ABSTRACT

Title of dissertation: EMERGING OPPORTUNITIES
AND CHALLENGES
IN HARDWARE SECURITY

Yuntao Liu
Doctor of Philosophy, 2020

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

Recent years have seen the rapid development of many emerging technologies in various aspects of computer engineering, such as new devices, new fabrication techniques of integrated circuits (IC), new computation frameworks, etc. In this dissertation, we study the security challenges to these emerging technologies as well as the security opportunities they bring. Specifically, we investigate the security opportunities in double patterning lithography, the security challenges in physical unclonable functions, and security issues in machine learning.

Double patterning lithography (DPL) is an emerging fabrication technique for ICs. We study the security opportunities that DPL brings at the layout level. DPL is used to set up two independent mask development lines which do not need to share any information. Under this setup, we consider the attack model where the untrusted employee(s) who has access to only one mask may try to infer the entire circuit design or insert additional malicious circuitry into the design. As a

countermeasure, we customize DPL to decompose the layout into two sub-layouts in such a way that each sub-layout individually exposes minimum information about the other and hence protects the entire layout from any untrusted personnel.

Physical unclonable functions (PUF) are a type of circuits for which each copy (of the same circuit structure) has a unique and unpredictable functionality. The unpredictable behavior is caused by the manufacturing variations of electronic devices. However, for many state-of-the-art PUF designs, we show that the device variations can be estimated using an optimization-theoretic formulation and hence the PUF's input-output behavior becomes predictable. Simulations show a substantial reduction in attack complexity compared to previously proposed machine learning based attacks.

Neural network (NN) is an emerging computation framework for machine learning (ML). It is increasingly popular for system developers to use pre-trained NN models instead of training their own because training is painstaking and sometimes requires private data. We call these pre-trained neural models *neural intellectual properties (IP)*. Neural IPs raise multiple security concerns. On the one hand, as the IP user does not know about the training process, it is crucial to ensure the integrity of the neural IP. To this end, we investigate possible hidden malicious functionality, *i.e., neural Trojans*, that can be embedded into neural IPs and propose effective mitigation techniques. On the other hand, the neural IP owner may want to protect the NN model from reverse engineering attacks. However, it has been

shown that hardware side-channels can be exploited to decipher the structure of neural networks. We propose both a novel attack approach based on cache timing side-channel and a defensive memory access mechanism.

NNs also raise challenges to conventional hardware security techniques. Specifically, we focus on its challenge to logic locking, a strong key-based protection of hardware IP against untrusted foundries by injecting incorrect behavior into the digital functionality when the key is incorrect. We formally prove a trade-off between the amount of injected error and the complexity of Boolean satisfiability (SAT)-based attacks to find the correct key. Due to the inherent error resiliency of NNs, state-of-the-art logic locking schemes fail to inject enough error to derail NN-based applications while maintaining exponential SAT complexity. To fix this issue, we propose a novel secure and effective logic locking scheme, called *Strong Anti-SAT (SAS)*, to lock the hardware and make sure that the NN modes undergo significant accuracy loss when any wrong key is applied.

EMERGING OPPORTUNITIES AND CHALLENGES IN HARDWARE SECURITY

by

Yuntao Liu

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
2020

Advisory Committee:
Professor Ankur Srivastava, Chair/Advisor
Professor Dana Dachman-Soled
Professor Manoj Franklin
Professor Gang Qu
Professor Michael C. Fu

Dedication

To my parents and my wife, for their love and support.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Ankur Srivastava, for his continuous guidance and support throughout my Ph.D. study. I'm extremely grateful for his patience, motivation and immense knowledge in all the time we spent on solving challenging research problems. He has always guided me to the correct path and supported my decision, which makes my Ph.D. experience productive and rewarding. The enthusiasm he has for his research will always be a great motivation for me. It is truly a pleasure to work with and learn from him.

I would like to thank Professor Dana Dachman-Soled for her guidance and help on our collaborated project. I was deeply impressed by her academic rigor and enthusiasm. I learned a lot from our collaboration and my following research also benefited from this experience.

I would like to express my gratitude to Professor Dana Dachman-Soled, Professor Manoj Franklin, Professor Gang Qu, and Professor Michael C. Fu for their time to serve on this committee and their valuable feedback on this dissertation.

I would like to extend my thanks to all my colleagues in Professor Srivastava's research group. My thanks first go to senior colleagues Yang Xie, Zhiyuan Yang, Chongxi Bao, Tiantao Lu and Caleb Serafy for showing me how to conduct research, set up experiments and write papers in the early stage of my Ph.D. study. Without them, I would not be able to adapt to the Ph.D. life so quickly. Thanks also go to my current colleagues Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Daniel

Xing, and Nina Jacobsen for the inspiring research discussions, for the late nights we were working together before deadlines, and for all the fun we have had in our lab.

Finally, I owe the deepest gratitude to my family. I want to express my appreciation to my parents for their endless love and devotion and to my wife Shimiao Liu who has been accompanying me during every hardship.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption System
ArbPUF	Arbiter Physical Unclonable Function
ASLR	Address Space Layout Randomization
CDF	Cumulative Density Function
CNF	Conjunctive Normal Form
CNN	Convolutional Neural Network
Conv	Convolutional
CPU	Central Processing Unit
CRP	Challenge-Response Pair
DI	Distinguishing Input
DNN	Deep Neural Networks
DPL	Double Patterning Lithography
DT	Decision Tree
DumMA	Dummy Memory Access
FC	Fully Connected
FFT	Fast Fourier Transform
FGSM	Fast Gradient Sign Method
FSC	Functionality Stripped Circuit
GAN	Generative Adversarial Nets
GANRED	GAN-based Reverse Engineering of DNNs
GEMM	Generalized Matrix Multiply
GPU	Graphic Processing Unit
HD	Hamming Distance
HE	Homomorphic Encryption
HRS	High Resistance State
HT	Hardware Trojan
IC	Integrated Circuit
IFM	Input Feature Map
IFP	Integer Feasibility Programming
ILP	Integer Linear Programming
IP	Intellectual Property
IQP	Integer Quadratic Programming

JSM	Jacobian Saliency Map
LLC	Last-Level Cache
LRS	Low Resistance State
LRU	Least Recently Used
MAC	Multiply and Cumulate
ML	Machine Learning
MPC	Multi-Party Computation
NN	Neural Network
OFM	Output Feature Map
ORAM	Oblivious Random Access Memory
OS	Oblivious Shuffle
PF	Point Function
PUF	Physically Unclonable Function
RAM	Random Access Memory
RAW	Read-after-Write
RMS	Root-Mean-Square
RU	Restore Unit
SAS	Strong Anti-SAT
SARLock	SAT Resistant Locking
SAT	Satisfiability
SFLL	Stripped Functionality Logic Locking
SGX	Software Guard Extensions
SVM	Support Vector Machine
TFUE	Trusted Foundry and Untrusted Employee
TTL	Tenacious and Traceless Locking
VLSI	Very Large Scale Integration
XNOR	Exclusive Nor
XOR	Exclusive Or
XORArbPUF	XOR Arbiter PUF

List of Publications

Book Chapter:

1. **Liu, Yuntao**, Yang Xie, and Ankur Srivastava. "Security in Emerging Fabrication Technologies" *Security Opportunities in Nano Devices and Emerging Technologies.* CRC Press, 2017. 197-214.

2. Bao, Chongxi, Yang Xie, **Yuntao Liu**, and Ankur Srivastava. "Reverse Engineering-Based Hardware Trojan Detection" *The Hardware Trojan War.* Springer, 2018. 269-288.

Journal:

1. **Liu, Yuntao**, Yang Xie, Chongxi Bao, and Ankur Srivastava. "A Combined Optimization-Theoretic and Side-Channel Approach for Attacking Strong Physical Unclonable Functions." *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)* 26.1 (2018): 73-81.

2. Abhishek Chakraborty, Nithyashankari Gummidipoondi Jayasankaran, **Yuntao Liu**, Jeyavijayan Rajendran, Ozgur Sinanoglu, Ankur Srivastava, Yang Xie, Muhammad Yasin, Michael Zuzak. "Keynote: a Disquisition on Logic Locking" *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* Early Access Article (2019)

Conference:

1. **Liu, Yuntao**, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava, "Strong Anti-SAT: Secure and Effective Logic Locking." *21st International Symposium on Quality Electronic Design (ISQED 2020)*

2. **Liu, Yuntao**, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing, and Ankur Srivastava, "A Survey on Neural Trojans." *21st International Symposium on Quality Electronic Design (ISQED 2020)*

3. **Liu, Yuntao**, Dana Dachman-Soled, and Ankur Srivastava, "Mitigating Reverse Engineering Attacks on Deep Neural Networks." *IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* IEEE, 2019.

4. Charkraborty, Abhishek, **Yuntao Liu**, and Ankur Srivastava. "TimingSAT: Timing Profile Embedded SAT Attack." *Proceedings of the 38th International Conference on Computer-Aided Design.* ACM, 2018.

5. **Liu, Yuntao**, Yang Xie, and Ankur Srivastava. "Neural Trojans." *Computer Design, 2017 IEEE International Conference on.* IEEE, 2017.

6. **Liu, Yuntao**, Chongxi Bao, Yang Xie, and Ankur Srivastava. "Introducing TFUE: The Trusted Foundry and Untrusted Employee Model in IC Supply Chain Security" *Circuits and Systems, 2017 IEEE International Symposium on.* IEEE, 2017.

7. **Liu, Yuntao**, Yang Xie, Chongxi Bao and Ankur Srivastava. "An Optimization-Theoretic Approach for Attacking Physical Unclonable Functions." *Proceedings of the 35th International Conference on Computer-Aided Design.* ACM, 2016.

8. Xie, Yang, Chongxi Bao, **Yuntao Liu**, and Ankur Srivastava. "A Security-Aware Design Scheme for Better Hardware Trojan Detection Sensitivity." *Microprocessor and SOC Test and Verification (MTV), 2016 17th International Workshop on* IEEE, 2016.

Poster:

1. **Liu, Yuntao** and Ankur Srivastava. "GANRED: GAN-based Reverse Engineering of DNNs via Cache Side-Channel." *Design Automation Conference.* 2020.

Chapter 1: Introduction

1.1 Security and Trust Issues in the IC Supply Chain

Many security issues have been raised in the globalized supply chain of integrated circuits (IC). We illustrate the supply chain and the security concerns in Fig. 1.1. For example, many companies sell intellectual properties (IP) to IC designers where the IP becomes a part of the IC design. In this case, the IP owner may wish to keep the design details of the IP as a secret and protect their IP from piracy by the IC designer or later stages of the supply chain. The same concern applies to the IC designers who do not own a foundry and outsource their designs for fabrication since the foundry is not under the designer's control and can be potentially untrusted. On the other hand, and the IC designer wants to verify the integrity of the IP and make sure that there are not malicious back doors. System manufacturers also have this concern when they deploy a chip that is bought from the open market in their systems.

1.1.1 The IP/IC Design Protection Problem

In the IC supply chain outlined in Fig. 1.1, the foundry knows every detail of the IC layout but is not in the designer's control. Therefore, the foundry is usually considered untrusted since there may be adversaries in the. Foundries usually have

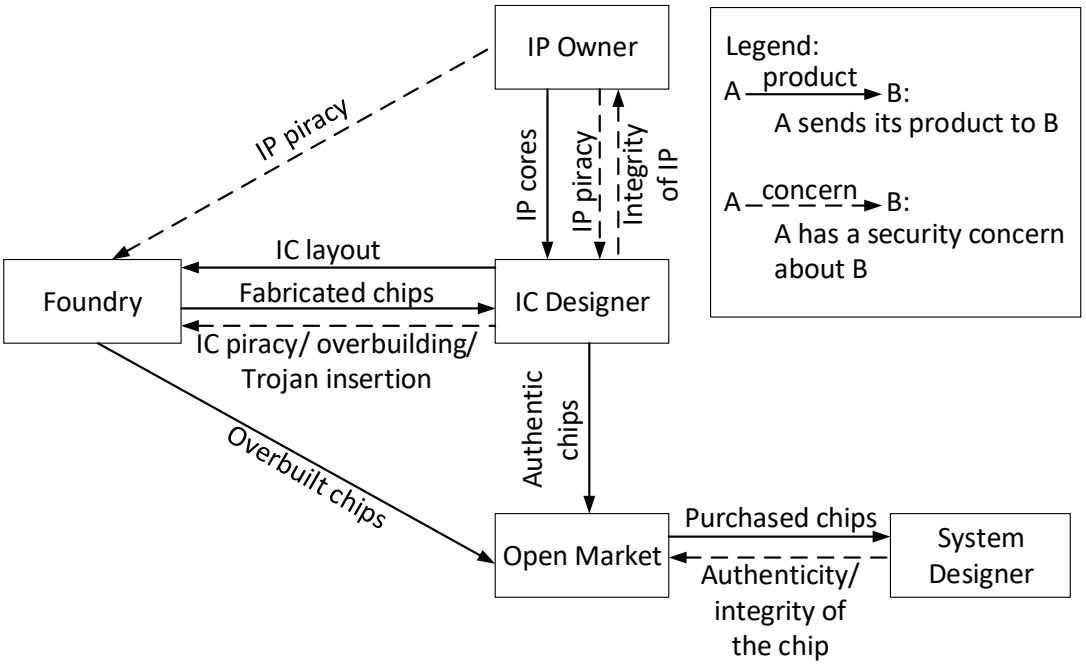


Figure 1.1: An overview of IC supply chain and the security and trust issues

complete information about the design to be manufactured. Therefore, it is crucial for the designer to protect the design from the possible attacks carried out by such informed adversaries. The following attacks are possible:

- **IC Overbuilding:** The foundry may fabricate more copies of the IC than it is supposed to. The foundry may sell the overbuilt ICs directly into the open market and harm the designer's interest.

- **IP Piracy:** The adversary is able to obtain the gate-level netlists of the design (*i.e.,* hardware IPs) by reverse-engineering the layout provided by the designer. The gate-level netlists are the details of the functionality of the circuit which the attacker may resale or reuse.

1.1.2 The Integrity Problem

When the designer receives an IP from the IP suppliers, he/she wants to ensure the integrity of the IP. Likewise, the fabricated ICs from the foundry may also be examined to make sure that no malicious modifications were made to the original IC design. These malicious modifications in IPs or ICs are usually called Hardware Trojans (HT). HTs are usually hidden and only affect the outcome of the circuit under rare circumstances. When an HT is triggered, the output of the circuit deviates substantially from the correct output. Otherwise, the circuit works correctly. This makes the harms of HTs severe and the detection difficult.

1.2 Focus of this Dissertation

In recent years, these hardware security and trust issues are becoming more complicated with the emergence of new technologies. These technologies are found in all the levels, from the physical level to the application level. For example, at the physical level, new fabrication technologies make the continuing scaling of semiconductor devices possible. At the device and circuit levels, new circuit designs that contain new devices provide many new desirable properties. At the application level, new computation frameworks have achieved much better performance on many tasks than state-of-the-art conventional algorithms. While most of these technologies are initially intended to overcome the drawbacks of their conventional

counterparts, they also come with a wide spectrum of implications on security including both opportunities and challenges. In this work, we investigate three specific problems at the intersection of security and emerging technologies:

1. Security opportunities in double patterning lithography (DPL), a new fabrication technology.

2. An attack against physical unclonable functions (PUF), a new type of circuit used for security.

3. Various security issues in neural networks (NN), a new computation framework that has been receiving a lot of interest in recent years.

1.2.1 Security Opportunities in Double Patterning Lithography

Although the foundries are not controlled by the designer, it might not be fair to assume the foundry as an untrusted black box. Indeed, many foundries have developed trusted fabrication lines [64, 1] where the foundries will have legal obligations to prevent any security risk. However, untrusted individual employees may exist and try to steal the IP or insert Trojans into the design. In our work, we take advantage of double patterning lithography (DPL), an emerging IC fabrication technology. When DPL is used, the layout is decomposed into two sub-layouts for mask development. Each sub-layout satisfies the shape spacing requirement which is specified by the fabrication technology node. Each sub-layout is processed in an independent mask production line. We further assume that there is no

collusion between employees on different lines. In addition to satisfying the process requirement, the objective of our DPL includes ensuring minimum information leakage about the entire circuit from individual sub-layouts. We define a measure of "distance" between the original layout and a sub-layout in order to quantify the information leakage. By decomposing in a way that maximizes the smaller distance between the one of the sub-layouts to the original layout, we ensure that minimum information about the original layout is leaked to the attacker no matter which sub-layout he/she has access to. We formulate an optimization-theoretic problem to determine how to partition the layout.

1.2.2 Security Issues with Physical Unclonable Functions

Physical unclonable functions (PUFs) are a type of circuits for which each copy (with exactly the same circuitry) has a unique and unpredictable functionality. This uniqueness comes from the manufacturing variations of electronic devices. The unique functionality of the PUF is characterized by the its input-output behavior: (i) for different PUF copies, the output according to the same input should be independent; (ii) for the same PUF, the output to different inputs should be independent. We call the PUF input as the "challenge", output as "response", and the input-output behavior as "challenge-response pair (CRP)". A typical application of the PUF is authentication. We give an illustrative authentication example as follows. In this scenario, the verifier wants to tell if the remote chip (which has a PUF inside) is authentic. We suppose that the verifier is the chip designer and has recorded a

large number of the PUF's CRPs before selling the chip. During authentication, the verifier (usually a server) sends a set of challenges to the remote device (who claims to have an authentic chip). In return, the remote device sends the PUF's responses to these challenges. If the responses are all correct, the authentication will be successful. As the PUF's response to each challenge is independent, for an n-bit input PUF, there are 2^n unique CRPs and the used CRPs need not be reused if n is large enough. This authentication protocol can be used in other scenarios such as verifying the authenticity of the edge devices in the Internet of things. It can also help the chip designer identify pirate/overbuilt copies in the market.

In this dissertation, the attack on PUFs, *i.e.,* to predict the PUF's responses to unknown challenges, is studied. We formulate a novel optimization-theoretic attacking approach and apply this attack on multiple types of PUFs. We found that our approach substantially reduces the complexity of the attack compared to conventional machine learning based attack.

1.2.3 Security Threats in Neural Networks

In recent years, neural networks (NN) have been an emerging computation framework that outperforms conventional methods on many tasks. As the performance of NNs continues improving, they are becoming larger and deeper as well. As a result, training the networks is becoming more and more expensive in terms of both time and computation resources. Therefore, it becomes increasingly worthwhile to buy a neural network intellectual property (IP) from a third party instead of

training one from scratch by oneself. The ramifications of such a supply chain shift on security is two fold. On the one hand, the neural IP buyer may be concerned about the integrity of the neural IP. On the other hand, the neural IP vendor may wish to only allow the buyer to use the IP without disclosing the underlying neural network model. In this case, reverse engineering attacks on neural networks is of the neural IP vendor's concern. In addition, due to the inherent error resilience of neural network models, error injection-based hardware IP protection schemes, such as logic locking [16], need to inject a higher amount of error in order to corrupt a neural network-based application. This will cause vulnerability to Boolean satisfiability (SAT)-based attacks [102], as we prove in this dissertation. There are the three aspects we focus on regarding the security of neural networks in this dissertation:

1. Neural Trojans: the malicious functionality that can be embedded in a neural IP.

2. Reverse engineering attacks on neural networks via hardware side-channels and countermeasures.

3. A novel logic locking scheme that has high application-level effects on neural network-based applications while maintaining high complexity against SAT-based attacks.

1.3 Contribution and Organization of the Dissertation

Chapter 2 presents the opportunities in layout-level obfuscation brought by DPL. We introduce the *trusted foundry and untrusted employee (TFUE) model* and develop a specialized version of DPL to thwart the attacks from untrusted employees.

Chapter 3 provides a novel optimization-theoretic attack on PUFs. Compared to existing machine learning-based attack, on average, our attack approach reduces the required PUF queries by 66% and takes 79.8% less time to achieve the same PUF response prediction accuracy.

In Chapter 4, we identify and demonstrate the threats of neural Trojans and propose countermeasures including input anomaly detection, retraining, and input preprocessing. Experiments show each countermeasure effective.

In Chapter 5, we theoretically prove a universal trade-off among all logic locking schemes between the hardware-level error injected by the locking circuitry and the complexity of SAT-based attacks. In order to protect hardware running neural network-based applications, which usually tend to be error-resilient, and maintain high SAT complexity, a novel logic locking scheme, called Strong Anti-SAT (SAS), is proposed. SAS achieves high application-level error injection effects while maintaining high SAT complexity.

We propose a cache side-channel based reverse engineering attack on neural networks in Chapter 6. Our attack specifically focuses on the structure of neural networks. Compared to existing approaches, our attack technique eliminates the need of shared main memory between the attacker and victim processes and achieve more accurate reverse engineering results.

In Chapter 7, we consider an even stronger attacker of neural network reverse engineering and provide a countermeasure based on various techniques including oblivious shuffle, address space layout randomization, and dummy memory accesses. We are able to exponentially increase the neural network structure search space.

Chapter 8 concludes this dissertation and discusses our future research plans.

Chapter 2: Security Opportunities in Double Patterning Lithography

In this chapter, we explore how to improve the security of proprietary design details of integrated circuit (IC) during fabrication brought by double patterning lithography (DPL). Specifically, we introduce the *trusted foundry and untrusted employee (TFUE) model* and develop a specialized version of DPL to thwart the attacks from an untrusted employee.

Almost all the previous studies on IC supply chain security label off-site foundries as untrusted. To thwart the attacks mentioned in Section 1.1 by untrusted foundries such as Trojan insertion, IC piracy, etc., countermeasures such as including logic obfuscation, split manufacturing, and post fabrication editing based approaches have been proposed [87, 81, 117, 93, 45]. Among these countermeasures, obfuscation based approaches suffer from various reverse-engineering-based attacks [102, 17], split manufacturing still requires a trusted foundry to fabricate some metal layers which incurs the cost of maintaining such a foundry. In 3D/2.5D integration technology, different dice manufactured by different foundries may not align well which reduces the yield. Post fabrication editing has to be done chip-by-chip and reduces the reliability of the chip and suffers from low efficiency.

In real-world scenarios, however, the "untrusted foundry" assumption may not be accurate. On one hand, the foundries are not necessarily incentivized to perform these attacks due to legal and financial liabilities. One the other hand, even for the foundries conventionally considered as trusted (*e.g.* owned by the designer), there can be rogue employees who has the ability to perform the attacks. Under the TFUE model, we try to secure the IC manufacturing process from the foundry's perspective.

2.1 Fundamentals of Double Patterning Lithography

In this section, we briefly introduce the principles of DPL. In the lithography step of IC fabrication, the minimum distance between two adjacent polygons in the layout is physically constrained by the wavelength of the light used in the lithography. We call this distance the *minimum feature spacing*, denoted as λ. The technology node of IC has scaled down to a point where λ can no longer provide enough resolution as required by the technology node. In order to continue the scaling of transistors, DPL has been developed.

DPL requires that the layout be decomposed into two sub-layouts, each of which satisfies the minimum spacing constraint. A mask based on each sub-layout is made in an independent mask development line. How each sub-layout is processed in its mask development line is Similar to how an entire layout is processed, the sub-layouts need to be edited to ensure that the masks are compatible with the foundry's fabrication process. After the masks are made, in the lithography stage,

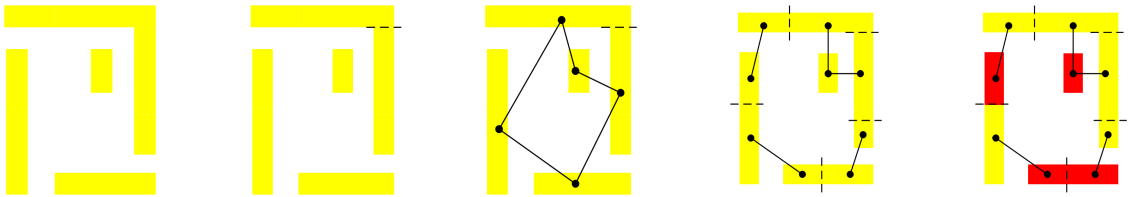


Figure 2.1: The steps of layout decomposition. (a): the original layout; (b): layout fracturing; (c): graph construction; (d): node splitting; (e): color assignment

there are two lithography steps, each putting one sub-layout onto the silicon wafer. After the two lithography steps, the fabricated IC will be the same as the original layout.

Layout decomposition algorithms that make each sub-layout satisfy the spacing constraint have been well developed. We describe the state-of-the-art layout decomposition algorithm proposed by Kahng *et al.* [48]. The layout of an IC is composed of rectilinear polygons. Each rectilinear polygon is called a "feature". The algorithm consists of layout fracturing, graph construction, node splitting, and graph update. These steps are illustrated in Fig. 2.1 and explained below.

1. **Layout fracturing.** In this step, any feature (rectilinear polygon) that is not a rectangle is partitioned into multiple rectangles. For example, in Fig. 2.1a, the upper right features is split into two rectangles as shown in Fig. 2.1b. Note that there can be more than one possible way to split a feature.

2. **Graph construction:** In this step, the *conflict graph* is built. In the conflict graph, each node represents a rectangle in the layout. An edge connecting a pair of nodes exists if the distance between the corresponding rectangles is smaller than λ. This indicates that the two rectangles cannot be on the same mask. A sample conflict graph is shown in Fig. 2.1c.

3. **Node splitting:** As an edge indicates a conflict, a node coloring problem can be formulated as follows. The nodes in the conflict graph need to be colored with two colors in a way that opposite colors must be assigned to two adjacent nodes (*i.e.,* those connected with an edge). Therefore, if any odd-length cycle exists in the conflict graph, the node coloring problem will have no feasible solution, *i.e.,* indicating that there is a conflict which must be resolved in order to get a feasible layout decomposition. An example of such a conflict is illustrated in Fig. 2.1c where we find a 5-node cycle. There is no feasible assignment of colors that will satisfy the constraint. In order to resolve this conflict, one node within the odd-length cycle needs to be split into two, indicating the splitting of a rectangle.

4. **Graph update and color assignment:** A new conflict graph is constructed based on the splitting. One possible way of splitting and the updated conflict graph is shown in Fig. 2.1d. Colors can be assigned when there is no more odd-length cycles in the graph. Fig. 2.1e gives one feasible color assignment.

How the color is often formulated as an integer linear programming (ILP) problem in [48] where the ILP focused on improving the yield of chips by enforcing the following criteria:

(a) Penalizing design rule violations

(b) avoiding assigning different colors in the same polygon ("stiches").

(c) Other factors that may improve the yield.

2.2 TFUE: the Trusted Foundry and Untrusted Employee Model

In this section, we justify the TFUE model [55, 104]. We discuss why this model is more realistic than assuming every foundry to be either completely trusted or completely untrusted.

- It is indeed risky for a foundry to undermine or counterfeit the IC designs massively. If this is noticed by the designer, the foundry will have legal troubles and lose business in the future. Every foundry wants a larger market share and may not risk this. Instead, even the foundries conventionally assumed untrusted, including offshore foundries, want the designers to trust their integrity. Bloom *et al.* proposed a way that allows the designer to attest the integrity of the foundry machinery and the produced chips to ensure their integrity [11]. The foundries' security policies and enforcement can also be reviewed by the designer and/or a licensed third party.

- Foundries that are conventionally assumed as trusted are not necessarily free from the risk of untrusted employees. There can be employees who collude with the foundry's or the designer's competitors who aim at undermining the fabricated chips or pirating the design. In this case, even a conventionally trusted foundry still need security measures against untrusted employees.

Due to the above reasons, we identify the foundry employees as the source of security threats in a foundry. We no longer classify the foundries into trusted ones and untrusted ones. Instead, we use TFUE as our primary assumption.

The following sections will present the threat model under TFUE and countermeasures utilizing DPL. In the next section, we will explain what an untrusted foundry employee can can do to undermine the IC supply chain security and explore the opportunities made possible by DPL to obfuscate the layout-level design details and defend the attacks.

2.3 Threat Model and Countermeasure under TFUE

2.3.1 Threat Model

We consider the threat model and countermeasures in the context of DPL since DPL is necessary for state-of-the-art technology nodes. The layout decomposition process can be automated, no employee needs to access the entire layout obtained from the designer before the decomposition. Secure machinery is used to decompose the layout into two sub-layouts. The two sub-layouts are then developed in two independent mask production lines by two independent groups of employees. The

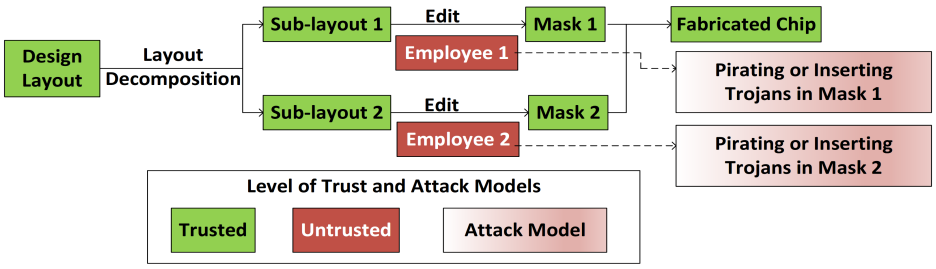


Figure 2.2: The attack model in the context of DPL under TFUE

independent mask production lines do not share any layout information. We further assume that there is no collusion between the employees on different lines. Each sub-layout is edited in its line to ensure that the mask is manufacturable. However, when an employee edits the sub-layout, he/she has the ability to perform various attacks such as Trojan insertion and piracy. In this context, we assume the following about the attacker:

- The attacker who is an employee of the foundry who works on one of the independent mask production lines. His/her knowledge about the entire IC design is limited to the what is on the sub-layout processed by that line.
- There is no collusion between employees on different lines.
- Although the employee's job is to edit the sub-layout to improve the manufacturability of the mask, he/she can actually make any change to the layout.
- The attacker may try to recover the original layout from the sub-layout that he/she works on and steal the entire chip design.

This attack model is illustrated in Fig. 2.2.

2.3.2 Countermeasure Formulation

the attacker needs to know to some extent about the original layout in order to insert Trojans into or counterfeit the IC design. In the attack model we consider, such knowledge is based on the inspection into one sub-layout. Intuitively, each sub-layout should look as 'different' as possible from the original layout in order to leak minimum information about the original layout. We will define a metric of "distance" between two layout images, based on which we propose a variant of layout decomposition algorithm which maximizes the minimum distance between each sub-layout and the original layout.

We follow the following convention of notations: lower case letters denote scalars (normal) and vectors (bold), and capital ones denote matrices. For example, x is a scalar, $\mathbf{x}$ is a vector and X is a matrix. We use lower scripts to denote the indices of vectors and matrices. A comprehensive list of symbols is given in Table 2.1. The symbols will be explained again when used in equations.

Our proposed countermeasure is a customized layout decomposition algorithm which follows the one specified in Section 2.1 up to node splitting until every conflict is resolved. We customize the color (sub-layout) assignment algorithm. We consider the layout of the design as a binary image. For each pixel in the image, its value is 1 it resides on a feature and the value is 0 otherwise.

