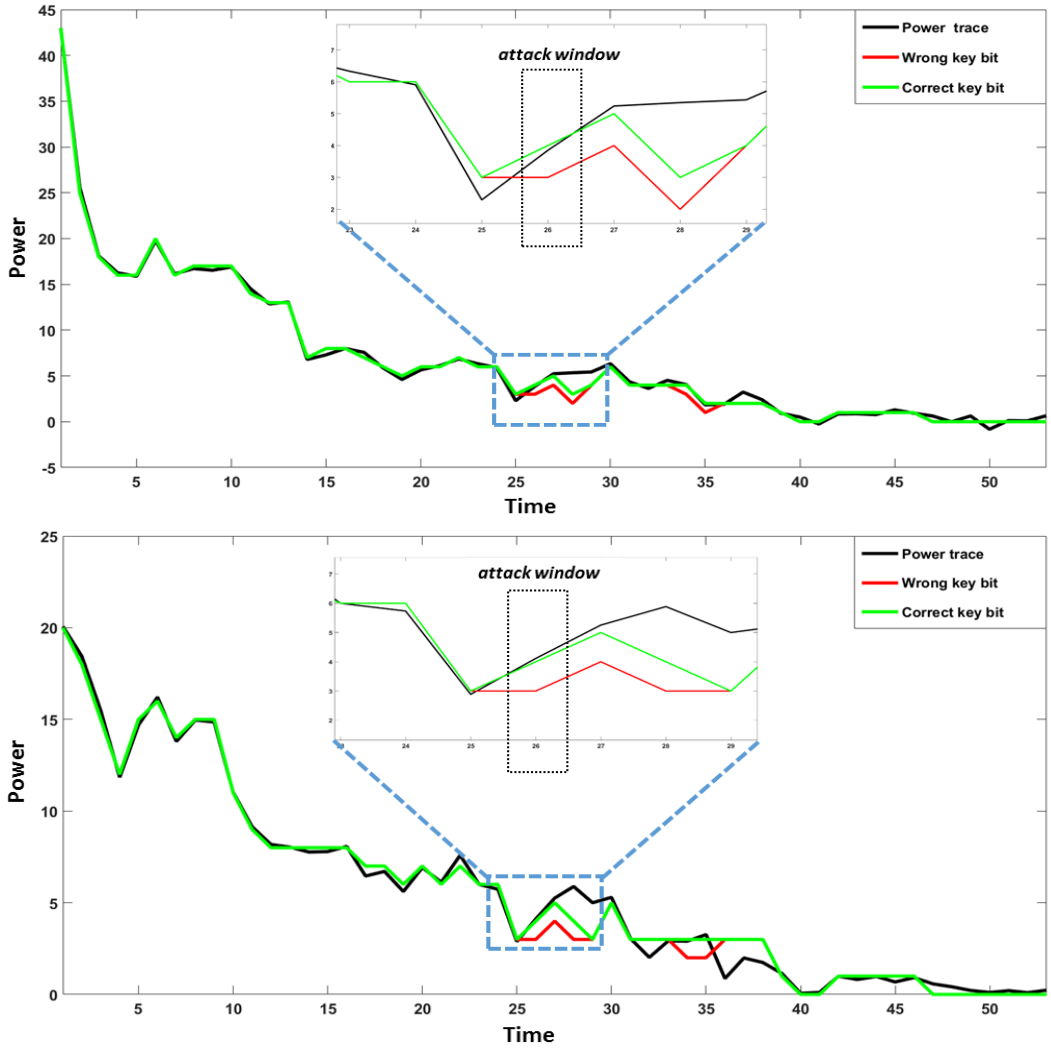


Figure 3.2: Power profiles for two random input vectors applied to $apex2\_enc05$ netlist targeting gate operation $xnor(keyinput24, n366)$ at logic depth 26 to recover $keyinput24$.

Multiple key-gates at same logic depth: As outlined in the previous section, if k key-gates are present in a logic depth, then we need to consider 2^k number of δ vectors are required to determine all the key inputs in the *attack window*. For example, to retrieve key inputs $keyinput1$ and $keyinput18$ at logic depth 6 of RLL locked $apex2\_enc05$ netlist, we consider four δ vectors δ^{00}_{depth6}, δ^{01}_{depth6}, δ^{10}_{depth6}, and δ^{11}_{depth6} which correspond to the key 2-tuple ($keyinput1$,$keyinput18$) guesses $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ respectively. Considering $m = 5$, we get the following δ vectors:

$$\delta^{00}_{depth6} = [+1.20, +1.09, -1.21, +0.93, -1.14] \tag{3.9}$$

$$\delta^{01}_{depth6} = [+0.20, -0.09, -0.21, -0.07, -0.14] \tag{3.10}$$

$$\delta^{10}_{depth6} = [+0.20, -0.09, -2.21, -0.07, -2.14] \tag{3.11}$$

$$\delta^{11}_{depth6} = [-0.80, -0.91, -1.21, -1.07, -1.14] \tag{3.12}$$

The values of ℓ_1-norms of above δ vectors are as follows:

$$||\delta^{00}_{depth6}||_1 = 5.58, \quad ||\delta^{01}_{depth6}||_1 = 0.72$$
$$||\delta^{10}_{depth6}||_1 = 4.72, \quad ||\delta^{11}_{depth6}||_1 = 5.13 \tag{3.13}$$

Of these ℓ_1-norms, the magnitude of $||\delta^{01}_{depth6}||_1$ is minimum and corresponds to the correct key 2-tuple ($keyinput1$,$keyinput18$)=(0,1) for the activated chip. Following this procedure, multiple key inputs can be successfully determined for a particular *attack window* when more than one key-gates are present at the targeted logic depth. As evident, higher number of power traces corresponding to different inputs (higher values of m) should be considered if the magnitudes of ℓ_1-norms for the correct and wrong key bit guesses are separated by *narrow* margins.

Table 3.3: Logic depths of $apex2\_enc05$ netlist key-gates locked using SLL scheme

key-gate output net	key-gate operation	key input	Logic Depth
$n105\$enc$	$xor(keyinput0, n105)$	$keyinput0$	6
$n118\$enc$	$xnor(keyinput1, n118)$	$keyinput1$	9
$n128\$enc$	$xnor(keyinput2, n128)$	$keyinput2$	11
$n119\$enc$	$xor(keyinput3, n119)$	$keyinput3$	11
$n127\$enc$	$xnor(keyinput4, n127)$	$keyinput4$	9
$pi16\$enc$	$xnor(keyinput5, pi16\$inv)$	$keyinput5$	2
$pi27\$enc$	$xnor(keyinput6, pi27\$inv)$	$keyinput6$	2
$n56\$enc$	$xor(keyinput7, n56)$	$keyinput7$	4
$n67\$enc$	$xor(keyinput8, n67)$	$keyinput8$	4
$pi23\$enc$	$xor(keyinput9, pi23)$	$keyinput9$	1
$pi28\$enc$	$xor(keyinput10, pi28\$inv)$	$keyinput10$	2
$n63\$enc$	$xnor(keyinput11, n63)$	$keyinput11$	3
$n68\$enc$	$xor(keyinput12, n68)$	$keyinput12$	4
$n76\$enc$	$xnor(keyinput13, n76)$	$keyinput13$	2
$n77\$enc$	$xnor(keyinput14, n77)$	$keyinput14$	4
$n104\$enc$	$xor(keyinput15, n104)$	$keyinput15$	4
$n110\$enc$	$xor(keyinput16, n110)$	$keyinput16$	12
$n111\$enc$	$xnor(keyinput17, n111)$	$keyinput17$	14
$n147\$enc$	$xor(keyinput18, n147)$	$keyinput18$	3
$n208\$enc$	$xor(keyinput19, n208)$	$keyinput19$	5
$n201\$enc$	$xnor(keyinput20, n201)$	$keyinput20$	28
$n209\$enc$	$xnor(keyinput21, n209)$	$keyinput21$	7
$n202\$enc$	$xnor(keyinput22, n202)$	$keyinput22$	30
$n135\$enc$	$xor(keyinput23, n135)$	$keyinput23$	5
$n136\$enc$	$xor(keyinput24, n136)$	$keyinput24$	7
$n261\$enc$	$xor(keyinput25, n261)$	$keyinput25$	3
$n120\$enc$	$xor(keyinput26, n120)$	$keyinput26$	13
$n294\$enc$	$xnor(keyinput27, n294)$	$keyinput27$	30
$n295\$enc$	$xor(keyinput28, n295)$	$keyinput28$	32
$n95\$enc$	$xnor(keyinput29, n95)$	$keyinput29$	5
$n124\$enc$	$xnor(keyinput30, n124)$	$keyinput30$	5

3.2.3.2 TA attack against other schemes

The crux of our proposed TA attack is the exploitation of the power side-channel information level-by-level to unlock the functionality of a chip. The presence of the key-gates at various logic depths of the locked netlist enable the adversary to determine the key inputs in an iterative manner by analyzing the power samples of the activated chip for different inputs in an *attack window*. In literature, other logic locking schemes such as *Strong Locking Locking* (SLL) [75], and Anti-SAT block [108] have also been proposed. In [117], a DPA attack strategy against netlists locked using RLL and SLL schemes has been outlined. Though the proposed DPA attack retrieves a significant portion of the key for the RLL approach, the attack could only resolve more than 50% of the key bits in only 25% of the benchmark circuits considered. The main drawback of the DPA technique is *key aliasing* effect which hinders the determination of correct key and becomes even more dominant with an increase in ratio of key inputs to primary inputs in a logic cone. The TA attack proposed in this work does not suffer from any such shortcomings as it exploits the power sample instants pertaining to only an *attack window* and recovers the key bits level-by-level.

(1) SLL scheme: The logic depths of various key-gates in *apex*2_*enc*05 netlist encrypted using SLL scheme is shown in Table 3.3. It can be easily observed from the table that the key-gates are located at various logic depths in this scheme also and hence, the circuit susceptible to the proposed TA attack based deobfuscation technique as outlined in section 3.2.2. We targeted *keyinput*9 (correct value in

activated chip being 0) driving a key-gate at logic depth 1 of the and set $m = 5$ to get the following δ vectors for $apex2\_enc05$ benchmark netlist (locked using SLL scheme):

$$\delta^0_{depth1} = [+0.09, +0.02, +0.10, +0.56, +0.06] \tag{3.14}$$

$$\delta^1_{depth1} = [+1.09, -0.98, -0.90, +1.56, -0.94] \tag{3.15}$$

The ℓ_1-norms of above δ vectors are as follows:

$$||\delta^0_{depth1}||_1 = 0.84, \quad ||\delta^1_{depth1}||_1 = 5.47 \tag{3.16}$$

As in previous case, the lower value of ℓ_1-norm (0.84) corresponds to the correct key bit ($keyinput9 = 0$) of the activated chip, while higher value of ℓ_1-norm (5.47) corresponds to the wrong key bit guess. *Similar distributed patterns of key-gate locations and successful ℓ_1-norm based key classifications were observed for other MCNC and ISCAS85 benchmark netlists (locked using RLL and SLL schemes) as well.* Thus, though SLL based locking is highly resistant to the DPA attack outlined in [117], such an obfuscation technique is still equally vulnerable to our proposed TA attack like the RLL scheme.

(2) Anti-SAT scheme: The Anti-SAT block based logic locking technique [108] comprises of two complementary *point-function* blocks and generates a wrong output by flipping the output of the circuit for few inputs for every possible key combinations, except for the correct key. Unlike the previous schemes, we observed that that several key-gates are located at same logic depths when a netlist is locked using Anti-SAT approach, and hence, applying the proposed TA attack based on logic depth would require an impractical number of power trace templates due to

Table 3.4: Arrival times at logic depth 1 of $c432\_OA4$ netlist

key-gate input	Arrival time	key-gate input	Arrival time
$keyinput10$	1.0091	$keyinput14$	0.9777
$keyinput11$	0.8558	$keyinput15$	0.9696
$keyinput12$	0.8731	$keyinput16$	0.8592
$keyinput13$	1.0482	$keyinput17$	1.1301

a large number of key guess combinations. For example, in Anti-SAT benchmark netlist $c432\_OA4$, 8 key-gates are present at logic depth 1 (see Table 3.5) However, as stated in section 3.2.2.1, for a real chip the differences in arrival times of input signals of gates located in the same logic depth will result in *distinct time windows* of their switching activities, resulting in low number of key guess combinations to be considered in an *attack window*. Let us consider an unit propagation delay of logic gates with 10% variation in input arrival times (due to wire delays) for the $c432\_OA4$ netlist. In order to target the 8 key-gates located at logic depth 1, we generated random numbers from a Normal distribution with mean 1 and standard deviation 0.1, and obtained key input arrival times as shown in Table 3.4. Assuming that an adversary is capable of collecting power traces with a resolution of 0.06 time units, we set the *attack window* to the time range of $(0.97, 1.03)$ units [denoted by $AW(0.97, 1.03)$] to retrieve $keyinput10$ value. In this time window, the power traces correspond to switching activities of key-gates with inputs $keyinput10$ (arrival time: 1.0091), $keyinput14$ (arrival time: 0.9777), and other non key-gates with arrival times within the range considered. We set $m = 5$ to obtain the following arrival time parameter based δ vectors for Anti-SAT locking scheme in the *attack window*

Table 3.5: Logic depths of $c432\_OA4$ netlist key-gates locked using Anti-SAT scheme

key-gate output net	key-gate operation	key input	Logic Depth
$G199gat\$enc$	$xor(keyinput0, AntiSATXOR)$	$keyinput0$	9
$G203gat\$enc$	$xnor(keyinput1, G203gat)$	$keyinput1$	11
$G296gat\$enc$	$xnor(keyinput2, G296gat)$	$keyinput2$	15
$G348gat\$enc$	$xor(keyinput3, G348gat)$	$keyinput3$	19
$G355gat\$enc$	$xnor(keyinput4, G355gat)$	$keyinput4$	19
$G356gat\$enc$	$xnor(keyinput5, G356gat)$	$keyinput5$	19
$G357gat\$enc$	$xnor(keyinput6, G357gat)$	$keyinput6$	21
$G381gat\$enc$	$xor(keyinput7, G381gat)$	$keyinput7$	25
$G386gat\$enc$	$xor(keyinput8, G386gat)$	$keyinput8$	25
$G393gat\$enc$	$xor(keyinput9, G393gat)$	$keyinput9$	25
$G1gat\$enc1$	$xnor(keyinput10, G1gat)$	$keyinput10$	1
$G56gat\$enc1$	$xor(keyinput11, G56gat)$	$keyinput11$	1
$G17gat\$enc1$	$xnor(keyinput12, G17gat)$	$keyinput12$	1
$G47gat\$enc1$	$xnor(keyinput13, G47gat)$	$keyinput13$	1
$G1gat\$enc2$	$xnor(keyinput14, G1gat)$	$keyinput14$	1
$G56gat\$enc2$	$xnor(keyinput15, G56gat)$	$keyinput15$	1
$G17gat\$enc2$	$xnor(keyinput16, G17gat)$	$keyinput16$	1
$G47gat\$enc2$	$xnor(keyinput17, G47gat)$	$keyinput17$	1
$gt2_1\$func$	$xnor(keyinput18, gt2_1)$	$keyinput18$	6
$gf2_1\$func$	$xnor(keyinput19, gf2_1)$	$keyinput19$	6
$gt1_2\$func$	$xnor(keyinput20, gt1_2\$struct)$	$keyinput20$	4
$gf1_2\$func$	$xor(keyinput21, gf1_2\$struct)$	$keyinput21$	4
$gt1_2\$struct$	$mux(keyinput22, G27gat, gt1_2)$	$keyinput22$	3
$gt1_1\$struct$	$mux(keyinput23, gt1_1, G146gat)$	$keyinput23$	3
$gf1_2\$struct$	$mux(keyinput24, G196gat, gf1_2)$	$keyinput24$	3
$gf1_1\$struct$	$mux(keyinput25, gf1_1, G34gat)$	$keyinput25$	3

$AW(0.97, 1.03)$, corresponding to the key 2-tuple ($keyinput10$,$keyinput14$):

$$\delta^{00}_{AW(0.97,1.03)} = [+0.94, +0.83, +0.90, -0.96, -0.80] \quad (3.17)$$

$$\delta^{01}_{AW(0.97,1.03)} = [-0.06, -0.17, +1.90, -1.96, +0.20] \quad (3.18)$$

$$\delta^{10}_{AW(0.97,1.03)} = [-0.06, -0.17, -0.09, +0.04, +0.20] \quad (3.19)$$

$$\delta^{11}_{AW(0.97,1.03)} = [-1.06, -1.17, +0.90, -0.96, +1.20] \quad (3.20)$$

The values of ℓ_1-norms of above δ vectors are as follows:

$$||\delta^{00}_{AW(0.97,1.03)}||_1 = 4.43, \quad ||\delta^{01}_{AW(0.97,1.03)}||_1 = 4.29$$

$$||\delta^{10}_{AW(0.97,1.03)}||_1 = 0.57, \quad ||\delta^{11}_{AW(0.97,1.03)}||_1 = 5.29 \quad (3.21)$$

Among the ℓ_1-norms of δ vectors considered, the magnitude of $||\delta^{10}_{AW(0.97,1.03)}||_1$ is minimum (0.57) and it corresponds to the correct key 2-tuple ($keyinput10$, $keyinput14$) = (1, 0) for the activated chip. Similar results were obtained for other Anti-SAT locked benchmark circuits as well.

3.3 Conclusion

In this chapter, we presented a template analysis based side-channel attack technique to deobfuscate the functionality of an IC locked using standard logic obfuscation schemes. The results of our experiments reveal that RLL, SLL, and Anti-SAT based locking schemes are all vulnerable to the proposed template analysis attack with a limited number of power side-channel traces per *attack window*. Our attack strategy exploits the fact that different key-gates of a locked netlist are located at different depths with respect to the arrival times of input signals, and

thus enabling the side-channel adversary to unlock the functionality of the circuit in a level-by-level manner. In practice, if the effect of noise is more significant in the collected power traces, then the adversary may need to consider more sophisticated metrics (such as Pearson's correlation coefficient) for successful template matching.

Chapter 4: Architecture-level Obfuscation

4.1 Introduction

Till date, combinational hardware obfuscation strategies have been mostly proposed at the circuit-level to thwart IC supply chain attacks. The strong security guarantees provided by such circuit-level logic locking schemes do not necessarily ensure acceptable IP security at the architecture-level of design abstraction. To the best of our knowledge, there is no available study which analyzes the resiliency of any obfuscated processor architecture (locked using standard circuit-level logic obfuscation techniques) to existing IC supply chain attacks. In this chapter, we first outline a SAT formulation based attack methodology [97, 89] against an obfuscated many-core Graphics Processing Unit (GPU) netlist to highlight the limitations of circuit-level locking approaches for protecting architecture-level IP. Unlike conventional attacks on logic locking schemes, an adversary can launch our proposed attack *without any activated hardware* and hence, this attack poses a major threat in the supply chain of such processor designs. Subsequently, we propose a couple of efficient architecture-level countermeasures which are highly resilient to such attacks. The main contributions of the chapter are as follows:

- We outline a technique to mount a SAT formulation based attack against a GPU core's locked pipelined netlist to approximately deobfuscate its functionality without any activated chip requirements. Our experiments on a real GPU testbed using NVIDIA's SASSIFI framework reveal that more than 95% of the application runs on such an approximately unlocked GPU result in correct outcomes with 95% confidence-level and 5% confidence-interval.

- To counter the proposed attack, we develop a countermeasure called cache locking which locks the cache block replacement policy in a GPU for wrong cache-key. This results in significant performance degradation of applications as evident from our experimental results, thus making the GPU hardware inefficient for fast application execution.

- We also develop a hardware/software co-design based obfuscation framework to provably safeguard the IP of hardware accelerator designs against SAT as well as removal/bypass type of attacks while still maintaining high output corruptability for applications.

4.2 Background

Modern GPU architectures have been developed to efficiently exploit the data-level parallelism in applications ranging from real-time 3D visualization to high-performance scientific computing. In this work, though we focus on widely used NVIDIA GPUs [10] to outline our proposed attack and defense strategies, the analyses are equally applicable to other processor architectures as well.

4.2.1 Overview of GPU architecture

In a NVIDIA GPU, blocks of threads are executed in Streaming Multiprocessors (SMs), which primarily consists of groups of streaming processors (SPs) or CUDA cores. The unit of execution flow in the SM is a collection of 32 threads, called *warp*. The threads in a warp follow the Single Instruction Multiple Thread (SIMT) mode, i.e., they execute the same instruction sequence but with different data. SPs are the primary computing elements of GPUs and corresponds to cores that perform scalar calculations. An SM, in addition to SPs, consists of other different types of functional modules such as load/store (LD/ST) units, Special Functional Unit (SFU), on-chip memory (instruction cache, configurable shared memory/ L1 cache, register files) and instruction control units (dispatcher, scheduler). In Fig. 4.1, we present a structural overview of an NVIDIA GPU architecture. In this work, we propose an attack against a locked GPU netlist to retrieve an *approx-key* and study the application-level impact of error propagation (due to the use of *approx-key*) utilizing NVIDIA's SASSIFI framework [44] on a real GPU.

