SECURING THE HARDWARE SYSTEM STACK: HARDWARE TO SOFTWARE LAYERS

by

Abhijitt Dhavlle
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Engineering

Committee:

_____ Dr. Sai Manoj PD, Dissertation Director

_____ Dr. Brian Mark, Committee Member

_____ Dr. Khaled Khasawneh, Committee Member

_____ Dr. Amlan Ganguly, Committee Member

_____ Dr. Monson Hayes, Department Chair

Date: _____ Summer Semester 2022
George Mason University
Fairfax, VA

Securing the Hardware System Stack

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Abhijitt Dhavlle
Master of Science
George Mason University, 2022
Bachelor of Engineering
Mumbai University, 2014

Director: Dr. Sai Manoj PD, Professor
Department of Electrical and Computer Engineering

Summer Semester 2022
George Mason University
Fairfax, VA

Dedication

I sincerely dedicate this thesis to my beloved Jagannath, Baladev, Subhadra and Srila Prabhupada. I dedicate the thesis to my loving parents, Additi and Avinash Dhavale, my brother Aniruddha, and my loving and supportive wife Mrunali; and to my kind and loving in-laws, Akshay, Sitarani Devi and Sudhakar Bhosale. Without their motivation and nurture it would have been impossible to keep treading in testing times. This cannot be complete without crediting my grandfather, Ambadas Dhavale, and grandmother, Kamal Dhavale. I also would like to thank my friends, Sanket, Gaurav, Manideep and many others for their constant support and encouragement. I cannot forget to express my deep gratitude to my mentors, Sandhini, Mahamantra, Narsimhalila , and Kanhaigopal for their loving care and best wishes. I want to deeply appreciate Rajeev and Vidula Saawant, Priya and Sharad Baagwe, Sachin Solase, my cousins, and all the relatives for their care and blessings. I extend my gratitude to everyone who has been instrumental in the success so far and much more to come.

Acknowledgments

I would like to thank the following special people who made this seemingly long journey possible. Dr. Sai Manoj PD, my Masters and PhD advisor for his kind support always in trying to make me a better version of myself. I have learned from him many qualities which are conducive to being an enthusiastic and sincere student. I would like to thank Dr. Jim Jones, Dr. Brian Mark, Dr. Khaled Khasawneh, Dr. Amlan Ganguly, Dr. Avesta Sasan, Dr. Aydin, Dr. Liling Huang, Dr. Houman Homayoun, Dr. Setareh Rafatirad, Ms. Jammie Chang, Ms. Patricia Sahs; Mr. Martin Voogel, Dr. Pierre Maillard and Dr. Paula Chen from AMD-Xilinx; Dr. Andrew Schmidt from USCISI, Ms. Anagha Malkapurkar from ACPCOE; Dr. Umesh Mhapankar, Mrs. Sonali Sherigar, Mrs. Raji MP from Fr. Agnels Polytechnic; and Mrs. Radhakrishnan from MSAS school, for being instrumental in their capacity as my advisors, professors, collaborators, trainers, teachers and much more.

Table of Contents

List of Tables

List of Figures

Listings

Abstract

SECURING THE HARDWARE SYSTEM STACK

Abhijitt Dhavlle, PhD

George Mason University, 2022

Dissertation Director: Dr. Sai Manoj PD

Computing systems have come a long way concerning speed, performance, optimization, and security. The state-of-the-art designs are deployed in the real world, targeting a variety of applications. The hardware security domain has experienced various attacks that proved a serious threat to computing systems. It is given that the underlying vulnerabilities in the system cannot be eradicated given the cost and design constraints. Attacks could differ based on the architectural component they target; Cache side-channel attacks, like, Flush+Reload, Prime+Probe, RowHammer, etc.; Malware-based attacks with reinforced evasion techniques; and Hardware Trojans that can camouflage and decipher sensitive information are all on the rise in the recent times. With the recent adaptation and pervasiveness of Machine Learning and Deep Learning techniques for improved performance and a better user experience, Hardware attacks have been improvised, too. This thesis focuses on development of hardware-assisted security defenses against malware, cache-targeted side-channels, and hardware Trojans.

Software-based malware detection have certain limitations such as performance overhead, requiring modification to software application, vulnerability to exploits, and so on;

to mitigate the limitations incurred by the traditional software-based malware detection techniques, Hardware-assisted Malware Detection (HMD) using machine learning (ML) classifiers has emerged as a panacea to detect malicious applications and secure the systems. To classify benign and malicious applications, HMD primarily relies on the generated low-level microarchitectural events captured through Hardware Performance Counters (HPCs). The dissertation discusses about an adversarial attack on the HMD systems to tamper the security by introducing the perturbations in the HPC traces with the aid of an adversarial sample generator application. We first deploy an adversarial sample predictor to predict the adversarial HPC pattern for a given application to be misclassified by the deployed ML classifier in the HMD. Further, as the attacker has no direct access to manipulate the HPCs generated during runtime, based on the output of the adversarial sample predictor, we devise an adversarial sample generator wrapped around a normal application to produce HPC patterns like the adversarial predictor HPC trace. With the proposed attack, malware detection accuracy has been reduced to 18.04% from 82.76%. To render the HMD robust against the attack, a hardening technique is proposed and evaluated. Hardening refers to the retraining of the HMD on adversarial samples to offer robustness against performance degradation; With hardening, the HMD performance is restored to 81%.

Many of the side-channel attacks target cache memories. To mitigate the attack, this thesis presents a random yet cognitive side-channel mitigation technique that is independent of the underlying architecture and/or operating system. In contrast to the existing randomization-based side-channel defenses, we introduce a cognitive perturbation-based defense, Covert-Enigma, where the introduced perturbations look legit but lead to an incorrect observation when interpreted by the attacker. To achieve this, the perturbations are injected at appropriate time instances to introduce additional operations, thereby misleading the attacker making the extracted data futile. To further make the attack more intricate for the attacker, the proposed Covert-Enigma offers two modes of operation, chosen by the user, to determine the kind of induced cognitive perturbations - arbitrary and cyclic modes. The cognitive perturbations are introduced in a wrapper application to the victim, thus

imposing no requirements on architectural level modifications nor soft updates/edits to the operating system. We report an evaluation of the proposed Covert-Enigma protecting RSA cryptosystem attacked by Flush+Reload crypto side-channel attack.

Offshore chip manufacturing adds a potential risk of hardware malware embedding. Interconnection networks for multi/many-core processors or server systems are the system's backbone as they enable data communication among the processing cores, caches, memory, and other peripherals. Given the criticality of the interconnects, the system can be severely subverted if the interconnection is compromised. Even by deploying naive hardware Trojans (HTs), an adversary can exploit the Network-on-Chip (NoC) backbone of the processor and get access to communication patterns in the system. This information can reveal important insights regarding the application suites running on the system, thereby compromising user privacy, and paving the way for more severe attacks on the entire system. In the dissertation, we demonstrate that one or more HTs embedded in the NoC of a multi/many-core processor can leak sensitive information regarding traffic patterns to an external malicious attacker, who, in turn, can analyze the HT payload data with machine learning techniques to infer the applications running on the processor. Furthermore, to protect against such attacks, we propose a Simulated Annealing-based randomized routing algorithm in the system. The proposed defense can obfuscate the attacker's data processing capabilities to infer the user profiles successfully. Our experimental results demonstrate that the proposed randomized routing algorithm could reduce the attacker's accuracy of identifying user profiles from $>$ 98% to $<$ 15% in multi/many-core systems.