In order to evaluate the distance between two images, we need a metric of "distance" in the first place. We use their difference in the frequency domain to characterize the distance. We transform the layout images using 2-dimensional

Table 2.1: List of symbols

Symbol and domain	**Meaning**
$\lambda \in \mathbb{R}_{++}$	the minimum feature spacing
n	the number of rectangles in the layout
$\mathbf{G} = (\mathbf{V}, \mathbf{E})$	the conflict graph
$\mathbf{V} = \{c_k \vert k = 1, \ldots, n\}$	the set of nodes (rectangles) in the conflict graph
$\mathbf{E}$	$\{(c_u, c_v) \vert$ the distance between c_u and $c_v \ \leq \lambda\}$
$p \in \mathbb{Z}_{++}$	the number of elements in each row and column of the Fourier coefficient matrices
$F^l \in \mathbb{C}^{p \times p}, l = 0, 1, 2$	the Fourier coefficient matrix of the FFT of the layout, with $l = 0, 1, 2$ corresponding to the original layout, the first sub-layout, and the second sub-layout, respectively
$\tilde{F}^k \in \mathbb{C}^{p \times p}, k = 1, \ldots, n$	the Fourier coefficient matrix of the FFT of the k-th rectangle's layout.
$s_l, l = 1, 2$	the distance between the original layout and sub-layout l
$\mathbf{x} \in \{0, 1\}^n$	the vector indicating the sub-layout that each rectangle belongs to, where $\mathbf{x}_k = 1$ if rectangle c_k is assigned to the first sub-layout, and $\mathbf{x}_k = 0$ if c_k is assigned to the second sub-layout
$\bar{\mathbf{x}} \in \{0, 1\}^n = \mathbf{1} - \mathbf{x}$	
$A^l, l = 0, 1, 2$	the real parts of F^l
$\tilde{A}^k, k = 1, \ldots, n$	the real parts of $\tilde{F}^k$
$B^l, l = 0, 1, 2$	the imaginary parts of F^l
$\tilde{B}^k, k = 1, \ldots, n$	the imaginary parts of $\tilde{F}^k$
$\alpha^{(ij)} \in \mathbb{R}^n$	$\alpha^{(ij)} = (\tilde{A}^1_{ij}, \tilde{A}^2_{ij}, \ldots, \tilde{A}^n_{ij})^T$
$\beta^{(ij)} \in \mathbb{R}^n$	$\beta^{(ij)} = (\tilde{B}^1_{ij}, \tilde{B}^2_{ij}, \ldots, \tilde{B}^n_{ij})^T$

Fourier transform (2D-FFT):

$$F_{\mu\nu} = \frac{1}{wl} \sum_{x=1}^{w} \sum_{y=1}^{l} I_{xy} e^{-2\pi j(\frac{x\mu}{w} + \frac{y\nu}{l})}$$

Let F^0, F^1 and F^2 be the Fourier coefficient matrices of the original layout and the two sub-layouts, respectively. Similar to the 1D-FFT, the linear property also holds for the 2D-FFT, *i.e.,*

$$F^0 = F^1 + F^2$$

This above equation must also hold element-wise, *i.e.,* $F^0_{ij} = F^1_{ij} + F^2_{ij}$ for any $i, j = 1, \ldots, p$. As the (sub-)layout comprises of multiple rectangles, its 2D-FFT Fourier coefficient matrix also equals the summation of those of its rectangles, *i.e.,*

$$F^0_{ij} = \sum_{k=1}^{n} \tilde{F}^k_{ij}, \; F^1_{ij} = \sum_{k=1}^{n} \mathbf{x}_k \tilde{F}^k_{ij}, \text{ and } F^2_{ij} = \sum_{k=1}^{n} \bar{\mathbf{x}}_k \tilde{F}^k_{ij}$$

Definition 2.1 (Distance)**.** The *distance* between two binary images is defined as the root-mean-square (RMS) value of the differences of their 2D-FFT coefficient matrix elements.

$$s_l = \sqrt{\frac{1}{p^2} \sum_{i=1}^{p} \sum_{j=1}^{p} |F^0_{ij} - F^l_{ij}|^2} \tag{2.1}$$

where s_1 and s_2 stand for the distances between the first/second sub-layout and the original layout, respectively. As we mentioned earlier, the distance is a measure of the attacker's difficulty to infer the entire layout from a sub-layout. Therefore, a larger difference indicates more security and it is desirable to partition the layout such that both distances are large, *i.e.,* no matter which sub-layout the attacker

can access, he/she always has great difficulties to attack. However, there may be a trade-off between the two distances. Considering this, the objective should be maximizing the smaller distance among the two (*i.e.,* s_1 and s_2):

$$\max_{\mathbf{x}\in\{0,1\}^n} \min\{s_1, s_2\}$$

This can be transformed into the following equivalent form:

$$\max_{\mathbf{x}\in\{0,1\}^n} \min\{\sum_{i=1}^{p}\sum_{j=1}^{p}|F_{ij}^2|^2, \sum_{i=1}^{p}\sum_{j=1}^{p}|F_{ij}^1|^2\} \tag{2.2}$$

Note that each Fourier coefficient is a complex number. The relationship between the magnitudes of the coefficient and its real and imaginary parts is $|F_{ij}^l|^2 = (A_{ij}^l)^2 + (B_{ij}^l)^2$ for $l = 0, 1, 2$, where A_{ij}^l and B_{ij}^l are the real and imaginary parts of F_{ij}^l, respectively. The real and imaginary parts also satisfy linearity individually:

$$A_{ij}^1 = \sum_{k=1}^{n}\tilde{A}_{ij}^k\mathbf{x}_k = \alpha^{(ij)T}\mathbf{x},\ A_{ij}^2 = \sum_{k=1}^{n}\tilde{A}_{ij}^k\bar{\mathbf{x}}_k = \alpha^{(ij)T}\bar{\mathbf{x}},$$
$$B_{ij}^1 = \sum_{k=1}^{n}\tilde{B}_{ij}^k\mathbf{x}_k = \beta^{(ij)T}\mathbf{x},\ B_{ij}^2 = \sum_{k=1}^{n}\tilde{B}_{ij}^k\bar{\mathbf{x}}_k = \beta^{(ij)T}\bar{\mathbf{x}}.$$

where $\alpha^{(ij)} = (\tilde{A}_{ij}^1, \tilde{A}_{ij}^2, \ldots, \tilde{A}_{ij}^n)^T$ and $\beta^{(ij)} = (\tilde{B}_{ij}^1, \tilde{B}_{ij}^2, \ldots, \tilde{B}_{ij}^n)^T$. The above equations give us each 2D-FFT coefficient, including the real and imaginary parts, of each sub-layout. To obtain the quantities that are required in (2.2), we have

$$\sum_{i=1}^{p}\sum_{j=1}^{p}|F_{ij}^1|^2 = \sum_{i=1}^{p}\sum_{j=1}^{p}(A_{ij}^1)^2+(B_{ij}^1)^2 = \mathbf{x}^T(\sum_{i=1}^{p}\sum_{j=1}^{p}\alpha^{(ij)}\alpha^{(ij)T}+\beta^{(ij)}\beta^{(ij)T})\mathbf{x} = \mathbf{x}^TQ\mathbf{x}$$

where

$$Q = \sum_{i=1}^{p}\sum_{j=1}^{p}\alpha^{(ij)}\alpha^{(ij)T} + \beta^{(ij)}\beta^{(ij)T} \tag{2.3}$$

For the same reason,

$$\sum_{i=1}^{p}\sum_{j=1}^{p}|F_{ij}^2|^2 = \bar{\mathbf{x}}^TQ\bar{\mathbf{x}}$$

Now the problem in (2.2) has been transformed into an integer quadratic programming (IQP) problem:

$$\begin{aligned} &\max_{\mathbf{x}\in\{0,1\}^n} \min\{\mathbf{x}^T Q\mathbf{x}, \bar{\mathbf{x}}^T Q\bar{\mathbf{x}}\} \\ &\text{subject to } \bar{\mathbf{x}} = \mathbf{1} - \mathbf{x} \\ &\mathbf{x}_i + \mathbf{x}_j = 1 \text{ if } (c_i, c_j) \in \mathbf{E} \end{aligned} \tag{2.4}$$

This problem is indeed difficult to solve for two reasons. First, an integer programming problem is hard to solve in general. Second and more importantly, even if we relax the problem (*i.e.,* the domain of $\mathbf{x}$) to be continuous, it is still difficult. By (2.3) we know that $Q \succ 0$. Therefore, $\mathbf{x}^T Q\mathbf{x}$ and $\bar{\mathbf{x}}^T Q\bar{\mathbf{x}}$ are convex functions of $\mathbf{x}$. However, in general, the pointwise minimum of two convex functions, like the one in (2.4), is neither convex nor concave. This makes gradient-based algorithms not applicable for this problem since we may get stuck at a local minimum, not the global one.

Fortunately, we found a good approximation of this problem. By inspecting Q, We found that almost every non-zero elements in Q are on the diagonal, *i.e., Q* is very sparse off-diagonal. In fact, for any benchmark, non-zero off-diagonal elements are less than 1%. In order to make the problem easier to solve, we simplify the problem by only considering the diagonal elements in Q. This is not likely to result in significant error. Since each element in $\mathbf{x}$ is either 0 or 1, when we ignore all the off-diagonal non-zero elements in Q, we have the following approximation:

$$\mathbf{x}^T Q\mathbf{x} \approx \mathbf{x}^T diag(Q_{11}, \ldots, Q_{nn})\mathbf{x} = \mathbf{d}^T\mathbf{x}$$

where

$$\mathbf{d} = (Q_{11}, \ldots, Q_{nn})^T$$

Then, the problem in (2.4) is approximately transformed into

$$\begin{aligned} &\max_{\mathbf{x}\in\{0,1\}^n} \min\{\mathbf{d}^T\mathbf{x}, \mathbf{d}^T\bar{\mathbf{x}}\} \\ &\text{subject to } \bar{\mathbf{x}} = \mathbf{1} - \mathbf{x} \\ &\mathbf{x}_i + \mathbf{x}_j = 1 \text{ if } (c_i, c_j) \in \mathbf{E} \end{aligned} \tag{2.5}$$

The problem in (2.5) is a well-studied integer linear programming problem, and there exists many efficient heuristic algorithms that can get good solutions in practice.

2.4 Experiment Setup and Results

We describe how we evaluate our proposed DPL algorithm by experiments in this section. It can be shown that the distances defined in Eq. (2.1) can be expressed as

$$s_1 = \frac{1}{p}\sqrt{\mathbf{x}^T Q \mathbf{x}}, \text{ and } s_2 = \frac{1}{p}\sqrt{\bar{\mathbf{x}}^T Q \bar{\mathbf{x}}},$$

If our approximation (*i.e.,* dropping the non-zero off-diagonal elements in Q) is reasonable, we should have

$$s_1^2 + s_2^2 \approx \frac{1}{p}(\mathbf{x} + \bar{\mathbf{x}})^T\mathbf{d} = \frac{1}{p}\mathbf{1}^T\mathbf{d} \approx \frac{1}{p}\mathbf{1}^T Q \mathbf{1}$$

In order to verify our approximation, we define s_{max} as

$$s_{max} = \frac{1}{p}\sqrt{\mathbf{1}^T Q \mathbf{1}}$$

We can verify whether $s_1^2 + s_2^2 \approx s_{max}^2$ holds. If so, we can be confident about the approximation.

Table 2.2: Experimental results on the proposed countermeasure

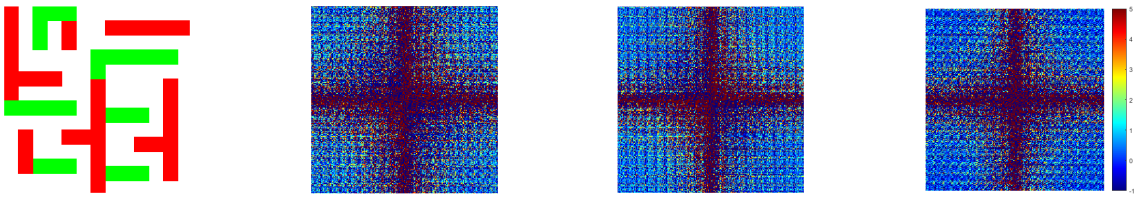
	Benchmark		Distances			Run
	Polygons	Rectangles	s_1^2	s_2^2	s_{max}^2	Time (s)
1	200	425	461	425	883	1.174
2	510	870	669	609	1272	2.004
3	990	1596	792	734	1558	4.807
4	1989	3094	1143	1167	2313	20.26
5	5081	8398	2257	2089	4339	348.0

Our approach is evaluated on 5 benchmarks up to 5000+ polygons and 8000+ rectangles. s_1^2, s_2^2, and s_{max}^2 as well as the running time for solving problem (2.5) are recorded.

It is shown in Table 2.2 that the s_1^2 and s_2^2 that we obtain by solving Eq. (2.5) are close to each other and sum up approximately equal to s_{max}^2, indicating that our formulation in (2.5) approximates the original problem (2.4) well. As $s_1^2 + s_2^2$ is roughly a fixed value (s_{max}^2), making them close to each other makes sure that the smaller between them is maximized. In this way, the attacker's difficulty is maximized, no matter he/she tries to insert Trojans into the design and/or stealing the design. The layout decomposition result with an illustrative benchmark is shown in Fig. 2.3.

2.5 Summary

In this chapter, we explored what opportunities DPL can bring to enhance the security of IC supply chain. Specifically, we investigate this problem under the TFUE model. Based on the state-of-the-art DPL algorithm, we developed our



(a) (b) (c) (d)

Figure 2.3: An illustrative benchmark and its layout decomposition results. (a)layout decomposition result. Each color indicates a sub-layout; (b) the 2D-FFT of the original layout; (c) the 2D-FFT of the first sub-layout (red in (a)); (d) the 2D-FFT of the second sub-layout (green in (a)) and color bar: the magnitude decreases from top to bottom.

version which takes security into consideration. Along with the secure machinery and security policies of the foundry, our approach of DPL can essentially make it hard for the attacker to perform any attack.

Chapter 3: Security Vulnerabilities in Physical Unclonable Functions

Physical unclonable functions (PUFs) were first proposed by Gassend *et al.* in 2002 [28] where the manufacturing variations of electronic devices are utilized as entropy sources to provide a unique signature of the circuit. The input to the PUF circuitry is called the *challenge* and the PUF's output is called the *response*. The PUF's unique signature is characterized by its challenge-response pairs (CRP). Let us consider a PUF of size $O(n)$ (*e.g.* number of devices in the PUF). We call a PUF a *strong PUF* if the PUF can produce an exponential (in n) number of CRPs. Otherwise, we call the PUF a *weak PUF*. The applications of weak PUFs include the storage of secret keys, the seeds of true randomness generators, etc. The most prominent use of strong PUFs is low-cost authentication [109, 110, 111]. An illustrative protocol is as follows. The server (verifier) maintains a database of (a subset of) the CRPs of the PUF before deploying/selling the chip that contains the PUF. During authentication, the server sends the chip one or more challenge vector(s). In return, the chip sends the server the PUF's response(s). As there are a huge number of unique CRPs for a strong PUF, the used CRPs need not be reused. Ideally, an eavesdropper should not be able to break the PUF since the used CRPs should not imply any any unused CRP. Unfortunately, this is not the

case. For example, Lim *et al.* found that the CRPs of the arbiter PUF (ArbPUF), a popular PUF design, can be characterized using a linear additive delay model [54] once the device variations are known. This property makes such PUF designs potentially vulnerable to optimization-theoretic attacks. In our work, we find that the Memristor Crossbar PUF (MXbarPUF) indeed has a similar linear behavior to the ArbPUF[58, 59].

In this chapter, we focus on attacking strong PUFs, *i.e.,* to predict their unknown CRPs. We assume that the attacker knows the PUF circuitry and can query PUF with challenge vectors of his/her choice. Provided the linear behavior of the above-mentioned PUFs, we formulate a novel optimization-theoretic approach to deciphering the internal device variations of the PUFs and predicting the unknown CRPs. Our approach substantially reduces the attack complexity compared to the existing attack based on machine learning (ML): compared to the ML-based attack, our optimization-theoretic approach reduces the known CRP requirement by 66% and takes 79.8% less time.

3.1 Physical Unclonable Functions

3.1.1 Memristor and Memristor Crossbar PUF

In 1971, Chua modeled the behavior of memristors in [24] although such devices did not exist at that time. The memristor's I-V characteristics, in short, is a hysteresis loop pinched at the origin [23]. The resistance of a memristor, called the memristance, can be adjusted between an upper bound M_H and a lower

bound M_L. When the memristance is M_H, the state of the memristor is called the *high resistance state (HRS)* and the state of M_L is called the *low resistance state (LRS)*. Memristors are polar and the direction of the applied voltage decides whether the memristance will be increased or decreased. This being said, when there no voltage is applied on the memristor, the memristance remains unchanged. This is called the *non-volatile property* of memristors which makes them desirable for many applications. After decades of search, Strukov *et al.* first fabricated a device that fits the properties of memristors in 2008 [101]. Since then, memristors have been extensively studied in many fields, including neuromorphic computation [46], computer memory systems [29] and hardware security [2].

The manufacturing variations of memristors, like those of conventional devices, can act as the source of entropy to build PUFs. Rose and Meade[85] proposed the *memristor crossbar PUF (MXbarPUF)*.

The way that MXbarPUF works can be split into the following stages (note that the $DONE$ signal is 0 until the PUF response is finalized):

- *The RESET Stage.* At the very beginning, all the memristors are reset to the HRS by a sufficiently long $RESET = 1$ signal which applies $-V_{DD}$ on all the memristors.

- *The SET Stage.* In this stage, $RESET = 0$, and $R/\overline{W} = CLOCK = 0$, indicating that the memristance of some memristors will be changed by the input challenge vector.

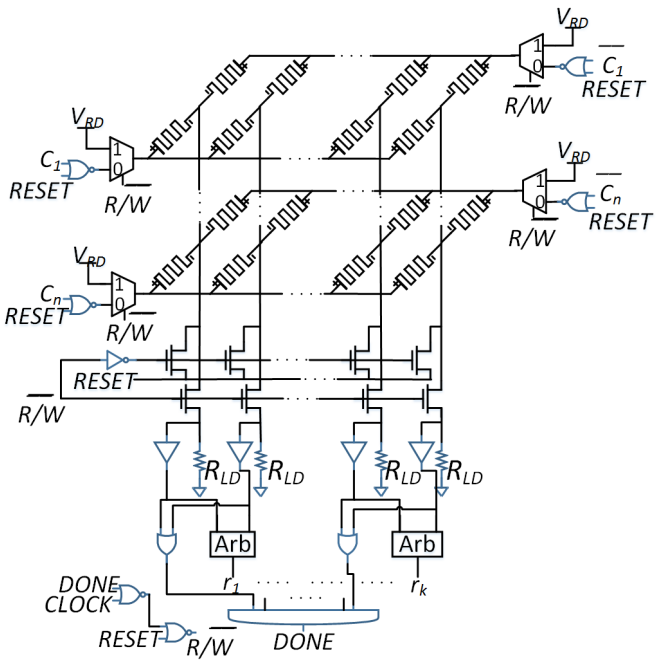


Figure 3.1: A MXbarPUF with k arbiters

- *The READ Stage.* When $CLOCK = 1$, V_{RD} is selected by each multiplexer. The voltage on R_{LD} reflects the total current in the column which is dependent on the memristors in that column. Each arbiter compares the voltage on two adjacent R_{LD}'s and outputs a response bit indicating which is higher.

We denote the finalized conductance of the memristor located at the a^{th} row and the b^{th} column as g_{ab} and define

$$G_j = \left(g_{1j}, g_{2j}, ..., g_{(2n)j}\right)^T \tag{3.1}$$

The resistance of column j (between V_{RD} and ground) is

$$R_j\left(G_j, C'\right) = \frac{1}{C'^T G_j} + R_{LD} \tag{3.2}$$

where

$$C' = \left(C_1, \overline{C_1}, C_2, \overline{C_2}, ..., C_n, \overline{C_n}\right)^T \tag{3.3}$$

is an expansion of the challenge vector. Then for column j, the voltage on R_{LD} is:

$$V_j\left(G_j, C'\right) = \frac{V_{RD} R_{LD}}{R_j\left(G_j, C'\right)} \tag{3.4}$$

The voltage difference between a pair of R_{LD} corresponding the same arbiter (say comparing columns $2i-1$ and $2i$), denoted by ΔV, is

$$\Delta V_i\left(G_{2i-1}, G_{2i}, C'\right) = \frac{V_{RD} R_{LD}\left(\frac{1}{C'^T G_{2i}} - \frac{1}{C'^T G_{2i-1}}\right)}{\left(\frac{1}{C'^T G_{2i}} + R_{LD}\right)\left(\frac{1}{C'^T G_{2i-1}} + R_{LD}\right)} \tag{3.5}$$

On the right side of (3.5), the denominator is always positive. V_{RD} and R_{LD} are also always positive. Therefore, the sign is dependent on the rest of the numerator which we transform while maintaining the sign as below:

$$U_i\left(G_{2i-1}, G_{2i}, C'\right) = C'^T G_{2i-1} - C'^T G_{2i}, \tag{3.6}$$

Because $\overline{C_i} = 1 - C_i$ for $i = 1, 2, \ldots, n$, we expand the vector multiplications in (3.6) as

$$\begin{aligned} U_i\left(G_{2i-1}, G_{2i}, C'\right) = &\sum_{l=1}^{n} C_l\left(g_{(2l-1)(2i-1)} - g_{(2l-1)(2i)} - g_{(2l)(2i-1)} + g_{(2l)(2i)}\right) \\ &+ \sum_{l=1}^{n}\left(g_{(2l)(2i-1)} - g_{(2l)(2i)}\right) \end{aligned} \tag{3.7}$$

We define

$$\begin{gathered} d_{li} = g_{(2l-1)(2i)} - g_{(2l-1)(2i-1)} - g_{(2l)(2i)} + g_{(2l)(2i-1)} \\ \text{for } l = 1, 2, \ldots, n \text{ and } i = 1, 2, \ldots, k \end{gathered} \tag{3.8}$$

where k is the number of arbiters. We further define

$$d_{(n+1)i} = \sum_{l=1}^{n}\left(g_{(2l)(2i)} - g_{(2l)(2i-1)}\right) \tag{3.9}$$

Then (3.7) can be simplified as

$$U_i\left(D_i, \Phi\right) = \Phi^T D_i \tag{3.10}$$

where

$$\Phi = (C_1, C_2, \ldots, C_n, 1)^T \in \{0,1\}^{n+1} \tag{3.11}$$

is the feature vector and

$$D_i = \left(d_{1i}, d_{2i}, ..., d_{(n+1)i}\right)^T \in \mathbb{R}^{n+1} \tag{3.12}$$

is the weight vector.

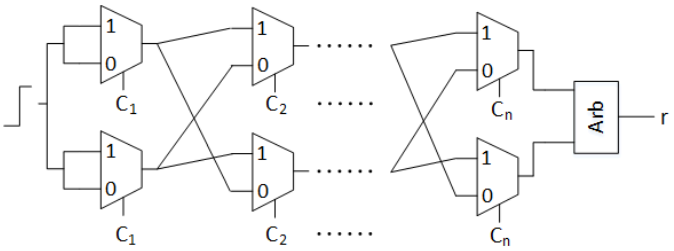
Note that each 2-column PUF can be separately attacked since each arbiter's response is explicitly given. Hence we hereafter omit the index 'i' of arbiters. Then, the an arbiter's response is:

$$r = \begin{cases} 1 \text{ if } \Phi^T D > 0 \\ -1 \text{ otherwise} \end{cases} \tag{3.13}$$

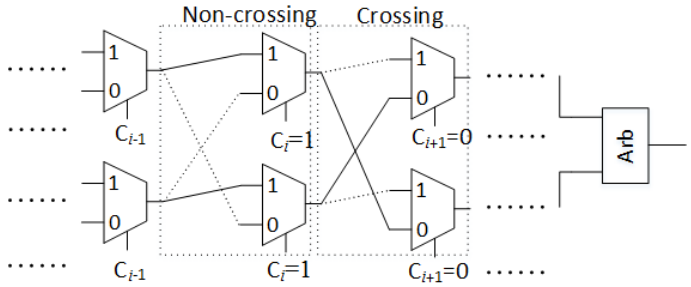
3.1.2 The Arbiter PUF

As shown in Fig. 3.2, the ArbPUF is composed of multiple cascaded stages, each comprising two multiplexers. Each challenge bit C_i selects which input to be propagated for both multiplexers in the i^{th} stage. The effects of C_i being 0 or 1 is shown in Fig. 3.2b. An initial pulse is given to all the inputs of the first stage. Process variations of devices will cause delay differences between the two delay paths. The total difference at the end of the two paths determines the PUF response through the arbiter.

The mathematical model of the ArbPUF is given below. We denote the cumulative delay difference of the two paths up to the i^{th} stage as Δ_i, and the incremental delay difference of the i^{th} stage as δ_i^1 or δ_i^0 for the non-crossing and



(a) The circuitry of the ArbPUF



(b) Illustration on the switching activity of the multiplexers

Figure 3.2: The switching activity of ArbPUF

the crossing case, respectively. The summation of the cumulative delay difference of the previous and the incremental delay difference of the current stage makes the cumulative delay difference of the current stage:

$$\Delta_i = \begin{cases} \Delta_{i-1} + \delta_i^1 & \text{if} \quad C_i = 1 \\ -\Delta_{i-1} + \delta_i^0 & \text{if} \quad C_i = 0 \end{cases} \quad \text{for } i = 1, 2, \ldots, n \tag{3.14}$$

where $\Delta_0 = 0$, $C = (C_1, C_2, ..., C_n) \in \{0,1\}^n$ is the challenge vector, and n is the number of stages in the ArbPUF (hence the length of C). We define the *feature vector* $\Phi = (\phi_1, \phi_2, \ldots, \phi_{n+1})^T \in \{-1,1\}^{n+1}$ and the *weight vector* $D = (d_1, d_2, ..., d_{n+1})^T \in \mathbb{R}^{n+1}$ as

$$\phi_i = \phi_i(C) = \prod_{j=i}^{n} (2C_j - 1) \text{ for } i = 1, 2, \ldots, n, \ \phi_{n+1} = 1 \tag{3.15}$$

$$
\begin{aligned}
d_i &= \frac{\delta^0_{i-1} + \delta^1_{i-1} + \delta^0_i - \delta^1_i}{2} \text{ for } i = 2, 3, \ldots, n, \\
d_1 &= \frac{\delta^0_1 - \delta^1_1}{2},\ d_{n+1} = \frac{\delta^0_n + \delta^1_n}{2}
\end{aligned}
\tag{3.16}
$$

One can verify that the final cumulative delay difference(*i.e.,* after the last stage) is:

$$
\Delta_n = \Phi^T D \tag{3.17}
$$

The arbiter determines the output bit by comparing the total delays of the two paths, *i.e.,* the sign of Δ_n. Note that we use '1' and '-1' to denote the responses (instead of '1' and '0') for simplicity.

$$
r = \begin{cases} 1 \text{ if } \Phi^T D > 0 \\ -1 \text{ otherwise} \end{cases} \tag{3.18}
$$

One can see that (3.13) has the same form as (3.18). The similarity in their challenge-response behavior makes our attack approach applicable to both the ArbPUF and the MXbarPUF.

3.1.3 The XOR Arbiter PUF

The XORArbPUF consists of multiple parallel and independent ArbPUFs. An XOR gate, as shown in Fig. 3.3, combines the outputs of all the ArbPUFs into one bit in order to produce non-linearity. Note that the challenge vectors of each ArbPUF is separate although they have the same length.

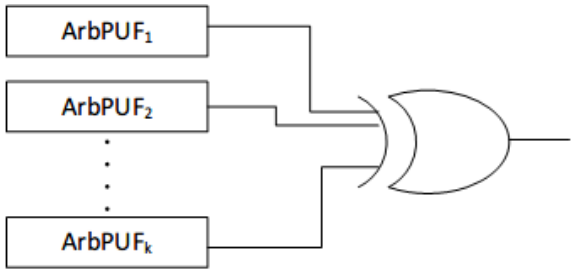


Figure 3.3: Illustration of XORArbPUF architecture

3.2 Existing Attacks on PUFs

3.2.1 Attacks on ArbPUF and MXbarPUF

It is assumed that the PUF is in the possession of the attacker who can query any challenge on the PUF and get the response. Lim *et al.* proposed a machine learning (ML) based attacking approach in the same paper where they proposed the ArbPUF architecture [54]. Specifically, they used support vector machine (SVM) for the attack. They were able to reduce the prediction error to below 5% (*i.e.,* percentage of incorrectly predicted unknown CRPs) with a small number of known CRPs. Due to the similar behavior of the MXbarPUF, the ML-based attack is also applicable to the MXbarPUF which we will show later in the experiments.

3.2.2 Attacks on XORArbPUFs

Ref. [54] suggested that the vulnerability of ArbPUF to such attack can be overcome by introducing nonlinearity into the PUF design. Although XORArbPUFs with sufficiently large size were suggested to be secure [88, 89], side-channels of them were later found which could be exploited by an attacker [63, 90, 122]. The first

side-channel boosted ML attack against XORArbPUFs was proposed by Mahmoud *et al.* [63] which is summarized as follows. The essence of the attack is to observe side-channel signatures when one arbiter switches from '0' (the initial state) to '1'. There will be some side-channel leakage since an amount of electric charge must be drawn from the power supply during the switch and a glitch can be observed in the power trace. Therefore, by monitoring the power trace, the attacker is able to tell the number of arbiters with the output of '1'. This side-channel information can be used to boost the ML-based attack [90].

3.3 Attack Formulation

An optimization-theoretic attacking approach on the ArbPUF and the MXbarPUF is formulated in this section. Our approach consists of two parts: (i) linear programming based weight vector estimation, and (ii) new challenge vector generation using the cutting-plane method. Subsequently, we combine this approach with the above-mentioned side-channel leakage and apply this side-channel boosted optimization-theoretic attacking approach on the XORArbPUF.

3.3.1 Linear Programming Based Weight Estimation

As noted in Sections 3.1.2 and 3.1.1, the feature vector Φ is

$$\Phi = (\phi_1, \phi_2, \ldots, \phi_{n+1})^T \in \{-1, 1\}^{n+1}$$

$$\phi_i = \phi_i(C) = \prod_{j=i}^{n} (2C_j - 1) \text{ for } i = 1, 2, \ldots, n, \ \phi_{n+1} = 1$$

for the ArbPUF, and

$$\Phi = (C_1, C_2, \ldots, C_n, 1)^T \in \{0,1\}^{n+1}$$

for MXbarPUF. The weight vector D of the ArbPUF and the MXbarPUF is defined in (3.16) and (3.8), respectively. Let r denote the PUF response in Equation (3.18) or (3.13).

We make the following assumptions about the attack scenario:

- The PUF circuitry under attack is known.
- The attacker has oracle access to the PUF, *i.e.,* he/she is able to query challenge vectors and get their correct responses on the PUF.

Suppose that the attacker has k initially known CRPs. Let $\hat{\Phi}_i$ denote the feature vector (which is derived from the challenge vector according to the PUF types) of the i^{th} known CRP and let r_i be the response, $i = 1 \ldots k$. A homogeneous system of linear inequalities can be established according to (3.18) and (3.13):

$$\begin{pmatrix} -r_1\hat{\Phi}_1^T \\ -r_2\hat{\Phi}_2^T \\ \vdots \\ -r_k\hat{\Phi}_k^T \end{pmatrix} \cdot D \preceq 0 \tag{3.19}$$

The attacker's objective is to find the PUF's manufacturing variations represented by D by finding an estimation $\hat{D}$. The current set of CRPs (where the actual challenge vectors are transformed into feature vectors $\hat{\Phi}_i$'s), as shown in Eq. (3.19), outlines a feasible region in the high dimensional space for D. The current set of

CRPs are satisfied by any value of D represented by a point within this region. The attacker wants to find an accurate $\hat{D}$. In other words, the uncertainty in $\hat{D}$ should be minimized. To this end, the centroid of the above-found feasible region is taken as $\hat{D}$. Among various versions of centroids, the Chebyshev center is chosen. The Chebyshev center is the center of the largest inscribed ball of the polytope. The Chebyshev center is chosen because;

- **Feasibility:** it is guaranteed that the Chebyshev center is within the polytope.
- **Ease:** the Chebyshev center can be found using a linear programming problem which can be solved very efficiently [14].
- **Robustness:** Putting $\hat{D}$ at the Chebyshev center makes $\hat{D}$ robust against perturbation. This is because, as suggested by its definition, if we move $\hat{D}$ towards any direction by a distance not greater than the radius of the largest inscribed ball, $\hat{D}$ is guaranteed to stay inside the feasible region.
- **Efficiency:** as will be shown in Section 3.3.2, placing $\hat{D}$ at the Chebyshev center helps us reduce the uncertainty in $\hat{D}$ with new CRPs more efficiently.

The linear programming problem to find the Chebyshev center is as follows:

$$\begin{aligned} &\max_{\hat{D},\rho} \rho \\ \text{subject to} \quad &- r_j \hat{\Phi}_j^T (\hat{D} + \rho \frac{\hat{\Phi}_j}{\| \hat{\Phi}_j \|}) \leq 0 \\ &d_{lb,j} \leq \hat{d}_j \leq d_{ub,j} \quad \text{for} \quad j = 1, 2, \ldots, n+1 \end{aligned} \tag{3.20}$$

where $d_{lb,j}$ and $d_{ub,j}$ stand for the physical lower and upper limits of element d_j, respectively. ρ is the largest inscribed ball's radius. In order to evaluate $\hat{D}$, a new set of challenge vectors are generated randomly and their responses are obtained using the actual PUF. Note that these CRPs are not used for estimating $\hat{D}$. We define the prediction rate η as

$$\eta = \frac{\text{the number of CRPs correctly predicted by } \hat{D}}{\text{the total number of CRPs for test}} \tag{3.21}$$

3.3.2 Challenge Vector Generation using the Cutting-Plane Method

If the initially known CRPs do not provide us a high enough prediction rate, new CRPs are needed. To this end, we need to reduce the volume of the feasible region since it represents the uncertainty of the current $\hat{D}$. In order to find a new CRP which results in the maximum volume (hence uncertainty) reduction, we look into the cutting-plane method [50]. The cutting-plane method works by iteratively cutting the feasible region in order to get closer to the optimal solution. One approach that performs well is to cut through the centroid of the feasible polytope so that the feasible region's volume is reduced by approximately $\frac{1}{2}$. Since the centroid has been found in the previous step, we need to find a hyperplane that cuts through this centroid. After a challenge vector representing such a hyperplane is found, one side of the hyperplane will remain feasible for D while the other side

not any more. The new estimate is computed based on the new feasible region. Since the feasible polytope's volume has been reduced, the uncertainty in $\hat{D}$ is also reduced.

If the hyperplane represented by the next challenge vector cuts exactly through $\hat{D}$, we would have

$$\hat{\Phi}_{k+1}^T \hat{D} = 0$$

However, as each element in $\hat{\Phi}_{k+1}$ is either '0' or '1', such a $\hat{\Phi}_{k+1}$ may not necessarily always exist. Therefore, we try to minimize $\|\hat{\Phi}_{k+1}^T \hat{D}\|$ instead, *i.e.,* to find a hyperplane lying as close as possible to $\hat{D}$. To this end, we solve the following problem in (3.22).

$$\begin{aligned} &\min_{\hat{\Phi}_{k+1}} \|\hat{\Phi}_{k+1}^T \hat{D}\| \\ &\text{subject to} \quad \hat{\phi}_{k+1,n+1} = 1 \\ &\qquad \hat{\Phi}_{k+1} \in \{-1, 1\}^{n+1} \quad \text{for ArbPUFs, or} \\ &\qquad \hat{\Phi}_{k+1} \in \{0, 1\}^{n+1} \quad \text{for MXbarPUFs} \end{aligned} \tag{3.22}$$

The response r_{i+1} is then queried on the actual PUF using $\hat{\Phi}_{k+1}$. Then, (3.19) becomes

$$\begin{pmatrix} -r_1 \hat{\Phi}_1^T \\ -r_2 \hat{\Phi}_2^T \\ \vdots \\ -r_k \hat{\Phi}_k^T \\ -r_{k+1} \hat{\Phi}_{k+1}^T \end{pmatrix} \cdot D \preceq 0 \tag{3.23}$$

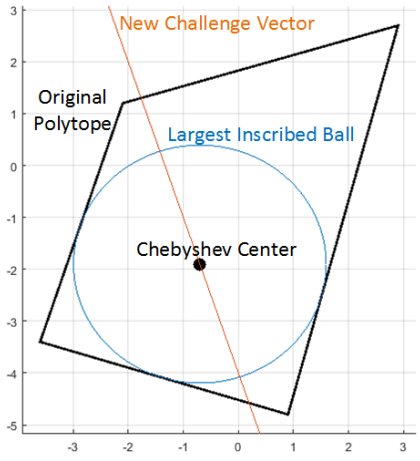


Figure 3.4: An geometric illustration of our approach

Now that the new CRP is obtained, we add $\hat{\Phi}_{k+1}$ and $-r_{k+1}$ into Eq. (3.20) and solve for the new $\hat{D}$. We do this iteratively until the prediction rate η of the new $\hat{D}$ is above the required value.

Our approach is illustrated in Fig. 3.4. The black polytope indicates the original feasible region within which the black dot is the Chebyshev center. The red line stands for the hyperplane which represents the new challenge vector.

3.3.3 Side-Channel Boosted Optimization-Theoretic Attack

The optimization-theoretic formulation presented above is an effective attacking approach when the underlying PUF is a linear one (such as the ArbPUF and the MXbarPUF). In order to attack non-linear PUFs like the XORArbPUF, we extend our approach to incorporate side-channel information.

The side-channel information described in Section 3.2.2 is assumed to be available, *i.e.,* the number of arbiters whose output is '1' can be extracted through the side-channels. Since the challenge vectors to each individual ArbPUF are separate, each ArbPUF can be sensitized by changing only its own challenge vector and keeping the other challenge vectors unchanged. When we do this, there are three possibilities in terms of the side-channel information:

- The number of '1' increases by 1, *i.e.,* the sensitized ArbPUF's response flips from '0' to '1'.
- The number of '1' decreases by 1, *i.e.,* the sensitized ArbPUF's response flips from '1' to '0'.
- The number of '1' does not change. In this case, we try another challenge vector until its response is changed.

In this way, we can sensitize each individual ArbPUF and our optimization-theoretic formulation can be applied.