4.2.2 Instrumentation of GPGPU applications

In this work, we focus on general-purpose GPU (GPGPU) applications based on the widely adopted NVIDIA's Compute Unified Device Architecture (CUDA) framework [9]. The CUDA programming framework adopts the SIMT model in hierarchies consisting of kernels, blocks, and threads. The CPU spawns the multi-threaded kernels onto the GPU, which subsequently allocates the blocks of threads to

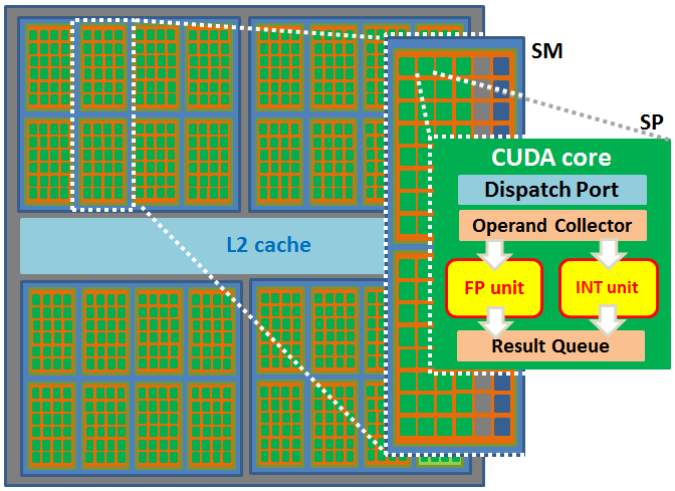


Figure 4.1: Block diagram of an NVIDIA GPU architecture

available SMs using internal schedulers. The parallel programs written in high-level language such as CUDA is compiled by a front-end compiler (NVIDIA's NVVM) to generate intermediate code in a virtual ISA called parallel thread execution (PTX). PTX abstracts the GPU as a data-parallel computing platform, but the PTX code does not run directly on the GPU. Another backend compiler optimizes and translates the PTX instruction in native machine code by either using ahead-of-time compilation of compute kernels via PTX assembler (ptxas) or using just-in-time compiler in the display driver to compile PTX representation of kernel available in binary format. In this work, we used NVIDIA's SASSIFI framework [44] to study the application-level error impact in a real GPU due the use of learned *approx-key* to unlock its core functionalities. The SASSI-based Fault Injector (SASSIFI) framework utilizes ahead-of-time backend compilation as the SASSI instrumentation is embedded in ptxas. SASSI is implemented as the final compiler pass in ptxas and uses *nvlink* to link instrumented applications with instrumentation handlers. The

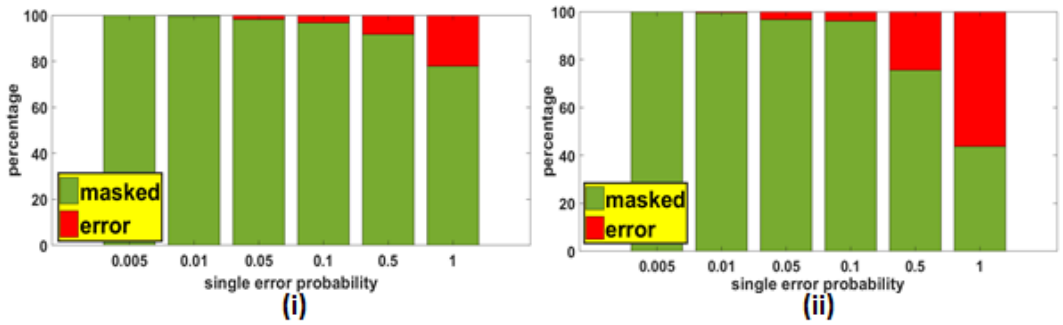


Figure 4.2: Application outcome vs. probability of single inst. being faulty for (i) datapath errors and (ii) controlpath errors in core

SASSI based application instrumentation requires two things to be specified: (i) *where* to insert instrumentation and (ii) *what* information to extract from each instrumentation site.

4.3 Proposed Attack on Obfuscated GPU

4.3.1 Obfuscation of GPU cores

In the inset of Fig. 4.1, we outline the structure of an NVIDIA GPU architecture's SP module which primarily consists of *inorder* integer and floating point pipelines. The SPs or CUDA cores are the most abundant computational elements in a GPU and are primarily responsible for its high throughput performance. Hence, as a natural choice, we assume that the designer inserts key-gates in the gate-level netlist between various SP pipeline stages to lock the overall functionality of the GPU, following the steps of any standard combinational logic locking approach [82, 75, 108, 115]. We also consider that all the SP modules in the GPU are locked using a *single key* so that the layouts of the cores are identical, thus having optimal

fabrication cost. Moreover, the use of separate keys for different cores will lead to an impractical key size as the number of cores in modern GPUs can be very large (for example NVIDIA's GeForce GTX Titan Z consists of 5760 CUDA cores).

In a logic obfuscated SP pipelined netlist, wrong inputs to key-gates will result in errors in outcomes of threads which utilize such *faulty* key-gates for their computations. Depending on locations of *faulty* key-gates and data, such errors will have varying impacts on multithreaded kernel executions in SIMT mode as follows:

(i) Datapath error: A *wrong* key bit input to a key-gate located in the datapath of pipeline will have an error propagation effect only in the fan-out cone of the *faulty* key gate. In other words, such a fault will have *thread localized* effects in computations, i.e., impacting only the threads which execute on that erroneous datapath.

(ii) Controlpath error: In SIMT mode, the decoder module of a SM decodes the opcode for all the active threads in a warp and individual threads execute the *same* decoded operations on different SPs or cores but with *different* data operands. Hence, a *wrong* key bit input to a key-gate located in the decoder module or controlpath will have an error propagation effect in the datapaths of *all the active threads* in a warp. Hence, controlpath errors will have *warp wide* effects in computations.

We consider a simple multithreaded *sum* application to study the effect of datapath and controlpath errors on an actual application-level output using NVIDIA's SASSIFI framework (more details in section 4.3.3). Based on the *key* inputs chosen, we will have different probabilities of errors occurring in instructions. In Fig. 4.2,

we illustrate the percentage of the application outcomes (out of several runs) being faulty (red) or correct/masked (green) due to a single *randomly selected* instruction being executed faulty (with different probabilities) for datapath and controlpath errors. From the plots it is evident that even error in a single thread due to a *wrong* key input may lead to faulty application outcomes. However, we observed that the *difference* in the number of faulty outcomes for datapath and controlpath errors is significant when the probability of an instruction being faulty is high, where as the *difference* becomes quite negligible with a decrease in the probability. We observe this effect on the experimental results for benchmark applications also as detailed in section 4.3.3. Hence, a smartly selected *approx-key* which injects very small error in instructions can indeed result in very accurate application outcomes despite not unlocking the hardware in its entirety.

As NVIDIA GPU's SP pipeline architecture/netlist details are proprietary, we instead consider a locked netlist of standard MIPS 5-stage inorder pipeline as substitute of the GPU netlist to perform our experimental analyses. We obfuscated the the control and data paths of MIPS pipeline using state-of-the-art Anti-SAT based logic locking scheme [108] integrated with Strong Logic Locking [76].

4.3.2 Attack on locked GPU cores

4.3.2.1 Attack Model

In *conventional* SAT attack [97], in addition to the netlist, full scan-chain access to an activated hardware is also required to deobfuscate a locked hardware. This is because the formulation of an iterative SAT attack utilizes the input-output truth tables of each of the locked modules. However, this is a *very strong* assumption as such privilege is *not available* in practice. First, when the untrusted foundry is trying to unlock a chip, the actual activated chip may not have been marketed yet. Second, even if they are in possession of the unlocked chip, the attacker needs to have full scan chain access into the internal combination modules. The designer who wishes to secure his design may just disable on-chip test structures before marketing the unlocked chip. **In our attack model, we allow the adversary to only possess a locked netlist, and do not grant her privileges to have full scan chain access to internal pipeline latches of SP modules in an activated chip.**

4.3.2.2 SAT formulation based attack

The primary challenge to deobfuscate the functionality of a locked SP netlist using *conventional* SAT attack approach [97] is the lack of knowledge of internal pipeline latch contents as per our attack model. In this section, we demonstrate how an adversary *can still successfully devise an iterative SAT formulation based attack* to effectively learn the *key* without an activated GPU hardware. The crux of

the ensuing attack strategy is the observation that the internal pipeline latches are *only responsible for performance speed-up* by dividing the long latency single-cycle datapath into low latency multi-cycle pipelined datapath. These pipeline registers play *no role* in determining the overall functionality of the pipelined netlist. Hence, for the sake of analyzing the functionality of the locked SP module, the adversary can model an equivalent netlist by transforming the multi-cycle pipelined datapath to a single-cycle datapath design. This equivalent netlist can be constructed easily by logically removing the pipeline latches and then simply connecting the input wires to corresponding output wires of the removed latches as shown in figure 4.3. The outcome of this transformation is the conversion of a locked sequential SP netlist to a *functionally equivalent locked combinational SP netlist*, which we analyze next using an iterative SAT formulation. Before we outline the details of our attack methodology, we define the following terminologies for convenience:

(i) f_{lock}**:** Functionally equivalent locked combinational SP netlist **(ii)** PI**:** Primary input to the locked SP netlist, consisting of opcode contents, source and destination register addresses, etc. as obtained from instruction binary (details later)

(iii) PO**:** Primary output of the locked SP netlist, consisting of destination register contents (for R-type or I-type MIPS instructions) or jump/branch address (for J-type MIPS instructions). From the locked netlist, the functional relationship among PI, key-gate inputs (K) and PO, i.e., $PO = f_{lock}(PI, K)$ is known to the attacker.

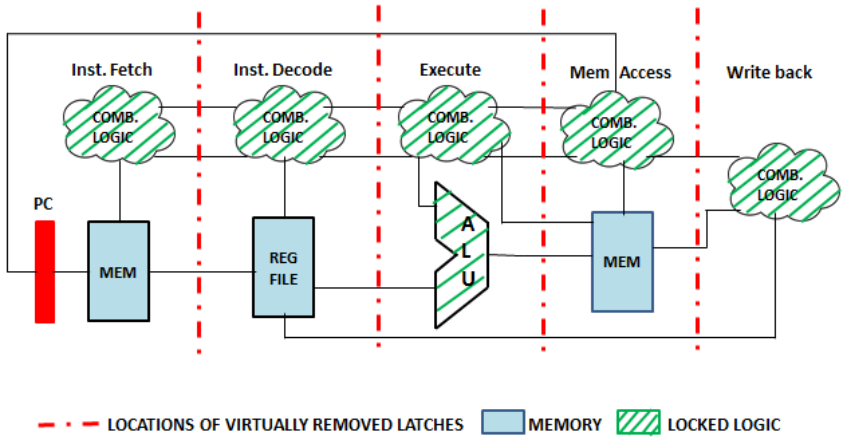


Figure 4.3: Multi-cycle to single cycle datapath transformation of locked pipelined netlist

The primary difference between *conventional* IC obfuscation and the obfuscation of a GPU core netlist is that the *correct* $PI - PO$ pairs are not known in the former case *without* an activated chip, whereas in the later case, the attacker can deduce the *correct* $PI - PO$ pairs for a SP netlist as explained below:

- The PI corresponding to each instruction is obtained by relating the human readable assembly instructions to binary information of the assembled application. For example, in case of NVIDIA GPUs, it is possible to successfully extract the PTX or SASS from a *cubin* or *executable* using the *cuobjdump tool* in CUDA Toolkit [9]. In addition, the attacker can utilize the NVIDIA's Nsight Visual Studio Edition to correlate between lines of CUDA C, PTX, and SASS [11]. Therefore, using the publicly available instruction set architecture (ISA), the adversary can determine PI for every instruction.

- Again, PO for each instruction is obtained from the corresponding PI and the ISA information because the PO depends on the result of operation (*known* from ISA) carried out on the source register contents (*known* from PI).

Therefore, the adversary has *prior knowledge* of *correct* $PI - PO$ pairs for every instruction of a compiled application being executed on a GPU core. For example, let us consider a simple assembly-level program fragment executed by a thread on a SP:

$$\ldots$$

$$\text{I1: ADD } R1,R2,\ R3 \qquad //R1{=}R2 + R3$$

$$\text{I2: ADD } R4,R1,\ R3 \qquad //R4{=}R1 + R3$$

$$\text{I3: MUL } R5,R2,\ R4 \qquad //R5{=}R2 * R4$$

$$\text{I4: SUB } R3,R5,\ R4 \qquad //R3{=}R5 - R4$$

$$\ldots$$

Let us suppose that the initial contents (prior to instruction I1 execution) of registers $R1$,$R2$,$R3$,$R4$, and $R5$ are $1, 2, 3, 4$ and 5 respectively. In instruction I1, the contents of registers $R2$ and $R3$ are added and written to register $R1$. Hence, using the PI information the adversary can easily calculate the expected value at the destination register, i.e., PO:[$R1$]=[$R2$]+[$R3$]=2+3=5. Now that the *correct* $PI - PO$ pair is known for each instruction, it is equivalent to having in possession an unlocked chip. Hence, SAT formulation based attack strategies [97, 89] can be utilized which use this information to iteratively identify distinguishing input-output

(DI) pairs. As noted in [97], each DI pair eliminates a subset of unique *wrong keys* for that SAT iteration, till we converge to the *correct key*. We can write an iterative SAT formulation for locked SP netlist as follows:

$$\begin{aligned}\mathbf{F_i} :=& \mathbf{C}(\tilde{\mathbf{PI}}, \tilde{\mathbf{K}}_\mathbf{1}, \tilde{\mathbf{PO}}_\mathbf{1}) \wedge \mathbf{C}(\tilde{\mathbf{PI}}, \tilde{\mathbf{K}}_\mathbf{2}, \tilde{\mathbf{PO}}_\mathbf{2}) \wedge (\tilde{\mathbf{PO}}_\mathbf{1} \neq \tilde{\mathbf{PO}}_\mathbf{2}) \\ & (\bigwedge_{\mathbf{j=1}}^{\mathbf{j=i-1}} \mathbf{C}(\tilde{\mathbf{PI}}^\mathbf{d}_\mathbf{j}, \tilde{\mathbf{K}}_\mathbf{1}, \tilde{\mathbf{PO}}^\mathbf{d}_\mathbf{j})) \wedge (\bigwedge_{\mathbf{j=1}}^{\mathbf{j=i-1}} \mathbf{C}(\tilde{\mathbf{PI}}^\mathbf{d}_\mathbf{j}, \tilde{\mathbf{K}}_\mathbf{2}, \tilde{\mathbf{PO}}^\mathbf{d}_\mathbf{j}))\end{aligned} \tag{4.1}$$

where, $\mathbf{F_i}$ denotes the i^{th} SAT iteration formulation, $\mathbf{C}(\tilde{\mathbf{PI}}, \tilde{\mathbf{K}}, \tilde{\mathbf{PO}})$ is the SAT formula for a locked circuit and $(\tilde{\mathbf{PI}}^\mathbf{d}_{\{\mathbf{1}\ldots\mathbf{i-1}\}}, \tilde{\mathbf{PO}}^\mathbf{d}_{\{\mathbf{1}\ldots\mathbf{i-1}\}})$ are the distinguishing input-output pairs that are found in previous $i-1$ iterations. Following such an attack strategy, if the adversary finds the *correct key* used to lock the original synthesized SP netlist, then all such PO responses for different instructions will be consistent with corresponding *correct* PO responses. In context of the aforementioned program fragment, the *correct key* will result in the contents of registers $R1, R2, R3, R4, R5$ being updated with values $5, 2, 8, 8, 16$ respectively just after the execution of instruction I4. To make the process of finding new distinguishing input-output pair *more efficient*, the adversary may develop customized microbenchmark applications consisting of a targeted set of operations carried out by the instructions.

To counter the feasibility of such a SAT attack, *point-function* based obfuscation approaches like Anti-SAT [108] and SARLock [115] have been proposed. Though the *point-function* based obfuscation scheme makes the SAT solving time exponential to obtain the correct key, a recent technique called the AppSAT attack [89] can retrieve an *approx key* to unlock the functionality of such a locked netlist for

almost all the primary inputs. In section 4.3.3, we present the results of the AppSAT attack against an Anti-SAT block based obfuscated netlist of MIPS pipelined design, which we considered as a functional substitute of the GPU core's netlist.

4.3.3 Experimental Results: AppSAT attack

The datapath and controlpath of the MIPS pipelined netlist were obfuscated using Anti-SAT scheme [108] integrated with Strong Logic Encryption (SLE) [75]. We used 5% XOR/XNOR key-gate overhead for locking the original synthesized netlist using SLE, and used additional key-gate inputs for obfuscation with Anti-SAT block, total key-size being 364 bits. Subsequently, we launched the AppSAT attack on the functionally equivalent locked single-cycle netlist (as outlined in section 4.3.2.2) with following parameters: a total of $5,000$ iterations of the SAT attack was performed, and at each iteration $10,000$ randomly generated patterns were queried to estimate the error rate $\mathcal{E}$, storing the distinguishing input/output pairs as constraints for successive iterations. In figure 4.4, we show the decreasing trend of $\mathcal{E}$ with the progress of SAT attack iterations. The *approx key* returned by the AppSAT attack consists of inputs to functional key-gates (inserted using SLE scheme) and key inputs for Anti-SAT block. We observed that there was an exact match between the portions of the original key and the *approx key* that correspond to the functional key inputs, while there were mismatches in the portions of keys that correspond to

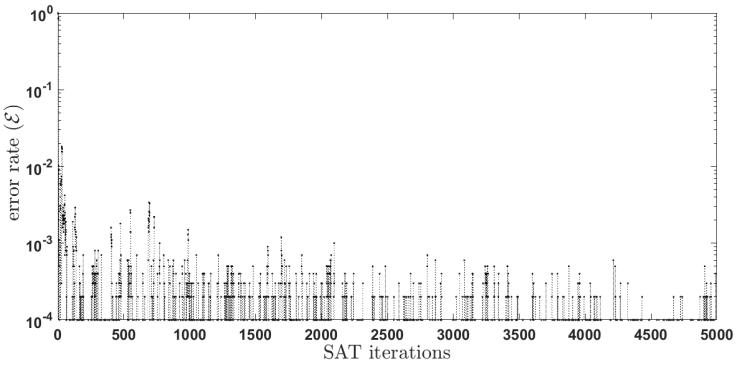


Figure 4.4: Error rate ($\mathcal{E}$) vs SAT attack iterations

the Anti-SAT block key-gates. However, as the output corruptibility of Anti-SAT block is *very low*, it has very limited effect on overall functionality retrieved by the *approx key*.

4.3.3.1 Experimental framework

We utilized the NVIDIA's SASSIFI framework [44] to capture errors on the application-level manifested due to the use of the retrieved *approx key* used to approximately unlock the inorder pipelined core netlist. As the SASSIFI tool injects errors in the architectural state, the outcomes of the injections are not dependent on specific GPU used, provided the binary file is not modified. We used NVIDIA's Maxwell architecture based GeForce GTX950M, CUDA 6.5 toolkit, and display driver version 352.63 for our experiments. We used 5 applications from Rodinia benchmark suite (version 2.3) [34] which include diverse workloads.

4.3.3.2 Error probability of faulty instructions

We applied the AppSAT attack as outlined in subsection 4.3.3 to deduce an *approx-key* for unlocking the pipelined cores of a GPU after converting the pipelined design to a functionally equivalent single-cycle one. For every application, we first noted the number of GPU assembly-level instructions that write to a General Purpose Register (GPR), $N = \#inst_{GPR}$. In order to estimate the number of GPR type instructions which are faulty for a benchmark application, we simulated the approximately unlocked SP core netlist with rN number of inputs. These inputs had the same opcode set as the benchmark instructions thereby capturing the spirit of the application instruction mix. Note that each benchmark has a different number and combination of instructions. Hence the error rate of each benchmark when executed on an approximately unlocked core could be different. For our experiments, we set $r = 20$ which resulted in simulating around hundreds of millions of inputs to the approximately unlocked core. Even for such a large test case, we found *no error* in *PO* values when compared with the correct responses for each instruction that we simulated. This illustrates the effectiveness of approximate unlocking the GPU chip using our proposed technique which does not require an unlocked chip. While our AppSAT based approximately unlocked GPU caused no measurable errors, we still wish to analyze the worst case scenario where 1 out of rN instructions are faulty to capture the *application-level* impact of the retrieved *approx-key*, though in practice the error probability will be even lower. We assumed that the number of faulty GPR instructions executed due to the use of *approx-key* follows Binomial distribution with

Table 4.1: Application-level impact of **datapath errors** due to *approx-key*

Application	Outcomes of injections (percentage)			
	Masked	DUEs	Pot. DUEs	SDCs
BFS	95.57	1.82	0.00	2.60
gaussian	99.47	0.52	0.00	0.00
hotspot	97.92	0.00	0.78	1.30
nw	96.09	1.30	0.00	2.60
pathfinder	95.57	2.34	0.00	1.82

error probability $\mathcal{E}_{AK} = 1/rN$ (because we assume that 1 out of rN instructions is faulty). Therefore, the probability that k number of GPR instructions are executed faulty can be expressed as follows:

$$P(X = k) = \binom{N}{k} \mathcal{E}_{AK}^{k} (1 - \mathcal{E}_{AK})^{N-k} \tag{4.2}$$

As $\mathcal{E}_{AK} << 1$ and $N >> 1$, we get $P(X = 1) \simeq N\mathcal{E}_{AK} = 1/r = 0.05$. It is to be noted that the probability that multiple GPR instructions will be executed faulty , i.e., $P(X >= 2)$, due to the use of *approx-key* is practically negligible. It is to be noted that in an actual scenario, the error propagation effect due to an *approx-key* will be restricted to only a few number of *low probability netlist paths*, and hence, the expected application-level error impact is even lesser. In our experiments, we *randomly* selected a GPR type instruction, to estimate effect of error propagation for different error injection sites.

Table 4.2: Application-level impact of **controlpath errors** due to *approx-key*

Application	Outcomes of injections (percentage)			
	Masked	DUEs	Pot. DUEs	SDCs
BFS	96.09	1.82	0.00	2.08
gaussian	98.41	0.52	0.00	1.04
hotspot	97.92	0.00	0.78	1.30
nw	95.31	1.82	0.00	2.86
pathfinder	95.83	1.30	0.26	2.60

4.3.3.3 Error impact on benchmark applications

Based on the analysis in previous subsection, we only considered the scenario where a single GPR type instruction is faulty with a probability of $p = 0.05$ for our experiments. For studying the application-level impact of errors due to the use of *approx-key* in data path and control path key-gates of the core netlist, we considered these cases separately. We used the SASSIFI framework to run error injections on 5 Rodinia benchmark applications [34] in Instruction Output Value (IOV) mode. In IOV mode, SASSIFI uses instrumentation handlers to inject errors into the destination register values of an instruction after they are executed. We performed 384 error injection runs for each of our application workloads so that the injection results have maximum error bars of 5% at 95% confidence level. In each error injection run, we *randomly* selected a dynamic instruction among all the GPR type instructions and either (i) *randomly* updated the destination register value of

a thread for studying the impact of a datapath error or (ii) *randomly* updated all the destination register values of all the threads in a warp for studying the impact of a controlpath error. The results of the injections were categorized [44] as follows:

(i) Masked: No error symptom detected and the application output with fault injection run is same as the original *error free* output.

(ii) DUEs: The application terminated with a non-zero exit status or application runtime crossed the *timeout threshold*.