Chapter 1: Introduction

1.1 Motivation

Computing systems since their origin have evolved and become pervasive in different fields of technology, extensively. The computing systems have evolved in regards to speed, performance, optimization, and security. In a computer architecture, evolution like the cache subsystem, branch prediction, faster memories have contributed to the speed and performance aspect; various algorithms towards the optimization aspect; while the security aspect was somewhat addressed in the software side of the system. Contributing researchers and industry designers have invested laborious efforts to maintain the pace of such an evolution given the avaricious demands of the state-of-the-art trends in technology. Yet, in recent decade, the computing system discipline has witnessed a diverse range of attacks, targeting both software and hardware, exploiting the underlying vulnerabilities in the systems [6–12].

Let us consider the stack as shown in Figure 1.1. The hardware stack comprises of all the physical components of the computer architecture, like, the branch predictor, memories, processor, cache, etc.; the software stack consists of the operating system or the kernel that collaborates with the hardware to support the application layer, where user-level software is executed. The software components, given certain vulnerabilities, has been susceptible to various exploits. But, since a decade, adversaries have used exploits to jeopardize computing systems by attacking the hardware [6–8, 13–20]. The vulnerabilities exist in the hardware inherently, and not created intentionally [6–8]. These attacks severely put our systems, and hence data, at increased risk of breach. It is, therefore, imperative that the hardware, in combination with the software, needs to be protected against such attacks. Hence, the hardware security domain has evolved as a panacea to protect against the attacks.

Research works, thus far, have focused on either protecting the software stack or the

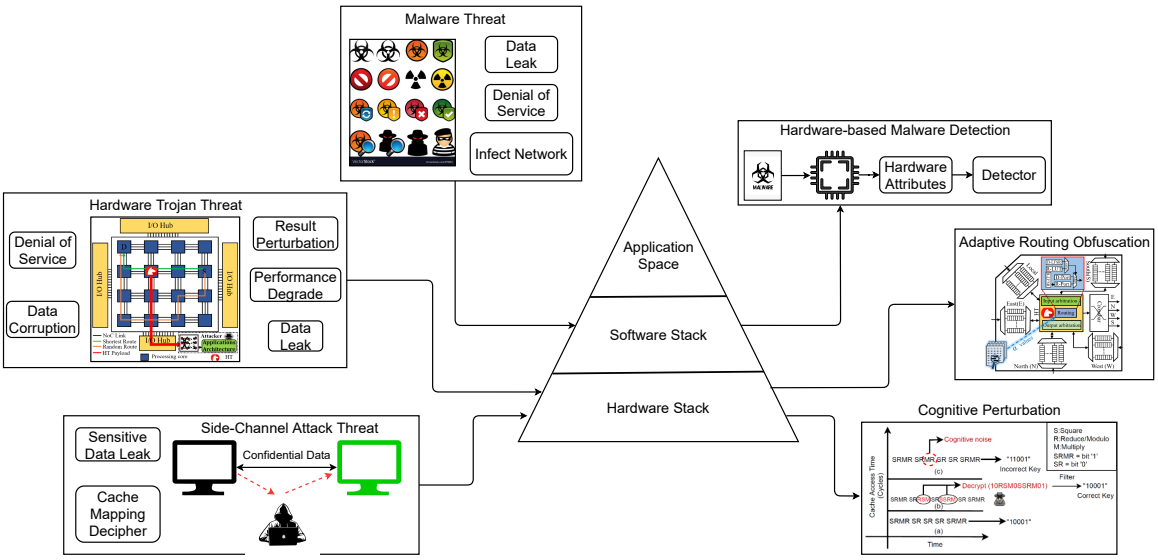


Figure 1.1: Hardware Software Stack Exposed to Vulnerabilities leading to various attacks

hardware stack. This has certain limitations. For example, a software mitigation in place could be overcome by one or multiple types of hardware exploits. An aggressive hardware mitigation, on the contrary, can lead to unreasonable overheads and performance drawbacks. To resolve this issue, solutions providing a golden balance - a mix of both the strategies - is mandated. The top threats that have proliferated in the recent decade are Malware, Side-Channel Attacks, and Hardware Trojan. Therefore, this thesis presents a comprehensive discussion on the three threats, their capability to disrupt the security barriers, and a proposed solution for each of the threats.

- **Malware Attack**: According to some Cybersecurity threat reports [21,22], it can be seen that the Malware threats have increased manifold over the years. Adversaries created not only novel malwares but also embedded sophisticated evasion techniques and enhanced disruption capabilities.

- **Side-Channel Attack**: Side-Channel attacks like Prime+Probe [23], Flush+Reload [7], and Flush+Flush [8] exploit the inherent vulnerability in the computing systems leading to data leaks. Prime+Probe is capable of deciphering the cache mapping of the victim process, causing data leaks; Flush+Reload works on the LLC cache and does not need to 'prime' the cache. It is capable of stealing the data by monitoring the cache access time of the flushed location. Flush+Flush supersedes the previous two by merely flushing the data, thus deciphering the location of the targeted information.

- **Hardware Trojan**: Offshore chip manufacturing led to the exposure of sensitive design information to adversaries. Hardware Trojans, embedded inside the chip, are capable of a multitude of attacks, like, denial-of-service, disrupting the functionality, corruption of sensitive information, targeted attack, etc. [13–20]. Some of the reports [24,25] of real-world hardware Trojan attacks have shown the disruptive nature of the threat.

With the above discussed threats and their features, it becomes necessary to propose mitigation strategies against such threats. Moreover, the performance overhead of such

mitigation must be within limits. In this thesis, I will be presenting my research works accomplished to address some of the exploits in software and hardware stack. I first present my research on malware detection, followed by side-channel attacks, and hardware Trojans. Ongoing research on topics like data augmentation for hardware Trojan, side-channel attack, RGRA-based CNN on FPGA, and attack/defense for neuromorphic DNN network is presented in the future works section. A brief introduction to malware detection, side-channel attacks, and hardware Trojan is presented next along with their contributions.

1.2 Contributions

Software-based Malware detection has been a popular conventional method. But, due to certain limitations, Machine Learning based detection that use hardware attributes evolved. The work proposes an adversarial attack on HMDs (Hardware-based Malware Detectors) in which the adversarial samples are generated through a benign code that is wrapped around a benign or malware application to produce a desired output class from the embedded ML-based malware detector. One of the main challenges to address is that the attacker or user has no direct access to modify the HPC and furthermore, manipulation of HPCs is highly complex to perform despite employing techniques like code obfuscation for executing malware [26, 27].