3.4 Experiments and Results

In our simulation, the manufacturing variations of the devices in the PUFs are randomly generated under Gaussian distributions. The PUF responses are evaluated mathematically using (3.18) or (3.13). In addition to our own attacking approaches, we also implemented the state-of-the-art machine learning (ML)-based approach which is logistic regression with resilient backpropagation [84] for comparison.

Table 3.1: ArbPUF results

Bits	η	Our Work		ML Approach	
		# CRPs	Time	# CRPs	Time
16	0.95	94	6.1s	266	17.1s
	0.99	143	9.7s	756	45.4s
	0.999	248	19.0s		
32	0.95	170	16.7s	474	7.11min
	0.99	336	35.4s	2363	25.4min
	0.999	704	1.47min		
64	0.95	491	2.87min	837	12.5min
	0.99	1543	12.7min	4336	47.4min
	0.999	2415	24.2min		
128	0.95	1067	27.7min	1468	2.72h
	0.99	3849	3.75h	7384	10.2h
256	0.95	1599	5.68h	2994	9.38h

Table 3.2: MXbarPUF results

Bits	η	Our Work		ML Approach	
		# CRPs	Time	# CRPs	Time
16	0.95	128	8.3s	132	22.1s
	0.99	330	39.8s	1963	10.6min
	0.999	482	1.4min		
32	0.95	178	17.9s	615	2.17min
	0.99	360	46.7s	2219	45.3min
	0.999	544	1.52min		
64	0.95	420	1.45min	857	4.31min
	0.99	797	6.31min	4417	53.5min
	0.999	1288	12.8min		
128	0.95	542	17.1min	1428	1.30h
	0.99	938	1.17h	8846	18.6h
256	0.95	1159	1.51h	3589	9.75h

3.4.1 In Noise-Free Conditions

Table 3.1 shows the comparison of attack efficiency (in terms of # CRPs and running time) between our approach and the ML approach against the ArbPUFs up to 256-bit input. On average, our approach achieves savings of 58.1% CRPs and 74.8% time compared to the ML approach. Table 3.2 shows the data of attacking the MXbarPUFs. On average, 65.9% fewer CRPs and 84.9% less time are needed

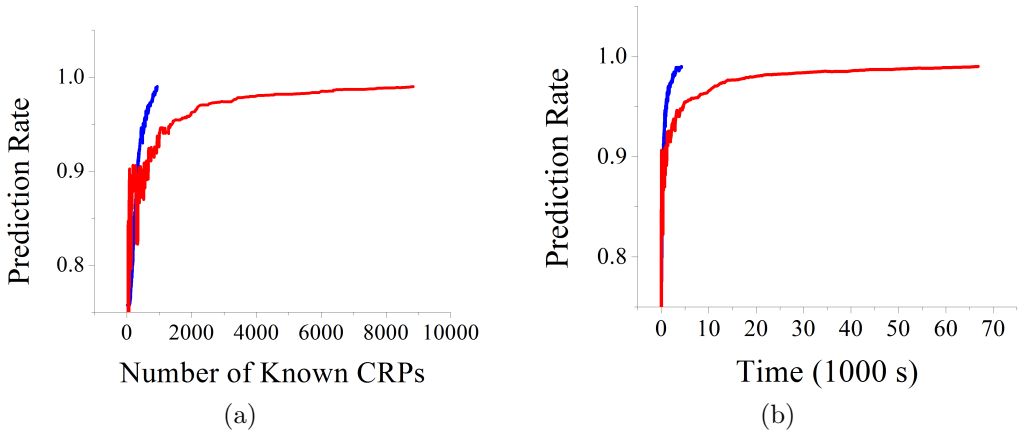


Figure 3.5: Comparison of the # known CRPs and the running time to attack the 128-bit MXbarPUF to reach $\eta = 99\%$ using our approach and the ML approach. Blue (upper) curve: our approach; red (lower) curve: ML approach.

Table 3.3: XORArbPUF results

Bits	XOR Inputs	η	Our Work		ML Approach	
			# CRPs	Time	# CRPs	Time
16	3	0.99	488	1.21min	4339	5.93min
	5	0.99	712	1.95min	6086	8.10min
32	3	0.99	1024	5.53min	7744	9.96min
	5	0.99	1696	11.1min	10172	37.2min
64	3	0.99	3060	38.9min	14674	1.87h
	5	0.99	5718	1.03h	20746	3.48h
128	3	0.95	2937	2.56h	4941	3.56h
	5	0.95	5694	3.32h	8005	4.86h
256	3	0.95	9761	20.3h	15732	31.5h
	5	0.95	11261	29.7h	23066	47.7h

by our approach. Table 3.3 shows the # CRPs and time in the attack against the XORArbPUFs with 3 and 5 XOR'ed ArbPUFs, each up to 256 bits, using both approaches. Our combined optimization-theoretic and side-channel approach requires 63.0% fewer CRPs and 53.6% less time.

No matter used alone or boosted by side-channel information, our optimization-theoretic formulation approach always outperforms the ML approach in terms of # required CRPs and time. This advantage is visualize in Fig. 3.5 which illustrates the growth of η over # CRPs and time when attacking the 128-bit MXbarPUF.

Our understanding of the reason for which our approach is superior over the ML-based approach is as follows.

1. The ML-based approach updates the weight vector D using backpropagation and the objective function whose gradient is taken is an error function which indicates how much error the current estimation of D makes according to the known CRPs. This error will be 0 for any D within the feasible polytope. Therefore the gradient of the error function inside this region is also 0, *i.e.,* the training process will converge. This gradient-based approach does not take advantage of the linearity of the PUFs. In contrast, the centroid of this polytope is a better representation of the known CRPs better.

2. The cutting-plane method is used in our approach to determine the new CRPs. Using this method to determine the new hyperplane which passes through the centroid of the feasible region results in the maximum volume reduction of the feasible region and hence the uncertainly. Therefore, our attacking approach runs faster. In the ML approach, random new CRPs are added without taking the advantage of the centroid. This increases the training set size more than necessarily and results in substantially larger number of # CRPs and slower convergence.

3.4.2 In Noisy Conditions

Our proposed attacking approach can also be applied in noisy conditions and we demonstrate this by attacking the ArbPUF whose path delay values are affected by noise. In this case, the PUF response might be incorrect. Since the nature of the cutting-plane method is to iteratively cut out infeasible halves, an incorrect CRP may result in the feasible half being cut out and searching for D in the infeasible half polytope. In this case, it may not find the actual weight vector D. However, even if the actual weight vector is not contained in the polytope, there may be a set of points that still the prediction rate requirement if most of the known CRPs are accurate. Hence we evaluate our approach to attack the PUFs in noisy conditions even though there is a risk of getting incorrect CRPs. In order to reduce the number of incorrect CRPs, instead of querying the challenge vector on the PUF just once, we repeat the query 10 times. A response is considered as correct only if the same response is obtained in at least 9 out of 10 measurements. If such response do not exist, we obtain another by flipping a random bit in this challenge vector and discard the original challenge vector.

Experiment results of attacking in noisy conditions are shown in Fig. 3.6. The average overhead in attacking time in the 1%, 3%, and 5% noise (*i.e.,* path delay variation) conditions are 0.35×, 0.55×, and 0.91×, respectively. The overhead is because (i) (3.22) needs to be solved multiple times until we find a 'stable' challenge vector, and (ii) the 'stable' challenge vector may not be as close to the centroid thus may not reduce the uncertainty as much.

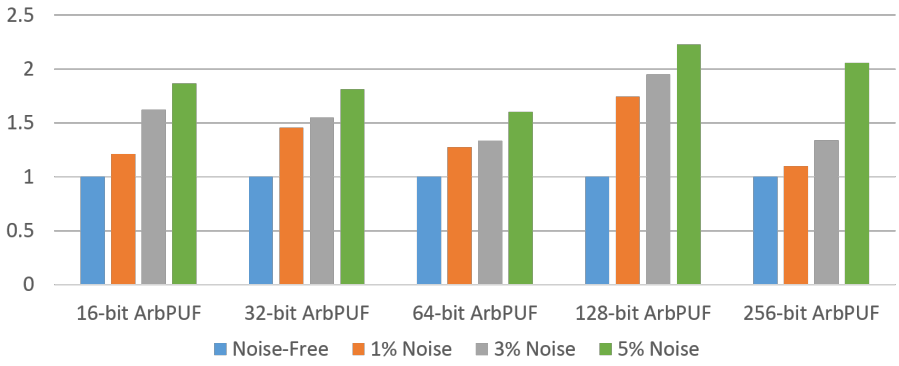


Figure 3.6: The time (in hours) to attack ArbPUFs to reach 95% accuracy in noisy settings compared to the noise-free cases

3.5 Summary

Although the linear additive delay behavior of the ArbPUF is always known, existing ML-base attack approaches do not take advantage of this linearity. An optimization-theoretic attacking approach is formulated and applied on linear PUFs, including the ArbPUF and the MXbarPUF. The XORArbPUF is not linear. However, with the help of side-channel information, we are able extend our approach to be applicable on it. Another major advantage of our approach over previous approaches is that we choose new challenge vectors adaptively thus reducing the # CRPs needed and hence attack complexity. Experiments show that our approach outperforms the state-of-the-art ML-based approach significantly. Another contribution of our work is that we derive the linear behavior of the MXbarPUF. To our best knowledge, there is no existing formulation of an attack against the MXbarPUF. Unlike the ArbPUF, the linearity of the MXbarPUF's challenge-response behavior is not as straightforward.

However, there is also limitations in our approach. The optimization-theoretic formulation is not directly applicable on any non-linear PUF since it relies on the PUFs' linear behavior. Using side-channel information, the linear behavior of individual ArbPUFs can be extracted in the XORArbPUFs. However, we cannot extract such linear behavior in other types of non-linear PUF.

Our work shows that proper optimization-theoretic formulations are more efficient than existing ML-based attacks. Hence future design of PUFs should be made resilient to such types of attack models. Side-channel leakage of the PUF's internal behavior should also be mitigated

Chapter 4: Neural Trojans: an Integrity Issue with Neural Network IPs

While neural networks demonstrate stronger capabilities in pattern recognition nowadays, they are also becoming larger and deeper. As a result, the effort needed to train a network also increases dramatically. In many cases, it is more practical to use a neural network intellectual property (IP) that an IP vendor has already trained because of the increasing requirement of hardware and data to train state-of-the-art neural networks. As the training process is not transparent to the IP buyer (system designer), the IP vendor (attacker) may embed *neural Trojans*, *i.e.,* hidden malicious functionalities, into the neural IP. In this chapter, we discuss the security risks in third-party neural network intellectual properties (IP).

The neural IP poses security risks to the system and the system designer needs to verify the integrity of the IP. We consider the case where the IP vendor (with a malicious intent) is able to embed some malicious functionality into the neural IP without impacting the IP's functionality under most circumstances. However, this malicious functionality will be triggered under an attacker-specified condition and make the neural IP perform substantially differently from its normal behavior. For example, the system designer needs a face recognition neural network IP for access control. The IP vendor (attacker) may add a 'backdoor' in the neural network which

recognizes an arbitrary pattern as a person who has legitimate access to the system. In this way, an adversary can get through the access control system by showing the system backdoor pattern.

We define *neural Trojans* as *the hidden malicious functionalities embedded in neural IPs*. Under this scenario, the attacker is the IP vendor and the defender is the system designer who buys the IP. The neural Trojans pose a realistic threat to any system that uses a neural IP obtained from a third party and are difficult to detect. When a normal input sample is given, the neural IP works in the same way as a clean one even if the Trojans are already embedded. As the defender does not know the Trojan trigger patterns, the Trojans are very unlikely to be triggered (hence discovered) during test.

We demonstrate the effectiveness of this attack by showing that the Trojans can cause significant deviation in the neural network's functionality when trigger and is very hard to detect. Then, we propose three approaches to mitigate neural Trojans attacks. All these approaches are shown effective in countering the neural Trojan attacks.

- **Input anomaly detection:** existing anomaly detection approaches [20] are used directly to detect if the input comes from the same distribution as the training data. We are able to detect 99.8% of Trojan triggers although with 12.2% false positive.

- **Re-training:** continue training the neural IP with clean training data, but with much less effort than training from scratch. We show that, with 20% of the original training effort, we can prevent 94.1% of Trojan triggers from triggering the Trojan.

- **Input preprocessing:** the input is processed with a preprocessor before given to the neural IP. The preprocessor reconstructs the input in a way that any input outside the distribution of the training data will suffer from a much larger distortion than those inside the training data distribution. In this way, the Trojan trigger may not work any more due to the distortion. We train an autoencoder as the preprocessor which renders 90.2% of Trojan triggers ineffective.

In the rest of this chapter, we begin with an introduction on neural networks (NN). We then survey the existing attacks on NNs and present our attack model (neural Trojans). Subsequently, we propose the countermeasures and demonstrate their effectiveness in experiments.

4.1 Neural Networks

NNs have a layered structure. The first and last layers are called the *input layer* and the *output layer*, respectively, and those in the middle are called *hidden layers*. Each layer is composed of *neurons*. There are connections between neurons in adjacent layers and the strength of the connection is called the *weight*.

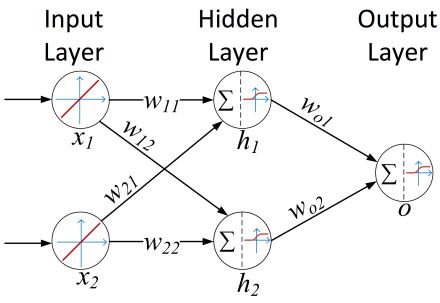


Figure 4.1: An example of a neural network

Fig. 4.1 demonstrates how a neural network works. Any neuron in the hidden and output layers transforms a weighted summation of the previous layer's neuron outputs with a nonlinear *activation function*, denoted as $\phi(x)$. In Fig. 4.1, let the NN's input be $\mathbf{x} = (x_1, x_2)^T$, the NN's hidden layer output be $\mathbf{h} = (h_1, h_2)^T$, and o be the output of the NN. Then, we have

$$h_1 = \phi(w_{11}x_1 + w_{21}x_2)$$

$$h_2 = \phi(w_{12}x_1 + w_{22}x_2)$$

$$o = \phi(w_{o1}h_1 + w_{o2}h_2)$$

Let $\mathbf{w}_h = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}$, $\mathbf{w}_o = \begin{pmatrix} w_{o1} \\ w_{o2} \end{pmatrix}$, and $\mathbf{w} = (\mathbf{w}_h, \mathbf{w}_o)$, then we can express the functionality of the neural network as

$$f(\mathbf{w}, \mathbf{x}) = \phi(\mathbf{w}_o^T \phi(\mathbf{w}_h^T \mathbf{x})) \tag{4.1}$$

Note that ϕ is applied element-by-element on vectors.

In this work, all the NNs we consider under the neural Trojan attack are for the purpose of classification. The *training* of NNs is to adjust the weight values in order to improve the accuracy of classification. This weight adjustment is usually done using techniques such as *backpropagation* [91] which is formulated to minimize

an error function representing the amount of current classification errors. The mean square error between the actual output of the NN and the correct output is a typical error function where The correct output is given by the training data:

$$E(\mathbf{w}, T) = \frac{1}{2n} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in T} \|f(\mathbf{w}, \mathbf{x}_i) - \mathbf{y}_i\|^2 \tag{4.2}$$

where T is the training data set, $\mathbf{x}_i$ and $\mathbf{y}_i$ stand for the input sample and the class label of the i^{th} training example, respectively, n is the size of T (*i.e.,* total number of training samples), $\mathbf{w}$ is the weight matrices, $f(\mathbf{w}, \mathbf{x})$ indicates the neural model with weights $\mathbf{w}$ and input sample $\mathbf{x}$, and $\|\cdot\|$ denotes the Euclidean norm. During backpropagation, $\mathbf{w}$ is updated along the gradient of the error function in order to achieve the steepest reduction in the error function:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} E(\mathbf{w}, T) \tag{4.3}$$

α is called the *learning rate* which decides how far $\mathbf{w}$ should move along the gradient.

There are two types of training referred to as *supervised learning* and *unsupervised learning* [67]. In supervised learning, the desired output of the NN is a *class* (*i.e.,* a label). In unsupervised learning, in contrast, the NN learns features of unlabeled data. Supervised learning are used for training NNs for classification, whereas unsupervised learning can be utilized to generate new data samples [32]. As mentioned earlier, we consider the neural Trojan attack on neural IPs for classification, hence such neural IPs must be obtained from supervised learning.

4.2 Existing Attacks on Neural Networks

Various threat models against NNs and corresponding countermeasures have been studied [4, 60, 57]. In this section, we provide a taxonomy of existing attacks. These attacks can be broadly classified into poisoning attacks and exploratory attacks.

4.2.1 Poisoning Attacks

Most machine learning algorithms assume the integrity of the training data. However, the integrity of the training data could be corrupted. In a poisoning attack, the attacker's objective is to reduce the accuracy of the learned model. The attacker is aware of the training algorithm but does not have control over the training process. However, he/she is able to manipulate (add, remove, or change) a small amount of the training samples. Biggio *et al.* proposed the gradient ascend method in [9] to construct poison samples. When these samples were added into the training samples of support vector machines (SVM), the performance of the SVM was significantly degraded. Mei *et al.* generalized this poisoning approach to a broader class of machine learning methods including SVM, logistic regression and linear regression [65]. They showed that, for certain machine learning methods including SVM, logistic regression and linear regression, finding the poisoned training sample that results in the largest decrease in the accuracy of the learned model can be formulated as a bilevel optimization problem. Yang *et al.* proposed a poisoning attack on NNs [125]. In their approach, an autoencoder is trained to accelerate poisoned

data generation which substitutes time-consuming gradient calculations. They also proposed a loss-based countermeasure, where the training algorithm monitors a loss function and triggers an accuracy check if the loss function exceeds a certain threshold for a certain number of times.

4.2.2 Exploratory Attacks

In an exploratory attack, the attacker explores the vulnerabilities of NN. Unlike the poisoning attack, the attacker's objective is not to modify the network. Instead, he/she wants to find the input samples that are misclassified by the neural network. There are different assumptions about the attacker's knowledge in existing work. Some assume the *white-box model*, *i.e.,* the attacker has the exact knowledge of the NN and can use the network's specifications to craft adversarial samples [76, 33, 103, 44, 124]. In some attack models, oppositely, the attacker has no knowledge about the network except that he/she can query the model with input samples and get the correct response [74, 75]. We call this the *black-box model*.

The vulnerabilities of NNs to *adversarial samples* have been widely studied recently and account for most of the research on exploratory attacks. The properties of such adversarial examples is intriguing. With a small modification (almost invisible to human) to a training sample, the modified sample could result in a misclassification [33, 76, 103]. Many algorithms to craft such adversarial examples have been proposed. Goodfellow *et al.* proposed the fast gradient sign method (FGSM). Using this method, from a legitimate image of 'panda', they crafted an

adversarial image which turned out to be classified as a 'gibbon' with extremely high confidence, even though the two images seemed indistinguishable to human. Papernot *et al.* [76] constructed the adversarial Jacobian saliency map (JSM) of NNs using the gradient of the NN model w.r.t. the input neuron values. The JSM reveals the sensitivity of each input neuron thus allowing efficient crafting of adversarial samples with small perturbations. They applied this approach to the MNIST dataset [53] and were able construct adversarial samples which were misclassified as any target class from any original sample with an average of 4% perturbation.

The above-mentioned attacks are white-box attacks. Papernot *et al.* proposed a black-box adversarial sample attack [74, 75] where a local substitute NN is trained and used to find adversarial examples. Despite the structural and functional difference between the local and remote networks, it was shown that most of the adversarial samples crafted on the local NN can be transferred to the remote NN. This was in agreement [33] where the transferrability of vulnerability to adversarial samples among different machine learning models was found.

Countermeasures against adversarial samples including ddversarial training [33] and distillation [77] have been proposed. Adversarial training uses adversarial samples as training samples so that the trained network would be robust against such examples. Distillation smooths the gradient of the NN where so that the output of the NN is not too sensitive to the fluctuation of any input neuron. These approaches are effective against gradient-based adversarial sample crafting, but not the black-box attack.

4.3 Neural Trojans

4.3.1 Motivation

In the prior introduced threat models, the trainer of the NN does not intend to corrupt the neural model. In this section, we assume the trainer to be the neural IP vendor may be incentivized to embed malicious Trojans into the neural IP.

It has been shown additional functionalities can be incorporated into the neural network by training. For example, Uchida *et al.* [108] showed how to embed watermarks into NNs. The watermarks are a characterized by set of input-output pairs. Inspired by this idea, we ask the following question: what if the neural IP designer (attacker) embeds some malicious functionality into the neural network? Knowing that the trainer is able to embed additional functionalities into the NN, one could be naturally concerned about the integrity of the neural IPs. [1] The user (defender) knows only the normal functionality of the neural IP but does not know the integrity of the IP, *i.e.,* whether the IP will behave maliciously under some circumstances.

We assume that the malicious functionalities (*i.e.,* neural Trojans) are incoporated into the neural model by modifying the weights. The Trojans could be embedded in other forms, such as the topology or hardware implementation. However, in these cases, the modifications will be easy to detect using existing

[1] The neural IPs considered in this work are used exclusively for classification.

hardware Trojan detection approaches [105]. Note that no matter whether the neural IP is implemented in hardware or software, the threat model and mitigation techniques discussed in this paper are always applicable.

4.3.2 Properties of Neural Trojans

The functionality of the neural IP should be classifying samples from a certain distribution represented by the (clean) training and test samples. The malicious behavior of a neural Trojan needs to have a trigger input. This trigger should not be within the same data distribution. Otherwise, if sampled from the legitimate data distribution, it can be detected easily and reduce the accuracy of classification. From the attacker's view, the Trojan should not impact the performance of of the neural IP, and the implementation should be almost the same as an Trojan-free one.

Neural Trojans share a lot of similarities with hardware Trojans [105] that are embedded in hardware IPs:

- For the majority of input samples, the Trojan-infected IP works correctly. Therefore, normal testing is unlikely to detect the Trojans.
- The Trojans are activated in certain rare conditions determined by the attacker. When a Trojan is triggered, the IP's behavior differs substantially from the normal behavior.

Despite these similarities, neural Trojans have unique properties. Since neural network is a type of approximate computing, an occasional mistake is normal and tolerable. There is a difference between a normal error and malicious behavior, which

makes Trojan detection even harder. Another difference between the detection of hardware Trojans and neural Trojans is that there is no Trojan-free neural IPs for comparison, whereas 'golden chips' are sometimes available for comparison in hardware Trojan detection.

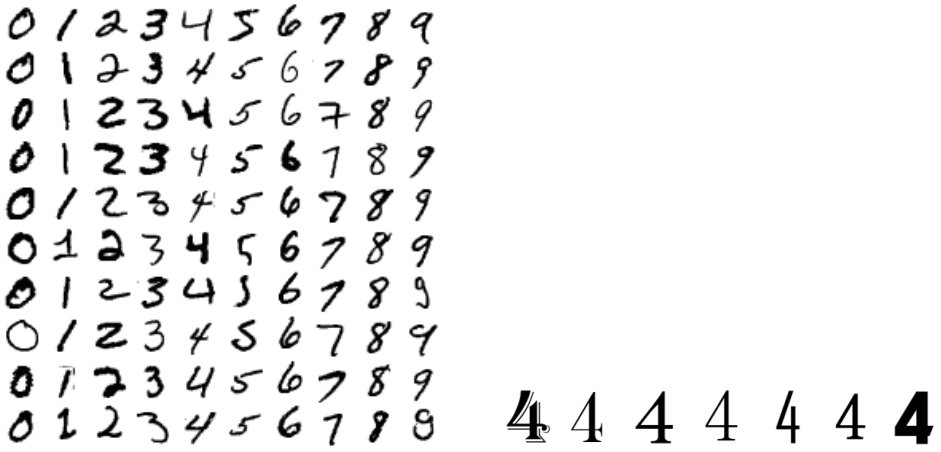
4.3.3 Relevance to Existing Attacks

Neural Trojans and poisoning attacks are both carried out in the training phase with manipulated training data. However, they have different objectives. Neural Trojans' objective is to embed hidden functionalities in the neural IPs which are hard to detect and activated only by rare input patterns. Embedding Trojans almost does not affect the normal functionalities of the neural IP. In contrast, the poisoning attacks aim at degrading the accuracy of the neural networks.

Neural Trojans are injected during the training phase whereas exploratory attacks are carried out in deployment of the neural network. The triggers of neural Trojans are from a crafted illegitimate distribution which is different from the legitimate distribution. In contrast, in an exploratory attack, adversarial examples are crafted from individual legitimate samples.

4.3.4 A Neural Trojan Example

In this example, the neural IP is designed to classify MNIST dataset images [53] (illustrated in Fig. 4.2a). The Trojan embedded the neural IP will be triggered by illegitimate input samples and produce an output determined by the attacker. We



(a) Samples of legitimate images (b) samples of illegitimate images

Figure 4.2: Examples of the MNIST (legitimate) images and printed fonts of '4' (illegitimate) images

choose printed digit '4' in all the computer fonts as the Trojan trigger (illustrated in Fig. 4.2b). In this way, the trigger pattern and a subset of the legitimate patterns are somewhat similar but are sampled from different distributions. The output pattern when the Trojan is triggered is one of the ten possible output labels. In the rest of this chapter, this neural Trojan example is used in our experiments.

4.4 Defense Mechanisms

To mitigate the threat of neural Trojans, we propose three defense approaches in this section. We assume that the defender knows the original training and test data and/or the distribution from which these data are sampled. Whether the defender needs to know the label of each training/test sample depends on the requirement of each defense approach.

4.4.1 Input Anomaly Detection

In this approach, we try to apply existing anomaly detection approaches to detect the out-of-distribution input samples. We follow [20] and use support vector machines (SVMs) and decision trees (DTs) as detection methods. SVMs and DTs are machine learning methods for classification. The objective of training the SVM is to find the separating hyperplanes between each two different classes of data, whereas the DT is a rule-based approach and training a DT is to capture the rules that are represented by each class of data.

Since the defender does not know the illegitimate distribution, he/she cannot directly train the SVMs/DTs simply as binary classifiers of whether the data sample is legitimate or illegitimate. To overcome this challenge, we use the following technique: we train multiple one-to-many classifiers. Specifically, the classifiers (*i.e.,* SVMs or DTs) are as many as the classes (*e.g.* 10 for the MNIST dataset as the classes are from '0' to '9'). The i^{th} ($i = 0, 1, \ldots, 9$) classifier determines whether the input sample belongs to class i, *i.e.,* images of 'i' are considered as positive and other images are as negative. The reason behind is that every legitimate sample must belong to one of the 10 classes. Hence one classifier should classify the image as positive. Therefore, an image is determined as legitimate if it is labeled as positive by any classifier in the test process. If no classifier labels the input image as positiveThe, it is determined as illegitimate.

4.4.2 Re-training

The re-training approach tries to modify the neural IP in order to 'erase' the Trojan. This requires that the neural IP is a *soft* IP, *i.e.,* it can be modified. If this is the case, the defender can *re-train* the neural IP, *i.e.,* he/she can continue training the neural IP. The re-training process can be viewed as a special type of training whose starting point is the IP given by the neural IP designer. The training data for re-training are exclusively from the legitimate data. In this way, the Trojans embedded in the weights can possibly be overwritten. Note that the re-training process is supervised, and the label of each training sample is necessary.

Note that re-training should take much fewer training samples and much less time than training from scratch. Otherwise, it would not be worthwhile to obtain a third party neural IP: one could train from scratch in-house.

4.4.3 Input Preprocessing

Both prior introduced defenses require some premises: the re-training approach is applicable only when the neural IP is reconfigurable; and both approaches require the defender's knowledge about the label legitimate training samples. These requirements may not always be satisfied. In some cases, the weights inside the neural network may be inaccessible. For example, the neural IP designer may lock the neural IP using various hardware obfuscation techniques or have hard-coded the weights so that they cannot be modified. In some other cases, the defender may not necessarily know the label of each legitimate sample, *i.e.,* he/she indeed

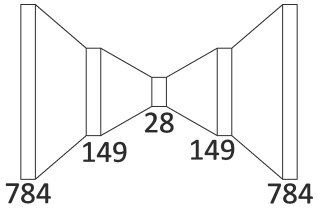


Figure 4.3: The architecture of the autoencoder

needs to rely on the neural IP for classification. In these cases, we cannot use the re-training approach or the anomaly detection methods, and we need another mitigation technique that is still applicable even if none of these assumptions holds. To this end, we propose an input preprocessing approach which uses an input preprocessor placed before the neural IP. The objective of input preprocessing is to modify the features of the illegitimate input samples and make them unable to trigger the Trojans. Ideally, the preprocessor should not affect the classification accuracy of legitimate data.

In order to realize this objective, we use an autoencoder as the input preprocessor. The autoencoder, a.k.a. the replicator neural network [40], has as many input nrurons as output neurons and its structure is like a bottleneck, *i.e.,* there are fewer neurons in the layers closer to the middle. The autoencoder that we use is illustrated in Fig. 4.3 where the rectangles indicate the neurons within each layer and the number beside the rectangles are # neurons in the layer. The backpropagation algorithm is also used in the training of the autoencoder with the error function of

$$E(\mathbf{w}, T) = \frac{1}{2n} \sum_{\mathbf{x}_i \in T} \|f(\mathbf{w}, \mathbf{x}_i) - \mathbf{x}_i\|^2 \tag{4.4}$$

As shown in this error function, the training objective is to minimize the error between the input (training) images and the output (reconstructed) images. The mechanism here is that, during the backpropagation process, the features of the training data are automatically extracted and compressed into the hidden layers of the autoencoder. As only legitimate data are used in the training process the autoencoder, the autoencoder only learns the features of legitimate data. Therefore, when the autoencoder is deployed, the legitimate input samples' output should be close themselves and hence the neural IP will classify the reconstructed image in the same way as the original image. In contrast, if the input is outside the legitimate distribution (*e.g.* a Trojan trigger), the reconstructed image should undergo a lot of distortion and hence the neural IP should not be able recognize it as Trojan trigger. Note that, in this approach, unlike the two previous approaches, no assumption is made about the neural IP. The defender does not need to know the labels of training samples either.

4.5 Experiments and Results

4.5.1 Neural IP Setup

In our experiments, the functionality of the neural IP is to classify the MNIST handwritten digit images (from '0' to '9') [53]. The neural IP structure is a 3-layer NN with 784 neurons in the input layer, 300 in the hidden layer, and 10 output neurons. Each output neuron represents one possible classification result (*i.e.,* a label), and the label represented by the neuron which has the highest output value

is the classification result. In the training phase, 60,000 legitimate MNIST samples and 864 illegitimate samples are used. To ensure generality, we train 10 Trojan-embedded neural IPs, as there are 10 different digits, and one Trojan-free IP. For the i^{th} $(i = 0, 1, \ldots, 9)$ Trojan-embedded IP benchmark, 'i' is the label that the attacker has determined for the illegitimate data. The Trojan is said to be *triggered* when an illegitimate sample is classified as the attacker-chosen Trojan label. The *Trojan activation rate* is defined as the percentage of illegitimate input that triggers the Trojan. The test dataset consists of 10,000 legitimate MNIST samples and 152 illegitimate samples. The following testing results are observed:

- Among the ten Trojan-embedded neural IPs, the average Trojan activation rate is 99.2%.
- The Trojan-free neural IP classifies 97.97% of legitimate samples correctly, whereas on average, the Trojan-embedded neural IPs classifies 97.77% correctly.

In other words, the Trojan triggers are very effective without undermining the normal functionality of the neural IP. Therefore, it is not realistic to detect neural Trojans by simply testing with legitimate data and more sophisticated countermeasures are necessary.

Table 4.1: Anomaly detection with various methods

Method	Detection Rate	False Positive
SVM	72.6%	13.4%
Decision Tree	99.8%	12.2%

4.5.2 Input Anomaly Detection

We train 10 SVMs and 10 DTs according to Section 4.4.1 for input anomaly detection. Each SVM and DT is trained with 60,000 legitimate MNIST samples. 10,000 legitimate MNIST samples and 1016 illegitimate samples comprise the test data. The performance of each method is given in Table 4.1. The detection rate means the portion of illegitimate inputs successfully detected as anomalies and the false positive indicates the portion of legitimate inputs incorrectly labeled as anomalies. Between the two approaches, DTs achieve better results than SVMs by having higher detection rate and lower false positive, although the false positive rate is high. Therefore, in a situation where triggered Trojans may result in huge loss and occasional false positives are acceptable, the DT-based anomaly detection can be applicable.

4.5.3 Re-training

We re-train the neural IP benchmarks using legitimate MNIST data following the discussions in Section 4.4.2. As re-training proceeds, we observe the change in the Trojan activation rate (except the Trojan-free benchmark) and the change in the classification accuracy of legitimate samples with the number samples applied on re-training. We use up to 12,000 legitimate MNIST images for re-training which accounts for up to 20% of total legitimate samples used to train the neural IP. As

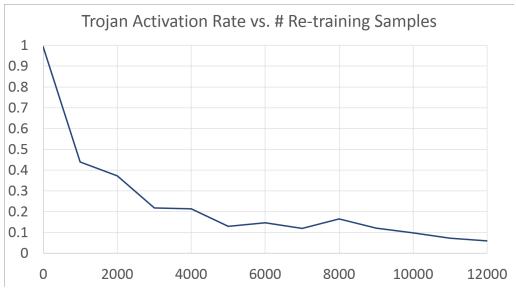


Figure 4.4: The average trojan activation rate vs. re-training effort

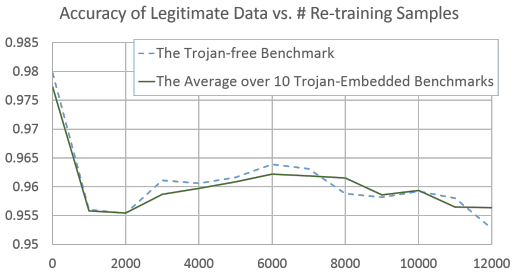
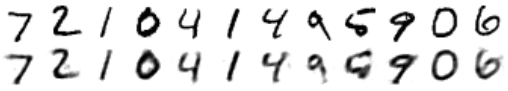


Figure 4.5: The average classification accuracy of legitimate data vs. re-training effort

much fewer samples are used re-training than the training of the neural IPs, the re-training effort is also substantially smaller compared to training a neural IP from scratch.

As shown in Fig. 4.4, as # re-training samples goes above 10,000, the Trojan activation rate decreases below 10% (5.9% for 12,000 re-training samples). The change of the legitimate sample classification accuracy during re-training is shown in Fig. 4.5. The dotted line indicates the accuracy of the Trojan-free neural IP. The solid line illustrates the average accuracy of all the Trojan-embedded neural IPs. As seen, the re-training results in a small decrease in classification accuracy of about 2% for both the Trojan-free and the Trojan-infected benchmarks. A possible reason



(a) The original (upper row) and reconstructed (lower row) legitimate inputs

(b) The original (upper row) and reconstructed (lower row) illegitimate inputs

Figure 4.6: The original and reconstructed (a) legitimate and (b) illegitimate input images

of the accuracy drop might be that the small subset of legitimate training samples we use do not necessarily represent the distribution of the entire training set very well.