(iii) Potential DUEs: Symptoms of unsuccessful kernel execution (detected by comparison of kernel exit status with *cudaSuccess*), explicit application error messages can be found in *stderr/stdout*.

(iv) SDCs: Application execution terminates without any crashes, hangs, or failure symptoms but output file/*stdout* is different compared to fault-free run.

In Tables 4.1 and 4.2, we report the results of such injection runs on the benchmark applications for datapath and controlpath errors. As evident from the statistics of the resulting outcomes, *almost all* of the injected errors are *masked* (95% or more depending on applications), implying that the *approx-key* is good enough to deobfuscate the functionalities of the locked SP or core pipelines such that there is very low effect of gate-level error propagation impact at the application-level. As highlighted in Fig. 4.2 earlier, the *difference* in erroneous application outcomes is negligible for datapath and controlpath errors due to a very low probability of an instruction being faulty.

4.4 Cache Locking Countermeasure

4.4.1 Basic Idea

The crux of the our proposed attack against the locked SP netlist is the iterative elimination of *wrong keys* based on evaluation of new DI pairs which satisfy the SAT formulation. The primary motivation behind our proposed cache locking countermeasure is that a *wrong cache-key* will result in slowdown of the GPU hardware even though it exhibits correct functionality, and thus being resistant to SAT formulation based attacks (including AppSAT [89]). The cache block replacement *protocols* of modern many-core GPUs are proprietary and hence, not known to an untrusted foundry. For example, several research related to performance analysis of NVIDIA GPUs has led to the conclusion that the cache block replacement protocols is neither of the standard ones commonly studied [65]. Therefore, we analyzed the effect of obfuscating *cache block replacement policy* to lock the overall performance of the GPU for wrong *cache-key* guesses. It is to be noted that locking the cache block replacement policy does not alter the expected (PI, PO) pairs corresponding to the SP units as the overall GPU still performs the correct functionality with an *approx-key* for the cores (retrieved using our proposed attack), but the overall application performance will suffer *significantly* due to drops in cache hit rates with a wrong *cache-key*. This is due to the fact that a wrong *cache-key* will lead to higher number of data fetch requests from slow off-chip memory which require a *significantly* large number of additional clock cycles [65].

In this work, to demonstrate our countermeasure, we assume that the cache replacement policy in place is *least recently used* (LRU). However, the proposed cache locking scheme can be applied to any other replacement policies as well and will continue to be immune to SAT type attacks. In order to lock the cache block replacement policy, we considered a hardware implementation of standard clock algorithm [95] which approximates LRU policy by augmenting an extra clock bit to a cache block to keep track of whether or not a block was accessed recently. If the i^{th} cache block was recently accessed the corresponding clock bit ($clock\_bit(i)$) is set to 1, whereas, on the other hand $clock\_bit(i)$ is reset to 0 if the block wasn't accessed recently. The cache blocks are assumed to be arranged as a circular queue with a current pointer or "clock hand" which cycles through this queue on every memory access. If the clock hand is currently pointing to $clock\_bit(i) = 1$, then as it moves to the next cache block the $clock\_bit(i)$ is reset to 0. The status of clock bit of i^{th} cache block is updated in a periodic manner as follows:

- On a **cache hit**, the $clock\_bit(i)$ is set to 1.
- On a **cache miss**, the clock hand moves to next available i^{th} block with $clock\_bit(i)$ set to 0 and replaces it by a data block fetched from lower memory hierarchy, followed by setting of $clock\_bit(i)$ to 1 to designate the recently written cache block.

To implement the Cache Locking scheme, we modified the aforementioned standard clock algorithm to a ***cache-key* dependent** block replacement policy. As per the modification, the i^{th} cache block will have an associated clock bit

($clock\_bit(i, K_i)$, $K_i \in \{0, 1\}$), which is set to K_i if it was recently accessed, whereas it is reset to $\overline{K_i}$ if the block wasn't accessed recently. Now if the input key matches the correct key then this approach is basically equivalent to the aforementioned LRU policy. However, if the input key bit for i^{th} cache block mismatches it's corresponding actual key bit, we invert this policy of setting and resetting $clock\_bit(i, K_i)$. This would end up scrambling the designation of the least recently accessed status for those cache blocks with wrong key-bit inputs. For every wrong key-bit (K_i) guess there will be either of the two *faulty scenarios* for the i^{th} cache block: **(i)** instead of the i^{th} cache block, some other j^{th} cache block will be replaced from the cache whose associated clock bit $clock\_bit(j, K_j)$ is set to $\overline{K_j}$ **(ii)** instead of replacing some other cache block with $clock\_bit(j, K_j)$ is set to $\overline{K_j}$, the i^{th} cache block is replaced. Therefore, this will result in drops of cache hit rates as such *faulty* cache block replacements will not be suitable for applications utilizing the cache locality principles. To have a practically reasonably *key-size*, the designer can also associate a single key-bit to multiple cache blocks.

This entire cache locking scheme is simple enough to be implemented in a lookup table (LUT) which can be configured after fabrication at test time by the designer. Hence, the attacker cannot simply remove the proposed countermeasure implementation since she is not aware of the locking mechanism as well as the original cache block replacement policy. In section 4.4.2, we present the experimental results highlighting the slowdown of applications due to cache locking countermeasure.

Table 4.3: Benchmark apps slowdown due to Cache Locking (CL) with $\delta_{hit\ rate}$=0.5, $\alpha_{mem} = 0.5$, and $\#penalty$=500

Application	$\#inst(*10^6)$	$\#mem(\%)$	Runtime (secs)		*slowdown*
			$t_{original}$	t_{CL}	
BFS	424.2	12	9.37	19.43	2.07
gaussian	246.3	5	0.80	3.30	4.13
hotspot	440.1	7	13.44	19.58	1.46
nw	123.2	32	0.79	8.55	10.82
pathfinder	436.9	18	4.23	19.54	4.62

Table 4.4: BFS slowdown due to Cache Locking (CL) vs. $\delta_{hit\ rate}$ with penalty cycles ($\#penalty$)=500 and $\alpha_{mem} = 0.5$

$\delta_{hit\ rate}$	Runtime (secs)		*slowdown*
	$t_{original}$	t_{CL}	
0.3	9.37	15.33	1.64
0.4	9.37	17.41	1.86
0.5	9.37	19.49	2.08
0.6	9.37	21.35	2.28
0.7	9.37	23.35	2.49
0.8	9.37	25.37	2.71

Table 4.5: BFS slowdown due to Cache Locking (CL) vs. penalty cycles (*#penalty*) with $\delta_{hit\ rate} = 0.5$ and $\alpha_{mem} = 0.5$

#penalty	Runtime (secs)		*slowdown*
	$t_{original}$	t_{CL}	
400	9.37	17.34	1.85
450	9.37	18.35	1.96
500	9.37	19.43	2.07
550	9.37	20.50	2.19
600	9.37	21.40	2.28

4.4.2 Experimental results: Cache Locking

We denote the drop in cache hit rate ($\delta_{hit\ rate}$) due to wrong *cache-key* guess as: $\delta_{hit\ rate} = hr_{original} - hr_{faulty}$, where, $hr_{original}$ and hr_{faulty} denote the cache hit rates corresponding to the designer's intended block replacement policy and attacker's faulty block replacement policy respectively. As an outcome of such a faulty policy, the application will incur additional clock cycles (*#add cycles*) which is estimated as follows:

$$\#add\ cycles = \delta_{hit\ rate} * \alpha_{mem} * (\#mem) * (\#penalty) \tag{4.3}$$

where, *#mem* denotes the number of memory access instructions (load or store) in the GPU assembly-level, *#penalty* is the number of penalty cycles to access slower off-chip memories, and α_{mem} corresponds to fraction of memory access instructions executed in parallel across multiple GPU cores. The value of the parameter α_{mem} will

depend not only on the application workloads but also on the number of GPU cores as well as on the thread scheduling policies. To evaluate the effect of cache misses on an actual NVIDIA GPU, we modified the CUDA codes of the applications to introduce *#add cycles* number of sleep cycles in the device. In Table 4.3, we report the relative slowdowns of various benchmark applications for a wrong *cache-key* setting parameters $\delta_{hit\ rate} = 0.5$, $\alpha_{mem} = 0.5$, and $\#penalty = 500$. It can be observed that the proposed Cache Locking countermeasure results in slowdowns ranging from factors of 1.46 to as high as 10.82 depending on applications. In Table 4.4, we report the variations in slowdowns of BFS application due to wrong *cache-key* with $\delta_{hit\ rate}$. The results from this table show that even if the attacker guesses a significant fraction of the *cache-key* correctly, resulting in small $\delta_{hit\ rate}$ (say 0.3), then also there is a notable slowdown (by a factor of 1.64) of the BFS application. In Table 4.5, we report the variations in slowdowns of BFS application due to wrong *cache-key* with *#penalty* (ranging over $400-600$ clock cycles) in order to capture the impact of cache locking scheme across different GPU architectures (with different off-chip memory configurations [65, 9]). As evident from the trends in the experimental results (see Table 4.3), the impact of the proposed countermeasure will become *even more prominent* for practical applications having large number of memory references ($\#mem$), thus defeating the *efficient utilization* of the GPU for high performance computing purposes for wrong *cache-key*.

4.5 Hardware-Software Co-Design Based Accelerator Obfuscation

The ever increasing demand for computational power by compute intensive applications such as speech recognition, computer vision, natural language processing, search ranking and other DNN applications have made architectural innovations crucial to achieve high performance and energy efficiency. GPUs as well as several domain specific hardware accelerators such as Google's Tensor Processing Unit (TPU) [50] and Diannao [37] have been developed which provide higher throughput while consuming much lower energy compared to general purpose processors. In this section, we propose a Hardware/software co-design based Accelerator Obfuscation (HSCAO) approach which renders an hardware accelerator design completely useless (unlike cache locking which only degrades performance) for running any application without proper activation by the designer post-fabrication. Also we would like to highlight that HSCAO scheme is applicable to any accelerator architecture, including the ones which do not have cache memory. In this work we use the TPU framework for the purpose of illustration, however our ideas are equally applicable to other accelerator designs as well.

Google TPU: In Google TPU framework [50], the host CPU sends instructions over PCIe bus to an instruction buffer for the TPU to execute rather than fetching them itself. The main computational component called the *Matrix Multiply Unit (MMU)* consists of 256X256 MACs which performs 8-bit multiply-and-adds on signed/unsigned integers. The inputs to the MMU are provided by *weight FIFO*

and *unified buffer* (UB) components. The MMU outputs 16-bit products which are collected in the *accumulator unit*, which are then passed on to the *activation unit*. Finally, the results are written back to UB. A DMA controller transfers data between the CPU host memory (HM) and UB.

4.5.1 Threat Model

We consider that an attacker in the *untrusted foundry setting* has access to the following three components for analysis:

- An *activated* hardware accelerator chip bought from the open market, used to obtain the *correct* input-output responses.
- The gate-level netlist of the hardware accelerator chip reverse-engineered from layout level details available in GDSII file.
- The hardware accelerator's software development kit (SDK) and details of its instruction set architecture (ISA).

It is to be noted that availability of the first two components have been assumed in several related works [75, 76, 82, 97, 108]. In addition, we also consider that the adversary has access to the SDK of the hardware accelerator as well as its ISA. This is a reasonable assumption due to the following reasons:

(i) The SDKs of most commercially available hardware accelerators are freely available for download (e.g., Nvidia's CUDA SDK [9]). An attacker can use such an SDK to generate executable corresponding to some developed microbenchmark

application. This enables her to observe *correct* input-output response pairs from the activated chip by mapping the instruction binaries to the corresponding register contents [31].

(ii) The operation of any processor or accelerator hardware is guided by its ISA. In this work, we assume a hardware accelerator design which is based on an open-source ISA standard (i.e., instruction formats and opcodes are *known*). While some conventional ISAs have been proprietary, recent works have touted the benefits of making the ISA open source [19]. The industry would benefit by making the ISA free as it will enable affordable processor designs to expand the IoT framework. Moreover, open-source ISA doesn't imply that commercial proprietary processor designs cannot use such an ISA. This is due to the fact that the though the ISA is standardized, the chip designer decides the micro-architectural features to be implemented as well as the logical and physical design approaches [12]. For example, Intel 64 processors [7], Codix-Bk3 [3] use open ISAs.

4.5.1.1 Root of Trust

Conventional locking approaches [82, 75, 118] rely exclusively on hardware keys to obfuscate a design. However, most hardware accelerators comprise of *proprietary* SDKs [9] without which these accelerator chips cannot be used. These SDKs represent substantial software development efforts and are developed by the design house (generally not exposed to the untrusted fab). The details of an SDK implementation can be easily hidden from users/fabs using software obfuscation

techniques [38]. For example, DexGuard tool [4] provides state-of-the-art software obfuscation features to protect an SDK against reverse-engineering. As per our threat model, the attacker can only use such an SDK as a *black box* without having access to its internal details. It is quite reasonable to assume that the attacker doesn't have the capability to develop a substitute SDK utilizing the GDSII file of a chip. This is because several architecture-level design specifications/protocols of accelerator designs are not publicly available [6, 9, 1].

4.5.2 Proposed HSCAO Framework

An overview of our proposed HSCAO framework is presented in Fig. 4.5. HSCAO relies on partitioning the obfuscation/deobfuscation task between the accelerator's SDK and the hardware. HSCAO framework consists of following three components:

- **Key sequencer:** It generates a pseudo-random key sequence using a secret key K_{seed} as initial seed value.
- **Software-level Obfuscation:** The *control bits* of instructions consisting of opcode and flag bits are locked by proprietary SDK and then communicated to the accelerator hardware design.
- **Hardware-level Deobfuscation:** Subsequently, the *control bits* are unlocked on-chip using a hardware-level deobfuscation module before further processing the instructions in other modules.

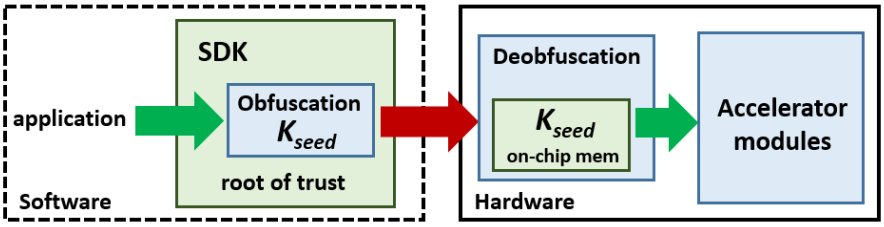


Figure 4.5: Hardware-software Co-design based obfuscation

The overall approach is to share the obfuscation/deobfuscation processes between the software and hardware components of an accelerator. The software portion obfuscates the instructions with *dynamic* keys generated using the key sequencer algorithm. The hardware portion replicates the key sequencer on-chip and is fully synchronized with its software counterpart to deobfuscate the locked instructions. The secret key K_{seed} is shared by the hardware and software counterparts of HSCAO framework: it resides in the SDK (root of trust) and in an on-chip tamper-proof memory (TPM). It is to be noted that such a hardware-software co-design based obfuscation approach aims to protect the design IP of an accelerator, not the information content of an user application running on it. Next, we present the details of different components in HSCAO framework.

4.5.2.1 Key sequencer

The key sequencer utilizes the secret key K_{seed} to generate a pseudo-random sequence of keys K_{seq} for locking/unlocking of instructions in software/hardware counterparts. According to the threat model considered (see section 4.5.1), the attacker has knowledge of the accelerator's instruction format: we consider that an instruction consists of n bits of opcode and control flags, referred to as *control*

bits. The remaining bits of the instruction consists of data handling and memory access related information, referred to as *non-control bits*. We lock the functionality of overall hardware accelerator by only obfuscating the *control bits* of instructions in an application. The *control bits* of the i^{th} instruction is locked by XORing it bit-wise with n-bits of K_i which is the i^{th} key in K_{seq}. The software/ hardware counterparts initializes their key sequencer implementations with the **same** K_{seed}, thus generating identical K_{seq} for locking/unlocking instructions.

Now, we describe the process of generating the pseudo-random key sequence K_{seq} from the secret key K_{seed}. In our design we use N cyclic shift registers (each n bits in length) as shown in figure 4.6. Both N and n are design parameters. These N shift registers are initialized with K_{seed} of size $n \times N$ bits. Figure 4.6 illustrates the state of the key sequencer for generating the first key K_1 in the sequence from the secret key K_{seed}. The m^{th} bit, $m \in \{1, 2, \ldots, n\}$, of K_1 (denoted by $K_{1,m}$) is obtained by XORing the m^{th} bits of all the N shift registers. For generating the next key K_2 in the sequence, all the shift registers are cyclically shifted by certain number of bits as specified in a *shift vector* $\vec{S} = [s_1, s_2, \ldots, s_N]$ where s_j, $j \in \{1, 2, \ldots, N\}$, corresponds to the number of bits the j^{th} register is to be shifted. Now, as before, the contents of all the registers are bit-wise XORed together to generate the key K_2. This process is repeated to generate the subsequent keys in K_{seq}. It is to be noted that the *shift vector* is randomly generated at run-time in the accelerator SDK for software-level obfuscation of an instruction, and its contents are

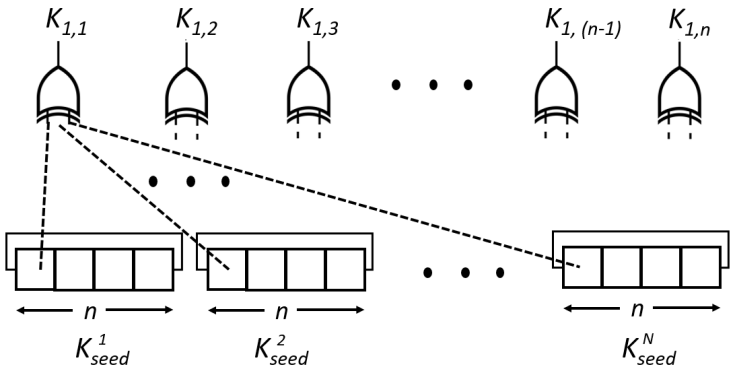


Figure 4.6: Cyclic shift register based key sequencer

not known to an attacker. For generating the same K_{seq}, not only K_{seed} must be common, but also these *shift vectors* needs to be shared between the software and hardware counterparts in the HSCAO framework.

4.5.2.2 Software-level Obfuscation

The accelerator's SDK (root of trust) generates K_{seq} by implementing the above key sequencer algorithm in software. Each key in K_{seq} is XORed bit-wise with the *control bits* of an instruction to obfuscate it. Thus, the application binary is locked in software as a function of secret key K_{seed} and *shift vectors* (dynamically generated per instruction). The SDK also locks the *shift vector* $\vec{S}$ to generate *locked shift vector* $\vec{S'} = [s'_1, s'_2, \ldots, s'_N]$ for every instruction, where the mapping from $\vec{S}$ to $\vec{S'}$ is obtained using a *secret* look-up table (*shift LUT*). This *shift LUT* is not available to an attacker who uses the SDK as a *black-box*. Note that, like K_{seed}, the contents of *shift LUT* is also shared between the software and hardware counterparts of HSCAO framework. For generating the same K_{seq} in hardware, the *locked shift vectors* are communicated to the chip using following ISA extensions:

- **INITK**: Initiate key instruction resets the N registers (each of length n bits) with the $n \times N$ bits of K_{seed}.

- **CSHFT**: Cyclic shift instruction shifts the contents of N registers as per the corresponding elements in *shift vector* $\vec{S} = [s_1, s_2, \ldots, s_N]$ only if the *shift LUT* is correctly configured on-chip, else a faulty mapping from $\vec{S'}$ to $\vec{S}$ will result in wrong operations. The CSHFT instruction format is as follows:

$$\text{CSHFT} \quad [s'_1, s'_2, \ldots, s'_N]$$

  The CSHFT instruction consists of an array of length N, where each element s'_i corresponds to a random number between $-r_{max}$ and $+r_{max}$.

In section 4.5.3, we show that an adversary will be practically unable to reconstruct the key sequence K_{seq} without the knowledge of K_{seed} and *shift LUT* contents. Thus in effect, the software component of HSCOA successfully locks the functionality of accelerator design.

4.5.2.3 Hardware-level Deobfuscation

The hardware-level deobfuscation module serves as a counterpart of the software-level obfuscation module. It replicates the key sequencer design on-chip to unlock the *control bits* of software-obfuscated instructions using the *same* secret key K_{seed} and *shift LUT* information (by bit-wise XORing keys with the locked control bits). On encountering an INITK instruction from the software interface, the hardware key sequencer design resets the cyclic shift registers with K_{seed} content. If a CSHFT

instruction is encountered, first the content of the *shift vector* $\vec{S}$ is retrieved from the *locked shift vector* $\vec{S'}$ (using *shift LUT* stored in on-chip TPM) and then, all N registers are cyclically shifted according to $\vec{S}$. This process allows perfect synchronization between the key sequencers in software and hardware counterparts of the HSCOA framework. Therefore, both the modules generate the *same* K_{seq} for performing instruction obfuscation/deobfuscation operations. As highlighted in figure 4.5, the above deobfuscation process is carried out on-chip before performing any application-specific computations in accelerator modules. Note that as both K_{seed} and *shift LUT* are configured by the designer post-fabrication in on-chip TPM, these are not known to an untrusted foundry.

4.5.2.4 Overall process

In summary, the proposed HSCOA framework obfuscates the functionality of a hardware accelerator chip as follows: The proprietary SDK locks the encoding of instructions and sends them to the accelerator chip, where they are deobfuscated using the shared K_{seed} and *shift LUT* information. By default, the software initially sends an INITK instruction to reset the on-chip shift registers in the hardware key sequencer module. The very first instruction of an application is locked using the key K_1 which depends on the state of the registers initialized with K_{seed}. The obfuscation of subsequent instructions using keys $\{K_2, K_3, \ldots, K_I\}$ in K_{seq} is governed by the *shift vectors* which are randomly generated at run-time in secure SDK. This information is communicated to the accelerator chip using CHSFT type instructions. Note that one does not need to obfuscate each of the subsequent

instructions with a separate key in K_{seq}. The designer can choose to lock blocks of instructions with common keys or lock a few randomly selected instructions, thereby *reducing the locking overhead*. In section 4.5.3, we provide theoretical analysis to demonstrate the resiliency of HSCOA framework against SAT formulation based attack. We also describe how this framework is also immune to removal [119] or bypass [112] types of attacks. We assume that the sizes (design parameters) of K_{seed} and *shift LUT* are large enough so that the attacker cannot devise any brute force based attack strategy. Our proposed hardware-software based obfuscation approach can also be seamlessly integrated with conventional logic obfuscation schemes [82, 76, 118] to lock other components (like MMU) of an accelerator chip to further enhance the security-level.