Firstly, we assume the victim's defense system to be a blackbox and perform reverse engineering to mimic the behavior of the embedded HMD or other security system and build a ML classifier. In order to determine the required number of HPCs to be generated through the application to be misclassified, we employ an '*adversarial sample predictor*' which predicts the number of HPCs to be generated to misclassify an application by the HMD. As aforementioned, the HPCs cannot be modified directly by the attacker, as such we craft an '*adversarial HPC generator*' application (code) that generates the required number of HPCs. The adversarial HPC generator application is wrapped around the application that needs to be misclassified. The main focus of this work is to create false alarms (malware classified as benign and benign classified as malware) in order to weaken the trust on the

embedded defenses, which increases the scope for attacks.

The proposed work benefits from the following:

- No need to tamper or modify the source code of the application around which the proposed adversarial sample generator code will be wrapped (i.e., executed in parallel).
- The crafted application has no malicious features embedded, thus not detectable by ML malware detectors.
- Scalable and flexible i.e., the crafted application can generate events as required to generate powerful adversary.

We previously discussed that the hardware stack is inherently vulnerable. Side-channel attack is a pivotal issue that functions by exploiting the vulnerabilities in the stack. Hence, this thesis presents a mitigation approach for cache-based side-channel attacks. In Covert-Enigma the victim application is coupled with a shield application that induces *cognitive* perturbations in the cache-access information. In Covert-Enigma, we induce cognitive perturbations in the victim's cache operations by executing dummy instructions that leave the victim's functionality unaltered yet scrambling the sequence observed by the attacker. As they are crafted to look legit when interpreted, we call them 'cognitive perturbations' in this work. The perturbations that cause scrambling of the observed leakage cannot be a simple random access or an arbitrary cache access operation as they would be trivial for the adversary to filter out as they do not translate to meaningful information when decrypted.

Our proposed Covert-Enigma tenders user-tunable parameters to offer the user with flexibility to fine-tune the level of complexity of the injected perturbations. Arbitrary and Cyclic are two operational modes that a user can select for its victim application. The *arbitrary* mode perturbs one or more bits cognitively in the sequence of operations chosen at runtime, whereas the *cyclic* mode selects bit(s) and then keeps perturbing[1] the same bits for a few execution cycles as determined by the user, post which the bit position changes.

The cardinal contributions of this work are:

[1]Perturbation or cognitive calls refer to dummy cache accesses that leads to meaningful decryption, yet incorrect

- Contrary to the existing works, the proposed Covert-Enigma enforces security on the covert channel by injecting cognitively crafted perturbations that imitate legit operations yet mislead the attacker.

- Render the attack more laborious by providing two modes of operation, Arbitrary and Cyclic, thus offering more flexibility in terms of the defense.

- Evaluate and compare the benefits of the proposed Covert-Enigma in terms of overhead and performance based on the key size, mode of operation, user-tunable parameters, and the number of bits recovered post attack on the victim.

The last chapter in this thesis discusses machine learning-assisted hardware-based malware attack and its proposed countermeasure. The Trojan is embedded in the NoC (Network-on-chip), a hardware component, responsible for carrying data traffic between various other hardware blocks. Considering the critical role played by the NoCs, embedding a HT that exploits the interconnection backbone can reveal the communication patterns in the system. This information when leaked to a malicious attacker can reveal important information regarding the application suites running on the system, thereby compromising the user profile. This information in turn, can enable further more severe attacks not just on the multi/many-core processor infected with the HT, but on the systems on which they are deployed. For instance, an adversary obtaining secure military information through a HT deployed in a router can subvert the military backbone, thus leading to a compromise of the national security [28].

In this regard, we first introduce a lightweight NoC-based HT, which, in its simplistic form, is a simple counter, which, when inserted in one or a few switches of the NoC can count the number of packets traversing the specific switches over a time window. The HT can then periodically, packetizes this count and send it to an external attacker program for payload analysis, severely compromising user profile confidentiality. This packetized count, which is the HT payload, can be subsequently analyzed by the external attacker using data processing techniques to infer the applications running on the system. To analyze the

retrieved information, the attacker trains a sophisticated Machine Learning (ML) algorithm, that can create training samples and maps packet traversal frequencies at specific switches to the application suites. We are able to demonstrate that the application suites running in the system can be detected with only 4 or 8 counter-based HTs with more than 98% accuracy using ML techniques. This is possible because specific routing protocols are proposed for these particular system configurations, which when adopted result in application-specific traffic patterns. Therefore, observing the traffic patterns with the help of the HTs can enable inferring the application(s) being executed in the system.

In order to defend against such a HT, we propose a novel Simulated Annealing (SA)-based randomized routing algorithm for the NoC which can obfuscate the HT-based attack discussed above. SA is a type of genetic algorithm that allows sub-optimal traversal of the search space for optimization to avoid being stuck in local optima [29]. Random packet routing over the interconnection can severely degrade performance of the system due to packets not being routed over shortest paths. Therefore, instead of simply adopting random routing, we propose a parameterized SA-based approach that can be tuned to achieve a desired trade-off between the defense against the attack and loss in performance. Due to SA-based random routing, the path for each packet is unpredictable and therefore, makes the mapping of packet traversal frequency through specific switches and corresponding applications unreliable. We demonstrate through cycle-accurate simulations that this SA-based randomized routing can reduce the effectiveness of the attack. *To the best of our work knowledge, this work is a first of its kind where it is shown that by monitoring traffic patterns in a NoC through HTs the user profile can be compromised; and defended the system against such an attack with controlled random routing.*

1.3 Flow of the Dissertation Proposal

The flow of the proposal is as follows: Chapter 2 discusses all the previous works related to the contributions made in this proposal; Chapter 3 discusses the Hardware-based Malware Detection, followed by Chapter 4 contributed to the proposed mitigation of Side-Channel

Attacks; Chapter 5 presents an attack and defense for Network-on-chip (NOC) embedded Hardware Trojan. Finally, the conclusion and future works are discussed in Chapter 6.

Chapter 2: Related Works

The chapter is dedicated to discussing the state-of-the-art works related to the research projects discussed in this proposal. First, I will discuss the related works for hardware-based malware detection, followed by the related works for side-channel attack, and, finally, the hardware Trojan attack and defense for NOCs, followed by the limitation of the previous works. A detailed survey on Hardware Trojan is included at the end of this chapter.

2.1 Hardware-based Malware Detection

This section discusses past works that have exploited HPCs for detecting application anomaly and differentiate malware against benign. Work in [27] was the first of its kind to use HPCs with ML-models for malware detection, but it does not mention runtime detection of malware with a limited number of features for resource-constrained systems. In [30, 31], authors have exploited information from HPCs to detect anomalies in the system attacked by ROP (return-oriented programming) and buffer overflow attacks by employing unsupervised learning which can be effective but requires complex hardware and software implementation with their overheads also. Work in [32] implements a single ML-classifier and demonstrated results to support their claim in detecting malware with high accuracy and performance, but it required 8 and higher HPCs to achieve the results, which is not feasible on small systems with fewer resources. Our previous work in [33] also includes analysis with 8 HPCs but only to show that higher HPCs may result in higher accuracy for some classes of malware, but runtime detection becomes difficult with 8 or higher HPCs in systems with less than 8 HPCs. Hence, we have also included results with 4 HPCs only and boosted the accuracy with ensemble learning.