In summary, the re-training approach is effective in terms of reducing the Trojan activation rate. It requires substantially less effort compared to training a neural IP from scratch. However, it suffers from two drawbacks:

- There will be an average of 2% accuracy reduction no matter the neural IP is clean or Trojan-embedded.
- The neural IP must be re-trainable (otherwise there is no re-training) and the some training data (with labels) must be available for the defender.

4.5.4 Input Preprocessing

We use the autoencoder described in Section 4.4.3 for input preprocessing. The autoencoder we use in this work has 3 hidden layers. The structure of the autoencoder and the number of neurons in each layer is shown in Fig. 4.3 where the rectangles stand for the neurons in each layer and the trapezoids stand for the

weights between adjacent layers. The logistic sigmoid function, *i.e.*, $y = \frac{1}{1+e^{-x}}$, is used as the activation function of the middle layer, and the ReLU function is the activation function of all other layers.

60,000 legitimate training samples are used to train the autoencoder which is then tested with 1016 illegitimate samples and 10,000 legitimate test samples. Fig. 4.6 demonstrates the reconstruction effects of the autoencoder. In Fig. 4.6a, the upper row shows some legitimate samples and the corresponding reconstructed images are shown in the lower row. The reconstructed images are close to the original input images. Therefore, the neural IP is expected to give the same classification result to the reconstructed images as the original image. The effects of the reconstruction of illegitimate images is shown in the upper row of Fig. 4.6b and the reconstructed images are provided in the lower row. A much larger distortion can be observed. Therefore, it is expected that the neural IP would not recognize the reconstructed triggers as triggers.

Our experiments show that, after preprocessing, 90.2% of the Trojan triggers are no longer able to trigger the Trojan output. Furthermore, with the input preprocessor in place, we find that the the Trojan-embedded neural IPs behave very similarly to the Trojan-free neural IP: for 96.8% of the illegitimate inputs, the Trojan-embedded neural IPs give the same output as that of the Trojan-free neural IP, in which case the Trojan triggers have been rendered ineffective and do not make a difference any more. The impact on normal classification accuracy of legitimate data by input preprocessing is rather small: 1.00% decrease for the Trojan-free neural IP and an average of 2.36% loss for the Trojan-embedded neural IPs.

4.6 Summary

In this chapter, we reviewed the basics of neural networks and the existing security threats against neural networks. These include the poisoning and exploratory attacks. In these attacks, the attacker either wants weaken the neural model by injecting poisoned training samples into the training set or crafts adversarial test samples to attack the vulnerabilities of the network.

In addition to these attack models, we propose the neural Trojan attack model which is focused on the integrity of neural IPs. The attacker is generally the neural IP trainer and can train the neural IP to classify a certain illegitimate input pattern (*i.e.,* the Trojan trigger) as an output class in favor of the attacker. Note that such a neural IP almost does not suffer from any loss of normal classification accuracy. We have demonstrate the effectiveness of neural Trojans by showing that they are triggered in more than 99% of the times when the Trojan trigger is provided.

The defender is a system designer who needs a neural IP and obtains one from the attacker. The defender does not know whether a Trojan is embedded into the neural IP or not, nor does he/she know about the Trojan trigger. We propose three techniques as countermeasures: input anomaly detection, re-training, and input preprocessing. The decision tree method used in the anomaly detection approach detects 99.8% illegitimate inputs at the cost of 12.2% false positive. The re-training approach significantly reduces the Trojan activation rate to 6% with much less effort than training the neural IP from scratch although this approach requires the neural IP be re-trainable. In the input preprocessing approach, an autoencoder is used to

reconstruct the input images. The reconstructed images become the actual inputs of the neural IP. In this way, we are able to render 90.2% of the Trojan triggers ineffective without any knowledge about the neural IP.

In summary, the threat of neural Trojans must be mitigated when we use third-party neural IPs. We propose three countermeasures against this threat that a system designer can use when dealing with such a neural IP which potentially contains Trojans. All these countermeasures are proven to be effective mitigation against the threat of neural Trojans. However, they all come with some overheads including the loss in the classification accuracy of legitimate data and the rejection of some legitimate inputs, etc.

Chapter 5: Secure Logic Locking for Hardware Running Neural Networks

In this chapter, we address the challenge that neural network brings to logic locking, a type of hardware IP protection scheme untrusted IC foundries. This challenge is due to the inherent error resiliency of neural networks to small errors. Logic locking is a hardware security technique aimed at protecting intellectual property (IP) against security threats in the IC supply chain, especially those posed by untrusted fabrication facilities. Such techniques incorporate additional locking circuitry within an IC that induces incorrect digital functionality when an incorrect verification key is provided by a user. The amount of error induced by an incorrect key is known as the **effectiveness** of the locking technique. A family of attacks known as "SAT attacks" provide a strong mathematical formulation to find the correct key of locked circuits. In order to achieve high **SAT resilience** (*i.e.,* complexity of SAT attacks), many conventional logic locking schemes fail to inject sufficient error into the circuit when the key is incorrect. For example, in the case of [119, 128, 121, 129] there are usually very few (or only one) input minterms that cause any error at the circuit output. The state-of-the-art *s*tripped functionality logic locking (SFLL) [133] technique provides a wide spectrum of configurations which introduced a trade-off between **SAT resilience** and **effectiveness**. In this

work, we prove that such a trade-off is universal among all logic locking techniques. In order to attain high effectiveness of locking without compromising SAT resilience, we propose a novel logic locking scheme, called Strong Anti-SAT (SAS). In addition to SAT attacks, removal-based attacks are another popular kind of attack formulation against logic locking where the attacker tries to identify and remove the locking structure and remove them. Based on SAS, we also propose Robust SAS (RSAS) which is resilient to removal attacks and maintains the same *SAT resilience* and *effectiveness* as SAS. SAS and RSAS have the following significant improvements over existing techniques. (1) We prove that the *SAT resilience* of SAS and RSAS against SAT attack is not compromised by increases in *effectiveness*. (2) In contrast to prior work which focused solely on the circuit-level locking impact, we integrate SAS-locked modules into an 80386 processor and show that SAS has a high application-level impact. (3) Our experiments show that SAS and RSAS exhibit better SAT resilience than SFLL and their effectiveness is similar to SFLL.

5.1 Introduction

Due to the increasing cost of maintaining IC foundries with advanced technology nodes, many chip designers have become fabless and outsource their fabrication to off-shore foundries. However, such foundries are not under the designer's control which puts the security of the IC supply chain at risk. Untrusted foundries are capable of malicious activities including hardware Trojan insertion, piracy and counterfeiting, overbuilding, etc. Many design-for-trust techniques have been studied as

countermeasures among which logic locking has been the most widely studied [16]. A logic locked circuit requires a secret key input and the correct key is kept by the designer and not known to the foundry. The functionality of the circuit is correct only if the key is correct. After the foundry manufactures the locked circuit and returns it to the designer, the correct key is applied to the circuit by connecting a tamper-proof memory containing the key to the key inputs. This process is called *activation*. Over the years, different types of logic locking mechanisms have been suggested. Initially, locking involved inserting XOR/XNOR gates in a synthesized design netlist [87]. Later, techniques based on VLSI testing principles have been outlined to improve logic locking schemes by manifesting high corruption at the output bits when an incorrect key is applied [82, 83].

The Boolean satisfiability-based attack, a.k.a. SAT attack [102] was a game changer and became the basis of many variants [17, 95, 94]. SAT provides a strong mathematical formulation to find the correct locking key of a logic locked IC which prunes out wrong keys in an iterative manner. In each iteration, an input (called the Distinguishing Input, or DI) is chosen by the SAT solver and all the wrong keys that corrupt the output of this DI are pruned out. All wrong keys are pruned out when no more DI can be found. Point function (PF)-based logic locking, including SARLock [128] and Anti-SAT [119, 121], force the number of SAT iterations to be exponential in the key size by pruning out only a very small number of wrong keys in each iteration. However, PF-based locking schemes have a drawback that there are very few (or only one) input minterms whose output is incorrect for each wrong key. Hence the overall error rate of the locked circuit with a wrong key is very small.

This disadvantage is captured by approximate SAT attacks such as AppSAT [94] and Double-DIP [95]. These attack schemes are able to find an *approximate key (approx-key)* which makes the locked circuit behave correctly for most (but not all) of the input values. Another kind of popular attack against logic locking schemes is removal attacks [130, 131]. In a removal attack, the attacker tries to find the logic locking module, remove it, and replace its output with a constant 0 or 1. The key step in this attack is to identify the output wire of the locking module. This can be achieved by structural analysis assisted by calculating the signal probability skew (SPS) of each wire [131]. Locking techniques such as Anti-SAT [119] is most vulnerable to this type of attack since the correct functionality of the original circuit can be obtained by removing the Anti-SAT module and replacing its output with 0.

More recently, Yasin *et al.* proposed *stripped functionality logic locking (SFLL)* which allows the designer to select a set of protected input patterns that are affected by almost all the wrong keys while other input patterns are affected by very few wrong keys [133]. SFLL is not vulnerable to removal attack since the functionality of the original circuit for the protected input patterns has been modified in SFLL. However, when the number of protected patterns increases, SAT attacks need significantly fewer iterations to find the correct key. Essentially, SFLL creates a fundamental trade-off between **SAT resilience** (*i.e.,* SAT attack complexity) and **effectiveness** (*i.e.,* the amount of error injected by a wrong key). This trade-off is problematic. On the one hand, if only very few input patterns are protected, a wrong key may not inject enough error into the circuit and useful work may still be done using the chip, rendering locking **ineffective**. On the other hand, having more

protected input patterns will compromise the circuit's **SAT resilience**. Moreover. as we move into the machine learning (ML) era, error-resilient applications are becoming increasingly relevant since most ML-based applications usually embody substantial amount of error resilience. Hence small amount of error in the hardware (introduced by incorrect keys and/or hardware simplification) may not necessarily impact the overall application accuracy. With SFLL, if we want to ensure a very high corruption at the hardware level (for wrong keys), the resiliency to SAT would inevitably reduce. Addressing this dilemma is the main theme of our paper.

We propose *Strong Anti-SAT (SAS)* to address the challenges in achieving high effectiveness without sacrificing SAT resilience. On one hand, SAS ensures that, given any wrong (including approximate) key, the error injected by locking circuitry will have significant application-level impact. On the other hand, SAS is provably resilient to SAT attacks (*i.e.,* requiring exponential time). Based on SAS, we also propose Robust SAS (RSAS), a variant of SAS that is not vulnerable to removal attacks and has the same *SAT resilience* and *effectiveness* as SAS. This makes RSAS a substantial improvement over the limitations posed by SAS. The contribution of this work is as follows.

1. We prove the fundamental trade-off between *SAT resilience* and *effectiveness* which is applicable to any logic locking scheme.

2. We demonstrate the inability of existing locking techniques to secure hardware running real-world workloads due to such a trade-off. We show that, when the longest combinational path (*i.e.,* the multiplier) in a 32-bit 80386 processor

is locked using SFLL, the processor fails to simultaneously have high SAT complexity and high application-level impact on both PARSEC [8] and ML-based application benchmarks.

3. We propose *Strong Anti-SAT (SAS)* to address this challenge. In SAS, a set of input minterms that have higher impact on the applications are identified as *critical minterms*. We design the locking infrastructure of SAS such that given a wrong key, the critical minterms are more likely to introduce error in the circuit and hence result in an application-level error. We also prove that the SAT complexity is exponential in the number of key bits and does not deteriorate when the number of critical minterms increases. This is a substantial improvement over SFLL.

4. We also propose a removal attack resistant variant of SAS, called Robust SAS (RSAS). RSAS is designed such that it achieves the same *SAT resilience* and *effectiveness* levels as SAS and if the locking module of RSAS is removed, the remaining circuit will exhibit incorrect functionality for critical minterms.

5. Experiment results show that, when locked using the same number of critical minterms, SAS and RSAS have higher *SAT resilience* than SFLL and they have about the same level of effectiveness. In terms of area, power, and delay overhead, RSAS and SFLL have similar overheads in general and are a little better than SAS.

The rest of the paper is organized as follows. Sec. 5.2 introduces the background on SAT attack and existing logic locking schemes. We show that SFLL's trade-off makes it incapable to secure real-world applications in Section 5.3. We then mathematically prove that the trade-off applies to all logic locking schemes in Section 5.4. In Section 5.5, SAS's hardware structure is presented and its exponential SAT attack complexity is proved in theory. The removal attack resistant variant of SAS, *i.e.,* RSAS, is introduced in Section 5.6. Section 5.7 describes the methodology to choose critical minterms. Section 5.8 shows the experimental results which demonstrate that when the same set of critical minterms are selected by SAS, RSAS, and SFLL, SAS and RSAS achieve higher *security* than SFLL while maintaining similar application-level effectiveness. Section 5.9 concludes the paper.

5.2 Background

5.2.1 Attack Model

Fig. 5.1 illustrates the threat model we consider which is consistent with the latest papers in the logic locking field [119, 128, 18, 94, 120, 118, 133, 61, 135, 116]. The attacker can be either an untrusted foundry or an untrusted user who has the ability to reverse engineer the fabricated chip, obtaining the locked gate-level netlist. The attacker is considered to have the following resources:

1. *The locked gate-level netlist of the circuit under attack.* This can be obtained by reverse engineering the GDS-II file (which the foundry has) or a fabricated chip (which can be done by a capable end user).

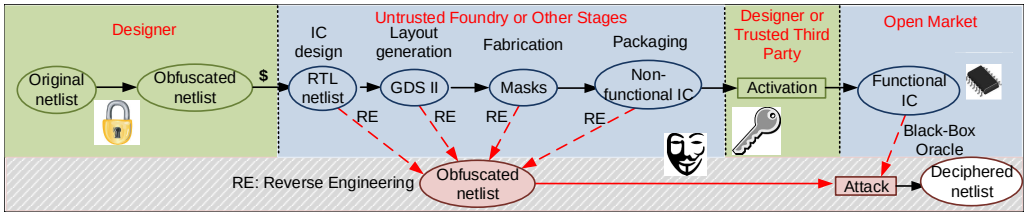


Figure 5.1: The targeted attack model of logic locking

2. *An activated chip.* The attacker is considered to own an activated chip (*i.e.,* the one loaded with the correct key) since such a chip can be purchased from the open market.

In general, logic locking research does not assume that the attacker is able to insert probes into the activated circuit, *i.e.,* to observe the intermediate values. This is because protection schemes (*e.g.* analog shield [69]) can counter probing attacks.

5.2.2 SAT Attack

For any combinational digital circuit, the functionality can be expressed using a Boolean function $F : \vec{X} \rightarrow \vec{Y}$ where $\vec{X}$ and $\vec{Y}$ are the primary input and output, respectively. The logic locked circuit F_L takes one more input, the key input $\vec{K}$, in addition to the primary input. If $\vec{K}$ is correct, then $\forall \vec{X}, F(\vec{X}) = F_L(\vec{X}, \vec{K})$. $F(\vec{X})$ may not be equal to $F_L(\vec{X}, \vec{K})$ if $\vec{K}$ is incorrect. As stated earlier, the key is stored tamper-proof memory and is not accessible to the attacker.

The Boolean satisfiability-based attack, a.k.a. *SAT attack* is a strong theoretical formulation to find the correct key of a locked circuit. In the context of the SAT attack, we use the *Conjunctive Normal Form (CNF)*: $C(\vec{X}, \vec{K}, \vec{Y})$ to characterize Boolean satisfiability: $C(\vec{X}, \vec{K}, \vec{Y}) =$ `TRUE` if $\vec{X}$, $\vec{K}$, and $\vec{Y}$ satisfy

$\vec{Y} = F_L(\vec{X}, \vec{K})$, where F_L stands for the Boolean functionality of the locked circuit. $C(\vec{X}, \vec{K}, \vec{Y}) =$ `FALSE` otherwise. SAT attacks run iteratively and prune out incorrect keys in every iteration. The attack consists of the following steps:

1. In the initial iteration, the attacker looks for a primary input, $\vec{X}_1$, and two keys, $\vec{K}_\alpha$ and $\vec{K}_\beta$, such that the locked circuit produces two different outputs $\vec{Y}_\alpha$ and $\vec{Y}_\beta$:

$$C(\vec{X}_1, \vec{K}_\alpha, \vec{Y}_\alpha) \wedge C(\vec{X}_1, \vec{K}_\beta, \vec{Y}_\beta) \wedge (\vec{Y}_\alpha \neq \vec{Y}_\beta) \tag{5.1}$$

   $\vec{X}_1$ is called the *Distinguishing Input (DI).*

2. The DI, $\vec{X}_1$, is applied to the activated circuit (the oracle) and the output $\vec{Y}_1$ is recorded. Note that $\vec{K}_\alpha$, $\vec{Y}_\alpha$, and $\vec{K}_\beta$, $\vec{Y}_\beta$ are not recorded. Only the DI and its correct output are carried over to the following iterations.

3. In the i^{th} iteration, a new DI and a pair of keys, $\vec{K}_\alpha$ and $\vec{K}_\beta$, are found. The newly found $\vec{K}_\alpha$ and $\vec{K}_\beta$ should produce correct outputs for all the DIs found in previous iterations. To this end, we append a clause to Eq. (5.1):

$$\begin{aligned} &C(\vec{X}_i, \vec{K}_\alpha, \vec{Y}_\alpha) \wedge C(\vec{X}_i, \vec{K}_\beta, \vec{Y}_\beta) \wedge (\vec{Y}_\alpha \neq \vec{Y}_\beta) \\ &\bigwedge_{j=1}^{i-1} (C(\vec{X}_j, \vec{K}_\alpha, \vec{Y}_j) \wedge C(\vec{X}_j, \vec{K}_\beta, \vec{Y}_j)) \end{aligned} \tag{5.2}$$

   In this way, all the wrong keys that corrupt the output of previously found DIs (*i.e.,* the output is different from that of the activated chip) are pruned out from the search space.

4. SAT solves Eq. (5.2) repeatedly until no more DI can be found, *i.e.,* Eq. (5.2) is not satisfiable any more.

5. In this case, there is no more DI. The output of the SAT attack is a key $\vec{K}$ that produces the same output as the activated circuit to all the DIs, which can be expressed using the following CNF:

$$\bigwedge_{i=1}^{\lambda} C(\vec{X}_i, \vec{K}, \vec{Y}_i) \tag{5.3}$$

where λ is the total number of SAT iterations.

Note that there can be multiple correct keys: some keys can be different from but functionally equivalent to the actual key in the activated chip.

Theorem 5.1. SAT is guaranteed to find a correct key $\vec{K}_c$ to the locked circuit.

Proof. This can be proved by contradiction: suppose the key returned by the last step of SAT attack is a wrong key. This implies that there must exist a primary input $\vec{X}$ such that

$$C(\vec{X}, \vec{K}_c, \vec{Y}_c) \wedge C(\vec{X}, \vec{K}, \vec{Y}) \wedge (\vec{Y}_c \neq \vec{Y})$$

where $\vec{K}$ is the actual key, $\vec{Y}_c$ is the output with returned key $\vec{K}_c$ and $\vec{Y}$ is the correct output according to the actual key $\vec{K}$. $\vec{X}$ cannot be a previously found DI because otherwise $\vec{K}_c$ will not satisfy (5.3). We can see that $\vec{X}$ qualifies for a DI: just assign $\vec{K}_\alpha = \vec{K}_c$ and $\vec{K}_\beta = \vec{K}$. This means that (5.2) is still satisfiable and contradicts the criteria that no more DI can be found before the SAT attack goes to the final step.

Hence proved. □

5.2.3 Existing Logic Locking Schemes

Multiple logic locking schemes have been proposed to thwart the SAT attack [132, 119, 121, 133, 128]. There are two ways to mitigate the SAT attack: one is to increase the time for each SAT iteration and the other is to increase the number of SAT iterations. The former requires either AES blocks [132] or reconfigurable logic [49], which is impractical for most circuits. The other approach is to exponentially increase the number of SAT iterations. This approach is also not perfect because a locking scheme must be rather ineffective to improve security. This is the case for Anti-SAT [119, 121], SARLock [128], and and TTL [129]. All these techniques are vulnerable to the approximate SAT attacks (such as AppSAT [94] and Double-DIP [95]).

The state-of-the-art, *stripped functionality logic locking (SFLL)* [133], explores the trade-off between security and effectiveness. SFLL comprises of two parts: a functionality stripped circuit (FSC) and a restore unit (RU). The FSC is the original circuit with the functionality modified for a set of *protected input cubes*. This modification makes SFLL resistant to removal attack. If the RU is removed, the FSC's functionality of protected input cubes is different from the original circuit, thus making the attack unsuccessful. The RU stores the key, compares the circuit's input with the key, and outputs a *restore vector* which is XOR'ed with the FSC output. If the key is correct, the restore vector will fix the FSC's output and the circuit will have correct output. There are two variants of SFLL: SFLL-HD and SFLL-flex. SFLL-HD has been successfully attacked by a functional analysis

based attack [98, 99]. As the latter remains secure, provides higher flexibility in selecting protected cubes, and is more relevant to SAS, *we focus on SFLL-flex* in this paper. An SFLL-flex configuration can be described using the number of protected cubes, c, and the number of specified bits of each cube, k, denoted as SFLL-flex$^{c \times k}$. The authors of [133] derived the following characteristics of a circuit locked with SFLL-flex$^{c \times k}$: (1) the fraction of input minterms whose output will be corrupted by a wrong key (*i.e.,* the "error rate" of a wrong key) is $c \cdot 2^{-k}$; and (2) the probability that a SAT attack finds the correct key within q iterations is $q \cdot 2^{\lceil log_2 c \rceil - k}$. We illustrate this relationship in Fig. 5.4. As a higher SAT success probability indicates weaker security, SFLL inherently suffers from a trade-off between security and effectiveness.

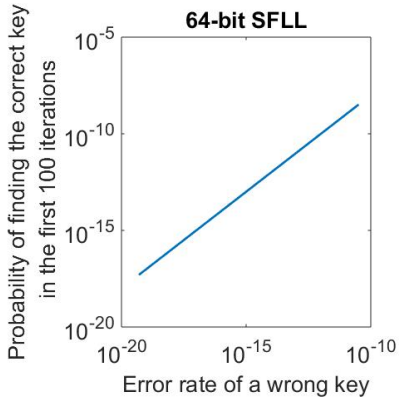


Figure 5.2: The positive correlation between the error rate of wrong keys and the probability that SAT finds the correct key in certain iterations

5.3 Insufficiency of SFLL for Real-World Applications

In this section, we investigate the application-level effectiveness of SFLL [133]. Specifically, we lock the multiplier within a 32-bit 80386 processor since it is the largest combinational component. The application-level impact is evaluated using both generic and neural network (NN)-based benchmarks. *We emphasize NN-based applications because they are inherently error-resilient and hence are more difficult to protect using logic locking.* Details of the benchmarks are listed in Table 5.1.

Table 5.1: Application benchmark details

Benchmark Type	**Quantity**	**Content**
Generic Applications	9	The PARSEC Benchmark Suite [8]
Neural Networks	5	MNIST [53], SVHN [68], CIFAR10 [51], ILSVRC-2012 [25], Oxford102 [70]

In order to evaluate the application-level impact of a logic locking scheme, we modify the GEM5 [10] simulator so that error is injected into the locked processor module according to the hardware error profile due to the wrong key. In this way, the circuit-level error induced by an incorrect (including approximate) key can be evaluated at the application level. This framework is illustrated in Fig. 5.3 which is similar to the strategy used in [19, 135].

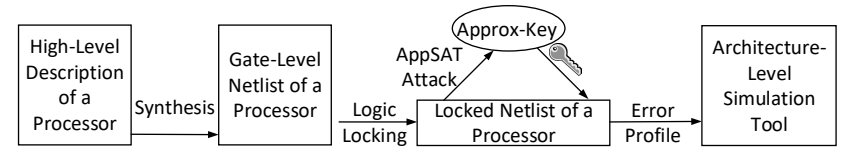


Figure 5.3: Our experimental framework

SFLL allows the designer to explore the trade-off between effectiveness and SAT resilience. We show that a "sweet spot" does not exist. In our experiments, we lock the multiplier with various SFLL configurations, each having a different level of SAT resilience, quantified by the average SAT iterations to unlock (as the X axis in Fig. 5.4). The effectiveness of each locking scheme is evaluated by running the PARSEC and NN benchmarks on the locked processors loaded with approximate keys. The percentage of PARSEC benchmark runs with incorrect outcome and the accuracy loss of NN models are used as the criteria to evaluate the effectiveness of each locking configuration. The trade-off is illustrated in Fig. 5.4.

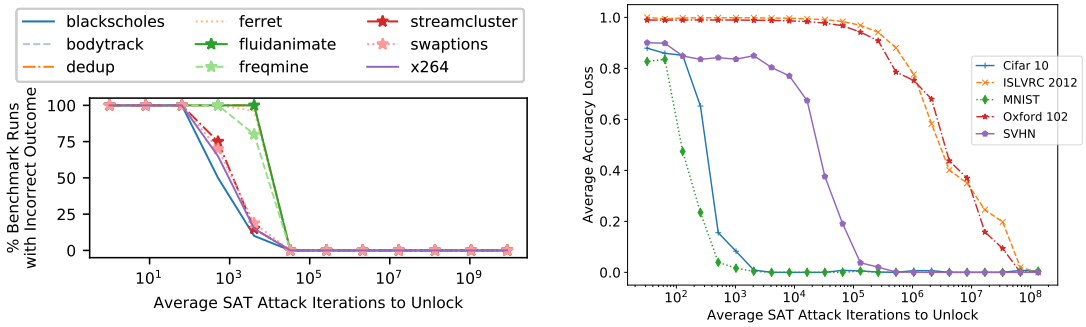


Figure 5.4: SAT resiliency vs. locking effectiveness trade-off. Left: PARSEC benchmarks. Right: NN benchmarks.

From Fig. 5.4, we observe that the wrong keys' impact decreases with the increase in SAT resiliency. In order to have a visible accuracy drop for the most error-resilient benchmarks, the SFLL locked processor cannot endure more than roughly 1000 SAT iterations. Such a locking scheme is extremely vulnerable since 1000 SAT iterations can be fulfilled within minutes. Therefore, a logic locking scheme that ensures high application-level impact without sacrificing SAT resiliency is needed.

5.4 Fundamental Trade-off for All Logic Locking Schemes

This section generalizes the trade-off of SFLL to all logic locking schemes. We start with definitions of concepts and then prove the relationship between SAT resilience and effectiveness.

Definition 5.1 (Corrupt)**.** We say that a key $\vec{K}$ **corrupts** a primary input minterm $\vec{X}$ if and only if the locked circuit produces a different output to $\vec{X}$ from the original circuit's output, *i.e.,* $F_L(\vec{X}, \vec{K}) \neq F(\vec{X})$.

Definition 5.2 (Error Rate)**.** The **error rate** $\epsilon_{\vec{K}}$ of a *wrong* key $\vec{K}$ is the portion of primary input minterms corrupted by the key $\vec{K}$.

Let $\mathcal{X}_{\vec{K}}$ be the set of input minterms corrupted by $\vec{K}$. Then,

$$\epsilon_{\vec{K}} = \frac{|\mathcal{X}_{\vec{K}}|}{2^n}$$

where n is the number of bits in the primary input. We use ϵ to denote the average error rate across all the keys. When the key is k bits long,

$$\epsilon = \frac{1}{|\mathcal{K}^W|} \sum_{\vec{K} \in \|\mathcal{K}^W|} \epsilon_{\vec{K}}$$

Definition 5.3 (Corruptibility)**.** The **corruptibility** $\gamma_{\vec{X}}$ of a primary input minterm $\vec{X}$ is the portion of *wrong* keys that corrupt this minterm.

Let $\mathcal{K}_{\vec{X}}$ be the set of wrong keys that corrupts the primary input minterm $\vec{X}$ and $\mathcal{K}^W$ be the set of wrong keys. Then,

$$\gamma_{\vec{X}} = \frac{|\mathcal{K}_{\vec{X}}|}{|\mathcal{K}^W|}$$

Let γ denotes the average corruptibility over all the input minterms, *i.e.,*

$$\gamma = \frac{1}{2^n} \sum_{\vec{X} \in \{0,1\}^n} \gamma_{\vec{X}}$$

Let us illustrate the above concepts with the following example. We consider a circuit with two primary input bits (x_0, x_1) and locked with a two-bit key (k_0, k_1), as shown in Fig. 5.5. Table 5.2 is the truth table for each possible primary input and key input combinations. If a key corrupts a primary input, the corresponding cell is marked with (✗).

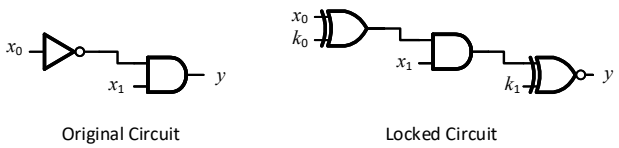


Figure 5.5: An example of logic locking, with the original circuit on the left and the locked circuit on the right.

Table 5.2: Truth table of the locked circuit in Fig. 5.5

	$\vec{K} = (0,0)$	$\vec{K} = (0,1)$	$\vec{K} = (1,1)$	$\vec{K} = (1,0)$	Correct y	$\gamma_{\vec{X}}$	γ
$\vec{X} = (0,0)$	1(✗)	0	0	1(✗)	0	$\frac{2}{3}$	$\frac{2}{3}$
$\vec{X} = (0,1)$	1	0(✗)	1	0(✗)	1	$\frac{2}{3}$	
$\vec{X} = (1,1)$	0	1(✗)	0	1(✗)	0	$\frac{2}{3}$	
$\vec{X} = (1,0)$	1(✗)	0	0	1(✗)	0	$\frac{2}{4}$	
$\epsilon_{\vec{K}}$	$\frac{1}{2}$	$\frac{1}{2}$	N/A	1			
ϵ	$\frac{2}{3}$						

In Table 5.2, we also calculate the error rate of each key, the corruptibility of each input minterm, and their averages. We can also observe that both the average error rate and average corruptibility equal $\frac{2}{3}$. It turns out that this equality is universal in logic locking:

Theorem 5.2. The average error rate of all wrong keys equals the average corruptibility of all input minterms, *i. e.* $\epsilon = \gamma$.

Proof. Recall that

$$\epsilon = \frac{1}{|\mathcal{K}^W|} \sum_{\vec{K} \in \mathcal{K}^W} \epsilon_{\vec{K}} = \frac{1}{|\mathcal{K}^W|} \sum_{\vec{K} \in \mathcal{K}^W} \frac{|\mathcal{X}_{\vec{K}}|}{2^n} = \frac{1}{2^n |\mathcal{K}^W|} \sum_{\vec{K} \in \mathcal{K}^W} |\mathcal{X}_{\vec{K}}|$$

and

$$\gamma = \frac{1}{2^n} \sum_{\vec{X} \in \{0,1\}^n} \gamma_{\vec{X}} = \frac{1}{2^n} \sum_{\vec{X} \in \{0,1\}^n} \frac{|\mathcal{K}_{\vec{X}}|}{|\mathcal{K}^W|} = \frac{1}{2^n |\mathcal{K}^W|} \sum_{\vec{X} \in \{0,1\}^n} |\mathcal{K}_{\vec{X}}|$$

Therefore, in order to prove $\epsilon = \gamma$, we only need to prove

$$\sum_{\vec{K} \in \mathcal{K}^W} |\mathcal{X}_{\vec{K}}| = \sum_{\vec{X} \in \{0,1\}^n} |\mathcal{K}_{\vec{X}}| \tag{5.4}$$

Let us consider the following bipartite graph $G = (\mathcal{X}, \mathcal{K}^W, \mathcal{E})$ where $\mathcal{X}$ is $\{0,1\}^n$ which is the set of all the possible input minterms, $\mathcal{K}^W$ is the set of wrong keys, and $\mathcal{E} = \{(\vec{X}, \vec{K}) | \vec{X} \in \mathcal{X}$ and $\vec{K} \in \mathcal{K}^W$, $\vec{K}$ corrupts $\vec{X}\}$. Both sides of Eq. 5.4 denote the total number of elements in $\mathcal{E}$ and hence must be equal. □

Let λ be the number of SAT iterations that a SAT attacker needs to find the correct key.

Theorem 5.3. The expected number of SAT iterations $E[\lambda]$ is lower bounded by $\frac{1}{\gamma}$.

Proof. In each SAT iteration, the average number of wrong keys pruned by the DI $\vec{X}$ is upper bounded by $\gamma |\mathcal{K}^{\mathcal{W}}|$ (because some of the wrong keys may have already pruned out by DIs of previous iterations). Therefore,

$$E[\lambda] \geq \frac{|\mathcal{K}^W|}{\gamma |\mathcal{K}^{\mathcal{W}}|} = \frac{1}{\gamma}$$

Hence proved. □

Theorems 5.2 and 5.3 explicitly point out that there exists an inverse relationship between ϵ and the lower bound of E[λ]. This quantifies the trade-off between them. This trade-off applies to any logic locking scheme. Note that different input minterms may inject a different amount of error at the application level. By assigning higher corruptibility to a few minterms with high application-level impact, we can achieve high effectiveness while maintaining high SAT resilience by keeping γ low and $E[\lambda]$ high. This is the main intuition behind SAS.

5.5 The Architecture and Properties of SAS

In Sec. 5.3 and 5.4, we demonstrated that two competing objectives exist for all logic locking schemes:

1. **Effectiveness:** Any incorrect key should have a high appli-cation-level error impact.

2. **SAT resilience:** The complexity of determining the correct key via SAT attacks should be very high.

In this section, we introduce *Strong Anti-SAT (SAS)* logic locking scheme which aims to achieve both objectives simultaneously. SAS guarantees an exponential expected SAT solving time while having a large impact on the accuracy of real-world applications. In SAS, instead of uniformly distributing the error across all possible inputs, we identify certain input patterns which potentially have a

higher impact on the overall application-level error. We call these inputs **critical minterms**. SAS is configured in such a way that any incorrect key corrupts at least 1 critical minterm. For the other minterms, the corruptibility is low.

5.5.1 The SAS Block

Let $\mathcal{M}$ be the set of critical minterms and $m = |\mathcal{M}|$ be the number of critical minterms. For the ease of implementation, we always choose m to be a power of 2. The basic locking infrastructure is the *SAS* block which is illustrated in Fig. 5.6. The key $\vec{K}$ of an n-bit SAS block consists of two n-bit sub-keys, $\vec{K}_1$ and $\vec{K}_2$. In order to describe the mechanism of the SAS locking scheme clearly, we use a reverse order and start our illustration from the output side.