4.5.3 Security Analysis of HSCAO

4.5.3.1 Resiliency to SAT attack

First we illustrate how the process of determining K_{seed} is computationally infeasible through SAT formulation based attack [97]. According to our threat model, an attacker has access to the netlist of the key sequencer design. Her objective is to find the key sequence K_{seq} for unlocking the software obfuscated instructions using SAT attack. Note that the very first instruction is obfuscated using K_1, which is derived from the state of the key sequencer initialized with K_{seed} (using default INITK instruction). Unlike subsequent keys in K_{seq} which are dependent on *shift vectors* generated at run-time (and thus varies from one application run

to another), K_1 is run-time independent. The attacker can deduce K_1 as follows: At first, she develops a microbenchmark application having knowledge of all the instruction types. Then, she finds K_1 by simply XORing bit-wise the locked *control bits* of first instruction with the correct opcode bits (known from ISA). This is because $(a \oplus b) \oplus a = b$, where a represents the opcode bits, $b = K_1$ and $\oplus$ denotes bit-wise XOR operation.

Note that K_1 is derived from K_{seed} using the key sequencer algorithm whose functionality is known to the attacker. Hence, she can use SAT solver (or any other Boolean solver) to find a key K_{eqv} belonging to the *equivalence class* of all keys that result in K_1. Note this K_{eqv} may or may not be equal to secret key K_{seed}. Although K_{eqv} correctly determines K_1, the entire key sequence generated assuming K_{eqv} was the initial seed of the key sequencer may not be the same as the actual key sequence K_{seq}. This is because the sequence of keys generated by the key sequencer design with separate initialization seeds (producing same K_1) will not be the same. Hence, finding a key K_{eqv} is not sufficient and the attacker needs to find the exact key K_{seed}. We show that the probability of K_{eqv} equals K_{seed} is exponentially small in terms of size of the secret key K_{seed}, thus making SAT attack (or other Boolean logic based attacks) against HSCOA as impractical as a brute force attack.

Theorem 4.1. The probability of finding K_{seed} using the above SAT attack approach is $1/2^{(N-1)n}$, where N is the number and n is the size of the cyclic shift registers.

Proof. Let v_i^j denote the value of i^{th} bit, $i \in \{1, n\}$, of the j^{th}, $j \in \{1, N\}$, cyclic shift register. Also, let S_i denote the set of all v_i^j, i.e., $S_i = \{v_i^1, v_i^2, \ldots, v_i^N\}$. Without loss of generality let us assume N is odd (similar arguments hold for N being even). As per the key sequencer design, the i^{th} key-bit of the first key $K1$ (denoted by $K_{1,i}$) is obtained by XORing all the elements of S_i. The value of $K_{1,i}$ is 0 whenever there are even number of ones in S_i, while $K_{1,i}$ is 1 when there are odd number of ones in S_i. Therefore, the number of possible combinations Q_i^0 of values of elements in S_i which result in $K_{1,i} = 0$ can be expressed as follows:

$$Q_i^0 = \binom{N}{0} + \binom{N}{2} + \binom{N}{4} + \ldots + \binom{N}{N-1} \tag{4.4}$$

Similarly, the number of possible combinations Q_i^1 of values of elements in S_i which result in $K_{1,i} = 1$ is as follows:

$$Q_i^1 = \binom{N}{1} + \binom{N}{3} + \binom{N}{5} + \ldots + \binom{N}{N} \tag{4.5}$$

Since $\binom{N}{k} = \binom{N}{N-k}$, $k \in \{0, N\}$, from equations (4.4) and (4.5) we get $Q_i^0 = Q_i^1 = Q_i = 2^N/2 = 2^{N-1}$. As the cyclic shift registers are initialized with K_{seed} and any two bits in a shift register are independent of eachother, any two bits of the key K_1 are also *independent* of each other. Therefore, the number of possible values of key K_{eqv} which results in the same K_1 (of size n bits) is $2^{(N-1)n}$. Essentially, the set of keys which result in the same K_1 has a size of $2^{(N-1)n}$. Only **one** of these keys is K_{seed}. Hence, the probability of finding K_{seed} from K_1 using SAT attack based approach is $1/2^{(N-1)n}$. □

From the above theorem, we see that the probability of finding the secret key K_{seed} is exponentially small in terms of the key-size. Note that using the *shift LUT* we end up hiding the randomly generated *shift vectors* as well. This adds to the security guarantee even further as both K_{seed} and the contents of *shift LUT* needs to be determined correctly to break the HSCOA framework.

4.5.3.2 Resiliency to other attacks

Our proposed HSCAO scheme is inherently secure to other types of attack on logic locking schemes, like removal attack [119] and bypass attack [112]. The underlying principle of such approaches is to either remove or bypass the protection circuitry to retrieve the netlist exhibiting correct functionality. Though the hardware-level deobfuscation logic of our proposed HSCAO scheme can be structurally identified, the removal/bypass of it won't neutralize the effect of software-level obfuscation performed by the proprietrary accelerator SDK (root of trust). Also, as highlighted in section 4.5.1.1, the attacker doesn't have the capability to develop a substitute SDK using the netlist information.

4.5.4 Experimental Results

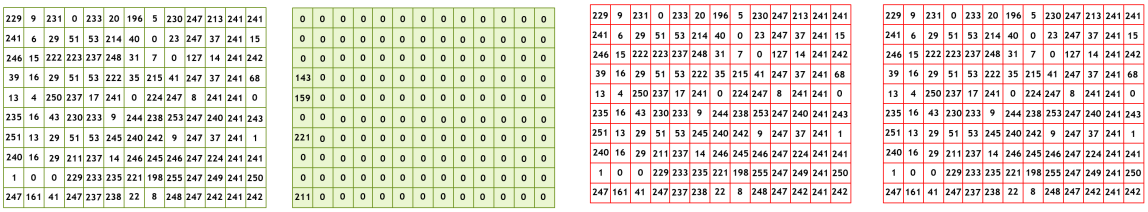
For our experiments, we used OpenTPU simulator [13] which is an open-source re-implementation of Google's TPU chip [50]. We considered a Tensorflow based implementation of Multi-layer Perceptron (MLP) regressor on the Boston Housing dataset [13]. In Fig. 4.7, we present the assembly-level program of such an application with detailed description of each instruction type. To augment the

```
(RHM)   RHM 0, 0, 10          # read from host mem addr 0, to UB addr 0, for length N = 10
(RW1)   RW  0                 # read weights from dram addr 0 to FIFO
(RW2)   RW  1                 # read weights from dram addr 1 to FIFO
(RW3)   RW  2                 # read weights from dram addr 2 to FIFO
(MMC1) MMC.SO 0, 0, 10        # Do MM on UB addr 0, to accumulator addr 0, for length 10
(ACT1)  ACT.R 0, 0, 10        # Do ACT ReLU on accumulator addr 0, to UB addr 0, for length 10
(MMC2) MMC.SO 0, 0, 10
(ACT2)  ACT.R 0, 0, 10
(MMC3) MMC.SO 0, 0, 10
(ACT3)  ACT.R 0, 0, 10
(WHM)   WHM 0, 0, 10          # write result from UB addr 0, to host mem addr 0, for length 10
(HLT)   HLT                   # halt execution
```

Figure 4.7: Assembly-level of MLP regression application

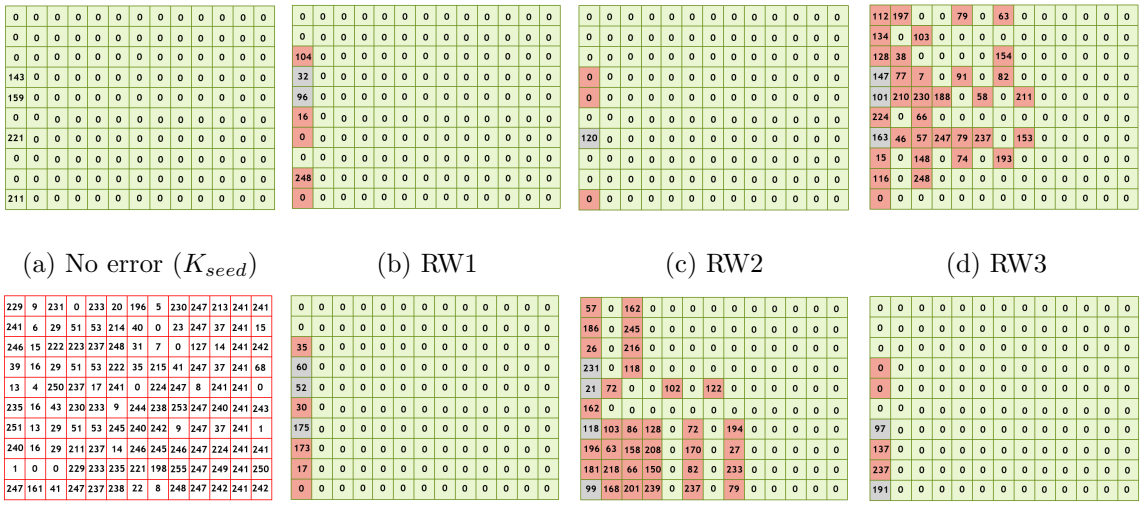
proposed HSCAO scheme with the OpenTPU simulator, we designed a *key sequencer* with N=9 cyclic shift registers (each $n = 16$ bits in length) to generate key sequence K_{seq} initialized with a randomly selected K_{seed} of size 144 bits. Each key belonging to K_{seq} was bit-wise XORed with the opcode bits (as specified in OpenTPU ISA) for performing obfuscation/ deobfuscation of an instruction in the software/ hardware counterparts of HSCAO framework. In Fig. 4.8a and Fig. 4.8b, we present the initial host memory content and the final host memory content (after running the application) of an unlocked TPU-like chip (activated using the correct key K_{seed}). Each small green square contains the correct value of a memory location.

To study the application-level error impact due to the use of an equivalent first round key for unlocking the TPU-like chip, we used two such keys K^1_{eqv} and K^2_{eqv} as initial seeds and ran the regression application (see Fig. 4.7). Note that for performing these experiments, we considered that the *shift LUT* is configured correctly. But in practice, the attacker will face additional challenge to determine the *shift LUT* contents. In Fig. 4.8c and Fig. 4.8d, we present the final host memory



(a) Initial HM (b) Final HM:K_{seed} (c) Final HM: K^1_{eqv} (exception) (d) Final HM: K^2_{eqv} (early termination)

Figure 4.8: Error impact on final host memory (HM) due to wrong instruction deobfuscation for different equivalent keys



(a) No error (K_{seed}) (b) RW1 (c) RW2 (d) RW3

(e) MMC1/WHM (f) ACT1 (g) MMC2,3/ACT3 (h) ACT2

Figure 4.9: Error impact on host memory (HM) due to single locked instruction

contents for using K^1_{eqv} and K^2_{eqv} respectively. With K^1_{eqv}, the application terminated with an exception that a new matrix multiply (MMC) type instruction cannot be dispatched while a previous instruction is still being issued, thus resulting in no memory update (as denoted by red squares). Similarly, with K^2_{eqv} also there

was no memory update as well due to early application termination (no exception raised). We observed that the reason behind this early termination being one of the keys (in the sequence generated by K^2_{eqv}) when bit-wise XORed with the corresponding obfuscated instruction opcode incorrectly resulted in the opcode for exit/halt condition (HLT). These results highlight that use of such equivalent keys fail to unlock the accelerator obfuscated using HSCOA framework.

The above approach of obfuscating every instruction, though effective, may incur significant delay for running applications due to updates of cyclic shift registers (depending on *shift vector* contents) per instruction. Therefore, we **locked only a single instruction** in the entire application assembly (apart from the first RHM instruction which is locked by run-time independent key K_1) and observed the resulting *corruption* in final memory contents. The outcomes of such experimental runs are presented in Fig. 4.9, where each subfigure shows the final host memory content for a particular locked instruction in the regression application. The states of memory locations are classified into 4 categories: (i) correct data update (ii) wrong data update which signifies data update in a faulty memory location (iii) corrupted data update where the memory update location is correct but the data content is wrong and (iv) no data update from initial data content. As observed from the figures, even locking a single instruction leads to significant errors in the application outcomes, highlighting the strength of our proposed HSCAO countermeasure to protect the IP of an accelerator chip design.

4.6 Conclusion

In this chapter, we first outlined an iterative SAT formulation based attack to *approximately* unlock the functionalities of pipelined GPU cores which are obfuscated using state-of-the-art logic locking scheme. The experimental results (obtained using NVIDIA's SASSIFI framework) reveal that the benchmark GPGPU applications exhibit high resiliency to error propagation effect due to use of a retrieved *approx-key* for unlocking the core netlists. Our proposed attack technique can be effectively utilized by an untrusted foundry to successfully deobfuscate GPU core netlist, even *without any requirement of an activated hardware*. Subsequently, we propose the cache locking scheme as a low-overhead countermeasure which significantly degrades the performance of applications for a wrong *cache-key*. In addition, we also propose a hardware-software co-design based obfuscation approach (called HSCOA) to render an unactivated accelerator chip functionally useless. Our proposed HSCOA scheme uses proprietary SDK as the root of trust for generating locked program binary which is subsequently deobfuscated in the hardware. The experimental results obtained by running a regression application on the OpenTPU simulator demonstrate the effectiveness of such an obfuscation framework.

Chapter 5: Hardware-assisted Obfuscation of Deep Neural Networks

5.1 Introduction

Deep learning (DL) algorithms are extensively used for analyzing big data in several domains including image classification, natural language processing, autonomous transportation, smart health, financial management, social networks, etc. [58, 41]. The key factors attributed to the unprecedented success of these algorithms are (i) availability of a *massive* and mostly labeled training dataset (ii) allocation of powerful computing resources as well as vast amounts of network training time and also (iii) substantial domain expertise of DL model developers to obtain highly accurate models. Therefore, well-trained DL models are considered to be IPs of the owner as *significant* cost is incurred behind their training process to gain a competitive edge in business [80, 36, 111]. In a *white-box* setting [103], the neural network architecture as well as the trained DL model parameters are made publicly available (e.g., Caffe's Model Zoo and Amazon's Alexa Skills) by a DL model owner [80, 17]. As the popularity of using such pre-trained models increases (especially with the deployment of MLaaS), IP protection as well as Digital Rights Management (DRM) of these distributed DL models are of major practical

concerns [80]. The prevention of *model piracy* is a key challenge in this field as there exists techniques (such as scaling, noising, fine-tuning, etc.) to cleverly modify model parameters without affecting the functionality or accuracy of the network and thus, helping attackers to claim false DL model ownership [98].

There has been a lot of research to address the privacy concerns of user data which are used to train Deep Neural Networks (DNNs) [16, 92, 121]. However, on the other hand, there is only a limited number of works which primarily focus on developing techniques to protect the IP of well-trained DL models rather than securing sensitive user data. Watermarking strategies for DL models have been proposed in recent literature [80, 17, 43, 46, 66] which help to claim the ownership of stolen models by embedding identification information into them. But such leaked DL models can be reused *privately* by the adversary, thus bypassing ownership inspection by the aforementioned watermarking techniques [120]. In order to further strengthen the IP security of DL models, an obfuscation technique for DNNs has been proposed in [111] which *structurally* obfuscates the network architecture. However, commonly raised DL model theft concerns are related to the stealing of well-trained weight parameters (or learned network functionality) and not due to the theft of DNN topology. This is because industrial applications typically use previously published DNN architectures which have demonstrated high modeling capabilities [98]. This strongly motivates us to develop a robust and efficient DNN obfuscation infrastructure which locks a DL model's weight parameters. Such an

obfuscated DL model should exhibit high prediction performance *only* if an end-user has legitimate access to it, whereas any unauthorized usage of the locked model should result in significant degradation of its prediction accuracy.

The above goal of IP protection of DL models can be achieved using provably-secure cryptographic schemes to encrypt the weight parameters. However, application of encryption/decryption on millions of model parameters (as present in modern DNNs) will incur large time/implementation overheads and thus, conflict with the strict response-time deadlines of DNN inference applications. In this chapter, we propose an obfuscation framework called Hardware Protected Neural Network (HPNN) as a *lightweight* alternative to achieve the desired IP security of DL models in a white-box setting. This framework ensures that only an authorized end-user who possesses a trustworthy hardware device (with the secret key embedded on-chip) is able to run intended DL applications using the published model. In addition, we also propose a novel watermarking technique called DynaMarks to protect the IPs of DL models against *model extraction* attacks performed by an authorized end-user using responses of the trusted hardware device to chosen queries. The main contributions of the chapter are as follows:

- We propose an obfuscation framework called HPNN to protect the IPs of DL models in a white-box setting. To the best of our knowledge, this is the first work which leverages hardware as a *root-of-trust* to achieve IP security of DL models.

- We provide a theoretical construct of a *key-dependent* backpropagation algorithm for training a neural network which doesn't sacrifice a model's prediction accuracy to gain security benefits.

- We perform extensive experimental evaluations across different DNN architectures and benchmark datasets to assess the robustness of obfuscated DL models in HPNN framework against model fine-tuning attacks.

- In addition, we also propose DynaMarks, a black-box watermarking technique to defend against model extraction attacks performed by authorized end-users. DynaMarks embeds watermark by dynamically changing the responses of the model's prediction API during the inference phase, without introducing any computational overhead in the training process.

5.2 Motivation

The development of a production-level DL model is not a trivial task as it requires a massive amount of training data along with high power computing resources. State-of-the-art DL models take several weeks of training over GPU clusters. In addition, designing a well-trained model requires significant machine learning expertise as well as long working hours to execute numerous trial runs to properly optimize the associated network hyper-parameters [80, 111]. The growing trend of deployment of well-trained DL models in public cloud infrastructure (MLaaS settings) opens the door for attackers to steal models and establish plagiarized

machine learning services. Such IP theft of DL models poses a major threat of substantial revenue loss in market share to its owners [17, 80]. Also, stolen DL models used in mission-critical operations (which may involve national security) can be sold to Darknet markets [120]. Therefore, there is a strong need to ensure the security of well-trained DL models from illegal usage.

Attacker's Goal. In this work, we assume that an attacker has access to a DL model's weight parameters either through public cloud platform or from an information breach via malicious malware infection or an insider source [120]. Also, we assume that the attacker has knowledge of the DNN architecture (or topology) used to train the model. This is reasonable assumption as industrial applications typically use published DNN architectures which have demonstrated high modeling capabilities [98]. The objective of the attacker is to either utilize the stolen DL model to provide a plagiarized cloud based service to end-users or to deploy it in a private network for running intended DL applications (as shown in Fig. 5.1). Though, in the former scenario, the DL model owner may still use watermarking techniques [80, 17, 43] to claim digital rights (if somehow she has obtained access privileges to the illegal cloud service), but in the latter attack scenario, the model owner won't have any provision to remotely query the DL model to extract watermarked contents [120]. This strongly motivates the development of a much more effective IP security solution for DL models which can thwart any sort of unauthorized usage scenarios.

A robust IP protection of DL models can be achieved using provably-secure cryptographic schemes where the DL model owner encrypts the model parameters before uploading them in a public cloud service. Only a legitimate end-user will be able to decrypt (using a *secret* key) the encrypted parameters to retrieve the trained DL model. However, this solution will be highly inefficient in practice as industrial DL models have millions of weight parameters [50] and applying cryptographic algorithms on such large-scale DNNs will incur huge time/implementation overheads. Instead, we propose HPNN framework as a *lightweight* alternative to secure IPs of DL models by obfuscating their weight parameters. Such obfuscated DL models can be openly distributed using public cloud infrastructure without any IP theft concerns.

5.3 Proposed HPNN Framework

5.3.1 Overall Flow

The global flow of HPNN framework is presented in Fig. 5.1. A DL model owner spends long working hours to train a network using a large annotated training dataset and high-performance computing platforms. The crux of IP protection guarantees provided by the HPNN framework relies on training a DNN using a *key-dependent* backpropagation algorithm (more details in section 5.3.3) which obfuscates the learned weight space of the model. Such a *key-dependent* training approach doesn't compromise the prediction accuracy of the obtained model to gain security benefits. Then, the obfuscated DL model is hosted on a public model

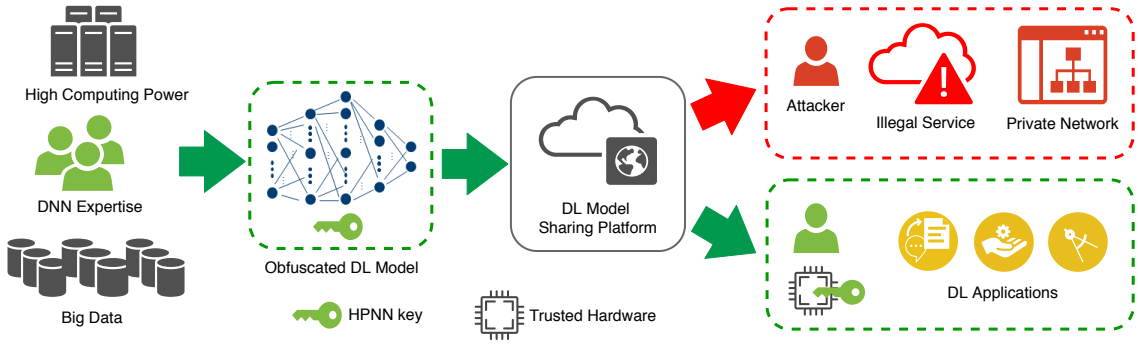


Figure 5.1: Proposed HPNN framework for IP security of DL models.

sharing platform (such as a cloud interface in MLaaS settings) to provide services to *only* authorized customers who have acquired the requisite licenses for model usage. In our proposed HPNN framework, licenses are distributed in the form of trustworthy hardware devices which securely embed the *secret* HPNN key on-chip [15, 116]. This scheme aims to guarantee state-of-the-art inference phase performance of a locked DL model only on such trusted hardware devices, while significantly degrading its prediction accuracy for any illegal usage. Note that a model owner can train several DNNs using the same HPNN key to obtain obfuscated DL models targeting different applications. Later in section 5.4.2, we also experimentally demonstrate the effectiveness of HPNN framework to thwart model fine-tuning type attack where an attacker tries to leverage the knowledge of the DNN architecture (white-box setting) and an available *thief* dataset to steal a well-trained DL model.