Panorama [34] uses an emulated environment and monitors hardware resources utilized

by the malware application by building a control flow graph. For detection to successfully happen, the proposed technique has to monitor applications within the emulation environment. The drawback of such a technique is that the attacker can craft malware that modifies the control graph such that it simulates the behavior of a benign application. Also, the malware can remain 'silent' until one cycle of detection is in progress, which means that the proposed method is not sufficiently resilient to the malware. In our work, irrespective of whether the malware is silent or not, as long as it executes at least once, HMD-Hardener will be able to detect it as our detection approach is based on capturing HPCs which no application -malware or benign- can evade. Work in [35] is of interest to us as it proposes to craft malware that does not alter any component of the operating system, thus thwarting any signature, behavior, or integrity checking defense mechanism or anti-virus program. Rather it relies on hardware state modifications to perform the attack. The paper vividly mentions that a defense mechanism that supports hardware integrity checks can successfully thwart the proposed attack. This drives us to the conclusion that using HPCs in HMDs will surely dissuade even such kinds of malware as 'Cloaker' [35].

Authors in [36] have proposed to use data obtained from HPCs to construct Singular Value Decomposition (SVD) matrix and then using different ML-classifiers to detect malign/benign. The paper clearly mentions that using hardware events is beneficial than software-based detection approaches as malware cannot compromise hardware events. But the work lacks to test the resiliency of the proposed methodology under attack by malware that can generate fake hardware events, thus derailing the entire foundation of the defense. Our work addresses this issue where we (as previously described) assume the role of an attacker, craft a next-generation malware and successfully profess that such malware can degrade the performance of HMDs. To provide a panacea to such next-generation malware attacks, we have also described in this work and included results for next-generation (aka adversarial) malware hardened HMD.

Work in [37] has exploited HPC data with different machine learning classifiers to propose ML-based HMD. The work describes the steps involved in building the ML-based

HMD and has included results of classification based on the number of HPCs used for classification.

2.2 Side-Channel Attacks

In order to secure the hardware systems against cache-side channel attacks, various defense techniques have been proposed that use different strategies. To address the challenges of cache-targeted SCAs, techniques such as static cache partitioning [2, 38], partition locked cache [1], non-monopolizable (nomo) cache architectures [39] and other defenses [2,38,40,41] are proposed. These techniques can tremendously reduce the interference between the attacker and victim's memory access, thereby providing a better defense. However, adopting such techniques require alterations in the cache design which may not be feasible [2]. To overcome such limitations, techniques such as cache-partitioning, randomization of cache architectures are introduced. Conventional fully associative cache is one of the preliminary randomization based cache, where a memory line can be mapped to any of the existing cache lines. Similarly, any of the cache lines can be evicted in random, thus, preventing the leakage of cache-access information. Despite its security benefits, this technique incurs large delays and is power hungry [2]. In a similar way, random permutation cache [1], newcache [4, 42], random fill cache [3], and random eviction cache [2] strategies are implemented. Compared to the cache-partitioning, the randomization based solutions have shown higher robustness, yet the above mentioned methods require modifications to the hardware and/or software and incurs performance penalties. We discuss the most relevant and prominent ones in this section.

2.2.1 Cache Partitioning Based Defenses

These defenses are based on eliminating the cache interference between the running processes. This way, running processes cannot snoop on each others' cache activity. He et al., [2] proposed to protect sensitive cache access (e.g., coming from sensitive data/operation) by reserving dedicated cache sets for those sensitive accesses. Thus, the sensitive cache access

will always index to the dedicated sets and all other cache access, including cache access from other running processes or threads will index to the rest of the cache sets. As the mapping from memory to a cache set involves the physical memory address, the proposed solution utilizes the operating system to organize physical memory into non-overlapping cache set groups, also called colors, and to enforce isolation policy on these groups. However, this approach leads to inefficient resource utilization and hardware overheads. Vladimir Kiriansky proposed dynamically allocated way guard (DAWG) [43], a generic mechanism for secure way partitioning of set associative structures including memory caches. DAWG endows a set associative structure with a notion of protection domains to provide strong isolation. When applied to a cache, unlike existing quality of service mechanisms such as Intel's Cache Allocation Technology (CAT), DAWG fully isolates hits, misses, and metadata updates across protection domains. DAWG enforces isolation of exclusive protection domains among cache tags and replacement metadata, as long as: 1) victim selection is restricted to the ways allocated to the protection domain (an invariant maintained by system software), and 2) metadata updates as a result of an access in one domain do not affect victim selection in another domain (are requirement on DAWG's cache replacement policy). DAWG protects against attacks that rely on a cache state-based channel, which are commonly referred to as cache-timing attacks, on speculative execution processors with reasonable overheads. The same policies can be applied to any set associative structure, e.g., TLB or branch history tables. DAWG requires additional techniques to block exfiltration channels different from the cache channel. Nonetheless, cache partitioning based defenses lead to hardware as well as performance overhead.

SGX Enclave Protection: Furthermore, Oleksenko et al. proposed *Varys* [44], a system that protects unmodified programs running in SGX enclaves from cache timing and page table SCAs. The *Varys* takes a pragmatic approach of strict reservation of physical cores to security-sensitive threads, thereby preventing the attacker from accessing shared CPU resources during enclave execution. This execution environment ensures that neither time-sliced nor concurrent cache timing attacks can succeed. Due to the lack of appropriate

hardware support in today's SGX hardware, Varys remains vulnerable to timing attacks on Last Level Cache (LLC). The paper also proposes a set of minor hardware extensions that hold the potential to extend Varys' security guarantees to L3 cache and further improve its performance. But the downside is it requires the application to monitor the SSA (SGX State Save Area) value, thus increasing the overhead and it introduces a window of vulnerability.

3D Integration: Chongxi Bao et al. in [45] show that 3D integration also offers inherent security benefits and enables many new defense mechanisms that would not be practical in 2D. The work is compatible with the ongoing trend of transition from 2D to 3D and enables designers to take security into account when designing future cache using 3D integration technology. Experimental results show that using their cache design, the side-channel leakage is significantly reduced while still achieving performance gains over a conventional 2D system.

Intel Cache Allocation Technology (CAT): Xiaowan Dong et al. present in [46] defenses against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel. They prototyped the solution in a system call Apparition, building on an optimized version of Virtual Ghost. To thwart LLC side-channel attacks, it leverage Intel's CAT in concert with techniques that prevent physical memory sharing. Apparition's control over privileged hardware state can partition the LLC to defeat cache side-channel attacks. Their defense combines Intel's CAT feature (which cannot securely partition the cache by itself) with existing memory protections from Virtual Ghost to prevent applications from sharing cache lines with other applications or the OS kernel. Similarly, authors in the paper [47] propose to utilize CAT (cache allocation technology) in Intel processors to provide a system-level protection to defend against SCAs on shared LLC. CAT is a way-based h/w cache-partitioning mechanism for enforcing quality to LLC occupancy. 'CATalyst' uses CAT to partition the LLC securely into a hybrid hardware-software managed cache to defend against SCAs.