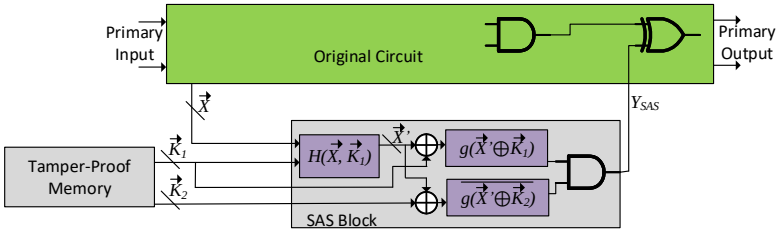


Figure 5.6: The Architecture of SAS Configuration 1 with the Details of the SAS Block

Y_{SAS} is the output of the SAS block. If $Y_{SAS} = 1$, a fault will be injected into the original circuit. g is a function with an on-set-size of 1, *i.e.,* only one input minterm will have output 1 and all others will have output 0. $\bar{g}$ has the opposite functionality of g. A function block $\vec{X}' = H(\vec{X}, \vec{K}_1)$ is inserted before g and $\bar{g}$ and it works as follows. If $\vec{X}$ is not a critical minterm, then $\vec{X}' = \vec{X}$. In this case, only one combination of $\vec{K}_1$ will make g output 1, therefore $\vec{X}$ has a low corruptibility.

If $\vec{X}$ is a critical minterm, then for a portion of $\vec{K}_1$, $\vec{X}$ is adjusted according to $\vec{K}_1$ to obtain $\vec{X}'$ such that $g(\vec{X}', \vec{K}_1) = 1$ and hence the corruptibility is increased. $\vec{X}' = H(\vec{X}, \vec{K}_1)$ further ensures that the wrong keys that corrupts each critical minterm are mutually exclusive and evenly partition the set of wrong keys. More specifically, as the partitioning is based on the $\vec{K}_1$ part of the key, we have the following. Let $\mathcal{K}^1_{\vec{X}} = \{\vec{K}_1 | \forall \vec{K}_2$ such that $(\vec{K}_1, \vec{K}_2) \in \mathcal{K}^W,\ (\vec{K}_1, \vec{K}_2) \in \mathcal{K}_{\vec{X}}\}$. Then we have

$$\forall \vec{X}_1, \vec{X}_2 \in \mathcal{M},\ |\mathcal{K}^1_{\vec{X}_1}| = |\mathcal{K}^1_{\vec{X}_2}|,\ \mathcal{K}^1_{\vec{X}_1} \wedge \mathcal{K}^1_{\vec{X}_2} = \emptyset, \text{ and } \bigcup_{\vec{X} \in \mathcal{M}} \mathcal{K}^1_{\vec{X}} = \mathbb{Z}_2^n \tag{5.5}$$

where n is the number of bits in $\vec{X}$, $\vec{K}_1$, and $\vec{K}_2$. This effect is illustrated in Table 5.3.

Table 5.3: Illustration of how m critical minterms partition the set of wrong keys

$\vec{K}_1$ of wrong keys		$\vec{k}_1$	$\cdots$	$\vec{k}_{\frac{2^n}{m}}$	$\vec{k}_{\frac{2^n}{m}+1}$	$\cdots$	$\vec{k}_{2\frac{2^n}{m}}$	$\cdots$	$\vec{k}_{2^n}$
	$\vec{X}_1$	•	•	•					
critical	$\vec{X}_2$				•	•	•		
minterms	$\cdots$						$\cdots$		
	$\vec{X}_m$							•	•
	$\vec{X}_{m+1}$	•							
non-									
	$\vec{X}_{m+2}$		•						
critical						$\ddots$			
minterms	$\cdots$								
	$\vec{X}_{2^n}$							•	

The 2 configurations of SAS will be introduced in the rest of this section.

5.5.2 Configuration 1: SAS with One SAS Block

This configuration is illustrated in Fig. 5.6. In this configuration, there is one SAS block. As the critical minterms evenly partition the set of wrong keys, the corruptibility of each critical minterm is $\gamma_c = \frac{1}{m}$. Below we derive the SAT resilience of this configuration assuming that the SAT solver chooses a DI uniformly at random in each iteration. This is a common assumption [129, 133, 92]. The SAT resilience is quantified using the expected number of SAT iterations $E[\lambda]$. To start with, we give 2 useful lemmas.

Lemma 5.4. Let $\mathcal{D}^i$ be the set of DIs that have been chosen in the first i iterations and $\vec{X}$ be a primary input minterm. If $\mathcal{K}_{\vec{X}} \subset \bigcup_{\vec{X'} \in \mathcal{D}^i} \mathcal{K}_{\vec{X'}}$, then $\vec{X}$ cannot be the DI of any SAT iteration beyond i.

Proof. Recall that Equation (5.2) gives the SAT formula for each SAT iteration:

$$C(\vec{X}_i, \vec{K}_\alpha, \vec{Y}_\alpha) \wedge C(\vec{X}_1, \vec{K}_\beta, \vec{Y}_\beta) \wedge (\vec{Y}_\alpha \neq \vec{Y}_\beta)$$
$$\bigwedge_{j=1}^{i-1} (C(\vec{X}_j, \vec{K}_\alpha, \vec{Y}_j) \wedge C(\vec{X}_j, \vec{K}_\beta, \vec{Y}_j))$$

To satisfy the first line, at one of $\vec{K}_\alpha$ and $\vec{K}_\beta$ must be a wrong key that corrupts $\vec{X}$. However, since any wrong key that corrupts $\vec{X}$ also corrupts at least 1 previously found DI, this wrong key cannot satisfy the second line. Hence such $\vec{X}$ cannot be the DI in future iterations. □

Lemma 5.5. For SAS Configuration 1, any critical minterm must exist in the set of DIs when SAT finishes: $\vec{X} \in \mathcal{D}^\lambda \ \forall \vec{X} \in \mathcal{M}$, where λ is the total number of SAT iterations and $\mathcal{D}^\lambda$ is the set of all DIs.

Proof. Recall that g has on-set size 1. Let $\vec{P}$ be the very input that makes $g(\vec{P}) = 1$. $\forall \vec{X} \in \mathcal{M}$, let $\vec{K_1} = \vec{X} \oplus \vec{P}$. Then, any $\vec{K} = (\vec{K_1}, \vec{K_2}) \in \mathcal{K}^W$ is a wrong key that only corrupts $\vec{X}$. Therefore, $\vec{X}$ has to be chosen as a DI to prune out this wrong key. □

Theorem 5.6. The expected number of SAT iterations of SAS Configuration 1 is

$$E[\lambda] = \frac{2^n + m}{2} \tag{5.6}$$

Proof. The total number of SAT iterations equals the total number of DIs. DIs consist of critical minterms and non-critical minterms. By Lemma 5.5, all the critical minterms must be in the set of DIs for SAT to terminate. Therefore, we only need to find the expected number of *non-critical minterms* that are chosen as DIs. As illustrated in Table 5.3, $\forall \vec{X'} \notin \mathcal{M}$, $\exists$ exactly one $\vec{X} \in \mathcal{M}$ such that $\mathcal{K}_{\vec{X'}} \subset \mathcal{K}_{\vec{X}}$. By Lemma 5.4, if this $\vec{X}$ is chosen as DI before $\vec{X'}$, then $\vec{X'}$ cannot be chosen in further iterations any more. In other words, $\vec{X'}$ will count towards the total number of iterations only when it is chosen before the critical minterm $\vec{X}$. By our assumption that the DI is chosen uniformly at random in each iteration, $\vec{X'}$ has a probability of $\frac{1}{2}$ to be chosen as DI before $\vec{X}$ is chosen. As this is true for any non-critical minterm, the expected number of SAT iterations is $E[\lambda] = \frac{1}{2}(2^n - m) + m = \frac{2^n+m}{2}$. □

5.5.3 Configuration 2: SAS with Multiple Blocks

In this configuration, we have l SAS blocks as illustrated in Fig. 5.7. Each SAS block takes an n-bit primary input $\vec{X}$, which is shared among all the SAS blocks, and a $2n$-bit key input. The output of each SAS block is XOR'ed with a wire in

the original circuit. Therefore, a fault is injected into the original circuit if any SAS block has output 1. Let $\mathcal{M}^j$ be the set of critical minterms for the j^{th} SAS block , $j = 1, 2, \ldots, l$. For ease of implementation, we choose l also to be a power of 2 and $l \leq m$. The relationship between $\mathcal{M}^j$ and the total set of critical minterms $\mathcal{M}$ is that $\mathcal{M}^1, \mathcal{M}^2, \ldots, \mathcal{M}^l$ *have the same cardinality, are mutually exclusive, and evenly partition* $\mathcal{M}$*, i.e.,*

$$|\mathcal{M}^1| = |\mathcal{M}^2| = \cdots = |\mathcal{M}^l|, \ \mathcal{M}^i \cap \mathcal{M}^j = \emptyset \ \forall i \neq j, \text{ and } \bigcup_{k=1}^{l} \mathcal{M}^k = \mathcal{M} \tag{5.7}$$

In this way, each SAS block has $\frac{m}{l}$ critical minterms. As each critical minterm receives high corruptibility from only one SAS block, the corruptibility of any critical minterm is $\gamma_c = \frac{l}{m}$.

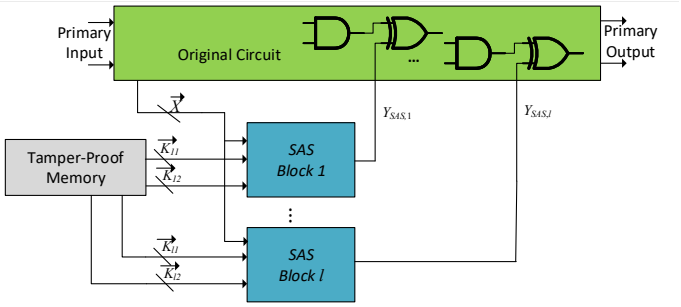


Figure 5.7: Configuration 2 with l SAS blocks

Lemma 5.7. For SAS Configuration 2, any critical minterm must exist in the set of DIs when SAT finishes: $\vec{X} \in \mathcal{D}^\lambda \ \forall \vec{X} \in \mathcal{M}$, where λ is the total number of SAT iterations and $\mathcal{D}^\lambda$ is the set of all DIs.

Proof. This is a natural extension to Lemma 5.5. Let $\vec{X}$ be a critical minterm and $\vec{X} \in \mathcal{M}^j$. Recall that g has on-set size 1. Let $\vec{P}$ be the very input that makes $g(\vec{P}) = 1$. $\forall \vec{X} \in \mathcal{M}^j$, let $\vec{k} = \vec{X} \oplus \vec{P}$. Then, let us consider the following wrong

key $\vec{K} = (\vec{K}^1, \vec{K}^2, \ldots, \vec{K}^l) \in \mathcal{K}^W$ which is composed as follows: $\vec{K}^j = (\vec{k}, \vec{K}_2^j) \in \mathcal{K}_j^W$ where $\mathcal{K}_j^W$ is the set of wrong keys for the j^{th} SAS block. For any $i = 1, 2, \ldots, l$ that $i \neq j$, $\vec{K}^i \in \mathcal{K}_i^C$ where $\mathcal{K}_i^C$ is the set of correct keys for the i^{th} SAS block. Such a key $\vec{K}$ is a wrong key that only corrupts $\vec{X}$. Therefore, $\vec{X}$ has to be chosen as a DI to prune out this wrong key. □

Below, we will analyze the SAT resilience of this configuration by deriving the expected number of SAT iterations.

Theorem 5.8. The expected number of SAT iterations of SAS Configuration 2 with l SAS blocks and m critical minterms is

$$E[\lambda] = \frac{l \cdot 2^n + m}{l + 1} \tag{5.8}$$

Proof. By Lemma 5.7, every critical minterm must count toward the total number of SAT iterations. Therefore, we only need to derive the expected number of non-critical minterms that are chosen as DIs.

For any non-critical minterm $\vec{X}' \notin \mathcal{M}$, in the i^{th} SAS block, there exists exactly one critical minterm $\vec{X}_i$ such that the set of wrong keys that corrupt $\vec{X}'$ in this SAS block, $\mathcal{K}_{i,\vec{X}'}$, is a subset of the set of wrong keys that corrupt $\vec{X}_i$, $\mathcal{K}_{i,\vec{X}_i}$. *i.e.,* $\mathcal{K}_{i,\vec{X}'} \subset \mathcal{K}_{i,\vec{X}_i}$. As the construction of the SAS block makes this true for any individual SAS block and the critical minterms for each SAS block are mutually exclusive, there are a total of l such critical minterms. When *all of these l critical minterms* are chosen as DI, they will cover the entire set of wrong keys that corrupt

$\vec{X}'$. Therefore, by Lemma 5.4, in order to include $\vec{X}'$ in the set of DIs, it must be selected before all l critical minterms are selected. This holds for any non-critical minterm.

By our assumption that the DIs are chosen uniformly at random in each SAT iteration, the probability that each non-critical minterm will be chosen as DI is $\frac{l}{l+1}$. Therefore, the expected number of SAT iterations is $E[\lambda] = \frac{l}{l+1}(2^n - m) + m = \frac{l \cdot 2^n + m}{l+1}$. □

The properties of both configurations of SAS are summarized in Table 5.4.

Table 5.4: Properties of the 2 configurations of SAS

Configuration	l	γ_c	$E[\lambda]$
1	1	$\frac{1}{m}$	$\frac{2^n+m}{2}$
2	$1 \leq l \leq m$	$\frac{l}{m}$	$\frac{l2^n+m}{l+1}$

5.6 Robust SAS: a Removal-Resilient SAS Variant

Although SAS achieves desirable SAT resilience and high corruptibility on critical minterms, it is still vulnerable to removal attack. In such an attack, the attacker can identify and remove each SAS block and replace their output wires with constant 0. In this way, the remaining part of the locked circuit will have correct functionality. In order to address this drawback, we introduce Robust SAS (RSAS), a variant of SAS that is resilient to removal attacks. In addition to adding an RSAS function block, RSAS modifies the functionality of the original circuit. Therefore, unlike SAS, one cannot obtain the correct functionality of the circuit

by identifying and removing the RSAS block. We will introduce the architecture of RSAS and show how any SAS configuration can be converted to a functionally equivalent RSAS configuration. Due to the equivalence in functionality, an RSAS configuration will have the same **SAT resilience** and **effectiveness** as its SAS counterpart.

5.6.1 RSAS Architecture and Relationship with SAS

A circuit locked by RSAS consists of an altered original circuit and one or more RSAS block(s). Fig. 5.8 illustrates the RSAS configuration with one RSAS block. Given the same set of critical minterms and the same number of locking function blocks, locking a circuit with RSAS and SAS will yield the same functionality. An RSAS-locked circuit can be obtained by converting a functionally equivalent SAS-locked circuit in the following way.

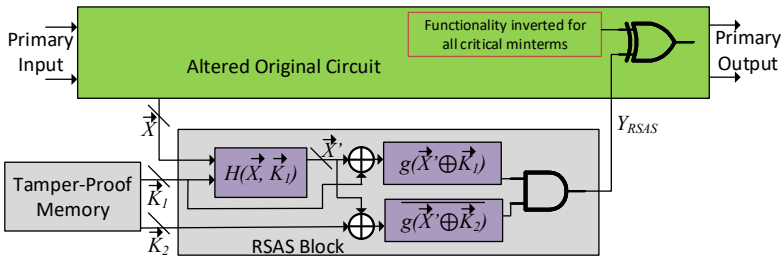


Figure 5.8: A circuit locked with one RSAS block, equivalent to SAS Configuration 1

5.6.1.1 Altering the original circuit

Recall that l is the number of SAS blocks in a SAS configuration. For the j^{th} SAS block, $j = 1, 2, \ldots, l$, the set of critical minterms it contains is denoted by $\mathcal{M}^j$ and $|\mathcal{M}^j| = \frac{m}{l}$, where m is the total number of critical minterms. In order to implement RSAS, we need to modify the original circuit's functionality. Notice that, for each SAS block, there is a wire in the original circuit that is XOR'ed with the SAS block's output. For the j^{th} SAS block, we locate this wire. For each critical minterm in $\mathcal{M}^j$, we invert the functionality of critical minterms at this wire. This needs to be done for each j in $j = 1, 2, \ldots, l$. This is illustrated in Fig. 5.9.

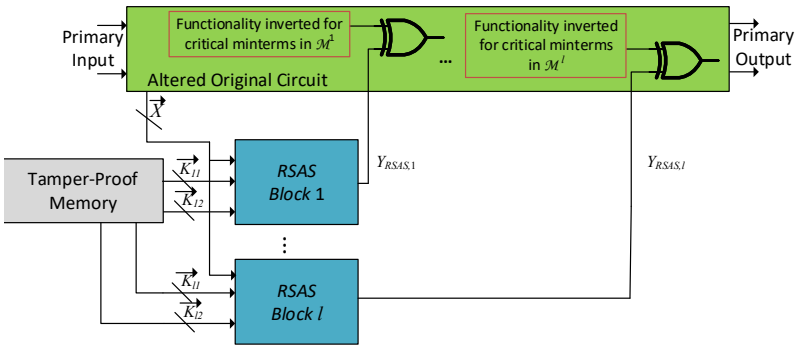


Figure 5.9: A circuit locked with multiple RSAS blocks, equivalent to SAS Configuration 2

5.6.1.2 Converting the SAS block into the RSAS block

The RSAS block is very similar to the SAS block and there is only one difference between them. For the j^{th} SAS block, $j = 1, 2, \ldots, l$, if the primary input is a critical minterm in $\mathcal{M}^j$, the output of RSAS block, $Y_{RSAS,j}$, is the inversion of the output of SAS block, $Y_{SAS,j}$. Recall that, for a SAS configuration with m critical minterms and l SAS blocks, each critical minterm's corruptibility is $\gamma_c = \frac{l}{m}$. Hence

for a portion of $\frac{l}{m}$ wrong keys, $Y_{SAS,j}$ is 1. This is achieved by the $\vec{X}' = H(\vec{X}, \vec{K}_1)$ function: if $\vec{X}$ is a critical minterm, then the $H(\vec{X}, \vec{K}_1)$ function makes sure that for γ_c portion of wrong keys, we will have $g(\vec{X}' \oplus \vec{K}_1) = 1$. For RSAS, since the functionality for critical minterms is inverted, the portion of wrong keys that makes $Y_{RSAS,j}$ be 1 is $1 - \gamma_c = \frac{m-l}{m}$. This means the functionality of $H(\vec{X}, \vec{K}_1)$ needs to be modified in the following way: if $\vec{X}$ is a critical minterm, then for $1 - \gamma_c$ portion of wrong keys, $g(\vec{X}' \oplus \vec{K}_1)$ will output 1. For non-critical input minterms, Y_{RSAS} behaves in the same as Y_{SAS}. This is illustrated in Table 5.5.

Table 5.5: Illustration of RSAS block's functionality. A '•' stands for $Y_{RSAS} = 1$.

$\vec{K}_1$ of wrong keys		$\vec{k}_1$	⋯	$\vec{k}_{\frac{2^n}{m}}$	$\vec{k}_{\frac{2^n}{m}+1}$	⋯	$\vec{k}_{2\frac{2^n}{m}}$	⋯	$\vec{k}_{2^n}$
	$\vec{X}_1$				•		•	•	•
critical	$\vec{X}_2$	•	•	•				•	•
minterms	⋯						⋯		
	$\vec{X}_m$	•	•	•	•		•		
	$\vec{X}_{m+1}$	•							
non-									
	$\vec{X}_{m+2}$		•						
critical									
minterms	⋯					⋱			
	$\vec{X}_{2^n}$								•

5.6.2 SAT Resilience and Effectiveness of RSAS

In Sec. 5.6.1, we introduced how to convert a SAS-locked circuit into an equivalent RSAS-locked circuit. These steps essentially invert the functionality of each critical minterm at two places: the first at the wire in the original circuit where RSAS is integrated, and the other at the RSAS block's output. Since these two wires are XOR'ed, the two inversions will cancel out which makes the RSAS-locked

circuit functionally equivalent to the SAS-locked circuit. Due to the equivalence in functionality, the derivations of SAS's *SAT resilience* and *effectiveness* will also hold for RSAS. Therefore, Table 5.4 is also the summary of these properties of RSAS.

5.7 Choosing Critical Minterms

The critical minterms for injecting large errors should be selected judiciously. A careful analysis of the workload would help identify these typical minterms. Generally these minterms would be very few as compared to the overall input space of the functional modules. Here we describe how to select the critical minterms. As mentioned in Sec. 5.3, we use PARSEC and NN models as application benchmarks. For the PARSEC (generic) benchmarks, we arbitrarily choose critical minterms from the input minterms that exist in all the application benchmarks. We take a similar approach for NN benchmarks. A significant part of an NN-based application is the weights of the NN model and it turns out that the weight values of most NN models follow a similar distribution. For example, Figure 5.10 shows the distribution of weights of the LeNet (MNIST dataset) and CaffeNet (ISLVRC-2012 dataset) models. These two are the smallest benchmark and the largest benchmark, respectively. The weight distributions are similar across NN benchmarks and many other NN models. This kind of similarity can be also found among generic applications.

We select a subset of weight values to be critical minterms based on their application-lavel impact. The selected critical minterms should cause significant application-level error. Fig. 5.10 also shows the accuracy loss of the NN model in the

following experiment: for each input minterm, we measure the accuracy loss of the NN model when every computation involving this very minterm is corrupted while no other minterm is corrupted. As the input minterm distributions are similar

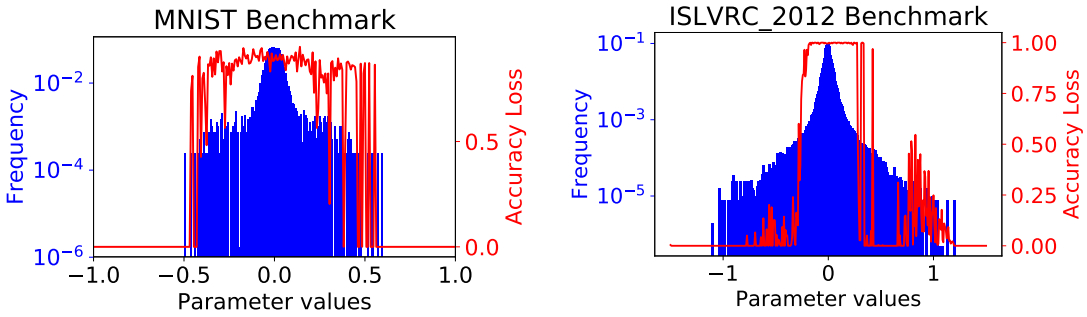


Figure 5.10: Weight distribution (blue histogram, left Y axis) and application-level accuracy loss (red line, right Y axis) of LeNet and CaffeNet when the corresponding input is locked

among the same type of applications, the flexibility of SAS allows the designer to choose a configuration and a combination of critical minterms that work well in securing the intended applications without compromising the SAT resiliency.

5.8 Experiments & Comparison with SFLL

This section shows the experimental results of SAS and RSAS as well as the comparison with SFLL. Recall that, as illustrated in Fig. 5.3, we obtain the gate-level netlists of the multiplier within a 32-bit 80386 processor by synthesizing the high-level description using Cadence RTL Compiler. Then we lock the netlist using various SAS and RSAS configurations and SFLL-flex with the same set of critical minterms. Note that the critical minterms are selected according to the method described in Sec. 5.7. The architecture-level simulation is conducted by a modified GEM5 [10] simulator where error is injected into the locked processor

module according to the hardware error profile due to the wrong key. We conduct the following experiment to verify the SAT resilience and effectiveness of SAS and RSAS and compare them with SFLL.

5.8.1 SAT Resilience

We first verify whether the SAT resilience of SAS/RSAS (*i.e.,* the actual number of SAT iterations) matches what we have derived in Sec. 5.5. The SAT resilience of SAS/RSAS and SFLL is also compared. We lock the multiplier in a 32-bit 80386 processor with SAS and RSAS as well as SFLL. Fig. 5.11 shows the actual and expected number of SAT iterations of multipliers locked with SAS and RSAS. These numbers are compared to the actual number of iterations of SFLL. In these locking configurations, we use 14 bits of primary input for locking purposes ($n = 14$) and experiment with each feasible configuration with up to 4 critical minterms. We can observe that SAS and RSAS have similar numbers of actual SAT iterations and they are both close to the expected value. When there is more than one critical minterms, SAS and RSAS exhibit higher SAT resilience than SFLL. This is because the corruptibility of each critical minterm in SFLL is almost 1 no matter how many critical minterms there are. This compromises its SAT resilience.

Fig. 5.12 compares the actual SAT iterations of SAS and SFLL. In Fig. 5.12a, it can be observed that SAS's SAT complexity is higher than that of SFLL by a roughly constant factor when m is fixed at 4. Note that the same set of four critical

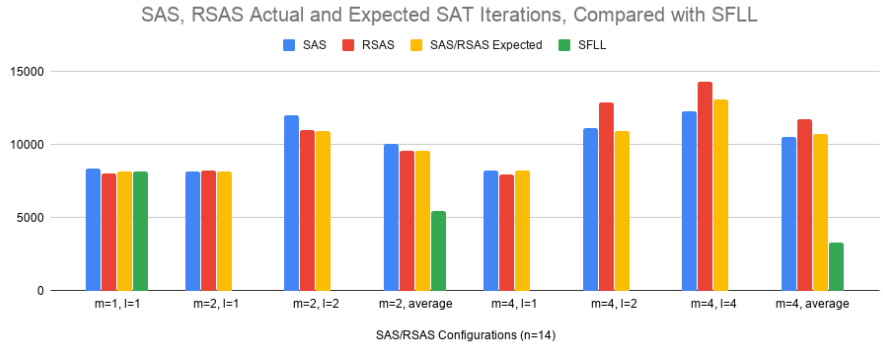
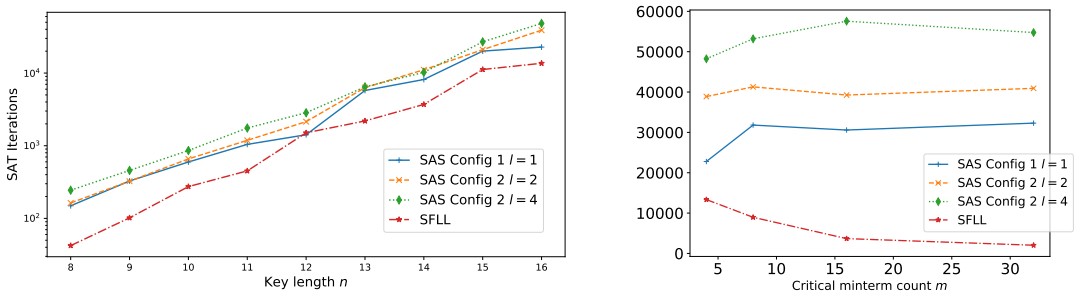


Figure 5.11: Actual and expected number of SAT iterations of SAS and RSAS, compared with SFLL.

minterms are used for each locking scheme. Among various SAS configurations, a larger l comes with higher SAT resilience as expected. In Fig. 5.12b, we vary the critical minterm count (m) from 4 to 32 and demonstrate its impact on the SAT resilience of SAS and SFLL. While SAS configurations become stronger with more critical minterms, SFLL becomes weaker. Therefore, SAS is more SAT resilient and gives designers more flexibility when more critical minterms are needed.



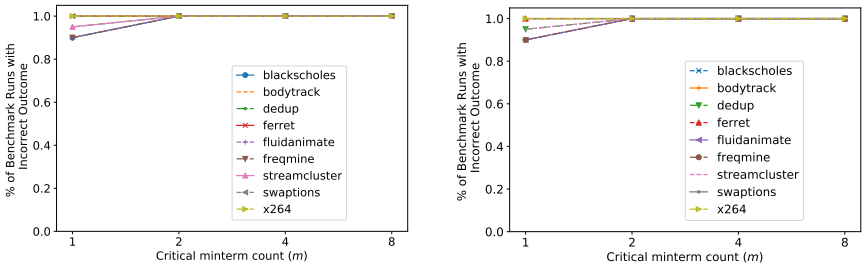
(a) Varying key length (n), fixing # critical minterms $m = 4$

(b) Varying # critical minterms (m), fixing key length $n = 16$

Figure 5.12: The observed SAT iterations of SAS and SFLL by varying key length and critical minterm count.

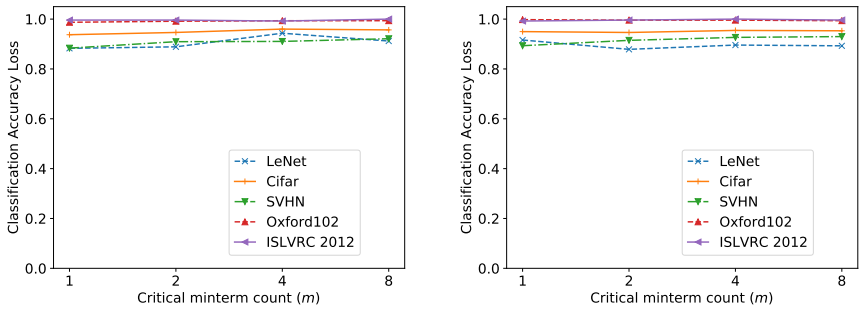
5.8.2 Effectiveness

We evaluate the effectiveness of SAS/RSAS and SFLL at the application level using PARSEC [8] and ML benchmarks as listed in Table 5.1. Due to the functional equivalence of SAS and RSAS, they will have the same architecture-level effects and we use the same functional model to perform architecture-level simulation of SAS and RSAS. In our experiments, various numbers of critical minterms are locked. The same set of critical minterms are used for SAS/RSAS and SFLL in each experiment. The critical minterms are chosen according to the methods described in Sec. 5.7. For SAS, we choose $l = 1$ when $m = 1$ and $l = 2$ when $m \geq 2$. Figs. 5.13 and 5.14 show that both SAS/RSAS and SFLL are effective at the application level for both generic and ML-based applications. SAS/RSAS achieves high application-level effectiveness and exponential SAT resiliency at the same time. Considering that SAS/RSAS's SAT resilience is not compromised with the increase in m as opposed to SFLL (as shown in Figs. 5.11 and 5.12b), SAS/RSAS is a significant improvement over SFLL.



(a) SAS/RSAS on PARSEC (b) SFLL on PARSEC

Figure 5.13: The application-level effectiveness of SAS/RSAS and SFLL on PARSEC and ML benchmarks



(a) SAS/RSAS on ML (b) SFLL on ML

Figure 5.14: The application-level effectiveness of SAS/RSAS and SFLL on PARSEC and ML benchmarks

5.8.3 Area, Power, and Delay Overhead of SAS, RSAS, and SFLL

Now that we have demonstrated the SAT resilience of SAS and RSAS and their application-level effectiveness, we evaluate their area, power, and delay overhead. The overhead is also compared with SFLL. In our evaluation, we use 32 bits from the primary input for locking ($n = 32$) and lock up to 4 critical minterms ($m = 1, 2, 4$). We synthesize the original and locked circuits using Cadence RTL Compiler using SAED 90nm process. Figs. 5.15, 5.16, and 5.17 show the area, power, and delay overhead values, respectively. Compared with SFLL, on average, SAS and RSAS have 2.22% and 1.49% more area overhead, 0.43% more and 0.04% less power overhead, 0.93% and 0.71% more delay overhead, respectively. These are not significant increases in overhead and should be worth the gain in SAT resilience.

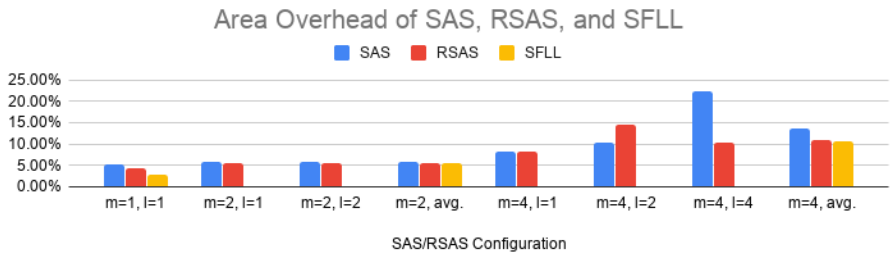


Figure 5.15: Area overhead of SAS and RSAS compared with SFLL

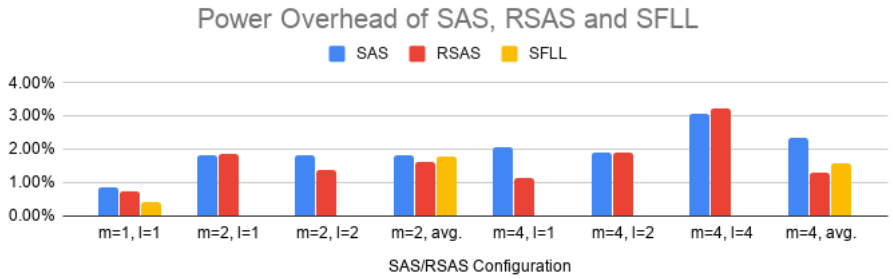


Figure 5.16: Power overhead of SAS and RSAS compared with SFLL

5.9 Summary

In this work, we investigate logic locking techniques to secure both generic and error-resilient workloads running on locked processors. We motivate our work by demonstrating the insufficiency of the state-of-the-art logic locking scheme in securing such applications. We point out that this is due to the fundamental trade-off between *SAT resilience* (SAT attack complexity) and *effectiveness* (error rate

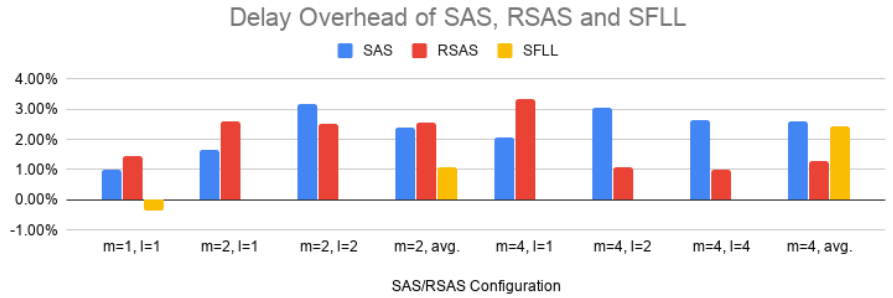


Figure 5.17: Delay overhead of SAS and RSAS compared with SFLL

of wrong keys) of logic locking. We formally prove this trade-off. In order to address this dilemma, we propose Strong Anti-SAT (SAS) where a set of critical minterms are assigned higher corruptibility in order to ensure high application-level impact. Based on SAS, we also propose Robust SAS (RSAS) to thwart removal attacks on logic locking. RSAS is functionally equivalent to SAS and has the same SAT resilience and effectiveness. Experimental results show that SAS and RSAS secure processors against SAT attack by ensuring exponential SAT attack complexity and high application-level impact simultaneously given any wrong key. We also evaluate the area, power, and delay overhead of SAS and RSAS and compare it with SFLL. It is shown that SAS and RSAS have modest increase in overhead. In summary, RSAS exhibits a higher SAT resilience than SFLL when multiple critical minterms are secured, while also maintaining equivalent effectiveness and removal attack resilience. Therefore, RSAS constitutes a significant improvment over SFLL-based locking.