Hardware root-of-trust. Our proposed HPNN framework relies on the utilization of a hardware root-of-trust (with *secret* HPNN key embedded on-chip) to provide services to authorized end-users. The rationale behind the assumption of availability of such trusted hardware devices are as follows: **(i)** Domain-specific hardware chips

(e.g., Google's Tensor Processing Unit [50], Intel's Neural Compute Stick [8], etc.) are being deployed in industrial settings for accelerating the inference phase in DNN applications. In our proposed HPNN framework also, the trusted hardware devices are utilized by authorized end-users for running only the DNN inference phase. Note that during the DNN training phase, the DL model owner just requires the knowledge of HPNN key value (no need for any trusted hardware device) to obfuscate the learned weight space of the model. **(ii)** Also, in order to counter emerging threats to IoT edge devices, applications are increasingly designed to rely on secure key storage facility provided by a hardware root-of-trust such as Trusted Platform Module (TPM) [15]. In addition to providing *stronger* security guarantees than their software counterparts, hardware-assisted protection mechanisms also incur significantly lower performance overhead [39, 83].

5.3.2 Neural Network Obfuscation

In this work, we assume that an attacker has knowledge of the details of a DNN architecture, i.e., the number and types of layers in the network as well as the connectivity graphs between the layers (white-box setting). Henceforth, we refer to such information as knowledge of the *baseline* DNN architecture. The goal of our proposed HPNN framework is to train a DNN in such a way that the learned weight space of the model is obfuscated as a function of *secret* HPNN key. To realize this objective, we *lock* any j^{th} neuron belonging to a nonlinear layer of the network by associating a HPNN key bit k_j as illustrated in Fig. 5.2(a). Such a neuron basically

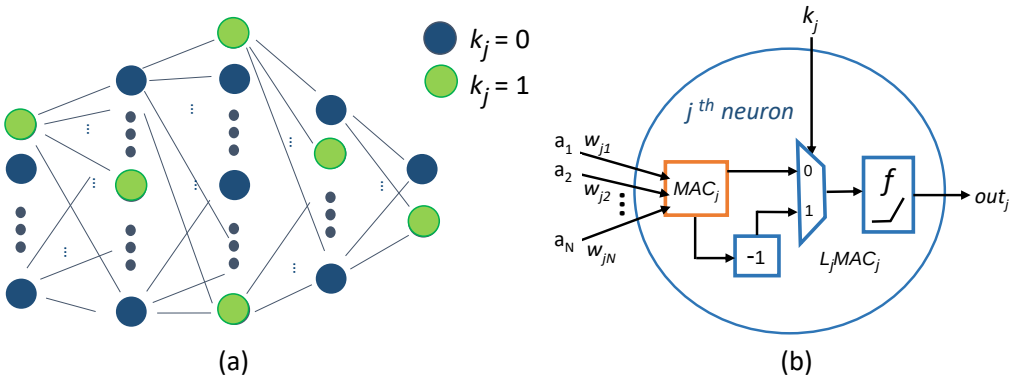


Figure 5.2: Obfuscation of a neuron in HPNN framework.

performs (i) multiply and accumulate (MAC) operation to compute a weighted sum of its inputs $(a_1, a_2, ..., a_N)$, i.e., $\text{MAC}_j = \sum_{i=1}^{N} a_i w_{ji} = \mathbf{a^T w_j}$ and (ii) then, passes MAC_j through a nonlinear activation function f to produce the neuron's output response, i.e., $out_j = f(\text{MAC}_j)$. Now, in order to lock the functionality of j^{th} neuron, we make out_j dependent on HPNN key bit k_j as follows:

$$out_j = f(L_j \text{MAC}_j) = f(L_j \mathbf{a^T w_j}) \tag{5.1}$$

where,

$$L_j = \begin{cases} +1 & \text{if } k_j = 0 \\ -1 & \text{if } k_j = 1 \end{cases} \tag{5.2}$$

The variable L_j is called the *lock factor* of j^{th} neuron, which governs the sign of MAC_j based on k_j value as shown in Fig. 5.2(b). If $k_j = 0$, then MAC_j remains the same, whereas (ii) if $k_j = 1$, then sign of MAC_j is flipped. Next, we study the implication of such key based obfuscation of neurons on the network training process.

5.3.3 Key-dependent Backpropagation

In order to train a neural network in the HPNN framework, we propose a *key-dependent* backpropagation algorithm which creates a model whose weight space is highly optimized as a function of the HPNN key. Such an obfuscated model strongly resists any attempts to illegally utilize it by concealing the learned decision boundaries. Next, we describe how the notion of HPNN key can be augmented to a *conventional* backpropagation based training approach.

Neural networks are typically trained using iterative, gradient-based optimizers with the objective of driving a desired *cost function* to a very low value. We consider the training of a network using *delta rule* which utilizes backpropagation algorithm to update network parameters such that the given cost function is minimized [41]. Let E^p denote the cost function which measures the discrepancy between the expected (or correct) output response and the output response produced by a network for the p^{th} training vector. Then, the learning rule for the i^{th} incoming weight of j^{th} neuron (w_{ji}) can be expressed as follows:

$$\Delta w_{ji} = -\eta \frac{\partial E^p}{\partial w_{ji}} \tag{5.3}$$

where, η is the learning rate. In HPNN framework, if we consider a mean squared error (MSE) cost function, i.e., $E^p = \frac{1}{2}\sum_j (t_j - out_j)^2$ with t_j being the correct output label, the above weight learning rule will be a function of *lock factor* L_j as shown below:

$$\Delta \mathbf{w}_j = \eta \delta_j \mathbf{a} \tag{5.4}$$

where,

$$\delta_j = \begin{cases} (t_j - out_j) f^{'}(L_j \text{MAC}_j) L_j & \text{if } j^{th} \text{ neuron} \in \text{output layer} \\ \Big(\sum\limits_{k \in O} w_{kj} \delta_k \Big) f^{'}(L_j \text{MAC}_j) L_j & \text{if } j^{th} \text{ neuron} \in \text{hidden layer} \end{cases}$$

with $f^{'}$ being the derivative of the activation function f, $\mathbf{a}$ denotes the input vector to the neuron, and O denotes the neuron's adjacent layer. The above backpropagation based learning rule can now be used to update the incoming weight vectors of all locked neurons in the proposed obfuscation framework. This will lead the entire network to learn an optimized weight space as a function of not only the training dataset but also the L_j values (which are derived from the *secret* HPNN key, see Eq. 5.2). As demonstrated later by experimental results outlined in section 5.4, such a locked model performs accurately during the inference phase *only* when the HPNN key is used to retrieve the correct functionalities of the locked neurons.

Model capacity. The capacity of a model describes the *complexity* of relationship it can map between the input patterns and output labels for a given dataset. The capacity of a DL model obtained by training a DNN using our proposed HPNN framework is *independent of any key value used*. To demonstrate this property let us first consider the case of a single layer fully-connected network, before we consider more complex DNN architectures. Note that two models (obfuscated using two different HPNN keys) have equivalent capacities if *there exists* equivalent weight assignments which lead to the *same* output predictions for any given input to the models. We show the existence of such equivalent weight assignments for a single layer fully connected network by establishing a relationship between the incoming

weight vectors of any j^{th} neuron (locked with different HPNN key bit values) which leads to the same neuron response out_j and hence, the same overall network's prediction for an input vector.

Definition 5.1. For any j^{th} neuron (with *lock factor* L_j and output response out_j), let $\mathbf{w}_j^{init}$ denote its initial incoming weight vectors (before training) and let $\mathbf{w}_{j,L_j}^N$ denote its incoming weight vectors after N training epochs.

Theorem 5.1. For a single layer fully-connected network initialized with all zero weight parameters (i.e., $\mathbf{w}_j^{init}$=0), we get $\mathbf{w}_{j,-1}^N$=$-\mathbf{w}_{j,1}^N$.

Proof. We prove this theorem using principle of mathematical induction. (i) *Base case* : Before any training epoch, we have $\mathbf{w}_{j,-1}^0$= $\mathbf{w}_j^{init}$= $\mathbf{0}$ = $-\mathbf{w}_{j,1}^0$ (ii) *Induction step* : Let us assume that $\mathbf{w}_{j,-1}^K = -\mathbf{w}_{j,1}^K$ after K training epochs. We now need to show $\mathbf{w}_{j,-1}^{K+1} = -\mathbf{w}_{j,1}^{K+1}$ in order to prove the lemma. In the $(K+1)^{th}$ training epoch with $L_j = 1$ and using Eqs. (5.1) and (5.4) we get,

$$\begin{aligned} \Delta\mathbf{w}_{j,1} &= \eta(t_j - f(\mathbf{a^T w}_{j,1}^K))f'(\mathbf{a^T w}_{j,1}^K)\mathbf{a} \\ \mathbf{w}_{j,1}^{K+1} &= \mathbf{w}_{j,1}^K + \Delta\mathbf{w}_{j,1} \end{aligned} \tag{5.5}$$

Similarly, with $L_j = -1$ we get,

$$\begin{aligned} \Delta\mathbf{w}_{j,-1} &= -\eta(t_j - f(-\mathbf{a^T w}_{j,-1}^K))f'(-\mathbf{a^T w}_{j,-1}^K)\mathbf{a} \\ &= -\eta(t_j - f(\mathbf{a^T w}_{j,1}^K))f'(\mathbf{a^T w}_{j,1}^K)\mathbf{a} \\ &= -\Delta\mathbf{w_{j,1}} \end{aligned} \tag{5.6}$$

$$\begin{aligned} \text{Therefore,} \quad \mathbf{w}_{j,-1}^{K+1} &= \mathbf{w}_{j,-1}^K + \Delta\mathbf{w}_{j,-1} \\ &= -(\mathbf{w}_{j,1}^K + \Delta\mathbf{w}_{j,1}) = -\mathbf{w}_{j,1}^{K+1} \end{aligned} \tag{5.7}$$

Hence, by the principle of induction the theorem holds true. □

It is non-trivial to derive similar relationships for modern DNN architectures which consist of multiple hidden layers. Also, a network is typically initialized with small random non-zero weight parameters for effective training [41]. But the following lemma still guarantees that the DL model capacity is unaffected by the choice of HPNN key used to train the DNN in our proposed obfuscation framework.

Lemma 5.2. DL models obfuscated using different HPNN keys have equivalent model capacities.

Proof. For a given DNN architecture, the manner in which a j^{th} neuron is locked in HPNN framework ensures that the same neural activation response out_j will be produced if we have incoming weight vectors of $\mathbf{w}_j$ for $k_j = 0$ and $-\mathbf{w}_j$ for $k_j = 1$, as evident from Eq. (5.1). This implies that *there exists* equivalent weight assignments for different HPNN keys which will lead to the *same* network prediction outcomes, which in turn implies that all such obfuscated DL models have equivalent capacities.

□

However, in practice, such *key-dependent* backpropagation based DNN training is likely to yield different incoming weight vector magnitudes of neurons for different HPNN keys due to network nonlinearity as well as random weight initialization. To ascertain the equivalence in capacities of DL models obtained by training the same DNN topology but locked using different HPNN keys, we performed the following experiment: First, we *randomly* generated 20 different HPNN keys, and then used these keys to train a given DNN architecture with the same training

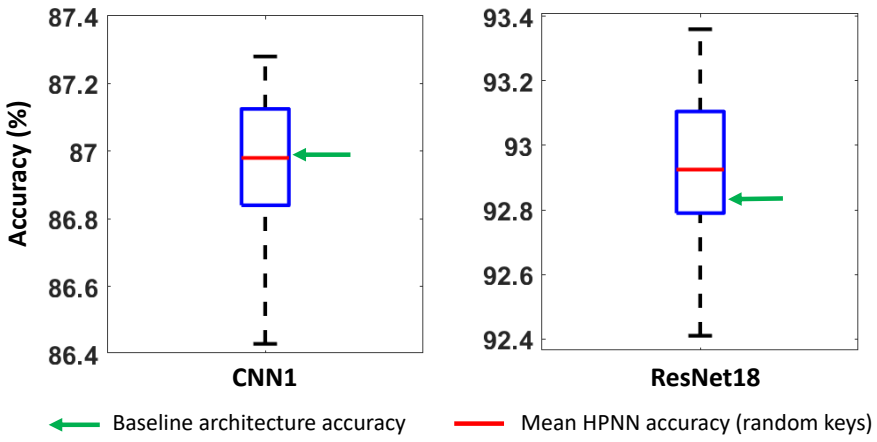


Figure 5.3: Performance of DL models locked using different HPNN keys.

dataset (Fashion-MNIST [5]) and hyperparameters combination. We considered the prediction accuracy of a DL model as the indicator of its modeling capacity. The experimental results are presented in Fig. 5.3 for two different DNN architectures, CNN1 (see Table 5.1 for network details) and ResNet18 [45]. Each of the box plots shows the distribution of prediction accuracy of 20 different DL models on the same test dataset. Such model accuracy distributions highlight the fact that DL models obtained using different HPNN keys perform on an equivalent scale. Also, the mean prediction accuracy (shown using red lines) for CNN1 and ResNet18 networks are 86.95% and 92.93% respectively, which are very close to the corresponding accuracy (shown using green arrows) of 86.99% and 92.83% of the *baseline* DL models (i.e., the models obtained using conventional backpropagation based training of *baseline* DNN architectures).

5.3.4 Role of hardware root-of-trust

In the proposed HPNN framework, an authorized end-user utilizes a trusted hardware device (which embeds the *secret* HPNN key) to run the DNN inference phase. In a modern DNN architecture, there are typically thousands of neurons belonging to nonlinear network layers and hence, associating a key bit with each such neuron (as presented in section 5.3.2) will lead to an impractically large HPNN key length. The hardware root-of-trust not only accelerates the DNN inference phase but also *facilitates the use of a practical size HPNN key.* This can be achieved by a simple modification in the MAC unit design of the trusted hardware device. For illustration purposes, let us consider a Google TPU-like chip [50] which will be deployed as a hardware root-of-trust in our proposed DL model obfuscation framework.

Google TPU design. The main computational component of a Google TPU chip is called matrix multiply unit (MMU) which performs MAC operations in a pipelined manner. MMU consists of 256X256 MACs which compute 8-bit multiply-and-adds on signed or unsigned integers. The resulting 16-bit products are first collected in 256 accumulator units and then passed on to an on-chip activation module which implements standard nonlinear operations (such as ReLU, sigmoid, etc.). For more details on TPU architecture, please refer to [50]. Next, we outline how the MAC design of such a chip can be modified to facilitate the use of a practical size HPNN key.

5.3.4.1 Key-dependent accumulator

We propose a low overhead design modification to make the MAC computation *key-dependent* as shown in Fig. 5.4(a). As specific design details of TPU are not publicly available, we make the following assumptions for the sake of illustration: (i) the design of an accumulator unit is based on a full-adder (FA) chain as shown in Fig. 5.4(b). (ii) all numbers are stored and operated on in their two's complement representation. Now, in order to lock the MAC computation of j^{th} neuron as a function of k_j, we introduce 16 additional XOR gates per accumulator unit as shown in Fig. 5.4(b). Each such gate takes as input – (i) a bit from the multiplier unit's 16-bit result and (ii) an HPNN key bit k_j which is supplied from a secure on-chip memory. The magnitude of k_j determines the functionality of the accumulation operation: If $k_j = 0$, then $\text{MAC}_j = \sum_{i=1}^{N} a_i w_{ji}$ is computed by performing a sequence of addition in the accumulator unit. On the other hand if $k_j = 1$, then MAC_j is converted to its two's complement by performing a sequence of subtraction, i.e., $\sum_{i=1}^{N} -a_i w_{ji} = -\text{MAC}_j$. This simple modification in the accumulator design makes the response of j^{th} neuron dependent on its lock factor L_j, i.e., $out_j{=}f(L_j MAC_j)$, as expected in Eq. (5.1).

5.3.4.2 HPNN key

As there are only 256 such accumulator units in a Google TPU-like architecture, the size of HPNN key will be 256 bits (a practical key length) and the total number of additional XOR gates required will be 256X16 = 4096. When

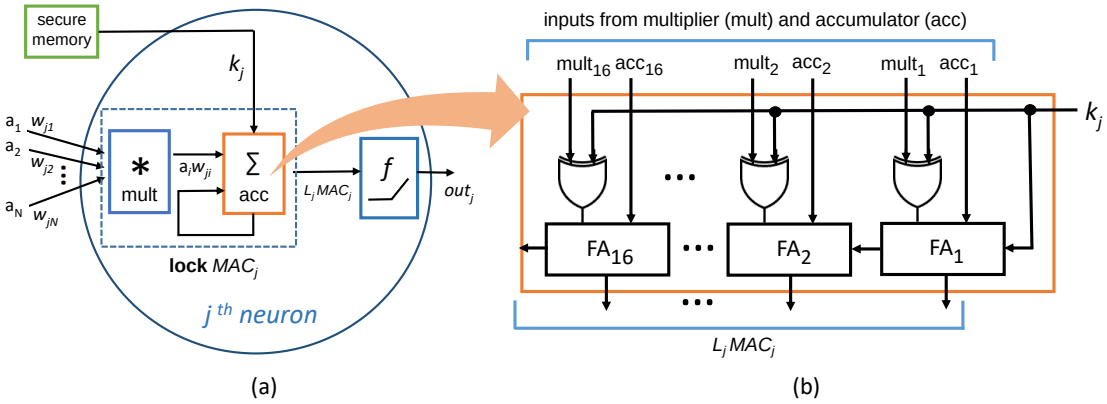


Figure 5.4: Hardware realization of neuron locking mechanism.

a large-scale DNN inference is run on such an accelerator chip, multiple locked neurons will be mapped to a single accumulator unit by using a hardware-specific scheduling algorithm. This implies that a single HPNN key bit will be associated with several locked neurons in the HPNN framework. During the training phase, a DL model owner needs to utilize the information from this hardware-specific scheduling algorithm to derive the key bits corresponding to all the locked neurons of a DNN from the 256-bit HPNN key. Note that the details of such scheduling used in the hardware root-of-trust will also be kept private to further enhance the security of HPNN framework.

5.3.4.3 Implementation overhead

The additional cost incurred to provide MLaaS using HPNN framework are as follows: (i) In the training phase, a DL model owner needs to perform a one-time preprocessing using the notion of hardware-specific scheduling algorithm to map subsets of DNN neurons to their corresponding HPNN key bits. (ii) In the inference

phase, which is carried out using the trusted hardware, **small area overhead** will be incurred due to introduction of additional XOR gates (4096 gates in case of Google TPU-like architecture) for modifying the accumulator design. If we consider a MMU implementation [62] which consists of gates in the order of 10^6, then the gate overhead due to our proposed design modification will be less than 0.5%. Also, there will be **no clock cycle overhead** (only combinational delay for calculating two's complement) due to the introduction of additional XOR gates. Hence, our proposed HPNN framework offers a *lightweight* IP security solution for DL models.

5.4 Evaluation of HPNN Framework

We evaluate the security benefits offered by the HPNN framework across 3 different benchmark datasets (Fashion-MNIST [5], CIFAR-10 [2], and SVHN [14]) and Convolution Neural Network (CNN) architectures (details in Table 5.1). We used Pytorch 3.1 to run simulations on a system consisting of an Intel Xeon CPU and a Nvidia Maxwell GPU with 32 GB and 2 GB memories respectively.

5.4.1 Performance of locked DL models

A DL model obtained using the *key-based* backpropagation algorithm (see section 5.3.3) should demonstrate high prediction accuracy *only* when it runs inference on a trusted hardware device. Such a hardware root-of-trust deobfuscates the locked neurons of a DNN to retrieve the network functionality using the *secret*

HPNN key[1]. The proposed HPNN framework aims to thwart any attempts by the attacker to run DNN inference with satisfactory prediction accuracy by loading the *baseline* DNN architecture with a stolen DL model. We performed experiments across different benchmark datasets to asses the robustness of HPNN framework in such an attack scenario. In columns 4 and 5 of Table 5.1, we report the accuracy obtained when running locked DL models on a hardware-root-of-trust (simulated by providing the *secret* HPNN key to retrieve the DNN functionality) and on the *baseline* DNN architecture (no key). In the latter case, we observe substantial accuracy drops of 79.88%, 80.17%, and 73.22% for Fashion-MNIST, CIFAR-10, and SVHN datasets respectively compared to the *original accuracy* as obtained by running the locked DL models on trusted hardware. Next, we evaluate the security offered by HPNN framework to protect the IP of a well-trained DL model in a stronger model fine-tuning attack scenario.

[1]In our experiments, we randomly assigned key bit values to neurons belonging to nonlinear layers of a DNN. However, in practice, the DL model owner needs to derive the key bits to be associated with such neurons from the HPNN key using hardware-specific scheduling information (see Sec. 5.3.4.2).

Dataset	Network Architecture (number and types of layers)	No. of neurons in nonlinear (ReLU) layers	Original accuracy	HPNN locked		Random fine-tuning		HPNN fine-tuning	
				accuracy	**%drop**	accuracy	**%drop**	accuracy	**%drop**
Fashion-MNIST	CNN1 (2 C, 2 MP, 2 ReLU, 1 FC)	4352	89.93	10.05	**79.88**	86.35	**3.58**	82.45	**7.48**
CIFAR10	CNN2 (6 C, 3 MP, 8 ReLU, 3 FC)	198144	89.54	9.37	**80.17**	78.87	**10.67**	78.53	**11.01**
SVHN	CNN3 (3 C, 3 MP, 4 ReLU, 2 FC)	29696	89.06	15.84	**73.22**	80.97	**8.09**	82.89	**6.17**

Table 5.1: Effectiveness of HPNN framework against model fine-tuning attack

(C: convolutional, MP: max-pooling, FC: fully-connected layers)

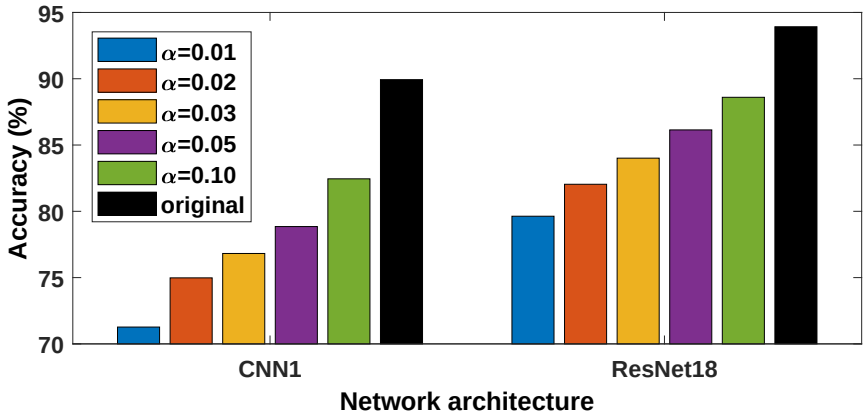


Figure 5.5: Accuracy vs. size of *thief* dataset (Fashion-MNIST)

5.4.2 Model fine-tuning attack

Model fine-tuning is a type of *transformation attack* strategy [70, 80] which drives the underlying neural network to converge to some other local minimum (different from the original model) and results in comparative performance in practical applications. To evaluate the effectiveness of our proposed HPNN framework against a model fine-tuning attack we consider the following threat model.