2.2.2 Randomization Based Defenses

To overcome limitations of hardware oriented approaches, randomizing the memory access is introduced in [1], thus, making the attack much harder, even impossible. For instance, [2] uses random memory-to-cache mappings. There is a permutation table for each process, which enables a dynamic memory address to cache set mappings. This makes the attacker hard to evict a specific memory line of the victim process. However, maintaining the mapping and updating mapping tables penalizes performance. It can also use software based compiler assisted approach to transform applications to randomize its memory access patterns.

Control Flow Randomization: Stephen Crane et al. in [5] explore software diversity as a defense against side-channel attacks by dynamically and systematically randomizing the control flow of programs. Existing software diversity techniques transform each program trace identically. This diversity based technique instead transforms programs to make each program trace unique. This approach offers probabilistic protection against both online and off-line side-channel attacks. It extends previous, mostly static software diversification approaches by dynamically randomizing the control flow of the program while it is running. Rather than statically executing a single variant each time a program unit is executed, they created a program consisting of replicated code fragments with randomized control flow to switch between alternative code replicas at runtime.

2.2.3 Detection Based Defenses

Computational Anomaly Detection: Work in [48] give an overview of the attacks on hardware, including the SCAs, and describes the panacea to thwart such attacks and secure the hardware. Sanket et al. in [49, 50] have proposed a unique methodology in detecting even stealthy malwares with hardware performance counters and image processing along with RNN-based ML to assist the detection process.

SCA Detection in the Cloud: Zhang et al. in [51] present an architecture where cores (processors) equipped with specialized signature detection techniques are employed to detect

SCAs based on the hardware performance counters (HPCs) these attacks generate in a system. Taesoo Kim et al. present in [52] a system-level protection mechanism against cache-based SCAs in the cloud named as 'STEALTHMEM'. This mechanism protects cache from unauthorized access by managing a set of locked cache lines per core that are never evicted from the cache. Thus, any virtual machine (VM) can hide its sensitive information from others. Work in [53] presents 'StopWatch' a system that defend against SCAs in cloud environment by triplicating each cloud-resident VM and using the timing of the I/O events at the replicas to determine the timings observed by each replica or the attacker. Shi et al. in their work in [54] propose a technique, they name as dynamic cache coloring, which notifies the VM when an application is executing secure-sensitive operations to swap the associated data to a safe an isolated cache line to protect the same against SCA attack by limiting its access. They presented the technique for multi-tenant based cloud environment.

2.3 Hardware Trojan Attack Related Works

Several researches have explored the design of HTs in NoC components like switches, links, and network interfaces (NIs) that can snoop or tamper the data in a NoC to launch a Denial of Service (DoS), data snooping, and performance degradation attacks. Although, such HTs pose major security threats to modern multi/many processor System-on-Chips (MPSoCs); post silicon detection of such HTs through physical inspection [55], functional testing[56], and/or side channel analysis [57] are challenging due to the increasing complex design and long manufacturing chains of the MPSoCs. Thus, most of the existing research focuses on designing solutions for mitigating the effect of these HTs. In [58] authors proposed a bit shuffling based encoding mechanism inside the NoC switches to counter HTs that mislead packets away from the destination cores to manifest a performance degradation attack. As a safegaurd against snooping attack to leak sensitive information by the HTs implanted in the NIs, authors in [59] also proposed encoding modules using algebraic manipulation detection (AMD) and cyclic redundancy check (CRC) codes. In [60], authors proposed the

design of an HT implanted in the NIs that can duplicate packets and launch a DoS attack. To detect and mitigate such attacks at runtime, authors proposed a multi-layer approach utilizing an encoding based snooping invalidation module (SIM) for duplication packet detection, a threshold voltage degradation based low power data snooping detection circuit called THANOS, and malicious application blacklisting mechanism. However, these encoding based solutions require additional hardware; which not only increases the communication overhead due to the existence of these encoding-decoding modules in the flit datapath but also complicates the design of NoC. Consequently, authors in [61] proposed a trojan detection mechanism and a trojan-aware routing algorithm to mitigate the effects of a misrouting HT, capable of launching a DoS attack by tampering the header flits of a packet. Although, these works focuses on the design of HTs capable of launching a DoS or performance degradation attack, very few researches investigates the effect of HTs and malicious applications working together to launch an attack on the MPSoCs. In [62] authors proposed an attacker model compromising of a HT implanted in the NoC switches and an accomplice application running on the MPSoC. Once the HT is triggered, the accomplice application can send commands to the compromised NoC component to launch a plethora of attacks including snooping, and DoS attacks. To encounter such attacks, a layered security architecture comprising of data scrambling layer, packet certification layer, and node obfuscation layer has been proposed by the authors. However, by accumulating enough encrypted packets, the one-time pad XOR cipher used by the data scrambler and packet certification layer can be compromised. Furthermore, this work fails to capture the severity of a snooping based HTs. In this work, for the first time, we demonstrate that by analyzing the payload from a lightweight HT, which, in its simplistic form, is a counter, a remote attacker utilizing deep learning based data augmentation and processing techniques, can determine the application profile in a MPSoC with a very high accuracy. The primary reason for an attacker being able to derive the application profile is due to the Dimension-ordered Routing mechanism in NoC architectures which is highly deterministic in nature. Secure routing mechanism like Region-based Routing (RBR) and Segment-based Routing (SBR), proposed in [63], are

not sufficient to mitigate the effect of such HTs as these routing mechanisms partitions the NoC into different security zones and minimizes the inter-zone traffic to ensure secure NoC communication. Furthermore, these mechanisms are also deterministic in nature, making them ineffective against our proposed attacker model. On the other hand, the NoC routing algorithm proposed in [64], uses west-first and adaptive XY routing to improve NoC security by offering more paths for routing and reducing interference with attacker. However, being partially adaptive, such routing algorithms are vulnerable to an attacker exploiting the routing paths. Priority-based routing mechanisms like Non-Interference Based adaptive (NIBR), as proposed in [65], is based on DOR. Therefore, such routing mechanisms are not impregnable by an attacker exploiting the packet count of the NoC switches. Hence, to safegaurd a NoC against such attacker, a randomized routing mechanism is essential. However, a complete randomized routing can result in severely degrading the performance of a NoC as the packets are not routed over shortest paths. Therefore, instead of simply adopting random routing, in this work, to encounter an attacker exploiting routing path information, we propose a parameterized SA-based routing approach that can be tuned to achieve a desired trade-off between the defense and performance degradation due to the non-optimal path taken by a packet.