Chapter 6: Cache Side-Channel-based Reverse Engineering of Neural Networks

In recent years, deep neural networks (DNN) have become an important type of intellectual property due to their high performance on various classification tasks. As a result, DNN stealing attacks have emerged. Many attack surfaces have been exploited, among which cache timing side-channel attacks are hugely problematic because they do not need physical probing or direct interaction with the victim to estimate the DNN model. However, existing cache-side-channel-based DNN reverse engineering attacks rely on analyzing the binary code of the DNN library that must be shared between the attacker and the victim in the main memory. In reality, the DNN library code is often inaccessible because 1) the code is proprietary, or 2) memory sharing has been disabled by the operating system. In our work, we propose GANRED, an attack approach based on the generative adversarial nets (GAN) framework which utilizes cache timing side-channel information to accurately recover the structure of DNNs without memory sharing or code access. The benefit of GANRED is four-fold. 1) There is no need for DNN library code analysis. 2) No shared main memory segment between the victim and the attacker is needed. 3) Our attack locates the exact structure of the victim model, unlike existing attacks

which only narrow down the structure search space. 4) Our attack efficiently scales to deeper DNNs, exhibiting only linear growth in the number of layers in the victim DNN.

6.1 Introduction

Deep neural networks (DNN) have demonstrated exceptional performance in a multitude of applications such as image classification and speech recognition, making them a valuable and important form of intellectual property. In order to protect DNN models, owners often host them on remote servers, restricting users only to querying the model. Hence, users do not have the details of the model (*i.e.,* architecture or weights). However, DNN model theft is still possible in this scenario. For example, an adversary can exploit side-channel information in order to reverse engineer the DNN [123, 42, 43, 5, 114, 6, 27, 107]. Under the remote host setting, cache side-channel shows the most promise. Because the last level cache (LLC) is shared among each processor core in most modern computer architectures, the attacker can infer the victim's cache usage even without interacting with the victim directly.

Existing cache-based attack focus on reverse engineering the structure of DNNs. As shown by the variety of prior research aimed at reverse engineering the structure of DNNs, such as [123, 42, 43], even if these attacks do not decipher the weight information, knowing the structure of DNNs enables enables weight extraction attacks [107] and membership inference attacks [96, 62] and improves black-box adversarial example attacks [75]. Therefore, unlocking the underlying DNN struc-

ture is a formidable attack. Hong *et al.* proposed DeepRecon which monitored calls to selected TensorFlow library functions and observed the layer sequence of DNNs [42]. Yan *et al.* proposed Cache Telepathy which substantially narrowed down the dimension parameter search space of DNNs by obtaining the number of generalized matrix multiplication (GEMM) operations via cache timing side-channels [123]. They were able to identify 16 possible structures for the VGG-16 DNN [97]. Both of these attacks required that the attacker and the victim share the DNN's library code (*e.g.* TensorFlow or GEMM library) in main memory (*i.e.,* the library code in the main memory is mapped to the virtual address spaces of both the attacker and the victim). However, memory sharing can be disabled by the server's operating system and the library code may be proprietary and inaccessible, thereby rendering these attacks infeasible. Moreover, neither of these attacks could give the DNN dimension parameters precisely. Instead, they return only the layer sequence or a set of possible parameter combinations. Since slight differences in DNN structure may result in a significant difference in accuracy under the same training effort [43], obtaining the exact structure of the victim DNN is crucial. Other existing DNN reverse engineering attacks require querying the victim DNN model [43, 107] or any physical side-channel probing [43, 5, 114, 6, 27]. These are not required by GANRED either. Therefore, GANRED can be carried out in a more realistic scenario.

6.1.1 GANRED Attack Overview

In our work, we develop GANRED, a novel generative adversarial nets (GAN)-based [32] Rereverse Engineering attack on DNNs which is capable of both fully recovering the dimension parameters of a DNN and does not require shared library access. For this attack, the victim DNN's cache side-channel information is measured by the attacker and acts as the **ground truth** of the GAN using a cache side-channel attack technique called *Prime+Probe* [73, 38, 79]. This technique does not require any shared main memory segment between the attacker and the victim.

The attacker builds another DNN and updates the structure of this DNN repeatedly to make its structure equivalent to the victim DNN. In the rest of the paper, **we refer to the victim's DNN as VDNN and the attacker's DNN as ADNN.** In order to achieve his/her objective, the attacker needs to find the correct structure of each layer before moving on to the next layer. This is done as follows. The attacker initializes the ADNN as a one-layer network. For each feasible structure of this layer, the **generator** measures the cache side-channel of the ADNN in the same way as the VDNN is measured (*i.e.,* using *Prime+Probe*).

The **discriminator** compares the cache side-channel information of the VDNN and the ADNN and indicates for how many clock cycles the two DNNs produce identical side-channel information. If the ADNN has the correct structure, *i.e.,* the same structure as the first layer of the VDNN, then the ADNN's cache side-channel

information should be identical to the VDNN's first layer, and the discriminator will indicate that the side-channel information of the two DNNs is identical throughout the period that the ADNN runs.

The **validator** compares the discriminator's output with a theoretical running time of the ADNN estimated using a linear regression analysis. This effectively rules out the ADNN structures that cause its cache side-channel to diverge from the the VDNN's in the middle of the ADNN's execution. The attacker chooses the structure that produces accurate cache side-channel data for the longest time as the first ADNN layer.

In order to search for the structure of VDNN's second layer, similar operations are done. Each feasible structure of the second layer is appended to the (now known) first layer to compose a two-layer ADNN whose cache side-channel is measured by the generator. The discriminator compares the cache side-channel information of the two DNNs and the validator determines whether the *added* matching time agrees with the theoretical runinng time of the second layer. The structure of each successive layer is recovered in this way until an ADNN is recovered that produces identical cache side-channel for the entirety of each DNN's execution. The attack is considered successful if ADNN's final structure is the same as the VDNN's structure.

6.1.2 Contributions

The contributions of this work are as follows:

- We propose the GANRED framework where DNNs are characterized by their accesses to a cache set over time. Our technique does not need any shared main memory segment between the victim and the attacker or analyze the DNN library codes on the server. Both resources were required by existing cache side-channel based DNN structure reverse engineering attacks [123, 42]. GANRED does not require querying the victim DNN model or any physical probing either, as required in other existing DNN reverse engineering attacks [43, 5, 114, 6, 27, 107]. Hence, GANRED can be carried out in a more realistic scenario where these privileges are not granted.

- We prove the following theoretical basis for GANRED. If the ADNN has the same structure as the first l layers of the VDNN, then both DNNs should produce identical cache side-channel information throughout these l layers.

- We show that our attack produces the exact structure of each VDNN model. This has not been achieved by existing DNN reverse engineering attacks based on cache side-channels [123, 42].

- We prove that the runtime of GANRED scales linearly in the number of DNN layers. This makes our attack scalable to much deeper DNNs.

6.2 Background

6.2.1 Dimension Parameters of Deep Neural Networks

Deep neural networks (DNN) are a supervised classification technique that consists of a sizable number of cascaded layers. Let i and l denote the layer number and the total number of layers, respectively, hence $i \in [l]$. In each layer, the **input feature map (IFM)** (a.k.a. the set of input neurons) is transferred into the **output feature map (OFM)** (a.k.a. the set of output neurons) via an operation which involves a set of *filters*. The IFM and OFM sizes (*i.e.,* number of contained neurons) of layer i are denoted by z_i^{in} and z_i^{out}, respectively. Note that the OFM of the previous layer is the IFM of the next layer.

Most DNNs consist of two types of layers: **fully connected (FC)** layers and **convolutional (Conv)** layers. The IFM and OFM of FC layers are (1-dimensional) vectors whose lengths are z_i^{in} and z_i^{out}, respectively. The weights consist of a matrix of dimension $z_i^{in} \times z_i^{out}$. The structure of a Conv layer is illustrated in Fig. 6.1. The IFM and OFM of a Conv layer are both 3-dimensional arrays. The width and height of a feature map are usually equal. There are a set of filters in the Conv layer and each of them is also a 3-dimensional array. Each filter is convolved with the IFM to obtain a *channel* in the OFM. A Conv layer can be characterized by a set of dimension parameters as listed in Table 6.1. *Note that a Conv layer can be followed by an optional pooling layer and, if so, we consider pooling as a part of the Conv layer.* We define the parameter P_i as the indicator of whether there is a pooling layer after layer i.

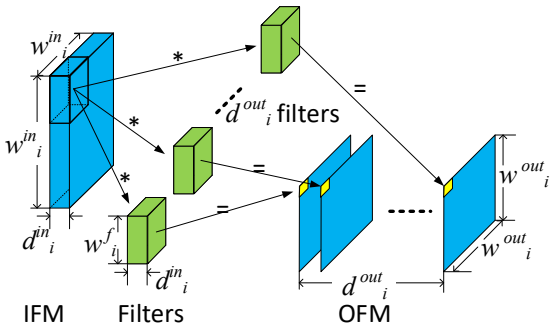


Figure 6.1: Illustration of a convolutional layer. "*" indicates inner product, each computing an output neuron.

Type of layer	Parameter	Definition (subscript $_i$ indicates layer i)
Conv layer	w_i^{in}, w_i^{out}	IFM/OFM width
	d_i^{in}, d_i^{out}	IFM/OFM depth (number of channels)
	w_i^f, δ_i	convolution filter width and stride
	P_i	indicator of pooling layer existence
FC layer	z_i^{in}, z_i^{out}, z_i^f	IFM/OFM/filter size

Table 6.1: List of dimension parameters of a layer

6.2.2 Cache Architecture Fundamentals

Cache is a type of on-chip storage for processors which temporarily stores a subset of the main memory's content in order to reduce memory access latency and improve the processor's efficiency. The basic component of cache is a cache block (also called a cache line). Most modern processors have a set associative cache where the cache is divided into multiple *ways*, each having the same number of blocks. For example, Fig. 6.2 illustrates a two-way set associative cache. The cache blocks in the same position of each way constitute a *set*. The organization of address bits is given in Fig. 6.2. When a block is to be moved into the cache, the cache controller will extract the set index bits from the block's address and put the block into an available slot in the according cache set. If no slot is available in the set,

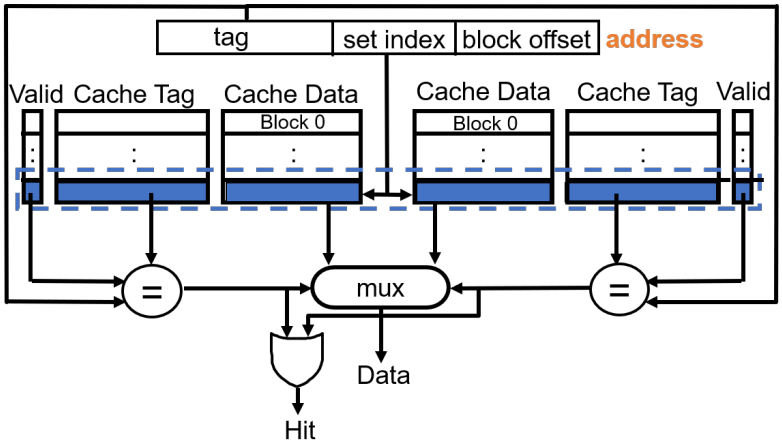


Figure 6.2: A two-way set associative cache example

the controller will select a block within the set to be replaced with the new block according to the *replacement policy*. The most commonly used replacement policy is to replace the *least recently used (LRU)* block.

In modern multi-core processors, the cache has a hierarchy of multiple levels. We specifically focus on the *last level cache (LLC)* since it is shared among all the processor cores. Hence the LLC is used by every program running on the processor, no matter which core the program runs on.

6.2.3 Cache Timing Side-Channel Attacks

In a cache timing side-channel attack, the attacker and the victim are two processes running on the same processor. The attacker seeks information leakage about the victim process by exploiting a fundamental property of cache: a cache hit is fast and a cache miss is slow. Although a lot of attack techniques have been proposed, most of them can be described as a three-step process [26]:

1. The attacker initializes the state of a cache location.

2. The victim program executes, which may modify the state of the attacker-initialized cache location.

3. The attacker accesses the same cache location again and observes the access latency. By doing so, he/she can infer whether the victim has accessed the initialized cache location.

These attacks can be categorized by whether data is shared between the attacker and victim processes.

6.2.3.1 Attacks based on Data Sharing

Flush+Reload is the major type of attack in this category [127, 126, 37]. These attacks require shared data between the attacker and the victim which can be achieved when the operating system allows multiple processes to map their individual virtual addresses to the same physical address for commonly required resources (*e.g.* library files) [39]. This sharing enables the attacker to obtain the victim's library usage information via cache timing side-channel. The 3 steps of Flush+Reload are as follows:

1. *Flush:* The attacker targets an address within shared memory and calls `clflush` (an X86 instruction) to flush the cache line (*i.e.,* block) that contains this address if such a cache line exists. Otherwise `clflush` has no effect.

2. The victim process runs. The content of the targeted address will be brought back to the same cache location if accessed by the victim.

3. *Reload:* The attacker accesses the targeted address and infers whether a cache hit or a cache miss occurs based on the access latency. If the victim has accessed the flushed address, a cache hit will occur. Otherwise, a cache miss occurs.

6.2.3.2 Attacks without Data Sharing

Many cache timing side-channel attacks work without shared main memory. Because there is a many-to-one mapping from main memory to cache, the attacker's and the victim's physical addresses can map to the same last level cache (LLC) location. In this way, the attacker can still detect the changes in cache state made by the victim. Examples of such attacks are Prime+Probe [73, 38, 79], Evict+Time [73], and cache collision-based attacks [12]. Among these attacks, Prime+Probe is the best known and most widely used. Its mechanism is as follows:

1. *Prime*: the attacker fills the cache sets of interest with his/her own data.

2. The victim program runs which may or may not overwrite the primed cache sets.

3. *Probe*: The attacker accesses the primed cache sets and observes timing. A cache miss indicates that the victim has accessed that cache set.

Note that *probing automatically primes the cache again* which enables the attacker to monitor the cache for a long period.

6.2.4 Existing DNN Reverse Engineering Attacks and Defenses

Reverse engineering of neural models has become a real threat which has attracted several researchers' attention. Among this body of work, a variety of distinct attack approaches have been explored to reverse engineering neural models. Yan *et al.* and Hong *et al.* independently proposed neural network reverse engineering techniques based on cache side-channels [123, 42]. In their attacks, the attacker needs to analyze the neural model's library code, extract the control flow, and select code lines to measure cache timing. These lines represent certain functions that are called when the neural network is running. By monitoring these function calls, information about the victim DNN's structure can be extracted. DeepRecon by Hong *et al.* inserted probes into the TensorFlow library code and was able to tell the number of layers and the type of each layer in DNNs [42]. In Yan *et al.* 's Cache Telepathy attack, the generalized matrix multiply (GEMM) backend libraries were monitored and they were able to reduce the DNN structure search space significantly [123]. For example, only 16 possible structures of the VGG-16 DNN [97] are still feasible after their attack. DeepRecon uses Flush+Reload which requires the attacker and victim to share the main memory segment that contains TensorFlow library files. Cache Telepathy can be done using either Flush+Reload

or Prime+Probe. However, even the Prime+Probe version requires a shared main memory segment for the GEMM library files. This might not be a realistic attack scenario: the operating system can disable memory sharing between different users or processes, and the attacker may not have access to the server's DNN library code.

In addition to cache side-channel, power/electromagnetic side-channels [5, 114, 6] and timing side-channel [27] have also been exploited to reverse engineer neural models. However, these attacks require physically probing the hardware and are feasible only when such probing is possible. Tramèr *et al.* proposed a technique to steal neural models from remote servers through prediction APIs provided by the server [107]. A countermeasure proposed by Juuti *et al.* detects such model extraction attacks with a statistical technique [47]. Hua *et al.* found out that the neural network can be reverse engineered from its memory access pattern [43]. Provably secure memory access protocol [56] and secure neural accelerator designs [113] can defend against this attack.

Many DNN protection techniques have been developed. Homomorphic encryption (HE) [13, 15, 30, 115] and secure multi-party computation (MPC) [86, 66, 72] have been employed to ensure the privacy of both the neural model and the input data. However, even the state-of-the-art HE and MPC algorithms are still too complex to use in practice. Additionally, several works have proposed the use of secure enclaves for DNN operations (such as Intel SGX) [106, 36]. However, DNNs running in these enclaves are vulnerable to cache side-channel attacks as well [112, 35]. In summary, there has not been an effective countermeasure against cache-based DNN reverse engineering.

6.3 Attack Model

In our attack model, the VDNN runs on a server alongside the attacker which is another process on the same server. **The attacker's goal is to reverse engineer the structure of the victim DNN model.** We consider a realistic threat model under which the attacker process does not have the privileges assumed by many prior works such as code access, memory sharing, or physical probing [123, 42, 43, 5, 114, 6, 27, 107]. The resources available to the attacker are as follows:

- **Shared last-level cache (LLC)** with the victim. This is the case for most state-of-the-art computer architectures. *Shared LLC enables the attacker to obtain cache side-channel information of another process, e.g. the victim DNN, using Prime+Probe.*

- **High-level APIs** of the machine learning framework. This is available to the attacker when he/she acts as a regular user on the server. *This enables the attacker to construct a DNN model.*

With these two resources, *the attacker can obtain the cache side-channel information of both VDNN and that of ADNN using Prime+Probe.* Note that these are only a small subset of the attackers' resources in prior attack models [123, 42, 43, 5, 114, 6, 27, 107]. We assume that **the attacker does not have any of the following privileges**:

- **Querying VDNN**. These are normally unavailable to the attacker unless the victim grants permission. However, [43, 107] both require this access.

- **The library code** of the machine learning framework (*e.g.* TensorFlow), since the library may contain intellectual property of the server and users only need high level APIs to build their neural model. Lacking code access makes the attacks in [123, 42] infeasible since they need to find specific functions in the code to insert probe.

- **Shared main memory** that stores the machine learning library code. This is also needed by [123, 42] in order to map the shared library to the attacker's own virtual memory space and implement Flush+Reload.

- **Physical access to the processor**. This access is not available when the server is not controlled by the attacker. This makes any side-channel other than cache side-channel impossible to measure and renders the attacks in [5, 114, 6, 27, 43] infeasible.

In summary, we assume a scenario where the resources available to an attacker are very constrained. None of the existing reverse engineering attacks [123, 42, 43, 5, 114, 6, 27, 107] are possible in this setting. However, in this work, we show that even under such constraints, there is still substantial information leakage about the DNN model through the cache timing side-channel. GANRED reverse engineers the DNN structure by utilizing this information. A server's security measures, such as restricting queries to the victim, and eliminating memory sharing or library code access, will disable existing attacks but not hide the side-channel information that is sufficient for GANRED.

6.4 Attack Methodology

In this section, we introduce the GANRED framework details. In Sec. 6.4.1, we present how to characterize a DNN using Prime+Probe results. Sec 6.4.2 describes how each component of the GANRED framework works. Sec. 6.4.3 introduces the overall algorithm of GANRED. Sec. 6.4.4 details how the validator utilizes a linear regression analysis to estimate the running time of a layer based on its structure. Sec. 6.4.5 proves the premise of GANRED that, if the ADNN has l layers and its structure is the same as the first l layers of the VDNN, then the ADNN should produce identical cache side-channel information as the VDNN's until the ADNN's execution ends.

6.4.1 Obtaining DNN's Cache Side-Channel Trace

Before we talk about the details of GANRED, we describe what side-channel information about a DNN can be obtained from Prime+Probe. **The discussion of this subsection holds for both the VDNN and the ADNN**.

During Prime+Probe, the attacker selects an arbitrary LLC set and focuses only on this set. This is because we find that each DNN that we study leaves almost the same access pattern on each LLC set. In the rest of this paper, unless otherwise noted, our discussion is focused on this very LLC set. Suppose that the DNN makes s memory accesses to this LLC set during its entire execution. Let us use t_j to denote the time at which the j-th access occurs, where j is the index of the access ($1 \leq j \leq s$). *Note that time is measured using CPU clock*

cycles throughout this paper. Also, note that t_j's are not deterministic. This is because the DNN's execution is scheduled by the computer's operating system and the scheduling can be affected by other programs running on the same computer.

Let us define X_j as the time between the DNN's $(j-1)$-th and the j-th memory accesses to the targeted LLC set:

$$X_j = t_j - t_{j-1} \tag{6.1}$$

For the sake of consistency, we define $t_0 = 0$ to be the clock cycle at which the DNN execution starts. Due to the randomness in t_j's, X_j's are also random variables.

Let $M(t)$ be the total number of times that the DNN accesses the targeted LLC set up to cycle t, a.k.a.

$$M(t) = \text{argmax}_j\{t_j \leq t\} \tag{6.2}$$

and its expected value be

$$m(t) = E[M(t)] \tag{6.3}$$

Since the above-introduced random variables can characterize the DNN's access pattern to the targeted LLC set and the pattern is dependent on the DNN's structure, it is desirable for the attacker to obtain information about the value of these variables. This can be done using Prime+Probe on the targeted LLC set. Specifically, let us suppose that **the attacker probes the targeted LLC set every c clock cycles for a total of p probes**. In each probe, every block in the targeted set is accessed. Assuming a least-recently-used (LRU) replacement policy, ideally, if the attacker accesses each block simultaneously, the LLC set will be filled

entirely with the attacker's data after each probe. The attacker measures the access latency of each block of the set. Since a cache hit would be a much lower access latency than a miss, the access latency will indicate whether the access was a cache hit or miss and the two are very unlikely to be confused.

Let y_k be the number of LLC misses the attacker observes in the k-th probe ($1 \leq k \leq p$). Assuming that other processes running on the same computer have a negligible probability of accessing the targeted LLC set between the $(k-1)$-th probe and the k-th probe, the number of missed blocks in the targeted LLC set indicates how many times the DNN has accessed this LLC set between time of the last probe, $(k-1)c$, and the time of the current probe, kc. Recall from Equation 6.2 that the DNN makes $M((k-1)c) - M(kc)$ accesses to the targeted LLC set in this period. Suppose the LLC is γ-way associative (*i.e.,* there are γ blocks in the targeted LLC set). Hence y_k is capped by γ and can be expressed by

$$y_k = \min\{\gamma, M(kc) - M((k-1)c)\} \tag{6.4}$$

Let us call $Y = (y_1, y_2, \ldots, y_p)$ the **cache side-channel trace** of a DNN. *Y is the cache side-channel information that can be directly observed from Prime+Probe.* Due to the randomness involved in the time of each access of the DNN, repeated measurements of the cache side-channel are made so that the average of the traces will be close to the expected value. Let $\mathcal{Y}$ be the set of traces obtained by repeated Prime+Probe measurements. $\mathcal{Y}$ characterizes the memory access pattern, and hence the structure, of the DNN.

The above description holds for both the VDNN and the ADNN. In the rest of this paper, we use superscripts "^{A}" and "^{V}" to denote variables of the ADNN and the VDNN, respectively, and use "$^{A/V}$" when an expression applies to both DNNs.

6.4.2 GANRED Components

The notation of some important components of GANRED framework are explained as follows.

$\mathcal{Y}^V$: the set of the VDNN's cache side-channel traces. This serves as the ground truth of the GANRED framework. The purpose of GANRED is to find a structure of the ADNN that makes the ADNN produce identical cache side-channel traces to $\mathcal{Y}^V$.

Θ: the set of estimated dimension parameters of the ADNN. Recall that the list of such parameters are listed in Table 6.1.

$G(\Theta)$: the generator that builds the ADNN with Θ and generates its cache side-channel traces as follows. (1) The ADNN is constructed with dimension parameters Θ and random weights. (2) The ADNN is executed and its cache side-channel trace is measured using Prime+Probe (*i.e.,* in the same way that the VDNN is sampled). (3) Step (2) is repeated multiple times in order to get a set of cache side-channel traces. Hence, the output of $G(\Theta)$ is a set of cache side-channel traces of the the ADNN, *i.e.,* $\mathcal{Y}^A$.

$D(\mathcal{Y}^V, \mathcal{Y}^A)$: the discriminator that compares the VDNN's traces, $\mathcal{Y}^V$, with the ADNN's, $\mathcal{Y}^A$. Recall that the length of each trace in $\mathcal{Y}^{A/V}$ is p. For each k such that $1 \leq k \leq p$, let $\bar{y}_k^{A/V}$ be the average number of cache misses in the k-th probe of ADNN/VDNN's cache side-channel traces. The discriminator's output, R, is also a p-element vector, *i.e.,* $R = (r_1, r_2, \ldots, r_p)$. We call R the **discriminator trace**. r_k is an indicator of how well the two traces match at the k-th probe. For this purpose, we could define r_k the difference between the two average cache misses, *i.e.,* $|\bar{y}_k^A - \bar{y}_k^V|$. However, experiment data can be noisy and make the discriminator trace R noisy. So instead, we take the two trace segments that are around the k-th probe of ADNN's average trace and VDNN's average trace and define r_k as the root-mean-square difference of the two trace segments. This will serve the discriminator's purpose better.

The validator is another important component of GANRED. Details of the validator are introduced in Sec. 6.4.4.

6.4.3 GANRED Framework

As a prerequisite of GANRED, the attacker repeatedly measures VDNN's cache side-channel using Prime+Probe and obtains a set of traces $\mathcal{Y}^V$. $\mathcal{Y}^V$ is then given to the GANRED framework, which takes the steps in Algorithm 6.1 to recover the victim DNN structure. In essence, GANRED determines the structure of the

Algorithm 6.1: GANRED Implementation

input : $\mathcal{Y}^V$; // VDNN's cache side-channel trace
output: Θ ; // ADNN's final dimension parameters

Initialization: $l \leftarrow 1$, $\Theta \leftarrow \emptyset$, $k_l \leftarrow 0$;
// l: estimated # layers in VDNN;
// k: the probe at which the traces starts to diverge
while $k_l < p$ **do**
  $l \leftarrow l + 1$;
  $\theta_l^* \leftarrow \emptyset$; // tracking optimal parameters of one layer
  $k_l^* \leftarrow k_{l-1}$; // k_l according to the current θ^*
  foreach $\theta_l \in \mathcal{S}_l$ **do**
    // $\mathcal{S}_l$: the set of all feasible parameter combinations of the l-th layer
    $\hat{\Theta} \leftarrow \Theta \cup \theta_l$;
    // Append enumerated parameters θ_l to existing parameters Θ
    $R = (r_1, r_2, \ldots, r_p) \leftarrow D(\mathcal{Y}^V, G(\hat{\Theta}))$;
    // Call the discriminator to compare traces of VDNN and ADNN
    $k_l' \leftarrow \text{argmax}_h r_1, r_2, \ldots, r_h < \eta$;
    // Given a threshold η, find how long the two sets of traces match from beginning
    if $k_l' > k_l^*$ **then**
      if $validate(\theta_l, k_{l-1}, k_l') == TRUE$ **then**
        // *TRUE* indicates a successful validation. Explained in Sec. 6.4.4
        $k_l^* \leftarrow k_l'$;
        $\theta_l^* \leftarrow \theta_l$;
      end
    end
  end
  $\Theta \leftarrow \Theta \cup \theta_l^*$;
  $k_l \leftarrow k_l^*$;
end

first layer before working on the second layer, determines the second before the third, and so on, until the two DNN's traces match entirely. We explain the procedure to determine the structure of each layer of the ADNN in detail as follows.

Recall that the set of parameters that need to be found for each layer is listed in Table 6.1. Notice that GANRED will work as long as the structure search space of each layer is finite. In this work, without loss of generality, we define the structure search space by the properties that state-of-the-art DNNs (*e.g.* AlexNet [52], the VGG family [97], and ResNet [41]) have in common:

1. If the l-th layer is a convolutional layer, then the filter width $1 \leq w_l^f \leq 11$, the output depth $d_l^{out} = 64 \times n$ where n is an integer and $1 \leq n \leq 32$, and the stride of convolution δ_l is 1 or 2.;

2. If the l-th layer is a fully connected layer, then the number of output neurons $z_l^{out} = 2^n$ where n is an integer and $8 \leq n \leq 13$.

Additionally, given that the user must provide input for and interpret output of the DNN in order to use it, the input and output dimensions of the VDNN will always be made available to the attacker. However, the attacker does not know the type (convolutional or fully connected) of each layer or the number of layers int the VDNN. Note that this is the same structure search space as considered by existing attacks [123, 42].

Suppose that GANRED is looking for the structure of the l-th layer, which means the first $l-1$ layers' structures have been determined. In this case, Θ contains the ADNN's parameters of the first $l-1$ layers. Let us use $\mathcal{S}_l$ to denote the

structure search space of layer l. The attacker enumerates through this space. For each structure within the search space, denoted as θ_l, an ADNN is constructed by appending a layer with dimension parameters given in θ_l to the already-determined $l-1$ layers (with parameters in Θ).

The generator then measures this ADNN with Prime+Probe repeatedly to obtain a set of cache side-channel traces, $\mathcal{Y}^A$. The discriminator then compares $\mathcal{Y}^A$ with $\mathcal{Y}^V$ and obtains the discriminator trace. Details of the generator and the discriminator have been described in Sec. 6.4.2. Each element of the discriminator trace is compared to a given threshold value η.

We say that the two traces *match* at probe k if $r_k < \eta$. Let k'_l be the last probe before the discriminator trace rises beyond η or ends. In other words, for any integer i within $1 \leq i \leq k'_l$, $r_i < \eta$. Recall that the attacker probes the targeted LLC set every c clock cycles. Hence the matching period stands for a time duration of $k'_l c$. We use k_{l-1} to denote the # of probes that the cache traces of the VDNN match the trace of the ADNN *without the l-th layer* (*i.e.,* the first $l-1$ layers of the ADNN with parameters in Θ).

If θ_l is the structure of the l-th layer that makes the two traces match for the longest period so far, it has the potential to be the correct structure of layer l. There is one caveat to be noticed. Due to the sequential nature of DNNs, the memory accesses of one layer must all finish before the next layer's accesses start. Therefore, if θ_l has the correct parameters of the VDNN's l-th layer, the added matching period due to the l-th layer, *i.e.,* $k'_l c - k_{l-1} c$, should be approximately the running time of the l-th layer of both the VDNN and the ADNN. However, the attacker does

not know which segment in the VDNN's trace corresponds to the l-th layer. The attacker can, though, verify whether the added matching period is close enough to the *theoretical* running time of a layer with parameters in θ_l. This technique can rule out θ_l if θ_l causes the ADNN's traces to diverge from the VDNN's traces in the middle of ADNN's execution. This is done by the *validator*. The details of how the validator calculates the theoretical running time of a layer is introduced in Sec. 6.4.4.

The successfully validated structure of the l-th layer that makes the two DNN's traces match for the longest time is chosen as the final structure of the l-th layer. If the two DNN's traces still do not match for the entire p probes, the attacker uses the same process to find the $(l+1)$-th layer. If the p probes have all matched, the ADNN's dimension parameters Θ is considered as the result of the attack.

6.4.4 Validating Reverse Engineered Parameter Combinations

During the reverse engineering of the l-th layer, if a structure denoted by θ_l makes the ADNN's traces and VDNN's traces match for the longest, the validator need to be invoked in order to verify whether θ_l is a "false positive" solution. Specifically, the validator will find whether the ADNN's traces deviate from the VDNN's in the middle of ADNN's execution, which should note be the case for the correct parameters of layer l. If the ADNN's traces match the VDNN's for k_{l-1} probes without layer l and k'_l probes with layer l, then layer l (with parameters θ_l)

makes the matched period increase by $(k'_l - k_{l-1})c$ clock cycles. This suggests that, if θ_l contains the correct dimension parameters of the l-th layer, the running time of the l-th layer is approximately $(k'_l - k_{l-1})c$ clock cycles.

The validator estimates the *theoretical* running time of a layer with parameters θ_l based on the following observation: *the execution time of a layer is linear in both its number of multiply-and-accumulate (MAC) operations and the number of cache misses.* [1] Hence, the validator uses a linear regression analysis to estimate the running time of a layer with parameters θ_l. Let $\hat{t}$ be the estimated running time. $\hat{t}$ is then compared to the increase in the length of matched period of the two DNNs' traces, $(k'_l - k_{l-1})c$. If the difference is below a certain threshold, then θ_l is accepted. Otherwise, θ_l is deemed a "false positive" and rejected. The validator proves to be an essential component of GANRED without which the correct structure of the VDNNs cannot be found.

In the rest of this subsection, we present the details of the linear regression process to estimate a layer's running time.

6.4.4.1 Convolutional (Conv) Layers

The operation of a Conv layer is illustrated in Fig. 6.1. When a filter is convolved with the input feature map (IFM), with each step that the filter moves, a new output neuron is computed. We assume that only the new input neurons (*i.e.,* that were not used in the last inner product) will result in cache misses. The number

[1] In Sec. 6.4.4, the notion of "cache misses" refers to the entire cache, not just the LLC set selected by the Prime+Probe attack.

of such new input neurons is $f_l \cdot d_l^{in} \cdot \delta_l$, where f_l is the filter width, d_in is the IFM depth, and δ_l is the convolution stride (see Table 6.1). We calculate the *theoretical number of cache misses* of the Conv layer, denoted as $u^{conv}(\theta)$, as the sum of two components: (a) the total number of "new input neurons" as described above for evaluating the entire OFM, and (b) the cache misses when each input neuron and weight is used for the first time.

$$\begin{aligned} u^{conv}(\theta_l) =&(((P_l+1)^2 w_l^{out,2} - 1) f_l d_l^{in} \delta_l \\ &+(f_l^2 + w_l^{in,2}) d_l^{in}) d_l^{out} \end{aligned} \tag{6.5}$$

Since a cache miss results in significantly longer latency than a cache hit, the number of cache misses will impact the Conv layer's running time.

Let us use $v^{conv}(\theta_l)$ to denote the # of MAC operations of a Conv layer, which can be given by

$$v^{conv}(\theta_l) = 2(P_l+1)^2 w_l^{out,2} f_l^2 d_l^{in} d_l^{out} \tag{6.6}$$

In order to show that the running time of a Conv layer's running time is linear in both the # of cache misses and the # of MAC operations, we conduct the following experiment. We take a population of Conv layers that is within our structure search space and measured the running time of these layers. A linear regression analysis is then conducted to verify the linearity. The regression shows that the linear scores of both # of cache misses and # of MAC operations to the running time are greater than 0.99 (1.0 is perfectly linear). In Fig. 6.3, we plot the layers with equal # of MAC operations on the same line and show that a Conv layer's running time linearly increases in the theoretical # of cache misses. Let $\hat{t} = \hat{A}^{conv} u^{conv}(\theta_l) + \hat{B}^{conv} v^{conv}(\theta_l) + \hat{C}^{conv}$ be the regression result equation.