Attacker's Capabilities. In addition to having the knowledge of the *baseline* DNN architecture, the attacker has the following privileges:

- Availability of a *thief* dataset (annotated) which constitutes a small fraction α (say 10%) of the original training dataset.
- Significant DNN expertise as well as powerful computational resources to train large network architectures.

Attacker's Limitation. The attacker doesn't possess a large amount of annotated training data as well as optimized model hyperparameters (which are responsible for its highly accurate performance). This is a reasonable assumption as DL model owners keep such information private to maintain a competitive edge in business [98, 80].

Attack Methodology and results. To perform a model fine-tuning attack, the attacker first loads the stolen DL model parameters to initialize the *baseline* DNN architecture and then utilizes the *thief* dataset to retrain the model. The attack is deemed successful only if such a retrained DL model performs equivalently, i.e., shows similar high levels of accuracy in its predictions as the owner's DL model running on a hardware root-of-trust (which embeds the HPNN key).

5.4.2.1 Impact of thief dataset size and network architecture

To analyze the impact of the size of the *thief* dataset on the success rate of a model fine-tuning attack, we assume the availability of different *thief* dataset fractions (α=1%, 2%, 3%, 5%, and 10%) to the attacker. In Fig.5.5, we present the experimental results of model fine-tuning attack across 2 different DNN architectures (CNN1 and ResNet18, see Table 5.1 for network topology) using the Fashion-MNIST dataset. It can be observed from the accuracy trends that as the size of the *thief* dataset increases, so does the success rate of a model fine-tuning attack. However, even with α=10%, the attacker reaches a fine-tuning accuracy of only 82.45% and 88.60% for CNN1 and ResNet18 whereas the corresponding accuracy obtained originally by DL model owner are 89.93% and 93.92% respectively. These results

highlight the effectiveness of our proposed HPNN framework to safeguard the IPs of DL models across different network architectures. Note that in the above set of experiments, we used the same hyperparameter configuration for performing model fine-tuning as used by the DL model owner to train the network.

5.4.2.2 Impact of hyperparameter

We varied both the learning rate (lr) and the number of training epochs to observe the best accuracy that can be attained using model fine-tuning attack. In Fig. 5.6, we present the results of such experiments using a *thief* dataset fraction α=10% across different datasets. The best accuracy achieved by such hyperparameter tuning on Fashion-MNIST and CIFAR-10 datasets are 85.91% and 79.61% respectively, which are significantly lower than their counterparts of 89.93% and 89.54% as obtained by the DL model owner. Also, we observed that increasing lr too much (for example setting $lr = 0.05$ on Fashion-MNIST dataset, see Fig. 5.6(top)) leads to poor generalization performance on the test dataset.

5.4.3 Information leakage from obfuscated DL model

A major challenge for HPNN framework is to ensure that a locked DL model doesn't leak any significant information related to the input-output mapping of the owner's DL model, beyond what can be exploited by the attacker using the *thief* dataset. In order to experimentally quantify the information leakage from an obfuscated DL model we performed two types of fine-tuning attacks under

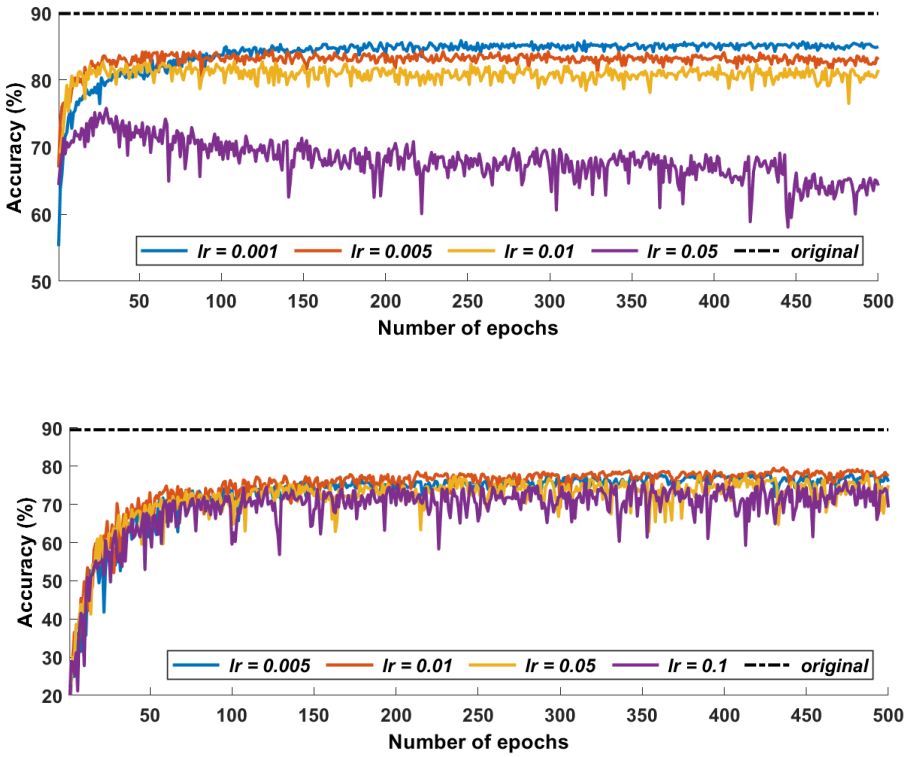


Figure 5.6: Effect of learning rate (lr) on fine-tuning **(top)** dataset:Fashion-MNIST, network:CNN1 **(bottom)** dataset:CIFAR-10, network:CNN2

the same hyperparameter settings (i) **Random fine-tuning** approach where we initialized the *baseline* DNN architecture with random *small* weight parameters and (ii) **HPNN fine-tuning** approach where we initialized the *baseline* DNN with the obfuscated DL model's weight parameters. The intuition behind such an experiment being that if the accuracy achieved by random fine-tuning and HPNN fine-tuning attacks are *similar*, then the obfuscated DL model doesn't leak any significant information related to the owner's DL model. The experimental outcomes for such fine-tuning attacks across different benchmark datasets (using a *thief* dataset fraction α=10%) are presented in the last four subcolumns of Table 5.1. We observe that both types of fine-tuning attacks could achieve accuracy levels

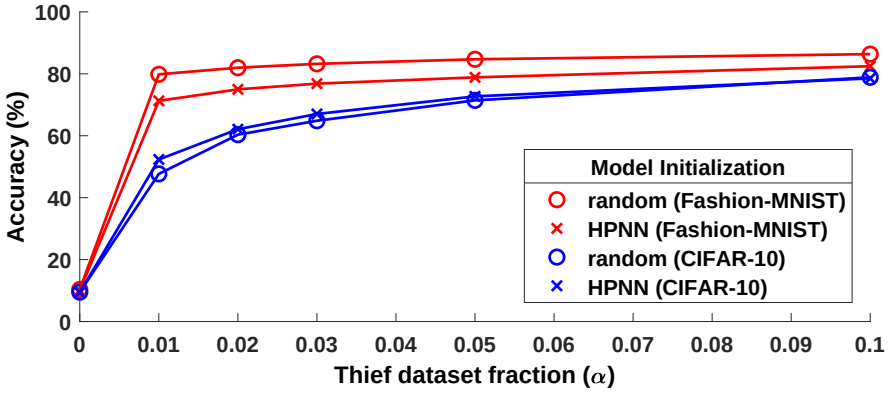


Figure 5.7: Impact of *thief* dataset size on fine-tuning attack.

which are significantly lower than the original accuracy obtained by the DL model owner. Also, both the attacks perform quite *similarly* in terms of the final accuracy achieved across different datasets. This indicates that initializing the network using weight parameters of an obfuscated DL model (which is trained on the *entire* annotated training dataset) doesn't provide any advantage compared to random weight initialization for performing fine-tuning attack.

We further investigated the effect of available *thief* dataset size on these two types of fine-tuning attacks. As we can observe from the experimental results reported in Fig. 5.7, both random and HPNN fine-tuning attacks perform very closely across different α values on the datasets considered. Note that α=0% corresponds to the scenario where the attacker doesn't possess any *thief* dataset to perform model fine-tuning. The accuracy trends signify that the success of attacker is limited by the size of the available *thief* dataset, irrespective of the

weight initialization used. Therefore, our proposed HPNN framework successfully thwarts IP theft attempts of DL models by unauthorized end-users even under a strong threat model which considers model fine-tuning attacks.

5.5 DynaMarks: Dynamic Watermarking to Defend Against Model Extraction Attacks

In this section, we consider the problem of IP security of DL models in a scenario where an authorized end-user attempts to steal the functionality of a well-trained model via *model extraction* attack. In a typical model extraction attack, an adversary (an authorized end-user in this case) queries the original or *victim* model (stored in the trusted hardware device) with inputs of her choice and uses the prediction responses to label a substitute dataset. Subsequently, the attacker uses this substitute dataset to train a *surrogate model* that replicates the functionality of the victim model [102, 70, 69]. We propose DynaMarks, a black-box watermarking technique to defend against such model extraction attacks on proprietary DL models deployed in edge devices. DynaMarks embeds watermark by dynamically changing the responses of the model's prediction API during the inference phase using low-cost hardware, without introducing any computational overhead in the training process.

5.5.1 Background

5.5.1.1 Model Extraction Attacks

A DNN classifier is a function $\mathcal{F} : \mathbb{R}^M \rightarrow \mathbb{R}^N$, where M and N are the number of input features and output classes respectively. The output of $\mathcal{F}(x)$ on input x is an N-dimensional vector $\overrightarrow{p_x}$ containing probabilities p_x^j that x belongs to class c_j for $j \in [N]$[2]. The predicted class $\mathcal{C}$ corresponds to the output component with maximum value as obtained by applying *argmax* function: $\mathcal{C} = argmax\ \mathcal{F}(x)$. In practice, the DNN classifier is trained using a massive annotated dataset along with an optimized set of hyperparameters such that $argmax\ \mathcal{F}(x)$ approximates the oracle function $\mathcal{O}$ which outputs the true class label for any input sample $x \in \mathbb{R}^M$.

Several model extraction attacks [67, 20] against machine learning models (including complex DNNs) have been proposed in recent literature which pose a major threat to the IP rights of their owners. In a model extraction attack, the objective of an attacker is to steal the functionality of a well-trained network $\mathcal{F}_{org}$ (referred to as the original or victim model) by querying it with a set $\mathbb{Q}$ of input queries and obtaining the corresponding set of predicted output probabilities $\mathcal{F}_{org}(\mathbb{Q})$. The attacker uses this information to train a surrogate model $\mathcal{F}_{sm}$ such that its accuracy is close to that of $\mathcal{F}_{org}$ on a test dataset, thus depriving the DL model owner of his business advantage. Typically, a model extraction attack is performed in black-box setting, i.e., the attacker doesn't have any knowledge of the weight parameters of $\mathcal{F}_{org}$, but has access to its prediction API which returns the

[2] $[N]$ denotes the set of first N natural numbers.

output probabilities for a given input query. The challenges associated with such an attack strategy are (i) lack of availability of annotated training data that comes from the same distribution as the data used to train $\mathcal{F}_{org}$ and (ii) no knowledge of the architecture of $\mathcal{F}_{org}$ or its training process. In this work, we focus on watermarking based approaches to defend against model extraction attacks on proprietary DL models.

5.5.1.2 Black-box DNN watermarking

Digital watermarking is a popular technique utilized to covertly embed a secret marker into the cover data such as images, videos, or audios. It enables free sharing of digital content, while providing proof of ownership of the cover data. Extension of watermarking approaches to deep learning offers an effective solution to defend against model theft by allowing the owner to claim IP rights upon inspection of a suspected stolen model. Several existing black-box DNN watermarking approaches [17, 80, 120] consist of *overfitting* a model $\mathcal{F}_{org}$ to outlier input-output pairs (known only to the DL model owner). Such watermarking techniques are based on the concept of backdoor insertion [17, 42] using a trigger set. If the DL model owner encounters a model which exhibits targeted misclassifications on this trigger set that was encoded by the watermark, then the owner can reasonably claim that the model is a stolen copy of $\mathcal{F}_{org}$.

Entangled Watermarks. In [49], the authors demonstrated a fundamental limitation of conventional DNN watermarking schemes based on outlier input-output pairs in the context of model extraction attack. In order to perform model extraction, an attacker does not directly steal the original model, but rather trains a surrogate model by using the information obtained by querying the original model. If the attacker queries a watermarked model $\mathcal{F}_{org}$ using inputs which are sampled from the task distribution, then the obtained surrogate model $\mathcal{F}_{sm}$ will only learn the victim model's decision surface relevant to the task distribution and will not retain the decision surface relevant to watermarking. Subsequently, the authors propose a new technique called entangled watermarks which trains a DL model to learn features common to both task distribution and watermark data by formulating a new loss function. However, this altered training process incurs a substantial increase (about $2\times$ compared to baseline model) in computational overhead. Moreover, the success of entangled watermarks scheme is heavily dependent on the training dataset as well as on hyperparameter tuning. In fact, such a watermarked model will incur severe performance degradation if the hyperparameters are not carefully selected, leading to a decrease in model utility.

DAWN. In [99], the authors propose a technique called DAWN which does not impose any alterations to the training process but selectively changes the responses of a model's prediction API in order to watermark a fraction of input queries. These watermarked queries then serve as a trigger set for a surrogate model $\mathcal{F}_{sm}$ trained using the API responses of the victim model $\mathcal{F}_{org}$. Unlike prior backdoor insertion based watermarking schemes, DAWN is resilient to model extraction attacks as

all the input queries including those which are watermarked belong to the task distribution, i.e., no outlier inputs are present in the trigger set. Although an effective technique to deter model extraction, the effectiveness of DAWN is limited to client-server model where a malicious client (attacker) submits queries to $\mathcal{F}_{org}$ hosted by the DL model owner using a trusted server. Such a scheme will not be applicable in scenarios where the model owner has no knowledge of the input queries made by the attacker, e.g., DL models deployed in remote edge devices. Also, returning false predictions with the objective of embedding watermarks can be unacceptable for certain applications, e.g., malware detection, medical applications such as cancer diagnosis, etc. [20, 105]. Moreover, DAWN does not secure against model extraction attack which utilizes several similar queries (multiple close images) where only one is assigned a false label [64]. This strongly motivates the need to develop an effective watermarking scheme which addresses the above drawbacks.

5.5.2 Problem Description

In recent years, there is a growing trend toward transition of the inference phase of deep learning to edge devices [60, 106]. This leads to improved user experience with reduction in inference time (low latency), less dependency on network connectivity, and increased energy efficiency of resource-constrained mobile devices. In addition, running inference on the edge also enables several deep learning services, e.g. Instagram features that involve real-time application of machine learning algorithms at image capture time [106]. In this new paradigm of edge

intelligence, an attacker can directly query a proprietary DL model deployed in an edge device without any need to redirect the queries to a trusted cloud server, thus rendering model protection countermeasures like DAWN [99] practically useless. Even the utilization of a trusted hardware in the edge as proposed in [28] will not be effective for protecting DL model IPs as any end-user possessing an authorized hardware will be able to mount model extraction attack. Therefore, the emerging trend of executing deep learning inference on edge devices poses major challenges to the security of well-trained DL models from IP infringement attempts. In this work, we propose a novel accuracy-preserving watermarking approach called DynaMarks as an effective IP protection mechanism for DL models deployed in edge devices.

Watermarking Requirements. The following requirements should be addressed while designing an effective black-box watermarking scheme for proprietary DL models deployed in edge devices:

- **Fidelity.** The performance or accuracy of the original DNN classifier should not degrade due to watermark embedding.
- **Robustness.** The embedded watermark should exhibit resiliency against model modifications such as compression/pruning and provide high detection confidence for proving model ownership.
- **Imperceptibility.** The watermark should not leave tangible footprints in the model, thus hindering any unauthorized detection.

The above set of requirements have also been considered in previous works on black-box DNN watermarking [17, 80, 120, 66]. In addition, we consider the following important watermarking requirement for defending against model extraction attacks.

- **Transferability.** The embedded watermark should survive model extraction attack, i.e., the watermark should get transferred from the original DL model to a surrogate model obtained from it.

Threat Model. The objective of an attacker is to extract the functionality of a well-trained DL model $\mathcal{F}_{org}$ without its watermark [49]. In practice, the attacker is data-limited [53, 68] and does not have sufficient number of inputs representative of the training set of $\mathcal{F}_{org}$. In this work, we assume an adversary with the following capabilities:

(i) access to γ fraction of the training data of $\mathcal{F}_{org}$ but not its labels, constituting the attacker's input query set $\mathbb{Q}$

(ii) knowledge of the network architecture of $\mathcal{F}_{org}$

(iii) ability to query $\mathcal{F}_{org}$ with any input sample and obtain the output probability for each class (black-box setting)

(iv) knowledge that $\mathcal{F}_{org}$ is watermarked but does not know the details of the watermarking procedure.

In addition, we also assume that $\mathcal{F}_{org}$ is deployed in a remote edge device and the

DL model owner does not have any influence on the query strategy or the training process adopted by the attacker to obtain the surrogate model $\mathcal{F}_{sm}$. Furthermore, the attacker is not constrained by memory or computational capabilities.

5.5.3 Proposed DynaMarks Technique

In this subsection, we present a new black-box watermarking technique called DynaMarks to defend against model extraction attacks on proprietary DL models. In order to perform model extraction, the attacker queries the original model with inputs from her query set $\mathbb{Q}$ and utilizes the predicted output probabilities to train a surrogate model. Previous works [102, 59] have shown that using output probabilities instead of labels drastically reduces (about 50-100×) the number of queries required to extract the model and also, improves the attack convergence as well as increases the converged model accuracy. The objective of our proposed DynaMarks scheme is to smartly alter these output probabilities in order to watermark a surrogate model without sacrificing the prediction accuracy of the original DL model.

5.5.3.1 Watermark Embedding

As per our threat model, the primary challenge for DynaMarks technique is to embed watermark into the original model $\mathcal{F}_{org}$ running on an edge device in such a way that during model extraction the watermark gets transferred to the surrogate model $\mathcal{F}_{sm}$. Since the DL model owner does not have any control over the input queries used by the attacker, she can no longer utilize the notion of a

trigger set for embedding watermark as adopted in several prior approaches [17, 80, 120, 99]. Instead, DynaMarks dynamically alters the output probabilities of $\mathcal{F}_{org}$ based on certain secret parameters at model inference runtime with the objective of watermarking $\mathcal{F}_{sm}$. Next, we present the details of the watermark embedding process.

Let us denote the output probability vector of $\mathcal{F}_{org}(x)$ for an input x by $\vec{p_x}$. Note that each component p_x^i of the vector $\vec{p_x}$ corresponds to the probability value predicted by the model for the i^{th} class, $i \in [N]$. Now, for every i^{th} class, the DL model owner defines a **secret** vector $\vec{V_i} = (v_i^1, v_i^2, \cdots, v_i^N)$ of length N such that the values of its elements specify the selection probabilities of the corresponding indices, e.g., the probability of selecting the j^{th} index of $\vec{V_i}$ is v_i^j. The process of embedding watermark into the model $\mathcal{F}_{sm}$ by utilizing these secret vectors is summarized in Algorithm 5.1. In order to extract the functionality of $\mathcal{F}_{org}$, the attacker uses samples from the input query set $\mathbb{Q}$ to query $\mathcal{F}_{org}$ and builds a substitute dataset D_{sub} with the returned responses (output probabilities). For each input $x \in \mathbb{Q}$, DynaMarks alters the output probability vector $\vec{p_x}$ as a function of vectors $\vec{V_i}$ as follows (lines 4-11 of Algorithm 5.1): If the i^{th} component of $\vec{p_x}$ has the maximum value, i.e., $argmax\ \vec{p_x} = i$, and the maximum value p_x^i lies within a certain range (α_i, β_i), then (i) generate a random variable Δp that follows a certain distribution, say a uniform distribution $U(a_i, b_i)$ (ii) select an index j from the vector $\vec{V_i}$ (iii) alter the probabilities of component pair (p_x^i, p_x^j) of the vector $\vec{p_x}$ by transferring an amount of Δp from p_x^i to p_x^j, i.e., $p_x^i = p_x^i - \Delta p$ and $p_x^j = p_x^j + \Delta p$. Note that such alteration to vector $\vec{p_x}$ does not affect the sum of its components which still equals 1. In

Algorithm 5.1: Watermark Embedding on Extracted Model

Input: (i) Network $\mathcal{F}_{org}$ initialized with weights $\mathcal{W}_{org}$

(ii) Vectors $\vec{V_i}$ and parameters (α_i, β_i, a_i, b_i), $\forall i \in [N]$

(iii) Attacker's input query set $\mathbb{Q}$

Output: Watermarked surrogate model $\mathcal{F}_{sm}$

`/*building substitute dataset*/`

$D_{sub} = \phi$

for $x \in \mathbb{Q}$ **do**

  $\overrightarrow{p_x} = \mathcal{F}_{org}(x)$

  if $argmax\ \overrightarrow{p_x} == i\ and\ p_x^i \in (\alpha_i, \beta_i)$ **then**

    Generate random variable $\Delta p \sim U(a_i, b_i)$

    Select an index $j \in N$ from $\vec{V_i}$

    `/*alter component pair` (p_x^i, p_x^j)`*/`

    $p_x^i = p_x^i - \Delta p$

    $p_x^j = p_x^j + \Delta p$

  end

  $D_{sub} = D_{sub} \cup (x, \overrightarrow{p_x})$

end

`/*watermark transferability*/`

$\mathcal{W}_{sm} \leftarrow$ Train $\mathcal{F}_{sm}$ using D_{sub}

return $\mathcal{F}_{sm}$

addition to the vectors $\vec{V_i}$, the parameters (α_i, β_i, a_i, and b_i), $\forall i \in [N]$, are also **secrets** chosen by the DL model owner. For a given set of inputs, such probabilistic changes in the output responses of $\mathcal{F}_{org}$ based on the secret parameters leads to a set of altered probability distributions over the set of outputs which constitute the watermark in our proposed DynaMarks scheme. For notational simplicity, we refer to this altered version of $\mathcal{F}_{org}$ as $\mathcal{F}_{alt}$.