2.4 Survey on Hardware Trojans

The proliferation of the deployment of sophisticated third-party Intellectual Property (3PIP) circuits in embedded, IoT, and cyber-physical systems (CPS) has exposed the devices to a host of security vulnerabilities. With the intensifying costs of circuit fabrication (fab) plants and the overheads involved, most IC designers prefer a fab-less setup. As a consequence, the devices are manufactured off-shore, where security cannot be guaranteed or verified. With such globalization of the manufacturing process, the community has seen an escalation in implanted hardware Trojans (HTs) in the devices [66–70]. An HT is a malicious modification made to an *authentic* design to cause malfunction, steal or leak sensitive data,

cause Denial-of-Service (DoS) attacks, or impede the normal functioning of the devices, to disrupting the device completely. The vulnerable computing devices are not limited to CPU but also extend to ASICs, FPGAs, and others [13, 14, 16, 17, 20, 71, 72].

As shown in Fig. 2.1(a), IC manufacturing is a multi-entity supply chain process. We have grouped some of the individual supply chain units into phases for simplification and brevity. Research on HTs has revealed that the vulnerability can be present in almost any IC manufacturing phase. Works in [73–80] discuss the design of HT for targeting the design phase; [81, 82] present research on HTs for the third-party IP (3PIP) and CAD tool phase, while [17, 83–86] and [87, 88] present the vulnerabilities and HTs for the fabrication and test phase respectively. The impact of HTs can be proven expensive when deployed in defense applications and can lead to national security concerns [66]. A plethora of Trojan design mechanisms exist; ring-oscillator [71,89], combination/sequential circuit [90,91], HDL software [92], single transistor [93], layout-level modification [93] based Trojans, to name a few.

There exist few works that survey the HT attacks and defenses [67–69, 94]. In contrast to the existing works, this work provides a novel categorization of the HTs based on the application/target platform and also provides insights on the HT designs, which has not been engulfed in the previous works. This survey aids the designers to understand the attacker's capabilities, the risk level at each phase of the supply chain, and the vulnerabilities exploited to ensure those are factored-in during the IC design and test cycle.

2.4.1 Attack model and countermeasures

2.4.1.1 Trojan Insertion in the Supply Chain

An adversary can insert a hardware Trojan at any IC supply chain phase [94]. In-house rogue designers are capable of inserting Trojan at the design phase. Attacker in a 3PIP vendor can insert stealthy HTs into the IP at RTL and/or netlist level or other design specification levels. Using the unauthenticated HT-inserted 3PIP in the original design compromises design integrity [94]. A design faces maximum trust issue when it is sent to

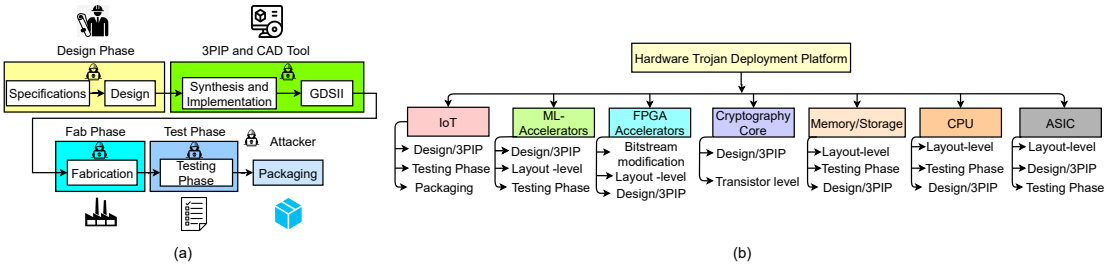


Figure 2.1: (a) Supply-chain and vulnerability; (b) Hardware Trojans categorized according to applications and vulnerabilities exploited

a third-party facility for fabrication. In an untrusted foundry, the original design could be exploited by inserting Trojan at the fabrication stage. In the testing stage, attacks can manipulate the Trojan detection method or test data that leads to false-negative results. Sometimes, the testing with traditional 'Automatic Test Pattern Generation' is ineffective as they cannot trigger certain stealthy Trojans [20]. Even after the testing phase, the attack surface is still open for the distribution phase attackers. An attacker can reverse engineer the design and replicate the design with a Trojan-embedded version during the distribution phase. The vulnerable phases in the IC supply chain are presented in Fig. 2.1(a)

2.4.1.2 Categorization of HT According to Platform

To the best of our knowledge, no previous work has been done for categorizing HT-attacks based on their platform-specific deployment. In this work, we spotlight on platform-specific HT design and their impacts on CPS systems. A HT can be designed for a range of application platforms such as ML-accelerators [13, 14], IoT devices [15, 16], FPGAs [89, 90, 92], ASICs [19,20], Cryptography cores [17,18], and CPUs [95,96]. If a HT is intended to be deployed on any of the aforementioned platforms, we classify that HT to be a threat against the deployment platform. Fig. 2.1(b) shows the hardware platforms and the corresponding

vulnerable phases in the supply chain. Categorizing Trojans gives a bird's-eye-view of the HTs that target the application platform, thus posing unintended security challenges on the deployed platform and the system.