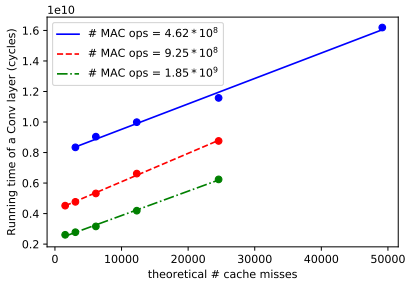


Figure 6.3: The linear relationship between Conv layer running time and theoretical cache misses

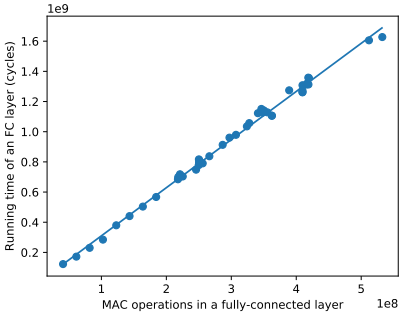


Figure 6.4: Linear regression on # MAC operations and trace length of an FC layer

6.4.4.2 Fully Connected (FC) Layers

In an FC layer, since there is no reuse of weights in the computation of different output neurons, the number of MAC operations is proportional to the theoretical number of cache misses. Therefore, we only look for the linear relationship between an FC layer's running time and the MAC operations. The # of MAC operations can be given by

$$v^{FC}(\theta_l) = 2z_l^{in} z_l^{out} \tag{6.7}$$

Similar to the analysis for Conv layers, we select a population of FC layers from the feasible structures and measure their running time. Then, a linear regression is conducted on the # of MAC operations to the running time. A clear linear relationship can be observed from Fig. 6.4 and we use $\hat{t} = \hat{A}^{FC} v^{FC}(\theta_l) + \hat{B}^{FC}$ to denote the regression result.

6.4.5 Mathematical Justification of GANRED

As we have described in Sec. 6.4.3, GANRED reverse engineers the DNN in a layer-by-layer manner. It must find the correct structure of the current layer before moving on to the next layer. To this end, we intuitively assumed that when the ADNN (with l^A layers) has the same structure as the first l^A layers of the VDNN (which has l^V layers in total, $l^V \geq l^A$), then the ADNN's cache side-channel traces $\mathcal{Y}^A$ should match with the VDNN's traces $\mathcal{Y}^V$ before the end of ADNN's execution. We justify this premise in this subsection. The following is assumed about a DNN's memory access:

1. The DNN layers are executed sequentially and hence the memory accesses of a layer must be completed before the next layer's memory accesses begin.

2. Each $X_j^{A/V}$, *i.e.,* the time between the ADNN/VDNN's $(j-1)$-th and j-th access to the targeted LLC set, is subject to a Gaussian distribution. For the VDNN, $X_j^V \sim \mathcal{N}(\mu_j, \sigma_j^2)$ where $1 \leq j \leq s^V$. If ADNN has the same structure as VDNN's first l^A layers, we also have $X_j^A \sim \mathcal{N}(\mu_j, \sigma_j^2)$ where $1 \leq j \leq s^A$.

3. Each $X_j^{A/V}$ is independent, *i.e.,* for any $1 \leq j_1 \neq j_2 \leq s^{A/V}$, $X_{j_1}^{A/V}$ and $X_{j_2}^{A/V}$ are independent.

If ADNN has the same structure as the first l^A layers of VDNN, the expected time at which ADNN's last access to the targeted LLC set would occur can be expressed as $\sum_{j=1}^{s^A} \mu_j$. Our objective is then to prove that, in this case, the difference between the cache side-channel traces of ADNN and VDNN before time $\sum_{j=1}^{s^A} \mu_j$ can be upper bounded by a small value. In order to prove this, we first bound the difference between the two DNNs' expected # of memory accesses up to time $\sum_{j=1}^{s^A} \mu_j$, *i.e.,* $|m^V(t) - m^A(t)|$, as follows.

Theorem 6.1. If $t < \sum_{j=1}^{s^A} \mu_j$, then $|m^V(t) - m^A(t)|$ can be upper bounded by

$$U(t) = \sum_{k=s^A+1}^{s^V} \left(\sqrt{\frac{\sum_{j=1}^{k} \sigma_j^2}{2\pi}} \cdot e^{-\left(\frac{-t+\sum_{j=1}^{k} \mu_j}{\sqrt{2\sum_{j=1}^{k} \sigma_j^2}}\right)^2} \right) \tag{6.8}$$

Proof of Theorem 6.1. We first find the expression of $m^V(t) - m^A(t)$ in terms of the CDF of between-access time.

$$\begin{aligned} m^{A/V}(t) =& E[M^{A/V}(t)] = \sum_{j=1}^{s^{A/V}} j \cdot \text{Prob}[M(t) = j] \\ =& \sum_{j=1}^{s^{A/V}} j \cdot (\text{Prob}[M(t) \leq j] - \text{Prob}[M(t) \leq j-1]) \\ =& \sum_{j=1}^{s^{A/V}} \text{Prob}[M(t) \leq j] \end{aligned}$$

$M^{A/V}(t) \leq j$ indicates that the j-th memory access occurs no later than time t, *i.e.*, $t_j^{A/V} = \sum_{i=1}^{j} X_i^{A/V} \leq t$. Because all X's are subject to independent Gaussian distributions as described above, we have

$$t_j^{A/V} \sim \mathcal{N}(\sum_{i=1}^{j} \mu_i, \sum_{i=1}^{j} \sigma_i^2)$$

Therefore, the probability of $M^{A/V}(t) \leq j$ can be calculated via the CDF of the above Gaussian distribution:

$$\text{Prob}[M^{A/V}(t) \leq j] = \frac{1}{\sqrt{2\pi \sum_{i=1}^{j} \sigma_i^2}} \int_{-\infty}^{t} e^{-\left(\frac{x-\sum_{i=1}^{j}\mu_i}{\sqrt{2\sum_{i=1}^{j}\sigma_i^2}}\right)^2} \mathrm{d}x$$

And hence

$$\begin{aligned} & m^V(t) - m^A(t) \\ = & \sum_{j=s^A+1}^{s^V} \text{Prob}[M(t) \leq j] \\ = & \sum_{j=s^A+1}^{s^V} \frac{1}{\sqrt{2\pi \sum_{i=1}^{j} \sigma_i^2}} \int_{-\infty}^{t} e^{-\left(\frac{x-\sum_{i=1}^{j}\mu_i}{\sqrt{2\sum_{i=1}^{j}\sigma_i^2}}\right)^2} \mathrm{d}x \end{aligned} \tag{6.9}$$

Since each integral is positive, we only need to prove $m^V(t) - m^A(t) < U(t)$. In Theorem 6.1, we specify that $t < \sum_{j=1}^{s^A} \mu_j$ and. Let $h_j = \sum_{i=1}^{j} \mu_i - t$. Due to the symmetry of the probability density of Gaussian distributions, we can rewrite the above expression as

$$\begin{aligned} & m^V(t) - m^A(t) \\ = & \sum_{j=s^A+1}^{s^V} \frac{1}{\sqrt{2\pi \sum_{i=1}^{j} \sigma_i^2}} \int_{-\infty}^{\sum_{i=1}^{j}\mu_i - h_j} e^{-\left(\frac{x-\sum_{i=1}^{j}\mu_i}{\sqrt{2\sum_{i=1}^{j}\sigma_i^2}}\right)^2} \mathrm{d}x \\ = & \sum_{j=s^A+1}^{s^V} \frac{1}{\sqrt{2\pi \sum_{i=1}^{j} \sigma_i^2}} \int_{\sum_{i=1}^{j}\mu_i + h_j}^{\infty} e^{-\left(\frac{x-\sum_{i=1}^{j}\mu_i}{\sqrt{2\sum_{i=1}^{j}\sigma_i^2}}\right)^2} \mathrm{d}x \end{aligned}$$

In each integral, since $x > \sum_{i=1}^{j} \mu_i + h_j$ and $h_j > 0$, we have $\frac{x - \sum_{i=1}^{j} \mu_i}{h_j} > 1$. Therefore, each integral can be upper bounded by

$$\begin{aligned}
&\int_{\sum_{i=1}^{j} \mu_i + h_j}^{\infty} e^{-\left(\frac{x - \sum_{i=1}^{j} \mu_i}{\sqrt{2 \sum_{i=1}^{j} \sigma_i^2}}\right)^2} \mathrm{d}x \\
&< \int_{\sum_{i=1}^{j} \mu_i + h_j}^{\infty} \frac{x - \sum_{i=1}^{j} \mu_i}{h_j} e^{-\left(\frac{x - \sum_{i=1}^{j} \mu_i}{\sqrt{2 \sum_{i=1}^{j} \sigma_i^2}}\right)^2} \mathrm{d}x \\
&= \sum_{i=1}^{j} \sigma_i^2 \cdot e^{-\left(\frac{\sum_{i=1}^{j} \mu_i - t}{\sqrt{2 \sum_{i=1}^{j} \sigma_i^2}}\right)^2}
\end{aligned}$$

Therefore, we can upper bound $m^V(t) - m^A(t)$ by

$$m^V(t) - m^A(t) < \sum_{j=s^A+1}^{s^V} \sqrt{\frac{\sum_{i=1}^{j} \sigma_i^2}{2\pi}} e^{-\left(\frac{\sum_{i=1}^{j} \mu_i - t}{\sqrt{2 \sum_{i=1}^{j} \sigma_i^2}}\right)^2} = U(t)$$

Hence proved. □

In order to illustrate that $U(t)$ is an extremely value in a straightforward manner, we estimate the parameters from a VDNN (AlexNet) and the ADNN with the same structure as the first 2 layers of the VDNN. These parameters include s^A, s^V, and μ_j. σ_j is assumed to be 20% of μ_j. Based on these parameters, $U(t)$ is plotted for the period close to the end of ADNN's execution as shown in Fig. 6.5. We could only plot this period because $U(t)$ monotonically increases in t as $t < \sum_{j=1}^{s^A} \mu_j$ and its value is so small before the plotted period that it is smaller than the smallest positive number that a floating point number can represent.

Recall that y_k is the number of observed LLC misses in the k-th probe and was defined in Equation 6.4. y_k stands for the cache side-channel information that GANRED uses. The k-th probe will occur before the ADNN's last access to the

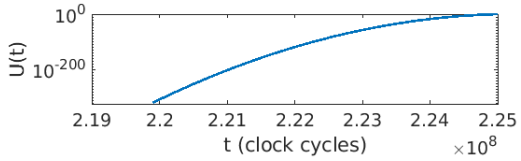


Figure 6.5: $U(t)$ vs. t given typical parameters in Equation 6.8.

targeted LLC set if $k < \sum_{j=1}^{s^A} \mu_j/c$. The following theorem indicates that, if the ADNN has the same structure as the first l^A layers of the VDNN, then the two DNNs should have very close cache side-channel traces as the probe index $k < \sum_{j=1}^{s^A} \mu_j/c$.

Theorem 6.2. If $k < \sum_{j=1}^{s^A} \mu_j/c$, then $|E[y_k^V] - E[y_k^A]|$ is upper bounded by $U(kc)$.

Proof of Theorem 6.2. By Theorem 6.1, we know that $m^V(kc) - m^A(kc) < U(kc)$. Note that $m^V(kc) - m^A(kc)$ can be expanded in the following way (note that $m^A(0) = m^V(0) = 0$):

$$\begin{aligned} & U(kc) > m^V(kc) - m^A(kc) \\ =& \sum_{i=1}^{k} (m^V(ic) - m^V((i-1)c)) - \sum_{i=1}^{k} (m^A(ic) - m^A((i-1)c)) \\ =& \sum_{i=1}^{k} \big((m^V(ic) - m^V((i-1)c)) - (m^A(ic) - m^A((i-1)c)\big) \end{aligned}$$

Let us first define $q_k^{A/V} = m^{A/V}(kc) - m^{A/V}((k-1)c)$. We can continue the above equation by

$$m^V(kc) - m^A(kc) = \sum_{i=1}^{k} (q_i^V - q_i^A) < U(kc) \tag{6.10}$$

Note that $q_k^V - q_k^A > 0$ given any k. This is because $q_k^V - q_k^A = [m^V(kc) - m^A(kc)] - [m^V((k-1)c) - m^A((k-1)c)]$ and, from Equation 6.9, it is clear that $m^V(t) - m^A(t)$ monotonically increases in t. Therefore, Equation 6.10 suggests that for any k that satisfies $1 \leq k < \sum_{j=1}^{s^A} \mu_j/c$, we have $q_k^V - q_k^A < U(kc)$, which is a necessary condition that their summation is smaller than $U(kc)$.

$y_k^{A/V} = \min\{\gamma, q_k^{A/V}\}$. Since $q_k^V > q_k^A$, $y_k^V \geq y_k^A$. Hence we only need to prove $E[y_k^V] - E[y_k^A] < U(kc)$.

$$
\begin{aligned}
& E[y_k^V] - E[y_k^A] = E[y_k^V - y_k^A] \\
=& (\gamma - \gamma)\text{Prob}[y_k^V \geq \gamma,\ y_k^A \geq \gamma] + \\
& (\gamma - y_k^A)\text{Prob}[y_k^V \geq \gamma,\ y_k^A < \gamma] + \\
& (y_k^V - y_k^A)\text{Prob}[y_k^V < \gamma,\ y_k^A < \gamma] \\
<& 0 + U(kc)\text{Prob}[y_k^A < \gamma] \\
<& U(kc)
\end{aligned}
$$

Hence proved □

From Fig. 6.5, we know that $U(kc)$ is a small value. The average number of cache misses in each probe will converge to the expected number given a sufficient number of repeated cache side-channel measurements. Therefore, if the ADNN has the same structure as the first l^A layers of the VDNN, the two DNNs will have very close average cache side-channel traces before ADNN's execution ends. *This means that comparing cache side-channel traces is a good way of determining whether the*

ADNN has correct parameters. Although it is still theoretically possible that two DNNs with different structures have indistinguishable cache traces, the sensitivity of cache traces to the structure makes this event rather unlikely.

6.5 Experiments

To evaluate the efficacy of the proposed GANRED framework, we have applied it to reverse engineer several state-of-the-art DNN structures on a real server. In our experiments, the VDNN is hosted on a Linux server which uses TensorFlow as the machine learning framework. The attacker logs into the server without sudo privilege. The APIs available to the attacker are those to construct the ADNN with convolutional, fully connect, and pooling layers. The server has an Intel i7-7700 CPU which has an 8MB, 16-way associative last-level cache (LLC). Each cache block contains 64 bytes (*i.e.,* 6 bits block-offset). Hence there are 8192 associative sets and the set index has 13 bits. The following practical challenges have been addressed in our experiments.

(1) Having to probe each block in the targeted LLC set significantly limits the frequency at which our attacker can probe the cache compared to a Flush+Reload attacker [123, 42]. Nonetheless, we achieve more accurate reverse engineering results than these attacks.

(2) The cache hit/miss result data has to be stored real-time but allocating another array for data storage will result in cache interference. Hence, we store the hit/miss result of each probe on the probed lines directly.

In our experiments, we use GANRED to reverse engineer state-of-the-art DNNs, including AlexNet [52] and the VGG family [97]. The generator repeatedly measures the cache side-channel trace of each DNN for 50 times. Fig. 6.6 shows the average cache side-channel traces of each VDNN and some traces of ADNNs that GANRED determines as having the correct structure in the progress. We also show the discriminator's output trace when comparing these ADNNs with the VDNN. As we can observe, these ADNNs' traces match well with the VDNNs' until the former are about to end. This agrees with what we derived in Sec. 6.4.5. The discriminator is able to capture the deviation as its output increases beyond the threshold η.

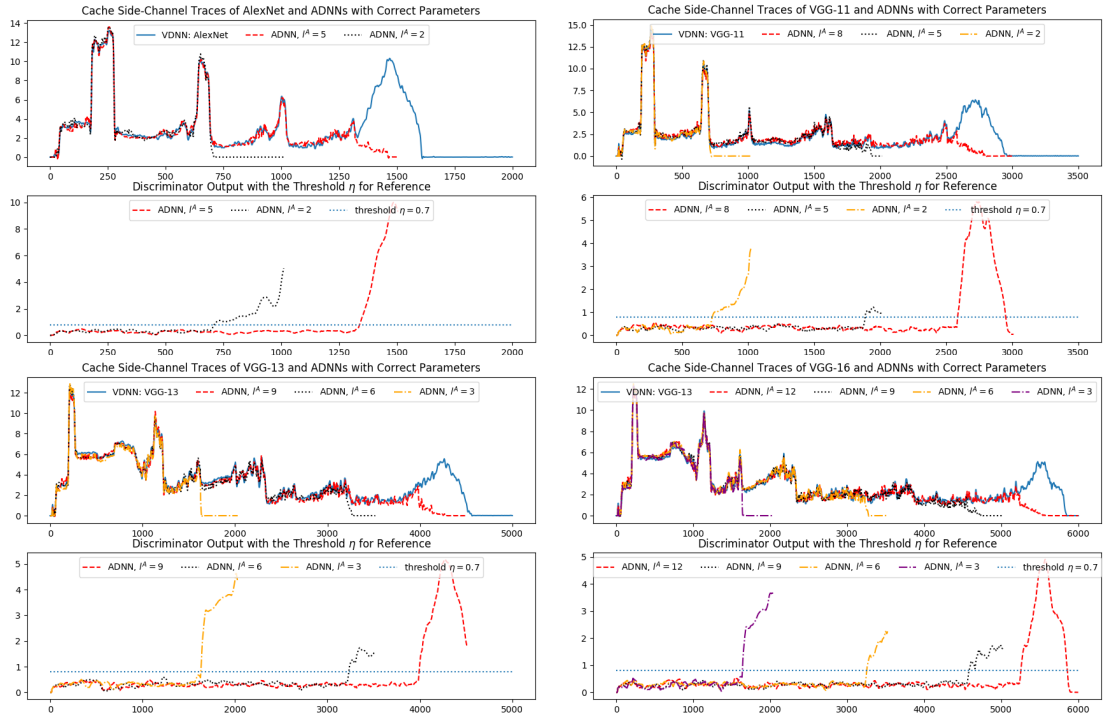


Figure 6.6: Upper images: the cache side-channel traces of AlexNet and VGG Nets and some ADNNs with correct parameters in the progress of GANRED. Lower images: the discriminator's outputs corresponding to the ADNNs. X-axis: number of probes.

In order to evaluate the performance of GANRED, we highlight the reduction in DNN architecture search space. To this end, we estimate the size of the original DNN structure search space (*i.e.,* without side-channel information) using the principles stated in Sec. 6.4.3 and assuming that the attacker knows the number of layers in the DNN. *Note that this is an underestimation, since the attacker under our attack model does not know the number of layers and is thus facing an even larger DNN structure search space.*

6.5.1 Attack Results

For each VDNN, GANRED is able to recover the precise structure. The # of possible structure of each VDNN benchmark and the attack results are shown in Figure 6.7. Recall that GANRED also eliminates the need for code access and shared main memory segments between the attacker and the victim. These are substantial improvements over existing attacks. Our attack is also scalable: the attack time increases linearly with the number of layers although the possible structure space grows exponentially. The reason for the linear growth in attack time is as follows. When reverse engineering any layer, the layer's IFM dimensions are always known, since the IFM is either the DNN's input (public knowledge) or the OFM of last layer (determined in the last layer). Since the same structure constraints apply to any layer, the number of ADNNs that need to be constructed and measured is the same. Hence the time spent on reverse engineering each layer is roughly the same.

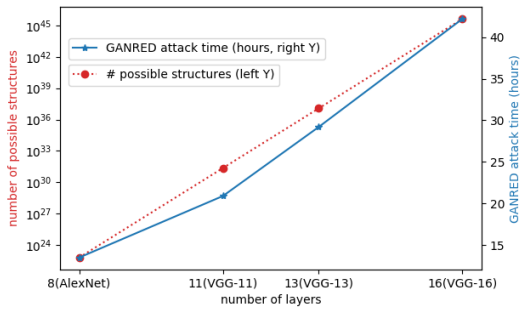


Figure 6.7: GANRED attack results

6.6 Summary

In this work, we develop GANRED, a GAN-based DNN structure reverse engineering framework which utilizes cache timing side-channel information. Unlike prior reverse engineering approaches which required shared library code in the main memory and other resources that may be unrealistic, our attack uses Prime+Probe and thus only requires minimal resources. GANRED compares the VDNN's cache side-channel trace with that of ADNN with estimated structure and converges when the two traces become identical. Experiments show that the precise structure of each VDNN benchmark has been found and the attack complexity scales linearly with the number of layers in VDNN. Therefore, we conclude that our attack is successful and scalable. The fundamental reason that GANRED produces more accurate results than existing attacks [123, 42] may be that the cache side-channel information used by GANRED inherently contains more information. Those existing attacks monitors certain library function calls, which only accounts for a tiny portion of DNN's memory access. In contrast, the cache side-channel traces measured by

GANRED contains information about the DNN's overall memory pattern. Such an attack method must be considered when the intellectual property of a DNN is concerned.

Chapter 7: Mitigating Reverse Engineering Attacks on Neural Networks

In Chapter 6, we studied the reverse engineering of DNNs via cache side-channel information. In this chapter, we consider a much strong attacker and propose a countermeasure. Instead of only observing some side-channels, the attacker now has the complete knowledge of the DNN's memory access pattern, based on which she can reverse engineer the DNN structure much more easily. In order to defend such an attack, we propose a defensive memory access mechanism which utilizes oblivious shuffle, address space layout randomization, and dummy memory accesses to counter such attacks. Experiments show that our defense exponentially increases the attack complexity with asymptotically lower memory access overhead compared to generic memory obfuscation techniques such as oblivious RAM (ORAM) and is scalable to larger DNNs.

7.1 Introduction

It has been shown that the DNN structure can be easily reverse engineered if the memory access pattern of the processor running the DNN is leaked [43]. This is a significant security concern. To the best of the authors' knowledge, no

efficient countermeasure has been proposed. Although applying an oblivious RAM (ORAM) protocol is a well-established approach to hide the memory access pattern, it comes with very high memory access overheads [100, 78, 3]. Because running DNNs is a memory intensive task, the speed of the DNN running in hardware is mostly constrained by the number of memory accesses [21, 22, 134]. This makes ORAM-based memory access obfuscation impractical for DNNs.

Oblivious shuffle also provably obfuscates the address space with lower overhead than ORAM albeit with weaker theoretical guarantees [71] (detailed in Section 7.3). In this chapter, we utilize oblivious shuffle to obfuscate a subset of the memory access patterns. The subset itself is customizable by the designer. A bigger subset (which could at most include the entire memory space) results in stronger obfuscation at the cost of higher memory access overheads. In addition, we use address space layout randomization (ASLR) on the entire memory space and add dummy memory access (DumMA) requests to the shuffled addresses for further improvements in security guarantees.

The contribution of this chapter is as follows:

- A novel defense strategy to obfuscate the processor's memory access pattern is proposed in order to reduce information leakage about the structure of the DNN being executed. This strategy utilizes three techniques: oblivious shuffle, address space layout randomization (ASLR), and dummy memory access (DumMA). Although these techniques have been existing, our innovation lies in combining them strategically to thwart the attack with low overhead.

- A modified attack based on that in [43] is formulated to reverse engineer the DNN structure in the presence of our defense in order to evaluate the security of our defense.

- Experimental results show that the complexity of the modified attack is very high thereby demonstrating the effectiveness of our defense.

- It is also shown that the memory access overhead of our defense is very low and does not increase with the DNN depth, making our approach scalable to deeper models.

7.2 Attack Model

The recent work of Hua, Zhang, and Suh [43] illustrates an elegant optimization theoretic attack based on the *memory access side-channels* of systems running DNNs. The attack model considered is as follows. The owner of the DNN model wants to enable the user to run the model on her own processor (*e.g.* a CPU, GPU, or DNN accelerator) without exposing the structure of the DNN model. The attacker is considered to be the user who is honest-but-curious, *i.e.,* she wants to know the details of the model but does not interfere with the normal execution of the DNN model. The processor is considered as secure, *i.e.,* the attacker cannot observe or interfere with the processor's internal operations.

However, the attacker is able to observe the **memory access patterns** of the processor, *i.e.,* a transcript of its memory accesses including the accessed addresses, the access types (*i.e.,* read or write), and the time of each access. The attacker also

knows the input and output of the DNN (since she has I/O access to the DNN). As shown in [43], a reverse engineering attack can be formulated under this attack model and the architecture of a DNN can be extracted.

7.2.1 Attack Setup

The specific type of deep neural networks (DNN) of interest to us is convolution neural networks. These networks comprise two types of layers: **convolutional (Conv)** layers and **fully connected (FC)** layers. If a pooling layer exists following a Conv layer, then we count it as a part of the Conv layer, as is consistent with Chapter 6. Each layer transforms a set of input neurons, called the **input feature map (IFM)**, into a set of output neurons, called the **output feature map (OFM)**. The OFM of the previous layer is the IFM of the next layer. Figure 6.1 illustrates the structure of a Conv layer. The structure of a Conv layer can be described by a set of hyper-parameters which are listed in Table 7.1. [1] The structure of a fully connected layer is much simpler. If layer i is an FC layer, its IFM and OFM are vectors of length z_i^{in} and z_i^{out}, respectively. A 2-D weight matrix of dimension $z_i^f = z_i^{in} \times z_i^{out}$ transforms the IFM to the OFM.

As in the attack model of [43], the DNN model is stored in a virtual address space that starts from 1 and each neuron or weight takes exactly 1 address to store. The way that the neurons and weights are aligned in the memory is as follows: The first feature map (*i.e.,* the IFM of layer 0, of size z_0^{in}) starts from address 1 and

[1] Table 7.1 includes more parameters than Table 6.1 since we deal with a stronger attack whose result is not confined to any specific family of DNNs in this chapter.

Table 7.1: List of Hyper-parameters of Each Layer

Parameter	Definition
w_i^{in}, w_i^{out}	width of the IFM/OFM of layer i
d_i^{in}, d_i^{out}	depth of IFM/OFM of layer i
z_i^{in}, z_i^{out}, z_i^f	size of IFM/OFM/filter of layer i
P_i	indicator of whether pooling exists in layer i
f_i^{conv}, f_i^{pool}	filter width of convolution/pooling (if existing) of layer i
s_i^{conv}, s_i^{pool}	stride of convolution/pooling (if existing) of layer i
p_i^{conv}, p_i^{pool}	padding of convolution/pooling (if existing) of layer i

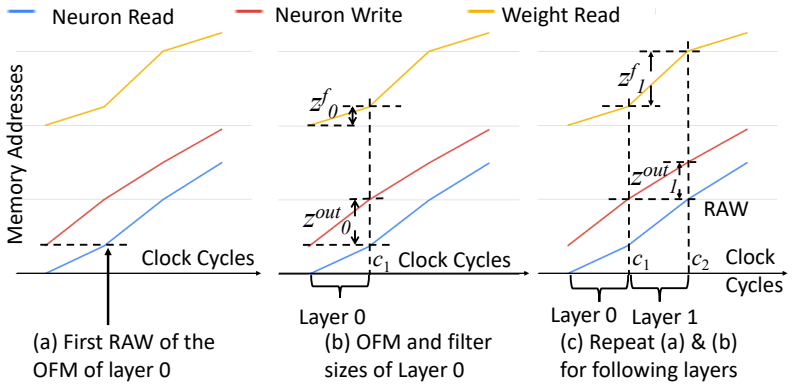


Figure 7.1: Illustration of the attack in [43] per Section 7.2.2.

ends at z_0^{in}. The second feature map (the OFM of layer 0 and the IFM of layer 1, of size z_0^{out} which equals z_1^{in}) starts at $z_0^{in} + 1$, and so on. The weights are stored in a separate area of memory and organized in a layer-by-layer configuration (similar to the neurons).

7.2.2 Attack Methodology

The attack to reverse engineer the structure of a DNN consists of 3 phases: determining the layer boundaries in the memory traces, solving for the feasible DNN structures which fit the memory trace, and training each feasible structure for the best match.

Phase 1: In this phase, the attacker determines the layer boundaries in the memory access transcript leaked by side-channels and obtains the IFM, OFM, and filter sizes of each layer. This process is illustrated in Figure 7.1.

The **layer boundaries** are expressed in terms of the clock cycles at which the first memory access of each layer occurs. Let $c_i,\ i \in [L]$ be the first clock cycle of layer i where L is the number of layers. The attacker determines the boundaries between layers by observing the first occurrence of **read-after-write (RAW)** in the memory. This is illustrated in Figure 7.1(a). Layer 0 writes its OFM to the memory and layer 1 needs to read the same memory location to get it as its IFM. Therefore, the first occurrence of RAW indicates that layer 0 has finished and layer 1 has started, thereby leaking c_1 (Figure 7.1(b)). In this way, by counting how many addresses have been written to before c_1, the attacker observes the OFM size of layer 0 z_0^{out} (Figure 7.1(b)). Similarly, the attacker can also observe the filter size of layer 0 z_0^f. The first RAW of layer 1's OFM marks the beginning of layer 2 (Figure 7.1(c)). By repeating the above procedure for all the subsequent layers, the IFM sizes, OFM sizes, filter sizes, the starting clock cycles of all the following layers can be observed.

Phase 2: After obtaining feature map and filter sizes in the previous phase, the attacker tries to obtain *all the feasible structures* of the DNN that conform with the observed access pattern. Each structure is described by a combination of the hyper-parameters of every layer (as listed in Table 7.1). These hyper-parameters are obtained by solving an **integer feasibility program (IFP)** problem that captures the relationship among the hyper-parameters for each layer with the information

obtained from phase 1. The IFP is defined by the following equations:

$$z_i^{in} = (w_i^{in})^2 \times d_i^{in}, \quad z_i^{out} = (w_i^{out})^2 \times d_i^{out} \tag{7.1}$$

$$z^f = (f_i^{conv})^2 \times d_i^{in} \times d_i^{out} \tag{7.2}$$

$$w_i^{out} = \frac{\frac{w_i^{in} - f_i^{conv} + p_i^{conv}}{s_i^{conv}} + 1 + P(p_i^{pool} - f_i^{pool})}{s_i^{pool} \times P + \bar{P}} \tag{7.3}$$

$$s_i^{conv} \leq f_i^{conv} \leq \frac{w_i^{in}}{2} \tag{7.4}$$

$$s_i^{pool} \leq f_i^{pool} \leq \frac{w_i^{in} - f_i^{conv} + p_i^{conv}}{s_i^{conv}} + 1 \tag{7.5}$$

$$p_i^{conv} < f_i^{conv}, \quad p_i^{pool} < f_i^{pool} \tag{7.6}$$

Phase 3: After all the feasible structures are obtained, the attacker trains each structure and picks the one with the highest accuracy as the final outcome. It was shown in [43] that the feasible structures obtained from the memory traffic were very few thereby significantly reducing the training effort.

7.2.3 Attack Complexity and Practicality

In this work, we consider 4 DNN benchmarks which are listed in Table 7.2. The complexity of the attack is measured using two metrics: the number of **IFP problems solved** and the total number of **feasible DNN structures**. The former represents the hardness involved with obtaining the set of feasible DNN structures (essentially the complexity of Phase 2). The latter represents the amount of training effort needed by the attacker to pick the best model (essentially Phase 3).

Table 7.2: Benchmark DNNs and attack complexity using the attack in [43]

Benchmark	input		# layers			attack complexity metrics	
	w_0^{in}	d_0^{in}	Conv	FC	Total	# IFPs solved	# feasible structures
DNN 1	28	1	2	2	4	4	4
DNN 2	32	1	3	2	5	5	6
DNN 3	32	3	4	3	7	8	1
DNN 4	64	3	5	3	8	9	8

We wrote a simulator to generate a processor's memory trace. The processor's memory trace is reverse engineered using the above-described attack method. The complexity of the attacks on the benchmark DNNs is also shown in Table 7.2. As seen, both metrics are low for all the benchmarks, indicating the low complexity of the attack.

It is important to note that the "exact" neural network with exactly the same weights and topology may not be the one synthesized by this attack. However, the attacker's objective would still be achieved since she would still be able to get substantially accurate classification performed by synthesizing the model based on the one running on the processor.

7.3 Cryptographic Preliminaries

The effectiveness of the above-mentioned attack necessitates a defense mechanism that reduces the information leakage of the DNN structure in the memory access patterns. Hiding memory access patterns is a well-studied problem and has been formalized via the notion of **Oblivious RAM (ORAM)** schemes. An ORAM scheme can be used to obfuscate the memory access patterns of any input RAM program and provides the strong theoretical guarantee that the obfuscated memory access patterns reveal no information about the input program [80]. However, even

the state-of-the-art ORAM protocol [100, 78, 3] incurs an **access overhead** of $\Omega(\log N)$, *i.e.,* the average number of memory accesses that have to be performed in order to access a single address in the original program is at least $\log N$, where N is the total number of address.

In our work, instead of ORAMs, we consider a different approach called **oblivious shuffle** whose overhead is much lower [34]. We define oblivious shuffle below followed by an explanatory example.

Definition 7.1 (Oblivious Shuffle)**.** A shuffle algorithm is an algorithm of the form $(\mathsf{Enc}(\pi(A)), \alpha) \leftarrow \mathsf{Shuffle}(A, \mathsf{Enc}, \pi)$ where A is an input array, Enc is a secure encryption algorithm, and π is a random, predetermined permutation function. The output of $\mathsf{Shuffle}$ is an encryption of the permutation of A according to π and a memory access transcript α. The $\mathsf{Shuffle}$ algorithm is an oblivious shuffle if α is independent of π.

Algorithm 7.1: A simple oblivious shuffle example

```
1 for i in [N] do
2 |   Read address i for A[i] and store A[i] in the secure on-chip memory of
  |     the processor
3 end
4 for i in [N] do
5 |   Write Enc(A[π^{-1}(i)]) to address i
6 end
```

A simple oblivious shuffle algorithm is shown in Algorithm 7.1. Using this algorithm, *the attacker will always see the same memory access pattern regardless of* π, which, in this case, is a read sequence followed by a write sequence, both in the address order of $0, 1, \ldots, N-1$. Hence the attacker cannot decipher π.

Reference [71] proposed the state-of-the-art oblivious shuffle algorithm, called the *Melbourne shuffle*, which is efficient and scalable: it only requires $O(\sqrt{N})$ private memory (*i.e.,* the memory not observable to the attacker) to shuffle an array of size $O(N)$ as proven in Theorem 5.1 in [71].

In this work, the permutation function π we choose 'looks' random and utilizes the internal randomness of the processor (which is fixed for the same shuffle but can vary for different shuffles). Similar to the above example, the attacker will also see a fixed memory access pattern α of Melbourne shuffle regardless of π.