When the attacker uses the responses of $\mathcal{F}_{alt}$ to the input query set $\mathbb{Q}$ for composing the substitute dataset D_{sub} and subsequently, trains a surrogate model $\mathcal{F}_{sm}$ using it, we expect the watermark to get transferred to $\mathcal{F}_{sm}$. This is because the extracted model $\mathcal{F}_{sm}$ tries to replicate the secret-dependent functionality of $\mathcal{F}_{alt}$ which maps a set of inputs to a set of altered probability distributions over the set of outputs. The above methodology of embedding watermark into $\mathcal{F}_{sm}$ doesn't require the DL model owner to have any knowledge of the attacker's query strategy. Hence, the proposed DynaMarks technique provides an effective IP security solution against model extraction attacks on proprietary DL models deployed in edge devices. Also, for a given dataset and network architecture, the model owner can choose the secret parameters in such a manner that the accuracy of the original model $\mathcal{F}_{org}$ is preserved.

Algorithm 5.2: Watermark Verification

Input: (i) Black-box access to network $\mathcal{F}_{sm}$ (ii) White-box access to networks $\mathcal{F}_{org}$ and $\mathcal{F}_{alt}$ (iii) Vectors $\vec{V_i}$ and parameters (α_i, β_i, a_i, b_i), $\forall i \in [N]$ (iv) Verification query set $\mathbb{V}$ (v) Watermark detection threshold τ

Output: Decision of watermark detection in $\mathcal{F}_{sm}$

/*Form NxN response distributions*/
Initialize response matrices R_{ij}^{sm}, R_{ij}^{org}, R_{ij}^{alt}, $\quad \forall i, j \in [N]$
for $(x, y) \in \mathbb{V}$ **do**
    $\vec{s_x} = \mathcal{F}_{sm}(x)$, $\vec{p_x} = \mathcal{F}_{org}(x)$, $\vec{q_x} = \mathcal{F}_{alt}(x)$
    for $j \in [N]$ **do**
        Append s_x^j, p_x^j, q_x^j to R_{yj}^{sm}, R_{yj}^{org}, R_{yj}^{alt}
    end
end
D_{ij}^{sm}, D_{ij}^{org}, $D_{ij}^{alt} \leftarrow$ Create distributions of R_{ij}^{sm}, R_{ij}^{org}, R_{ij}^{alt}
/*Calculate distance metrics*/
$\delta_{sm}^{org} \leftarrow 0$, $\delta_{sm}^{alt} \leftarrow 0$
for $i \in [N]$ **do**
    for $j \in [N]$ **do**
        $\delta_{sm}^{org} = \delta_{sm}^{org} + \text{JSD}(D_{ij}^{org} \parallel D_{ij}^{sm})$
        $\delta_{sm}^{alt} = \delta_{sm}^{alt} + \text{JSD}(D_{ij}^{alt} \parallel D_{ij}^{sm})$
    end
end
/*Determine if model is watermarked*/
$\eta = \delta_{sm}^{org} \ / \ \delta_{sm}^{alt}$
if $\eta > \tau$ **then**
    return Watermark Detected in $\mathcal{F}_{sm}$
end

5.5.3.2 Watermark Verification

In order to verify the presence of watermark in $\mathcal{F}_{sm}$, the DL model owner compares the distributions of output responses of the models $\mathcal{F}_{sm}$ and $\mathcal{F}_{alt}$ using a verification query set $\mathbb{V}$. An element of the set $\mathbb{V}$ consists of a tuple (x, y), where x denotes an input query and y denotes its known output label. In our experiments (details later in section 5.5.4), we consider the entire test set of a benchmark dataset as the verification query set $\mathbb{V}$. If the model $\mathcal{F}_{sm}$ is stolen from $\mathcal{F}_{alt}$ using model extraction, we expect the distributions of their output responses to be *similar* as both of them will be functions of the secret parameters used to embed the watermark. The process of watermark detection in a suspected surrogate model $\mathcal{F}_{sm}$ is summarized in Algorithm 5.2. In the first phase (lines 1-9), the DL model owner defines a data structure called *response matrix* R^{sm} (corresponding to model $\mathcal{F}_{sm}$) of dimension NxN whose each element $R^{sm}_{ij}, \forall i, j \in [N]$, is a variable-length list. The process of populating the elements of R^{sm} is as follows: The model $\mathcal{F}_{sm}$ is queried with an input x from the set $\mathbb{V}$ to obtain an output probability vector $\vec{s_x}$ in black-box setting. If the actual label corresponding to input x is y, then the components s^j_x of $\vec{s_x}$, $\forall j \in [N]$, are appended to the respective lists R^{sm}_{yj} along the y^{th} row of matrix R^{sm}. The repetition of this step for all the elements of set $\mathbb{V}$ results in clustering of the output responses of the model $\mathcal{F}_{sm}$ according to their actual class labels along the rows of the matrix R^{sm}. Subsequently, the DL model owner forms a *response distribution* D^{sm} of dimension NxN using the response matrix R^{sm} by creating individual probability distributions D^{sm}_{ij} from the contents of the corresponding list

R_{ij}^{sm}, $\forall i, j \in [N]$. Similarly, the response distributions D^{org} and D^{alt} are also formed by querying the models $\mathcal{F}_{org}$ and $\mathcal{F}_{alt}$ respectively using the *same* verification set $\mathbb{V}$. Note that the model owner has white-box access to both $\mathcal{F}_{org}$ and its altered version $\mathcal{F}_{alt}$ for constructing their respective response distributions.

In the second phase (lines 10-17), the DL model owner calculates a distance metric δ_{sm}^{org} by iteratively adding the Jensen-Shannon divergence[3] JSD($D_{ij}^{org} \parallel D_{ij}^{sm}$) between the probability distributions D_{ij}^{org} and D_{ij}^{sm}, $\forall i, j \in [N]$. Similarly, another distance metric δ_{sm}^{alt} is also obtained by considering the response distributions D^{alt} and D^{sm}. In the final phase (lines 18-22), the DL model owner calculates a parameter η which is the ratio of δ_{sm}^{org} to δ_{sm}^{alt}. If the value of the parameter η is greater than a certain threshold τ (empirically determined after experimental evaluations), then the presence of watermark is detected in the model $\mathcal{F}_{sm}$. The rationale behind such a decision being that the response distribution D^{sm} will be more *similar* to D^{alt} (smaller δ_{sm}^{alt}) than compared to D^{org} (larger δ_{sm}^{org}) if $\mathcal{F}_{sm}$ is trained using the output responses of $\mathcal{F}_{alt}$. This will lead to a large value of parameter η (greater than τ) implying that $\mathcal{F}_{sm}$ is indeed extracted from $\mathcal{F}_{alt}$. As evident from the above discussion, the success of such watermark detection in DynaMarks scheme strongly depends on how effectively the dynamically generated watermark gets transferred from the output responses of $\mathcal{F}_{alt}$ to the output responses of $\mathcal{F}_{sm}$ due to model extraction.

[3]Jensen-Shannon divergence JSD($P \parallel Q$) is a metric for measuring the similarity between two probability distributions P and Q.

Implementation efficiency. The proposed DynaMarks technique can be implemented in edge devices using low-cost hardware components with negligible impact on the performance of deployed DL models. The alterations in the output probabilities of a model can be easily performed using hardware implementations of pseudo-random number generators along with standard digital adder/subtractor circuits. The incorporation of such simple hardware designs will have insignificant effects on the model inference time as well as on the energy-efficiency of an edge device. The secret parameters used in the DynaMarks scheme can be stored securely in a tamper-proof chip such as Trusted Platform Module (TPM) [15] embedded into the edge device.

Dataset	**Fidelity (accuracy)**		**Averaging Attack**			**Pruning ($\kappa = 10\%$)**			**Different Architectures**		
	$\mathcal{F}_{org}$	$\mathcal{F}_{alt}$	δ_{sm}^{org}	δ_{sm}^{alt}	η	δ_{sm}^{org}	δ_{sm}^{alt}	η	δ_{sm}^{org}	δ_{sm}^{alt}	η
Fashion MNIST	90.72%	90.72%	45.32	17.99	**2.52**	41.83	16.03	**2.61**	73.72	28.71	**2.57**
CIFAR-10	84.98%	84.98%	41.07	16.02	**2.56**	33.25	14.63	**2.27**	34.56	17.94	**1.93**

Table 5.2: Evaluating the fidelity and robustness properties of DynaMarks scheme.

5.5.4 Evaluations

5.5.4.1 Experimental Setup

Datasets. We evaluate DynaMarks technique on two popular image datasets, Fashion MNIST [5] and CIFAR-10 [2], using PyTorch 1.7 framework. The classification tasks on these datasets are much more complex compared to the classic MNIST dataset, e.g. models achieving 99% accuracy on MNIST only attain about 90% accuracy on Fashion MNIST dataset. Fashion MNIST dataset consists of $70,000$ samples (training set of $60,000$ examples and test set of $10,000$ examples) of 28x28 grayscale images, whereas CIFAR-10 dataset consists of $60,000$ samples (training set of $50,000$ examples and test set of $10,000$ examples) of 32x32 colour images. For both these datasets, an image is associated with a label from 10 classes (N=10).

Network Architectures. In order to obtain the well-trained model $\mathcal{F}_{org}$, we use Convolution Neural Networks (CNNs) for both the datasets. In case of Fashion MNIST dataset, the architecture is composed of 2 convolution layers with 16 and 32 5x5 kernels respectively with 2x2 max pooling operations, followed by a fully-connected layer. For CIFAR-10 dataset, the architecture is composed of 6 convolution layers with 32, 64, 128, 128, 256, and 256 3x3 kernels respectively with 2x2 max pooling operations, followed by three fully-connected layers. Both the networks were trained to attain satisfactory performance on the image classification tasks considered.

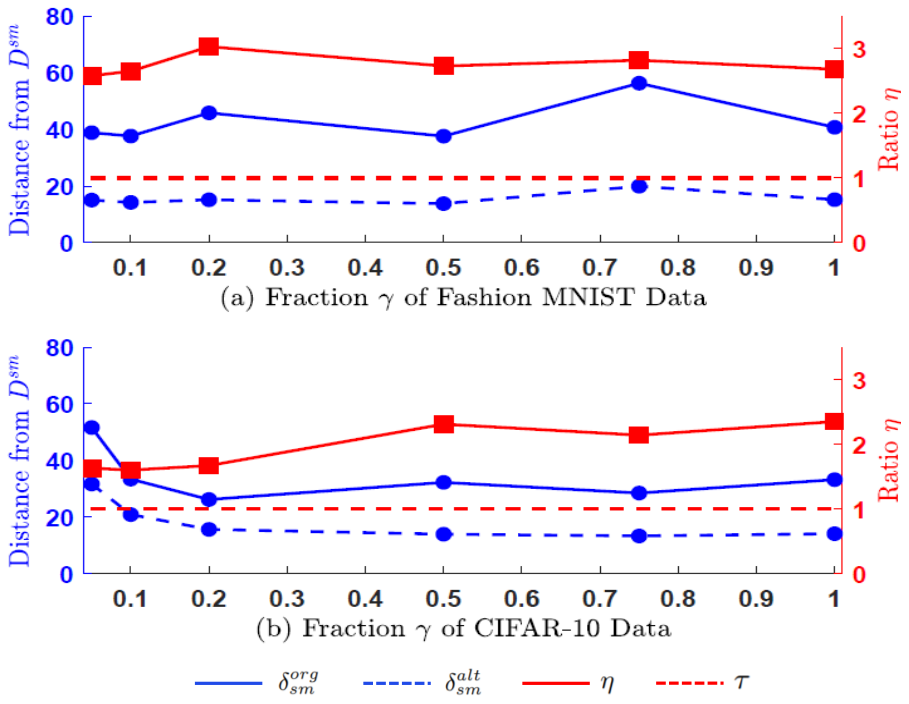


Figure 5.8: Transferability of DynaMarks for different training fractions.

5.5.4.2 Validating DynaMarks

Next, we perform experiments to evaluate the effectiveness of our proposed DynaMarks approach in the context of the watermarking requirements stated in section 5.5.2.

Impact on accuracy (Fidelity requirement). The accuracy of a well-trained model $\mathcal{F}_{org}$ should not degrade due to watermark embedding as otherwise its utility will be impacted [59]. In DynaMarks scheme, a DL model owner has the flexibility to calibrate the secret parameters (vectors $\vec{V_i}$ along with α_i, β_i, a_i, and b_i, $\forall i \in [N]$) such that the accuracy of the original DNN classifier is preserved. In our experiments, we construct each $\vec{V_i}$ by setting the selection probability of a single index (randomly selected from $[N]$) to 2/11 and the selection probabilities of the remaining indices

to 1/11. Also, we set $\alpha_i = 0.9$, $\beta_i = 1$, $a_i = 0.01$, and $b_i = 0.19$, $\forall i \in [N]$, for both the datasets such that $\mathcal{F}_{alt}$ exhibits accuracy-preserving outcomes as reported in the second and third subcolumns of Table 5.2.

Detectability in surrogate model (Transferability requirement). A very important criteria for designing watermarking schemes for proprietary DL models deployed in edge devices is the property of watermark transferability from the well-trained model $\mathcal{F}_{alt}$ to a surrogate model $\mathcal{F}_{sm}$ which is extracted from the former. In fact, the watermark verification process of the DynaMarks scheme (as outlined in Algorithm 5.2) is also strongly dependent on the detectability of the watermark at the output responses of $\mathcal{F}_{sm}$. In Fig. 5.8, we study the transferability property of DynaMarks by evaluating the parameter η (shown using solid red line) across different fractions γ of the training data available to the attacker. For both the datasets, we used the same CNN architecture for $\mathcal{F}_{alt}$ and $\mathcal{F}_{sm}$ and considered the entire test set as the verification set $\mathbb{V}$. In the subfigures, we also plot the variations of the distance metrics δ_{sm}^{org} and δ_{sm}^{alt} using solid and dashed blue lines respectively. We observe that the parameter η is always greater than the threshold $\tau = 1$ (no false negatives), implying that there is strong watermark transferability from the output responses of $\mathcal{F}_{alt}$ to the output responses of $\mathcal{F}_{sm}$ due to model extraction. In order to assess the false positive rate in our watermark verification process, we trained a benign model $\mathcal{F}_{bm}$ using 50% of the original training dataset and recalculated the parameter η by taking the ratio of the distances of the output response distribution of $\mathcal{F}_{bm}$ from those of $\mathcal{F}_{org}$ and $\mathcal{F}_{alt}$. In this case, we found that η is always lesser than the threshold $\tau = 1$ ($\eta = 0.19$ for Fashion MNIST and $\eta = 0.54$ for CIFAR-10), implying

that there is no false positives in the watermark detection process. The value of the threshold τ was chosen empirically after performing experimental evaluations on the datasets. In summary, DynaMarks exhibits strong watermark transferability as required from an ideal watermarking scheme to defend against model extraction attacks.

Resiliency to model modifications (Robustness requirement). We analyze the robustness of DynaMarks scheme against the following types of watermark removal strategies.

(i) *Averaging Attack.* If an attacker queries $\mathcal{F}_{alt}$ repeatedly using the same input, then according to Algorithm 5.1 she will get different output probabilities for different input queries as the model response is dependent on a couple of randomness factors (lines 6 and 7 of Algorithm 5.1). The attacker can then calculate the average of such output probabilities to get a representative response for an input which are used to populate the substitute dataset D_{sub} for training $\mathcal{F}_{sm}$. The attacker succeeds if the obtained surrogate model $\mathcal{F}_{sm}$ does not retain any information pertaining to the watermark. In order to evaluate the outcome of such an attack for a data fraction $\gamma = 1$ (entire training data), we queried $\mathcal{F}_{alt}$ repeatedly using the same input for 100 times to get a representative sample in the substitute dataset D_{sub}. Then, we obtained the distance metrics δ_{sm}^{org} and δ_{sm}^{alt} as reported in subcolumns 4 and 5 of Table 5.2. We observe that their ratio η (see subcolumn 6) is still greater than the watermark detection threshold $\tau = 1$, implying that DynaMarks is resistant to such averaging attack.

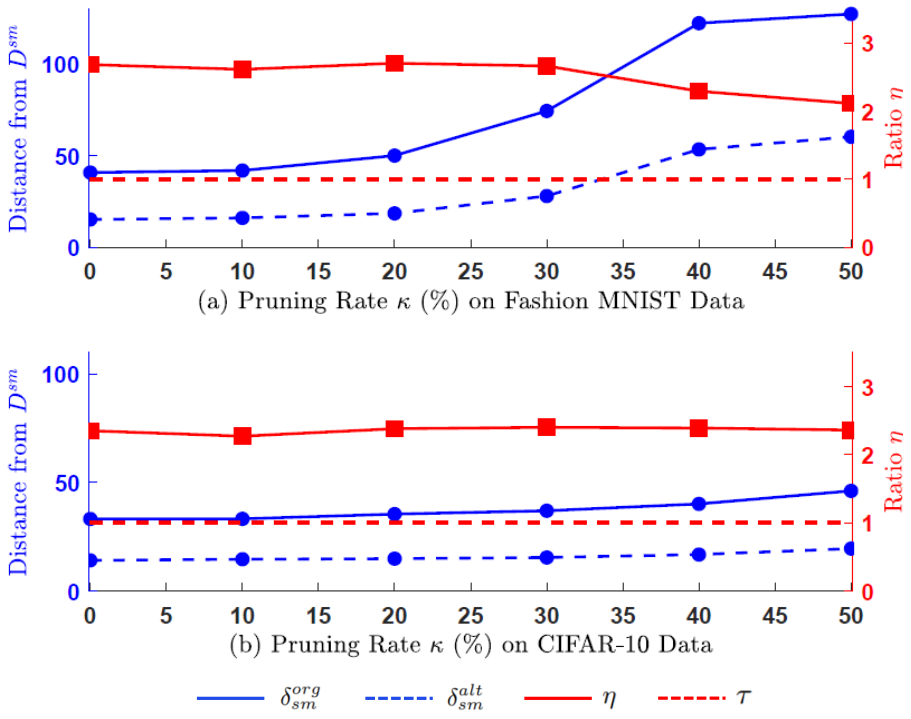


Figure 5.9: Transferability of DynaMarks for different pruning rates.

(ii) *Model Compression.* The attacker can also adopt a model pruning approach to compress the surrogate model $\mathcal{F}_{sm}$ with the objective of removing the watermark [49, 80]. In our experiments, to perform model compression, we eliminated the lowest κ% of the network connections across all the layers of $\mathcal{F}_{sm}$ (trained using data fraction $\gamma = 1$) using the global pruning method of PyTorch framework. We report the outcomes of such model pruning with a pruning rate $\kappa = 10\%$ in subcolumns 7-9 of Table 5.2. In this case also we observe that the watermark detection parameter η is above the threshold $\tau = 1$ for both the datasets, highlighting the robustness of DynaMarks scheme against model compression. We further varied the pruning rate κ from 10% up to 50%; we find in Fig. 5.9 that even then the watermark can be successfully detected ($\eta > \tau$) across all the pruning rates. It is to be noted

that high proportions of pruning result in significant accuracy degradation of the extracted model $\mathcal{F}_{sm}$ with negligible impact on the detection parameter η, e.g., in case of Fashion MNIST dataset the model accuracy drops by 11.17% with $\kappa = 50\%$ compared to the uncompressed model accuracy even though for the pruned model the parameter $\eta = 2.11$ remains well above the threshold $\tau = 1$.

(iii) *Different architectures.* All the previous set of experiments were performed by keeping the same CNN architecture for the victim and the surrogate models. In order to study the influence of the choice of architecture on DynaMarks technique, we use Resnet-18 [45] network for obtaining a surrogate model $\mathcal{F}_{sm}$ from a CNN-based victim model $\mathcal{F}_{alt}$ with a data fraction $\gamma = 1$. From the last three subcolumns of Table 5.2, we observe that even in this setting the watermark detection parameter η is sufficiently greater than the threshold $\tau = 1$ for both the image datasets considered. This highlights the fact that the watermark generated by altering the responses of the victim DL model using Algorithm 5.1 transfers to the responses of the surrogate model $\mathcal{F}_{sm}$ irrespective of its architectural choice. This is because $\mathcal{F}_{sm}$ replicates the input-output mapping of the model $\mathcal{F}_{alt}$ (which is a function of the secret parameters chosen by the DL model owner) provided that the network architecture of $\mathcal{F}_{sm}$ is sufficiently complex.

Imperceptibility requirement. Unlike prior black-box watermarking approaches [49, 99], DynaMarks does not utilize any trigger inputs to detect watermark in a surrogate model $\mathcal{F}_{sm}$ obtained using model extraction attack. Hence, state-of-the-art backdoor detection schemes such as Neural Cleanse [104] are not applicable to DynaMarks which generates watermark by dynamically altering the responses of

the prediction API of a protected DL model at inference runtime. Also, it seems very unlikely that a data-limited attacker will be able to revert back such alterations without any notion of the secret parameters used in the watermark generation process. As future work, we plan to investigate the security offered by DynaMarks technique against steganalysis and watermark overwriting attacks.

5.6 Conclusion

In this chapter, we proposed a lightweight obfuscation framework called HPNN for IP protection of DL models. In this framework, a DL model owner utilizes a novel *key-dependent* backpropagation algorithm to train a network such than only an authorized end-user who possesses a trusted hardware (with the *secret* HPNN key embedded on-chip) will be able to effectively run the DNN inference phase. The experimental outcomes across different benchmark datasets and DNN architectures highlight the fact that any unauthorized usage of such locked DL models will lead to a substantial degradation of the model prediction accuracy. In addition, we also performed extensive evaluations to demonstrate the robustness of obfuscated DL models (trained using HPNN framework) against model fine-tuning attacks.