Table 2.1: Details of Hardware-Trojan works based on implemented platform

Work	Target	Evaluated Benchmark	Platform	Risk Level					Payload	Trigger	Detection Mechanism
				Design	3PIP	Fabrication	Test	Distribution			
[14]	DNN Misclassification	ImageNet dataset	Memory Controller IP realized on a 28nm technology	✓	✓	✗	✗	✗	Circuit-based input data poisoning in intermediate layers of classifier	Geometric feature-based mechanism	Avoid untrusted 3PIP vendors; thorough functional testing;
[13]	Perturbation of target operations in a NN	MNIST	ReLu computation block	✓	✓	✗	✗	✗	Combinational circuit	Training inputs from dataset fed to combinational circuit	Nearly impossible with traditional detection methods
[15]	Compromise security by leaking ROPUF frequency to adversary	N/A	IoT platform; ML-assisted attack;	✓	✓	✗	✗	✗	Sequential Trojan circuit in ROPUF; 130 nm and 128-bit CMOS Trojan	Sequential + combinational circuit	Monitor the distribution of sensed data; Trojan-infected ROPUF sensor chips have non-Gaussian distribution, in contrast to Gaussian for Trojan-free sensors
[97]	Cause degradation of NN performance	CIFAR10 and MNIST	Convolutional NN on NVIDIA Tesla GPU	✓	✓	✓	✗	✗	Multiplexer-based combinational circuit	Custom designed combinational circuit- based trigger	Combination of adversarial training and hardware Trojan detection
[71, 89]	Bitstream modification for device ageing and failure	128-bit AES cipher	FPGA	✓	✓	✗	✓	✓	RO-based always-ON circuit	Training inputs from dataset	Thorough functional test; temperature test; optical test
[90]	Recover secret key during cipher operation	PRESENT cipher	Xilinx Spartan-6	✓	✓	✓	✓	✓	Combination circuit for fault injection	1-bit signal activating the payload	-
[98]	Recover secret key/ Weaken cyrptographic algorithms	AES, DES and 3-DES	FPGA	✗	✓	✗	✗	✗	Malicious modifications to LUT, S-Box and Embedded Memory post bitstream generation	N/A	Built-in self-test, forced decomposition and white-box cryptography all of which have limited uses
[92]	Compromise Design and 3PIP/CAD softwares to inject HTs in multiple designs	N/A	Yosys/Arachne-pnr/Project IceStorm/FPGA	✓	✓	✗	✗	✗	Malicious modifications to synthesis and bitstream generation software	Custom malicious trigger cells	Exhaustive functional or formal verification
[93]	Compromise security of the keys generated with RNG, and reduce the attack complexity	NIST test suite	Intel Ivy bridge	✗	✗	✓	✗	✗	Dopant level modifications at sub-transistor level	N/A	Unspecified
[17]	Key recovery from ciphers	AES cipher	FPGA	✓	✓	✗	✗	✗	XOR gate	Combinational circuit	Compare layout images with GDSII; Keep place-and-route density > 80%
[20]	Ring oscillator on ASIC	ISCAS s9234	ASIC	✓	✗	✗	✗	✗	N/A	Internal	N/A
[19]	Ring oscillator on ASIC	ISCAS'89 s9234	ASIC	✓	✗	✗	✗	✗	Leak frequency information	Random test patterns	Timing-based detection
[99]	Exploit DRAM	RocketChip SoC	Memory	✗	✓	✗	✗	✗	Fault-injection, DoS attack	Accessing pre-defined address	N/A
[91]	RRAM	ASU RRAM Verilog-A	Memory	✓	✗	✓	✗	✗	Information leakage, Fault injection or DoS attack	accessing pre-selected address	N/A
[72]	L1 d-cache	GEM5 simulator	Memory	✓	✗	✓	✗	✗	Fault injection/DoS attacks	Certain data pattern for a certain number of times	Extensive delay/power profiling
[100]	USB flash drive	XTS-AES-256	Memory	✗	✗	✗	✗	✓	Create covert, Remote channel	External	Remains undetected
[96]	CMOS gate	Ivy Bridge processors, SBox	CPU	✗	✗	✓	✗	✗	Leak secret keys	Unspecified	N/A
[95]	Secret processor information	8051 microcontroller	CPU	✗	✗	✓	✗	✗	Leak software IP, Leak encryption key, Cause system malfunction	Specific sequence of data, Specific instruction	N/A
[101]	Secret processor information	GEM5 simulator	CPU	✗	✗	✓	✗	✗	Data leakage from TLB	L1 cache access, Certain data pattern	Memory image analysis
[102]	Processor register	OR1200 processor	CPU	✗	✗	✓	✗	✗	Full control over the processor	Sequence of unlikely events	N/A
[103]	Power constrained IoT device	ISCAS85	IoT	✓	✓	✓	✓	✗	Deviates from original functionality for certain input	Attacker chosen set of vectors	Remains undetected

2.4.2 Hardware Trojan Design Classification

In this section, we categorize and explain the state-of-the-art HT attacks according to the target hardware, such as Machine learning accelerators, Memory subsystem, FPGA, CPU, ASIC, Cryptocores, and IoT devices as shown in Fig. 2.1(b).

2.4.2.1 FPGA Platforms

FPGAs are increasingly deployed in many public and defense applications due its reconfigurability, programmability, and reliability. However, FPGAs are no exception to HT attacks. Authors in [89] demonstrate insertion of a ring-oscillator (RO)-based HT in an unencrypted bitstream, synthesized for an FPGA platform used for 128-bit AES cipher operation. The ROs are always-ON type HTs that increase the device's temperature (by dissipating power), thus causing aging and ultimately failing the FPGA much earlier than its estimated life. A similar HT is presented in [71], with the difference that it inserts the HT during the HDL design/3PIP phase. Work in [90] presents a multiple-fault injection attack-based HT to flip four nibbles in the penultimate round of the PRESENT cipher to steal the secret key using two faulty ciphertexts. The HT injects the fault to perform the attack. Work in[98] proposed insertion of HTs in FPGA bitstream. The attack works by detecting and modifying the S-box in the bitstream, thus affecting the AES and 3-DES cipher algorithms. Work in [92] takes a two-step approach to insert HTs during the design phase. To avoid detection, malicious functionality is injected during the HDL process that incorporates camouflaging circuitry. Post bitstream generation, malicious logic is activated by reconfiguring the circuitry that triggers the HT.

2.4.2.2 Machine-Learning Accelerators

Machine-learning accelerators (MLA) have emerged as a panacea to process emerging and state-of-the-art ML applications with high throughput. Given a wide range of systems using MLAs, security becomes a crucial aspect for the system. Authors in [14] propose a Deep NN (DNN) architecture layer-level modification as an HT inserted during the design and 3PIP phase in the memory controller IP capable of snooping on the data bus connecting the off-chip memory. The goal is to force the DNN (Deep Neural Network) accelerator to output incorrect classification results after the Trojan trigger. The attack proposes to leverage memory access patterns to identify the input data. Attack delivers high accuracy even amidst preprocessing and system noise.

Another work targeting a Neural Network (NN) accelerator is [13], where a combinational circuit type HT payload is injected during the design phase in the computation block (ReLu) to achieve perturbation on targeted operations. HT is triggered based on the training inputs from MNIST data. Without modifications to training data, the attack achieves high accuracy in perturbing the output results and remains stealthy; thus, bypassing detection due to effective negligible area overhead.

Another proposed HT design, in combination with ML, is presented in [15]. Ring oscillator physical unclonable functions (ROPUFs) are employed in the wireless sensor as a safeguard against invasive-attacks. The authors in [15] propose to insert an HT to leak the oscillating frequency of the ROPUF to an adversary. With the knowledge of the frequency, the input, and the output challenge, an accurate ROPUF can be modeled with the help of the ML model to break the security barrier. With 200K leaked samples, [15] depicts cracking a 128-bit ROPUF -based sensor. [97] presents similar work.

2.4.2.3 Cryptographic Cores/IPs

CPS systems extensively use cryptocores or IPs to enhance computation and communication security. These hardware blocks are required to process crypto-operations such as AES, DES, RSA for data security and privacy. An HT implanted in such CPS systems can target to leak private keys, bypass crypto-blocks inside the hardware, perform a DoS attack, or disrupt security measures enforced in the hardware to benefit the adversary [17, 93, 104]. In this subsection, we discuss some of the works focused on HT implants in the hardware.

HT attack explained in [93] performs modifications to the transistor's physical properties. The attack modifies the digital post-processing of the design at the sub-transistor level to compromise the security of the keys generated by a random-number generator (RNG) inside an Intel Ivy Bridge. The modified Trojan RNG design passes the built-in-self-test and generates random numbers that pass the NIST test suite. Yet, the Trojan remains stealthy, evading functional and optical tests. A differential fault analysis (DFA)-based AES HT is discussed in [17]. A 128-bit AES is attacked with the help (payload) of a single XOR

gate, which faults one bit of the crypto operation in its eighth round. Based on the fault analysis, the complete key is recovered by the adversary. Similar cryptographic attacks are presented in [75, 85, 104].

2.4.2.4 ASIC Platforms

Application-Specific Integrated Circuits (ASICs) used in a wide range of computing platforms are often produced in large volumes that gives the adversary a good motivation to insert a Trojan circuit in the original design, thus affecting a million devices or even more.