Under this condition, if a processor is running a DNN model and switches between Melbourne shuffle phases and regular DNN phases, there will be three types of phases in the memory access pattern: (i) the Melbourne shuffle, (ii) the DNN accesses *inside* the shuffled addresses, and (iii) the DNN access *outside* the shuffled addresses. The attacker can distinguish these phases because, no matter what memory access pattern is generated by the DNN application, that of the Melbourne shuffle will always be α. The attacker can hence also observe the addresses that are shuffled. Note, however, that the attacker has no information about π, or the actual memory addresses accessed by the DNN application during type (ii) phases since she does not know the internal randomness of the processor, nor does α leak any information of π.

Running DNNs is a memory-intensive task where every neuron and weight needs to be accessed. To compare the memory access overhead of ORAM and that of the Melbourne shuffle, we take an array A of length N and require that every element in A be accessed once. With ORAM, the total # accesses will be $\Omega(N \log N)$. With

Melbourne shuffle, the memory accesses consists of two parts: those of the shuffle and those of the actual accesses. The Melbourne shuffle takes $O(N)$ memory accesses. Unlike the ORAM, once the oblivious shuffle is completed, there is no additional access overhead: one simply needs to access the new address. Hence the total # accesses will be $O(N) + N = O(N)$. *The overhead of the Melbourne shuffle is therefore a constant multiplicative factor*, making it asymptotically lower than that of the ORAM.

7.4 Defense Methodology

In this section, we propose a memory access strategy for processors to run DNN models with minimal leakage of structural information. The defense should fulfill two competing objectives:

- The resulting attack complexity should be very high.
- The memory access overhead should be low.

As discussed in Sec. 7.3, using ORAM will satisfy the first objective but fail the second one. In order to achieve both objectives, our proposed defense strategy utilizes 1) the *Melbourne shuffle* , 2) *address space layout randomization (ASLR)* [31], and 3) adding *dummy memory accesses (DumMA)*. A modified attack based on [43] to find the feasible structures of the DNN is also formulated in order to evaluate the security of our defense.

7.4.1 Utilizing Oblivious Shuffle

In order to obfuscate all the layer boundaries, a DNN with L layers needs $L-1$ oblivious shuffles (one for each layer boundary) during its execution. Note that we do not need to shuffle the entire memory: in the i-th shuffle, only the memory addresses accessed near c_i need to be shuffled (recall that c_i is the clock cycle at the beginning of layer i). Varying the number of shuffled addresses enables us to explore a spectrum of trade-offs between the memory access overhead and the security of the defense. In this subsection, we present how to determine when and where to shuffle and model the attacker's knowledge based on the new memory access pattern.

7.4.1.1 Oblivious Shuffle Strategy

We use the following method to determine where and when to shuffle. For the reasons described in Section 7.3, we assume a strong attacker who can distinguish whether an access is a regular DNN-based request vs. a Melbourne shuffle request. Note that this assumption only strengthens the attacker and therefore designs in this threat model yield *more secure* strategies. In our explanation below, we express the timescale in terms of the clock cycles in the *original memory access pattern* (such as in Figure 7.1) and ignore the clock cycles that are spent on Melbourne shuffle as if it is done instantly. *In the rest of this paper, all the mentions of "memory accesses" are referred to those of the DNN model, NOT those of Melbourne shuffle.*

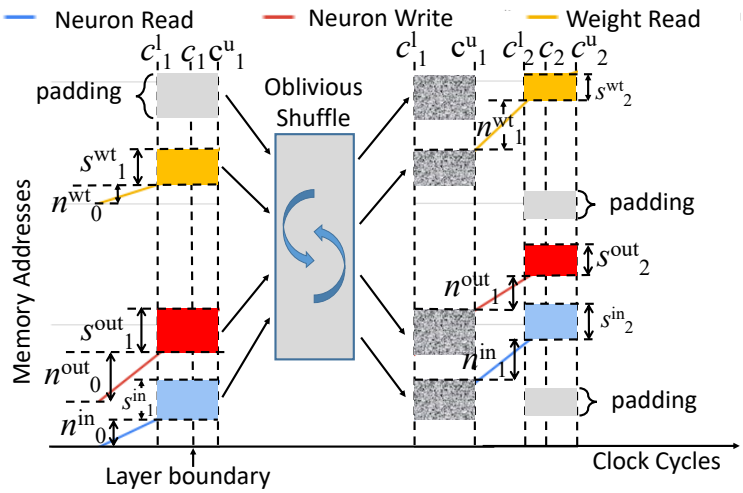


Figure 7.2: Illustration of oblivious shuffle in the memory access pattern of a DNN. The variables that are observable to the attacker are also illustrated.

Step 1: determining the shuffle budget. In order to control the memory access overhead of the Melbourne shuffle, we shuffle at most 2^{b_s} addresses in each shuffle. b_s is called the **shuffle budget**.

In order to obfuscate c_i, *all the memory addresses that are accessed within a certain range of clock cycles containing* c_i *are obfuscated using Melbourne shuffle.* To this end, we determine the above-mentioned clock cycle range with an upper bound and a lower bound of c_i, denoted as c_i^l and c_i^u, respectively. We take $\hat{c}_i^l \sim U(c_i - 2^{b_s}, c_i)$ and $c_i^l = \lceil \hat{c}_i^l \rceil$ where U stands for the uniform distribution (assuming there is at most one memory access per clock cycle). Let $c_i^u = \lfloor c_i^l + 2^{b_s} \rfloor$. This makes sure that the total # memory address accessed from c_i^l to c_i^u does not exceed 2^{b_s}.

All the memory addresses that are accessed within $[c_i^l, c_i^u]$ are to be shuffled. We illustrate this in Figure 7.2 to which we encourage the readers to refer as the explanation proceeds. We call the set of shuffled memory addresses at c_i^l the **shuffled regime** i, which include s_i^{in} input neurons addresses, s_i^{out} output neurons addresses,

s_i^{wt} weight addresses, and a set of additional addresses.These additional addresses are randomly chosen and can be any address in the memory space allocated for the DNN, as shown in Figure 7.2. This padding action will also enable us to add dummy accesses in the shuffled regimes to improve security as will be shown later.

Step 3: choosing the permutation π. Although the Melbourne shuffle will not leak information about π, π should 'look' random enough in order to obfuscate the memory accesses within the shuffled regimes. In addition, π must be easy to compute, otherwise we would spend too much time computing π. To meet these requirements, we use the *Feistel-based format-preserving encryption* [7] algorithm for π.

7.4.1.2 Information Leakage

We model the attacker's *best-case* knowledge in order to analyze the *worst-case* security guarantees. As noted before, we assume a strong attacker who can tell the difference between a Melbourne shuffle access and a regular DNN access. Therefore, c_i^l and the addresses within the shuffled regime i are known to the attacker. She can also infer c_i^u since after c_i^u, the memory accesses of the DNN will come out of the shuffled regime. As described in Section 7.2, by default, the neurons are stored in consecutive addresses and the weights too. In line with the *best-for-the-attacker* principle, we assume that each kind of memory access (*i.e.,* neuron read, neuron

write, or weight read) within each shuffled regime is of consecutive addresses. In this case, by calculating the difference of accessed addresses before c_i^l and after c_i^u, the attacker is able to infer s_i^{in}, s_i^{out}, and s_i^{wt}.

The memory access pattern outside the shuffled regimes are directly visible to the attacker. Let n_i^{in}, n_i^{wt}, and n_i^{out} denote the # read neurons, # read weights, and # written neurons, respectively, between clock cycle $c_i^u + 1$ and $c_{i+1}^l - 1$ (essentially the region between two adjacent shuffled regimes, note that in this region layer i is active) as illustrated in Figure 7.2. Let $\hat{s}_i^{in}$ and $\hat{s}_i^{wt}$ be the # shuffled input neurons and weights in the shuffled regime i, respectively, that are accessed *before* c_i (*i.e.,* belong to layer $i-1$). In order to find all feasible structures of layer i, the attacker needs to enumerate all the possible (integer) combinations of $c_i \in [c_i^l, c_i^u]$ and $c_{i+1} \in [c_{i+1}^l, c_{i+1}^u]$ (since she does not know exactly where the c_i, c_{i+1} lie within these boundaries). Let $r_{t_1}^{t_2}$ and $w_{t_1}^{t_2}$ be the # read and written addresses, respectively, within $[t_1, t_2]$. The following equations hold:

$$r_{c_i}^{c_{i+1}} = z_i^{in} + z_i^f \tag{7.7}$$

$$w_{c_i}^{c_{i+1}} = z_i^{out} \tag{7.8}$$

$$\hat{s}_{i+1}^{in} = z_i^{in} - n_i^{in} - (s_i^{in} - \hat{s}_i^{in}) \geq 0 \tag{7.9}$$

$$\hat{s}_{i+1}^{wt} = z_i^f - n_i^{wt} - (s_i^{wt} - \hat{s}_i^{wt}) \geq 0 \tag{7.10}$$

Equation (7.7) states that the total # read addresses between the layer boundaries is equal to the summation of the IFM size and the filter size. Similarly, (7.8) is based on the fact that the OFM is the only thing that is written back to the

memory. Equations (7.9) and (7.10) are because the read accesses of each layer consist of 3 parts: (a) those in the previous shuffled regime (i), (b) those that are not shuffled, and (c) those in the current shuffled regime ($i+1$), and those in (c) must be non-negative.

Equations (7.7) through (7.10) gives the possible combinations of z_i^{out} and z_i^f. Each possible combination is plugged in the IFP problem in Equations (7.1) through (7.6). The more possible combinations of z_i^{out} and z_i^f, the more IFPs to be solved and hence the greater attack complexity. In order to increase the # possible combinations and hence attack complexity, we propose to use ASLR and DumMA to relax the constraints on z_i^{out} and z_i^f imposed by Equations (7.7) through (7.10).

7.4.2 Address Space Layout Randomization

ASLR was initially proposed to counter the buffer overflow attack [31]. *In our work, ASLR is only done once at compile time to randomize the entire memory space.* Each address is permuted using a permutation function π_{init} which maps an address addr to be initially stored in address $\pi_{init}(\mathsf{addr})$. We use the same type of algorithm for π_{init} as the π for the Melbourne shuffle. In this way, the access pattern even outside the shuffled regimes will look random and the continuity of the address space is broken. Therefore, the attacker is not able to infer n_i^{in}, n_i^{out}, or n_i^{wt}. As a result, Equations (7.9) and (7.10), which require these variables, are not applicable any more. In this way, the constraints on z_i^{out} and z_i^f are reduced to only Equations (7.7) and (7.8). This will result in more possible combinations of

z_i^{out} and z_i^f and hence force the attack to solve more IFPs. Note that ASLR does not increase the number of memory accesses at run time since it works only as a mapping from the requested address to the actual address. Also note that ASLR just by itself does not mitigate the RAW type attack and needs to be combined with oblivious shuffle.

7.4.3 Dummy Memory Accesses

In this technique, we add dummy memory access within the shuffled regimes. When dummy memory accesses (DumMA) exist in the shuffled regimes, the attacker cannot tell a real access from a dummy one. However, she still gets an *upper bound* of # real read/write addresses since they must not exceed the total # read/write addresses (*i.e.,* real+dummy). One question is how many dummy accesses should be added. This is answered as follows. Since repeated accesses to *the same address of the same type* are observable, these accesses will not increase the upper bound of # real read/write addresses and do not improve the level of obfuscation. Therefore, there is no need to add more dummy accesses when every address in the shuffled regime is both read and written once.

7.4.3.1 DumMA Without ASLR

In this case, the attacker is able to infer each type of shuffled addresses: s_i^{in}, s_i^{out}, and s_i^{wt} because they only rely on c_i^l, c_i^u. For this reason, Equations (7.9) and (7.10) still hold. However, Equations (7.7) and (7.8) need to be changed to

reflect the "upper bound" effect caused by DumMA: the # of reads and writes between the layer boundaries are the upper bounds of $z_i^{in} + z_i^f$ and z_i^{out}, respectively (since many of these accesses are dummies).

$$r_{c_i}^{c_{i+1}} \geq z_i^{in} + z_i^f \tag{7.11}$$

$$w_{c_i}^{c_{i+1}} \geq z_i^{out} \tag{7.12}$$

Compared to Equations (7.7) and (7.8), (7.11) and (7.12) change equalities to inequalities (with the equal sign), and thus increasing the possible combinations of z_i^{out} and z_i^f.

7.4.3.2 DumMA With ASLR

Due to ASLR, Equations (7.9) or (7.10) does not hold any more for the same reason as described in Sec. 7.4.2. z_i^{out} and z_i^f are hence only constrained by Equations (7.11) and (7.12).

7.4.4 Summary of Defense Techniques

Three techniques have been introduced in the formulation of our defense: oblivious shuffle (OS), address space layout randomization (ASLR), and dummy memory accesses (DumMA). Each OS obfuscates the accessed memory addresses within a certain range of clock cycles containing a layer boundary. The following information remains leaked to the attacker: (i) the nature of each memory access outside the shuffled regimes, (ii) the # shuffled input neurons s_i^{in}, output neurons s_i^{out}, and weights s_i^{wt} in each shuffled regime, and (iii) the (actual) # read and

Table 7.3: Information leaked under various combinations of defense techniques

Techniques	Availability to the attacker		Equations
	s_i^{in}, s_i^{out}, s_i^{wt}, n_i^{in}, n_i^{out}, n_i^{wt}	$r_{c_{i-1}}^{c_i}$, $w_{c_{i-1}}^{c_i}$	
OS	Yes	Exact	(7.7) ∼ (7.10)
OS + ASLR	No	Exact	(7.7), (7.8)
OS + DumMA	Yes	Upper bound	(7.9) ∼ (7.12)
All the above	No	Upper bound	(7.11), (7.12)

written addresses of the DNN model. (i) and (ii) are obfuscated by ASLR and (iii) by DumMA. The information leakage under four cases is summarized in Table 7.3: OS only, OS + ASLR, OS + DumMA, and OS + ASLR + DumMA.

7.4.5 Attacking the Proposed Defense

As mentioned earlier, we assume that the attacker knows which defense techniques are in place and is able to attack accordingly. The new attack of layer i is shown using the procedure shown in Algorithm 7.2.

Algorithm 7.2: Procedure to reverse engineer layer i in the new attack

```
for (c_i, c_{i+1}) ∈ [c_i^l, c_i^u] × [c_{i+1}^l, c_{i+1}^u] do
    for Each feasible structure of layer i − 1 ending at c_i − 1 do
        Find all the possible combinations of z_o and z_f according to the
         equations summarized in Table 7.3;
        for Each possible (z_o, z_f) pair do
            Solve the IFP defined by Equations (7.1) through (7.6).
             Concatenate all the found feasible structures of layer i to the
             currently used structure of layer i − 1.
        end
    end
end
```

When Algorithm 7.2 finishes for the last layer, all the feasible structures of the DNN will be obtained.

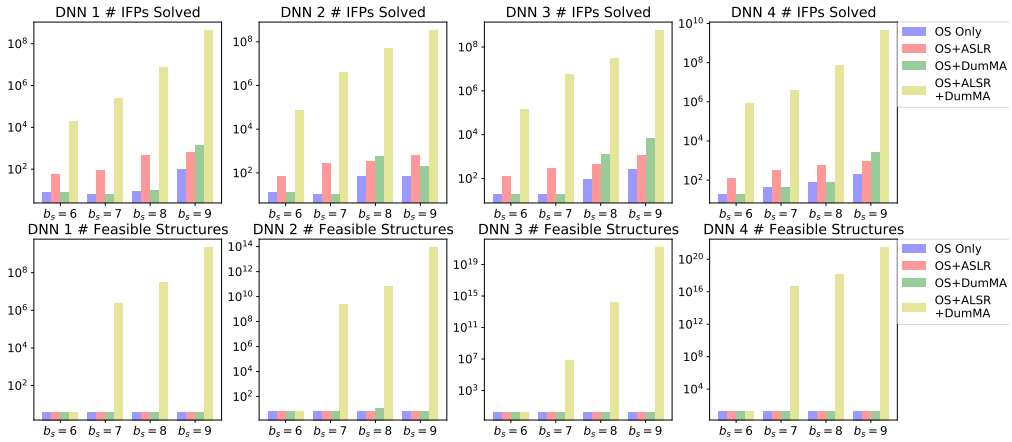


Figure 7.3: # IFPs solved and # feasible structures of the DNN benchmarks under various defense techniques

7.5 Experiments and Results

In this section, we evaluate the effectiveness of our proposed defense strategy using the complexity of the *modified* attack and measure the memory access overhead. The attack complexity is evaluated in the same way as described in Section 7.2.3, with the two metrics being the number of IFP problems (defined by Equations (7.1) $\sim$ (7.6)) (Phase 2) to be solved and the total number of feasible DNN structures that need to be trained and evaluated (Phase 3).

The shuffle budget b_s ranges from 6 to 9 in our experiments. Under each b_s, we generate the memory traces for each combination of defense techniques. When DumMA is used, we add dummy accesses into each shuffled regime such that each shuffled address is both read once and written once.

The two complexity metrics of the 4 benchmarks are reported in Figure 7.3. We observe that the combination of all three techniques yields the highest security level:

Table 7.4: The overhead of our proposed defense

Benchmark		$b_s = 6$	$b_s = 7$	$b_s = 8$	$b_s = 9$
DNN 1	Without DumMA	4.10%	10.89%	10.94%	33.63%
	With DumMA	4.36%	11.35%	12.07%	35.21%
DNN 2	Without DumMA	8.85%	25.20%	36.24%	86.66%
	With DumMA	9.31%	26.44%	37.95%	87.58%
DNN 3	Without DumMA	2.38%	3.72%	9.03%	16.42%
	With DumMA	2.49%	3.99%	9.41%	17.16%
DNN 4	Without DumMA	1.12%	1.83%	5.18%	13.97%
	With DumMA	1.76%	3.12%	6.79%	17.41%
Average	Without DumMA	4.11%	10.41%	15.35%	37.67%
	With DumMA	4.48%	11.23%	16.56%	39.34%

1. Both metrics are many orders of magnitude better than any other combination of techniques within the same benchmark and under the same shuffle budget.

2. For each benchmark, both metrics grow exponentially with the shuffle budget.

3. The # possible structures tend to grow exponentially as the DNN gets deeper.

The overheads of our proposed defense under each shuffle budget from 6 to 9 are listed in Table 7.4. As seen, very high attack complexity can be achieved at the cost of low access overheads. Moreover, the memory access overhead does not increase when the DNN gets larger, making our technique easily scalable to larger DNN models. This is because the overhead is roughly determined by the ratio of the size of each shuffled regime to that of each layer. This scalability is a key advantage of our approach over ORAM. ORAM requires an $\Omega(\log N)(\times 100\%)$ access overhead where N is the size of the memory (and must be least the size of the DNN), which means that the access overhead must increase as the DNN becomes larger. The effectiveness and scalability of our defense strategy make it practical to defend the reverse engineering attacks on DNNs. Note that the secure encryption algorithms (for which we use AES) and the permutation function π (which is a simple

transformation from AES) are not considered as significant sources of overheads because AES accelerators have been integrated into most processor architectures nowadays which allow very efficient computation of AES functions.

7.6 Summary

A novel defense strategy against the reverse engineering on DNNs is proposed in this chapter. The targeted attack model analyzes the memory access pattern of the processor running the DNN and solves an integer feasibility program to obtain all the possible structures of each layer. In our defense strategy, three techniques are utilized to obfuscate the memory access pattern, including oblivious shuffle, address space layout randomization, and dummy memory access. A modified attack based on the original attack is also formulated in order to evaluate the security of the proposed defense. Experiments show that, by combining all the three defense techniques, very high attack complexity can be achieved with low overheads. It is also shown that our defense approach easily scales to larger DNN models. Therefore, we conclude that the DNN reverse engineering attacks based on memory access patterns can be effectively countered using our proposed defense approach.

Chapter 8: Conclusion and Future Research Directions

In this dissertation, we studied the security vulnerabilities and opportunities in various emerging technologies. In Chapter 2, we explored the opportunities to enhance IC supply chain security brought by double patterning lithography (DPL). In Chapter 3, we formulated an optimization-theoretic attack against physical unclonable functions (PUF). From Chapter 4 to Chapter 7, we investigated multiple security issues in which neural networks were involved. In Chapter 4, we identified the security threats of neural Trojans, *i.e.,* hidden malicious functionalities in neural network IPs, and proposed effective countermeasures. In Chapter 5, we studied the ramifications of the inherent error resiliency of neural network-based applications on logic locking. Specifically, we proved the unavoidable trade-off between *security* and *effectiveness* among all logic locking schemes and found that, in the presence of such error resiliency, no existing logic locking scheme can achieve the two competing objectives at the same time. Hence we developed Strong Anti-SAT (SAS), a novel logic locking scheme that magnifies hardware module-level error to the application level while maintaining exponentially high SAT attack complexity, thus achieving the two goals simultaneously. Chapters 6 and 7 discussed side-channel-based neural network reverse engineering attacks and defenses, respectively. In Chapter 6, we

proposed GANRED, a novel cache side-channel attack to reverse engineer deep neural networks. Unlike existing attacks, GANRED does not need code access or shared maim memory content with the victim, but achieved more accurate results nevertheless. In Chapter 7, we consider an even stronger attacker model where s/he has access to the entire memory access record of the neural network. We proposed a countermeasure using oblivious shuffle, address space layout randomization, and dummy memory access. Experiments show that the countermeasure exponentially increases the number of feasible network structures found by the attack.

8.1 Future Work

There are many new research directions that can be extended from the security issues we have studied in this paper. We summarize these directions in the rest of this chapter.

8.1.1 Security Opportunities in 3D IC

3D IC is another emerging semiconductor technology which stacks multiple silicon dice together to form complete circuitry. Along with significant performance gains, this physical structure provides many unique opportunities that may benefit its security. For example, the top layer is a natural shield of probing or various side-channels for the lower layers. It also enables split manufacturing, where each layer is manufactured in a separate foundry. In this case, an approach to determine

how to split the circuitry onto different dice that resembles our proposal in Chapter 2 may be developed in order to minimize the information leakage about the entire design.

8.1.2 Architecture and Application Aware Logic Locking

In Chapter 5, we emphasized that a logic locking scheme must corrupt the application running on the protected hardware when a wrong key is applied. This is because, if locking has no application-level effects, a pirated/overbuilt copy of the chip will simply be as good as an authentic chip. Although there have been a myriad of logic locking techniques, most of them are focused on the module level. However, in order to derail an application from its correct behavior when a wrong key is present, the locking induced error must be propagated from the locked module to the processor's architecture and then to the application. Such propagation should be taken into consideration when deciding deciding the locking scheme and the location in the place to lock.

8.1.3 Hardware-Neural Network Co-Design for Security

Both the protection of neural models from hardware side-channel attacks and the hardware security challenges brought by neural network-based applications have been studied in this dissertation. In order to address all these issues simultaneously, a neural network-hardware co-design framework would be desirable. For example, let us consider the following scenario where the hardware designer collaborates with

the neural model owner to protect both the hardware and the neural model. The hardware designer implements a key-dependent operation in the hardware and the neural model owner trains the model with the knowledge of the key. Without the correct key, no unauthorized copy of the hardware can run the neural network correctly, nor can the end user decipher the neural model. Detailed protocols and implementations to this end can make interesting research topics.

Bibliography

[1] Accredited suppliers by the defense microelectronics activity.

[2] MT Arafin, Carson Dunbar, Gang Qu, N McDonald, and L Yan. A survey on memristor modeling and security applications. In *Quality Electronic Design (ISQED), 2015 16th International Symposium on*, pages 440–447. IEEE, 2015.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, and Elaine Shi. Optorama: Optimal oblivious RAM. *IACR Cryptology ePrint Archive*, 2018:892, 2018.

[4] Marco Barreno, Blaine Nelson, Anthony D Joseph, and JD Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[5] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. Csi neural network: Using side-channels to recover your artificial neural network information. *arXiv preprint arXiv:1810.09076*, 2018.

[6] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. {CSI}{NN}: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 515–532, 2019.

[7] Mihir Bellare, Phillip Rogaway, and Terence Spies. The ffx mode of operation for format-preserving encryption. *NIST submission*, 20, 2010.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[9] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

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

[11] Gedare Bloom, Bhagirath Narahari, and Rahul Simha. Fab forensics: Increasing trust in ic fabrication. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 99–105. IEEE, 2010.

[12] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[13] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[14] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[15] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

[16] 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.

[17] Abhishek Chakraborty, Yuntao Liu, and Ankur Srivastava. TimingSAT: timing profile embedded SAT attack. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018.

[18] Abhishek Chakraborty, Yang Xie, and Ankur Srivastava. Template attack based deobfuscation of integrated circuits. In *Computer Design (ICCD), 2017 IEEE International Conference on*, pages 41–44. IEEE, 2017.

[19] 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.

[20] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[21] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[22] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[23] Leon Chua. Resistance switching memories are memristors. *Applied Physics A*, 102(4):765–783, 2011.

[24] Leon O Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507–519, 1971.

[25] J Deng, A Berg, S Satheesh, H Su, A Khosla, and L Fei-Fei. Ilsvrc-2012, 2012. *URL http://www. image-net. org/challenges/LSVRC*, 2012.

[26] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Analysis of secure caches and timing-based side-channel attacks. *IACR Cryptology ePrint Archive*, 2019:167, 2019.

[27] Vasisht Duddu, Debasis Samanta, D Vijay Rao, and Valentina E Balas. Stealing neural networks via timing side channels. *arXiv preprint arXiv:1812.11720*, 2018.

[28] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[29] Amirali Ghofrani, Siddharth Gaba, Melika Payvand, Wei Lu, Luke Theogarajan, Kwang-Ting Cheng, et al. A low-power variation-aware adaptive write scheme for access-transistor-free memristive memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 12(1):3, 2015.

[30] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[31] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490, 2012.

[32] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[33] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[34] Michael T Goodrich and Michael Mitzenmacher. Anonymous card shuffling and its applications to parallel mixnets. In *International Colloquium on Automata, Languages, and Programming*, pages 549–560. Springer, 2012.

[35] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[36] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[37] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

[38] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 38–55. IEEE, 2016.

[39] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[40] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 170–180. Springer, 2002.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[42] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security analysis of deep neural networks operating in the presence of cache side-channel attacks. *arXiv preprint arXiv:1810.03487*, 2018.

[43] Weizhe Hua, Zhiru Zhang, and G Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, page 4. ACM, 2018.

[44] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

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

[46] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B Bhadviya, Pinaki Mazumder, and Wei Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, 10(4):1297–1301, 2010.

[47] Mika Juuti, Sebastian Szyller, Alexey Dmitrenko, Samuel Marchal, and N Asokan. Prada: Protecting against dnn model stealing attacks. *arXiv preprint arXiv:1805.02628*, 2018.

[48] Andrew B Kahng, Chul-Hong Park, Xu Xu, and Hailong Yao. Layout decomposition approaches for double patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):939–952, 2010.

[49] 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.

[50] James E Kelley, Jr. The cutting-plane method for solving convex programs. *Journal of the society for Industrial and Applied Mathematics*, 8(4):703–712, 1960.

[51] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[53] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[54] Daihyun Lim. Extracting secret keys from integrated circuits. 2004.

[55] Yuntao Liu, Chongxi Bao, Yang Xie, and Ankur Srivastava. Introducing tfue: The trusted foundry and untrusted employee model in ic supply chain security. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pages 1–4. IEEE, 2017.

[56] Yuntao Liu, Dana Dachman-Soled, and Ankur Srivastava. Mitigating reverse engineering attacks on deep neural networks. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 657–662. IEEE, 2019.

[57] Yuntao Liu, Ankit Mondal, Abhishek Chakraborty, Michael Zuzak, Nina Jacobsen, Daniel Xing, and Ankur Srivastava. A survey on neural trojans. In *Twenty-first International Symposium on Quality Electronic Design*, pages 33–39. IEEE, 2020.

[58] Yuntao Liu, Yang Xie, Chongxi Bao, and Ankur Srivastava. An optimization-theoretic approach for attacking physical unclonable functions. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 45. ACM, 2016.

[59] Yuntao Liu, Yang Xie, Chongxi Bao, and Ankur Srivastava. A combined optimization-theoretic and side-channel approach for attacking strong physical unclonable functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(1):73–81, 2017.

[60] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 45–48. IEEE, 2017.

[61] Yuntao Liu, Michael Zuzak, Yang Xie, Abhishek Chakraborty, and Ankur Srivastava. Strong anti-sat: Secure and effective logic locking. In *Twenty-first International Symposium on Quality Electronic Design*, pages 199–205. IEEE, 2020.

[62] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889*, 2018.

[63] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. Combined modeling and side channel attacks on strong pufs. *IACR Cryptology ePrint Archive*, 2013:632, 2013.

[64] Sonny Maynard. Trusted manufacturing of integrated circuits for the department of defense. In *National Defense Industrial Association Manufacturing Division Meeting*, 2010.

[65] Shike Mei and Xiaojin Zhu. Using machine teaching to identify optimal training-set attacks on machine learners. In *AAAI*, pages 2871–2877, 2015.

[66] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[67] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[68] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

[69] Xuan Thuy Ngo, Jean-Luc Danger, Sylvain Guilley, Tarik Graba, Yves Mathieu, Zakaria Najm, and Shivam Bhasin. Cryptographically secure shield for security ips protection. *IEEE Transactions on Computers*, 66(2):354–360, 2017.

[70] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[71] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming*, pages 556–567. Springer, 2014.

[72] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 619–636, 2016.

[73] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[74] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[75] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[76] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

[77] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 582–597. IEEE, 2016.

[78] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. *IACR Cryptology ePrint Archive*, 2018:373, 2018. to appear in FOCS 2018.

[79] Colin Percival. Cache missing for fun and profit, 2005.

[80] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual Cryptology Conference*, pages 502–519. Springer, 2010.

[81] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Logic encryption: A fault analysis perspective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 953–958. EDA Consortium, 2012.

[82] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of the 49th Annual Design Automation Conference*, pages 83–89. ACM, 2012.

[83] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *Computers, IEEE Transactions on*, 64(2):410–424, 2015.

[84] Martin Riedmiller and I Rprop. Rprop-description and implementation details. 1994.

[85] Garrett S Rose and Chauncey A Meade. Performance analysis of a memristive crossbar puf design. In *Proceedings of the 52nd Annual Design Automation Conference*, page 75. ACM, 2015.

[86] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018.

[87] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Epic: Ending piracy of integrated circuits. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 1069–1074. ACM, 2008.

[88] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 237–249. ACM, 2010.

[89] Ulrich Rührmair, Jan Solter, Frank Sehnke, Xiaolin Xu, Ali Mahmoud, Vera Stoyanova, Gideon Dror, Jurgen Schmidhuber, Wayne Burleson, and Srinivas Devadas. Puf modeling attacks on simulated and silicon data. *Information Forensics and Security, IEEE Transactions on*, 8(11):1876–1891, 2013.

[90] Ulrich Rührmair, Xiaolin Xu, Jan Sölter, Ahmed Mahmoud, Mehrdad Majzoobi, Farinaz Koushanfar, and Wayne Burleson. *Efficient Power and Timing Side Channels for Physical Unclonable Functions*. 2014.

[91] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[92] Abhrajit Sengupta, Mohammed Nabeel, Muhammad Yasin, and Ozgur Sinanoglu. Atpg-based cost-effective, secure logic locking. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2018.

[93] Bicky Shakya, Navid Asadizanjani, Domenic Forte, and Mark Tehranipoor. Chip editor: leveraging circuit edit for logic obfuscation and trusted fabrication. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 30. ACM, 2016.

[94] Kaveh Shamsi and et al. Appsat: Approximately deobfuscating integrated circuits. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, pages 95–100. IEEE, 2017.

[95] 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.

[96] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[97] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[98] 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.

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

[100] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):18, 2018.

[101] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

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

[103] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[104] Mark Tehranipoor, Domenic Forte, Garrett S Rose, and Swarup Bhunia. *Security Opportunities in Nano Devices and Emerging Technologies*. CRC Press, 2017.

[105] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27(1), 2010.

[106] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[107] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[108] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. *arXiv preprint arXiv:1701.04082*, 2017.

[109] Qian Wang, Mingze Gao, and Gang Qu. A machine learning attack resistant dual-mode puf. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 177–182, 2018.

[110] Qian Wang, Mingze Gao, and Gang Qu. Puf-passse: A puf based password strength enhancer for iot applications. In *20th International Symposium on Quality Electronic Design (ISQED)*, pages 198–203. IEEE, 2019.

[111] Qian Wang and Gang Qu. A silicon puf based entropy pump. *IEEE Transactions on Dependable and Secure Computing*, 16(3):402–414, 2018.

[112] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.

[113] Xingbin Wang, Rui Hou, Yifan Zhu, Jun Zhang, and Dan Meng. Npufort: a secure architecture of dnn accelerator against model inversion attack. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 190–196. ACM, 2019.

[114] Lingxiao Wei, Bo Luo, Yu Li, Yannan Liu, and Qiang Xu. I know what you see: Power side-channel attack on convolutional neural network accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 393–406. ACM, 2018.

[115] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

[116] Yang Xie, Chongxi Bao, Yuntao Liu, and Ankur Srivastava. 2.5 d/3d integration technologies for circuit obfuscation. In *Microprocessor and SOC Test and Verification (MTV), 2016 17th International Workshop on*, pages 39–44. IEEE, 2016.

[117] Yang Xie, Chongxi Bao, and Ankur Srivastava. Security-aware design flow for 2.5 d ic technology. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*, pages 31–38. ACM, 2015.

[118] Yang Xie, Chongxi Bao, and Ankur Srivastava. Security-aware 2.5 d integrated circuit design flow against hardware ip piracy. *Computer*, (5):62–71, 2017.

[119] Yang Xie and Ankur Srivastava. Mitigating sat attack on logic locking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 127–146. Springer, 2016.

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

[121] 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.

[122] Xiaolin Xu and Wayne Burleson. Hybrid side-channel/machine-learning attacks on pufs: a new threat? In *Proceedings of the conference on Design, Automation & Test in Europe*, page 349. European Design and Automation Association, 2014.

[123] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[124] Chaofei Yang, Beiye Liu, Hai Li, Yiran Chen, Wujie Wen, Mark Barnell, Qing Wu, and Jeyavijayan Rajendran. Security of neuromorphic computing: thwarting learning attacks using memristor's obsolescence effect. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 97. ACM, 2016.

[125] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. Generative poisoning attack method against neural networks. *arXiv preprint arXiv:1703.01340*, 2017.

[126] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[127] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

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

[129] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Ttlock: Tenacious and traceless logic locking. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 166–166. IEEE, 2017.

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

[131] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security analysis of anti-sat. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 342–347. IEEE, 2017.

[132] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016.

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

[134] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 548–560. ACM, 2017.

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