We also presented a novel watermarking approach called DynaMarks as an effective IP security solution against model extraction attacks performed by authorized end-users in the HPNN framework. Unlike existing defenses, DynaMarks does not introduce any computational overhead in the model training phase nor does it sacrifice the victim model's prediction accuracy to gain security benefits. The

experimental outcomes on Fashion MNIST and CIFAR-10 datasets demonstrate the effectiveness and robustness of DynaMarks technique against different types of watermark removal strategies.

Chapter 6: Conclusion and Future Research Directions

This dissertation focuses on developing obfuscation based design techniques to enhance hardware-oriented security and trust at multiple levels of design abstractions. These approaches can be utilized to build trust between the IC design houses and the fabrication companies which are located in different parts of the world.

6.1 Conclusion

In Chapter 1, we presented an overview of different hardware-based security threats in the various stages of an IC design's life cycle. We outlined several design-for-trust approaches as proposed in the related literature to protect the IPs of hardware implementations from a malicious foundry as well as end-users. In addition, we also discussed about existing logic obfuscation based security schemes which aim to protect the IPs of hardware designs by introducing key-gates in their synthesized netlists.

In Chapter 2, we evaluated the security of delay locked designs at the circuit-level of design abstraction. We developed a novel SAT formulation based attack approach called TimingSAT to deobfuscate the functionalities of such delay locked designs. TimingSAT attack works in two stages: In the first stage, the attacker

finds the functional key using conventional SAT attack approach and in the second stage, the attacker finds the delay key using a timing profile embedded SAT formulation of the circuit. We performed experiments across several benchmark circuits to demonstrate the effectiveness of the TimingSAT attack to break delay-locked benchmarks within a few hours. Subsequently, we proposed a countermeasure called stripped-functionality delay locking (SFDL) which not only thwarts TimingSAT attack but also resists all known attacks against circuit-level obfuscation techniques. The security guarantees of the proposed countermeasure were validated with extensive experimental evaluations. In Chapter 3, we investigated the susceptibility of conventional circuit-level obfuscation schemes to side-channel analysis attacks. We demonstrated how an adversary can adopt a template analysis based side-channel attack to successfully deobfuscate the functionality of a locked circuit using a low number of power side-channel traces.

In Chapter 4, we first highlighted the limitations of circuit-level obfuscation schemes to provide reasonable security guarantees at the architecture-level of design abstraction. We formulated a oracle-less approximate SAT attack against an obfuscated many-core GPU design whose cores were assumed to be locked using state-of-the-art circuit-level logic locking schemes. Such an attack translates the multi-cycle GPU core netlist to a functionally-equivalent single-cycle netlist and utilizes the GPU instruction set architecture to compute distinguishing input-output pairs as required for SAT formulation based attacks. Then, we proposed a countermeasure called cache locking which modifies the cache block replacement policy as a function a secret key such that incorrect keys results in significant drops in cache hit

rates, thus degrading the performance of the applications running on a locked GPU. Subsequently, we proposed a second countermeasure which uses hardware/software co-design based obfuscation approach to provably safeguard the IPs of hardware accelerator designs from an untrusted foundry. The attack resiliency of such a scheme is manifested by using a sequence of keys to obfuscate instruction encoding for an application. The effectiveness of both the countermeasures were demonstrated using experiments performed on standard benchmark applications.

In Chapter 5, we explored the possibility of using a hardware root of trust to obfuscate the functionalities of well-trained deep learning (DL) models, thus enhancing application-level security. We proposed a framework called Hardware Protected Neural Network (HPNN) in which a deep neural network (DNN) is trained as a function of a secret key and then, the obfuscated DL model is hosted on a public model sharing platform. The HPNN framework ensures that only authorized end-user who possess trusted hardware device (with the secret key embedded on-chip) can run DL applications with high accuracy using the published model. We also provided a theoretical construct of a key-dependent backpropagation algorithm for training a DNN which doesn't sacrifice a model's prediction accuracy to gain security benefits. The outcomes of extensive experimental evaluations across different DNN architectures and benchmark datasets demonstrates the efficacy of the HPNN framework as well as validates the robustness of obfuscated DL models against model fine-tuning type attacks. Subsequently, we addressed the threat posed by an authorized end-user trying to steal a DL model via model extraction attack by proposing a watermarking technique called DynaMarks. DynaMarks embeds

watermark into a surrogate model by dynamically changing the output responses from the original model's prediction API based on certain secret parameters at inference runtime. The integration of DynaMarks scheme with the HPNN framework allows a DL model owner to reliably prove model ownership even under a strong attack scenario.

6.2 Future Research

In this dissertation, we have developed several obfuscation based design techniques to enhance hardware-oriented security and trust at different levels of design abstractions. The following discussion summarizes potential future research directions to extend the work presented in this thesis.

6.2.1 Improved SAT Attack

A strong logic obfuscation scheme must provide reasonable security guarantees against SAT formulation based attacks. In the related literature, the resiliency of logic locking techniques has been analyzed using classical SAT attack approach which utilizes a serial algorithm to iteratively prune out unique subsets of wrong keys till a correct key is found. A potential research direction is to develop an improved version of the SAT attack which uses a parallel algorithm to prune out subsets of wrong keys concurrently and converges to find a correct key in a much

shorter time. An ideal logic obfuscation scheme should provide strong security guarantees against such optimized versions of SAT attack as well as other existing attacks on logic locking while incurring minimal implementation overheads.

6.2.2 Recommender System for Obfuscation

Currently a designer applies a chosen logic locking algorithm to obfuscate the synthesized netlist of a sub-module of a design. This approach will incur substantial area overhead if there is a large number of sub-modules in the design, as typically present in a modern system on a chip (SoC). Therefore, in order to reduce the implementation overhead, the designer needs to judiciously select a subset of possible locations for inserting the locking construction. Such selection of locations is not a trivial task as the designer needs to try several possible combinations to insert key-gates in the synthesized netlist such that the output corruptibility of the entire system-level design is high for wrong key inputs. An interesting future research direction would be to build a recommender system for suggesting probable locations for inserting key-gates in the netlist of a large-scale design such that the obfuscated design is not only secure against SAT/other types of attacks but also exhibits high output corruptibility for incorrect key inputs while incurring minimal implementation overhead. In order to develop such a recommender system, the designer needs to build a library consisting of (i) different locking algorithms (ii) topology information of previously locked netlists and their output corruptibility for wrong key inputs (iii) corresponding resiliency to SAT/other types of attacks

(iv) associated overhead due to introduction of additional locking constructions in the netlist. The end objective of such a recommender system will be to suggest the best possible locations to introduce locking constructions in the synthesized netlist of the entire design which simultaneously provides strong security guarantees, high error rate for wrong key inputs, and minimal implementation overhead.

6.2.3 DNN Obfuscation Using PKI

There is a strong need to develop a secure framework for distribution and deployment of Deep learning (DL) models, especially with the growing use of such models in numerous industrial products and services. In Chapter 5, we proposed HPNN framework which realizes this objective by utilizing hardware root-of-trust which ensures that only authorized end-users can run inference phase of deep neural networks (DNNs) with high accuracy. However, the HPNN framework assumes that all the trusted edge devices use the same obfuscation key (HPNN key) embedded on-chip. In practice, different edge devices may use different keys (securely stored in respective trusted platform modules) and hence, the HPNN framework needs to be adapted to provide an efficient model IP protection mechanism in such a setting. One possible solution is to perform the following steps (i) train a DNN using a reference obfuscation key K_{ref} using the key-dependent backpropagation algorithm to obtain a DL model M_{ref} (ii) use Public Key Infrastructure (PKI) to securely exchange a device-specific obfuscation key K_{device} between the server and the end-user device using Diffie-Hellman protocol (iii) for each neuron of the DNN, if

the associated key bits derived from K_{ref} and K_{device} are different, then flip the sign of all of its incoming weight parameters in M_{ref} to obtain an equivalent DL model M_{device} for the device. It is to be noted that the above solution requires training of only one DL model M_{ref} using key-dependent backpropagation algorithm and the other device-specific models M_{device} can be obtained by adjusting the sign bits of the weight parameters of M_{ref}, thus providing a scalable approach to adapt HPNN framework in a setting where different edge devices uses different keys.

Another interesting future research direction will be to explore the opportunity to extend the HPNN framework in a federated learning setting [61] where several edge devices store and process data locally (only intermediate updates being communicated periodically to a central server) with the end objective of learning a single global DL model. Such a framework should aim to enhance privacy of federated learning with minimal impact on model performance and system efficiency.

Bibliography

[1] AI chip. https://www.gyrfalcontech.ai/solutions/2801s/.

[2] CIFAR-10 dataset. https://www.cs.toronto.edu/ kriz/cifar.html.

[3] Codix-Bk3. https://www.codasip.com/risc-v-processors.

[4] DexGuard. https://www.guardsquare.com/en/products/dexguard.

[5] Fashion MNIST. https://github.com/zalandoresearch/fashion-mnist.

[6] Google Edge TPU. https://cloud.google.com/edge-tpu/.

[7] Intel ISA. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-\instruction-set-reference-manual-325383.pdf.

[8] Intel NCS. https://software.intel.com/en-us/neural-compute-stick.

[9] NVIDIA CUDA. http://www.nvidia.com/cuda/.

[10] NVIDIA Maxwell. https://developer.nvidia.com/maxwell-compute-architecture. Accessed: 2017-11-15.

[11] NVIDIA Nsight assembly correlation. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-view-assembly-code-correlation-nsight\-visual-studio-edition/. Accessed: 2017-11-15.

[12] Open ISA based processor. https://www.codasip.com/2017/08/08/does-risc-v-mean-open-source-processors/.

[13] OpenTPU. https://github.com/UCSBarchlab/OpenTPU.

[14] SVHN dataset. http://ufldl.stanford.edu/housenumbers/.

[15] TPM. https://trustedcomputinggroup.org/.

[16] Martin Abadi et al. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[17] Yossi Adi et al. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *USENIX*, pages 1615–1631, 2018.

[18] Y. Alkabani and F. Koushanfar. Active Hardware Metering for Intellectual Property Protection and Security. *USENIX Security Symposium*, pages 291–306, 2007.

[19] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS, UCB, Tech. Rep. UCB/EECS-2014-146*, 2014.

[20] Buse Gul Atli et al. Extraction of complex dnn models: Real threat or boogeyman? *arXiv preprint arXiv:1910.05429*, 2019.

[21] Kimia Zamiri Azar et al. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–122, 2019.

[22] Kimia Zamiri Azar et al. Threats on logic locking: A decade later. In *Great Lakes Symposium on VLSI*, pages 471–476. ACM, 2019.

[23] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. Preventing IC piracy using reconfigurable logic barriers. *IEEE Design & Test of Computers*, 2010.

[24] Andrew E Caldwell, Hyun-Jin Choi, Andrew B Kahng, Stefanus Mantik, Miodrag Potkonjak, Gang Qu, and Jennifer L Wong. Effective iterative techniques for fingerprinting design ip. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 843–848, 1999.

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

[26] Abhishek Chakraborty, Yuntao Liu, and Ankur Srivastava. Timingsat: timing profile embedded sat attack. In *Proceedings of the International Conference on Computer-Aided Design*, page 6. ACM, 2018.

[27] Abhishek Chakraborty, Yuntao Liu, and Ankur Srivastava. Evaluating the security of delay-locked circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[28] Abhishek Chakraborty, Ankit Mondai, and Ankur Srivastava. Hardware-assisted intellectual property protection of deep learning models. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[29] Abhishek Chakraborty and Ankur Srivastava. Hardware-software co-design based obfuscation of hardware accelerators. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 547–552. IEEE, 2019.

[30] Abhishek Chakraborty, Yang Xie, and Ankur Srivastava. Template Attack Based Deobfuscation of Integrated Circuits. *IEEE International Conference on Computer Design*, pages 41–44, 2017.

[31] Abhishek Chakraborty, Yang Xie, and Ankur Srivastava. Gpu obfuscation: attack and defense strategies. In *Proceedings of the 55th Annual Design Automation Conference*, page 122. ACM, 2018.

[32] Prabuddha Chakraborty et al. Surf: Joint structural functional attack on logic locking. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 181–190, 2019.

[33] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *CHES 2002*, pages 13–28. Springer, 2003.

[34] Shuai Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, IEEE International Symposium on*, pages 44–54, 2009.

[35] Pinhong Chen and Kurt Keutzer. Towards true crosstalk noise analysis. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 132–138. IEEE Press, 1999.

[36] Huili Chen et al. Deepmarks: A digital fingerprinting framework for deep neural networks. *arXiv preprint arXiv:1804.03648*, 2018.

[37] Tianshi Chen et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *Sigplan Notices*, 49(4):269–284, 2014.

[38] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[39] Jeffrey Dwoskin et al. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 389–400. ACM, 2007.

[40] Muhammad Yasin et al. Security analysis of anti-sat. Cryptology ePrint Archive, Report 2016/896, 2016. `http://eprint.iacr.org/2016/896`.

[41] Ian Goodfellow et al. *Deep Learning*. MIT press, 2016.

[42] Tianyu Gu et al. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[43] Jia Guo et al. Watermarking deep neural networks for embedded systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[44] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, pages 249–258. IEEE, 2017.

[45] Kaiming He et al. Deep residual learning for image recognition. In *Conference on computer vision & pattern recognition*, pages 770–778. IEEE, 2016.

[46] Dorjan Hitaj et al. Have you stolen my model? evasion attacks against deep neural network watermarking techniques. *arXiv:1809.00615*, 2018.

[47] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara. Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation. *ACM USENIX Security Symposium*, pages 495–510, 2013.

[48] R.W. Jarvis and M.G. McIntyre. Split Manufacturing Method for Advanced Semiconductor Circuits, 2007. US Patent no. 7,195,931.

[49] Hengrui Jia et al. Entangled watermarks as a defense against model extraction. *arXiv preprint arXiv:2002.12200*, 2020.

[50] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[51] A.B. Kahng, J. Lach, W. H Mangione-Smith, S. Mantik, I.L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking Techniques for Intellectual Property Protection. *IEEE/ACM Design Automation Conference*, pages 776–781, 1998.

[52] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 89. ACM, 2019.

[53] Sanjay Kariyappa et al. Defending against model stealing attacks with adaptive misinformation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2020.

[54] Ramesh Karri et al. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46, 2010.

[55] D. Kirovski and M. Potkonjak. Local Watermarks: Methodology and Application to Behavioral Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1277–1283, 2003.

[56] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[57] F Koushanfar. Provably Secure Active IC Metering Techniques for Piracy Avoidance and Digital Rights Management. *IEEE Transactions on Information Forensics and Security*, 7(1):51–63, 2012.

[58] Yann LeCun et al. Deep learning. *Nature*, 521(7553):436, 2015.

[59] Taesung Lee et al. Defending against machine learning model stealing attacks using deceptive perturbations. *arXiv preprint arXiv:1806.00054*, 2018.

[60] En Li et al. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pages 31–36, 2018.

[61] Tian Li et al. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[62] Y Lin et al. Data and hardware efficient design for convolutional neural network. *IEEE Trans. on Circuits and Systems*, pages 1642–1651, 2017.

[63] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[64] Nils Lukas et al. Deep neural network fingerprinting by conferrable adversarial examples. *arXiv preprint arXiv:1912.00888*, 2019.

[65] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel & Distributed Systems*, 28(1):72–86, 2017.

[66] Erwan Merrer et al. Adversarial frontier stitching for remote neural network watermarking. *arXiv preprint arXiv:1711.01894*, 2017.

[67] Tribhuvanesh Orekondy et al. Knockoff nets: Stealing functionality of black-box models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4954–4963, 2019.

[68] Tribhuvanesh Orekondy et al. Prediction poisoning: Utility-constrained defenses against model stealing attacks. *arXiv preprint arXiv:1906.10908*, 2019.

[69] Soham Pal et al. A framework for the extraction of deep neural networks by leveraging public data. *arXiv preprint arXiv:1905.09165*, 2019.

[70] Nicolas Papernot et al. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia CCS*, pages 506–519, 2017.

[71] P.Kocher, J.Jaffe, and B.Jun. Differential Power Analysis. In *CRYPTO*, pages 388–397, 1999.

[72] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security Analysis of Logic Obfuscation. *IEEE/ACM Design Automation Conference*, pages 83–89, 2012.

[73] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri. Security Analysis of Integrated Circuit Camouflaging. *ACM SIGSAC Conference on Computer & Communications Security*, pages 709–720, 2013.

[74] J. Rajendran, Huan Zhang, Chi Zhang, G.S. Rose, Youngok Pino, O. Sinanoglu, and R. Karri. Fault Analysis-Based Logic Encryption. *IEEE Transactions on Computer*, 64(2):410–424, 2015.

[75] Jeyavijayan Rajendran et al. Security analysis of logic obfuscation. In *Proceedings of 49th Annual Design Automation Conference*, pages 83–89, 2012.

[76] Jeyavijayan Rajendran et al. Fault analysis-based logic encryption. *Computers, IEEE Transactions on*, 64(2):410–424, 2015.

[77] Amin Rezaei et al. Cycsat-unresolvable cyclic logic encryption using unreachable states. In *Asia and South Pacific Design Automation Conference*, pages 358–363. ACM, 2019.

[78] Shervin Roshanisefat et al. 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.

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

[80] Bita Darvish Rouhani et al. Deepsigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In *Proceedings of International Conference on ASPLOS*, pages 485–497. ACM, 2019.

[81] J.A. Roy, F. Koushanfar, and Igor Markov. Ending Piracy of Integrated Circuits. *IEEE Computer*, 43:30–38, 2010.

[82] Jarrod A Roy et al. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 1069–1074. ACM, 2008.

[83] Ahmad-Reza Sadeghi et al. Security and privacy challenges in industrial internet of things. In *52nd Design Automation Conference (DAC)*, 2015.

[84] SEMI. Innovation is at Risk Losses of up to $4 Billion Annually due to IP Infringement. `www.semi.org/en/Issues/IntellectualProperty/ssLINK/P043785`, 2008. Last accessed on 08/01/18.

[85] Abhrajit Sengupta and Ozgur Sinanoglu. Cas-unlock: Unlocking cas-lock without access to a reverse-engineered netlist. *Cryptology ePrint Archive*.

[86] Abhrajit Sengupta et al. Atpg-based cost-effective, secure logic locking. In *36th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2018.

[87] Abhrajit Sengupta et al. Customized locking of ip blocks on a multi-million-gate soc. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2018.

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

[89] Kaveh Shamsi et al. Appsat: Approximately deobfuscating integrated circuits. In *IEEE Symp. Hardware-Oriented Security and Trust*, 2017.

[90] Kaveh Shamsi et al. Cyclic obfuscation for creating sat-unresolvable circuits. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 173–178, 2017.

[91] Yuanqi Shen and Hai Zhou. Double dip: Re-evaluating security of logic encryption algorithms. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 179–184. ACM, 2017.

[92] Reza Shokri et al. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321. ACM, 2015.

[93] Deepak Sirone and Pramod Subramanyan. Functional analysis attacks on logic locking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 936–939. IEEE, 2019.

[94] Deepak Sirone et al. Functional analysis attacks on logic locking. *IEEE Transactions on Information Forensics and Security*, pages 2514–2527, 2020.

[95] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.

[96] Thomas Popp Stefen Mangard, Elisabeth Oswald. *Power Analysis Attacks revealing the secrets of Smart Cards*. Springer, 2007.

[97] Pramod Subramanyan et al. Evaluating the security of logic encryption algorithms. In *Hardware Oriented Security and Trust, 2015*, 2015.

[98] K Szentannai et al. Mimosanet: An unrobust neural network preventing model stealing. *arXiv:1907.01650*, 2019.

[99] Sebastian Szyller et al. Dawn: Dynamic adversarial watermarking of neural networks. *arXiv preprint arXiv:1906.00830*, 2019.

[100] Desta Tadesse et al. Accurate timing analysis using sat and pattern-dependent delay models. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1018–1023. EDA Consortium, 2007.

[101] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 363–381. Springer, 2009.

[102] Florian Tramèr et al. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium*, pages 601–618, 2016.

[103] Yusuke Uchida et al. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 269–277. ACM, 2017.

[104] Bolun Wang et al. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019.

[105] Tianhao Wang et al. Robust and undetectable white-box watermarks for deep neural networks. *arXiv preprint arXiv:1910.14268*, 2019.

[106] Carole-Jean Wu et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[107] K Xiao et al. Hardware trojans: lessons learned after one decade of research. *ACM TODAES*, 22(1):6, 2016.

[108] Yang Xie and Ankur Srivastava. Mitigating sat attack on logic locking. In *CHES 2016*, pages 127–146. Springer, 2016.

[109] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[110] Yang Xie et al. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *54th Annual Design Automation Conference*, 2017.

[111] Huin Xu et al. Deepobfuscation: Securing the structure of convolutional neural networks via knowledge distillation. *arXiv:1806.10313*, 2018.

[112] Xiaolin Xu et al. Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017.

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

[114] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu. SARLock: SAT Attack Resistant Logic Locking. *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 236–241, 2016.

[115] Muhammad Yasin er al. Sarlock: Sat attack resistant logic locking. In *Hardware Oriented Security and Trust (HOST), 2016*, pages 236–241. IEEE, 2016.

[116] Muhammad Yasin et al. On improving the security of logic locking. *CAD of Integrated Circuits & Systems, IEEE Transactions on*, 2015.

[117] Muhammad Yasin et al. Security analysis of logic encryption against the most effective side-channel attack: Dpa. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2015*, pages 97–102. IEEE, 2015.

[118] Muhammad Yasin et al. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1618. ACM, 2017.

[119] Muhammad Yasin et al. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[120] Jialong Zhang et al. Protecting intellectual property of deep neural networks with watermarking. In *Proceedings of Asia CCS*, pages 159–172, 2018.

[121] Qingchen Zhang et al. Privacy preserving deep computation model on cloud for big data feature learning. *IEEE Transactions on Computers*, 2015.

[122] Hai Zhou et al. Resolving the trilemma in logic encryption. *International Conference on Computer-Aided Design. ACM, 2019.*