Work in [20] reports a Trojan design that has several heterogeneous Trojan circuits. The smallest Trojan circuit is designed with three AND gates and one inverter, totaling 4 gates with 26 transistors. The largest Trojan has 182 transistors in the design. Several Trojan circuits become active depending on the trigger signal, and critical frequency information is leaked from the ring oscillator circuit.

Similarly, authors in [105] present the hardware Trojan insertion into ASIC design during the fabrication phase. The Trojan design consists of a serial interface of "kill-sequence" observatory circuit by coupling D flip-flops back-to-back and gets input from the original circuit. Once a pre-defined kill sequence is observed, the Trojan causes a functional failure that leads to a denial-of-service (DoS) attack. The Trojan is embedded in the I/O location of the design, providing gate-level abstraction access when triggered externally.

2.4.2.5 CPU

Central Processing Unit (CPU) is another platform of interest for inserting Trojans as all the processing is performed under the hood of the CPU, and an erroneous command could pose a serious security challenge to the system. Authors in [95] design a sequential Trojan circuit where the Trojan circuit is modeled as a finite state machine (FSM). During field operation, if a sequence of rare events happen, the corresponding Trojan stage is activated. The attack leaks secret keys or sensitive data from the CPU.

In [101], authors present HarTBleed, a Trojan design embedded in a CPU that has a

capacitor-based triggering circuit accompanied by a novel payload circuit. The Trojan is triggered when a pre-defined address of the L1 cache is accessed 1800 times. Data leakage occurs from the processor's translation look-aside buffer (TLB). Similar capacitor-based triggering is applied in [102] where the victim flip-flop is set to a certain value once the triggering capacitor is fully charged. Work in [102] inserts an HT in an OR1200 chip. The Trojan remains undetected with the state-of-the-art defense mechanism.

2.4.2.6 Memory/Storage Unit

As sensitive information leakage is one of the primary rationales behind HT attacks, designers are leaning towards the memory/storage level Trojan insertion to decrypt critical data. Authors in [106] inserted two versions of HTs into an SRAM array in the design phase. The proposed stealthy design can successfully evade the testing phase. Type-1 Trojan is designed by concatenating a series of nMOS pass transistor (PT) sequentially where gates are connected to the trigger nodes. Another Trojan type, Type-2, involves one additional pMOS PT in addition to Type-1 that breaks a pull-up path in the memory cell. The attack uses multiple data patterns as a triggering method for the Trojan, post which the SRAM cell becomes dysfunctional. Authors in [99] target a DRAM memory by inserting a Trojan that exploits advanced properties of DRAM such as the underdrive (protects from retention failure) and the overdrive (supports write operation) properties. Here, the HT has a capacitor-based trigger circuit that activates a range of payloads such as Denial-of-Service, information leakage, and fault-insertion. Similar work targeting RRAM memory is presented in [91]. A recent study [107] shows Trojan insertion in an STT-MRAM based memory cell by exploiting the inherent dynamic faults. Here, a poly-based resistor is inserted as a Trojan in between nodes of a bit-cell. The Trojan is triggered under the circumstances when n-number of the read operation is performed from a pre-selected memory address showing a certain data-pattern. Unlike previous designs where HT insertion is done in the supply chain ranging from the design phase to the testing phase, authors in [100] present an attack during distribution phase. [100] discusses HT inserted into a FIPS-140 level-2

certified USB flash drive that uses AES-256 encryption by means of reverse-engineering the FPGA bitstream and an ARM CPU firmware. Altering the ARM CPU code eventually manipulates the AES algorithm, leading to deciphering the user information in the flash drive by applying a couple of plaintext-cipher pairs.

2.4.2.7 IoT Platforms

Inserting a tiny Trojan into lightweight IoT devices which lack built-in security mechanism such as sensors or network ICs might lead to a range of attacks into the whole network such as Denial-of-Service attack and remote attacks [108].

As FPGA, ASIC, CPU, System on Chip (SoC) can be used as an IoT device, a Trojan can be inserted into any of the platforms mentioned above to gain desired adversarial impact. An HT is proposed in [103] where the Trojan design has zero area and power overhead. The attack uses an asynchronous counter-based design that modifies an input signal of a Trojan-free circuit when triggered. The triggering circuit is a 3-bit counter with a very small power overhead. State-of-the-art power analysis tools are unable to detect this Trojan. As IoT devices are power constrained, these types of Trojans can easily be inserted on IoT devices with zero area and power overhead and remain undetected. Work in [109] reports similar low-power Trojan for IoT devices. A sequential Trojan is inserted in a ring oscillator physical unclonable function (ROPUF) for IoT sensor network that leaks critical oscillating frequency information [15].

Discussion: Below, we present a brief discussion on the HT design; additional details are included in Table 3.4, where 'target' refers to the motive/intention of attack, 'Risk Level' for potential insertion phases during the supply chain; 'Payload' refers to additional malicious hardware serving as Trojan, while 'Trigger' is a mechanism that observes a physical parameter inside the target device, and activates the payload.

So far, we have studied the feasibility of HT attacks on different platforms ranging from FPGA to ASICs. Of the works surveyed for the HT attacks on FPGAs/ML-accelerators, it is observed that the feasibility of HT insertion and impact is widely feasible at design and

3PIP integration phases compared to fabrication, as inserting HTs during the fabrication phase is a tedious and non-trivial job, though possible. For HT attacks [71, 89], they are plausible on encrypted bitstream by combining with an encrypted bitstream recovery attack as explained in [110]. Similarly, for the FPGA- and Cyrptocore- based HT attacks, it is observed that the adversary can insert payloads at different phases of the supply-chain; with sophisticated 3PIP/CAD vendor tools, the attack surface broadens to implant a variety of modifications to internal blocks, such as the Sbox, AES/DES/RSA crypto blocks, internal communication bus, layout, netlist and so on with negligible area and power overheads, which makes detection with conventional mechanisms (during test phase) a non-trivial job. An HT can be inserted at the distribution phase by reverse-engineering the design and replace it with a Trojan-injected version. A similar attack is presented in [100] on a consumer-level USB flash drive at the distribution phase to leak sensitive information. HT attack paradigm is shifting towards the IoT devices because of the large attack surface and payoffs. HT attacks on IoT sensor networks pose a security risk to the overall network. Analyzing recent works on HT Trojan design, we gather HT design trend and risk challenges in the supply chain. ASIC designs are more prone to Trojan insertion at the design and the fabrication phases. For Memory/Storage platform, 3PIP phase is exploited, in addition to the design and fabrication phases. For HT design targeting IoT devices, low area footprint, and power-based Trojans are most dominant while 3PIP vendors in the supply chain are most untrustworthy.

Chapter 3: Hardware-Assisted Malware Detection

The ever-increasing complexity of modern computing systems has resulted in the growth of security vulnerabilities, making such systems an appealing target for sophisticated attacks. Computing systems today are employed to deliver high performance and efficiency while protecting users' data. Attackers take advantage of malware's capabilities to harm the system, steal users' data, disrupt the system, or a combination of it. Malware, also known as malicious software, is a program or application to infect the computing systems without the user agreement for serving harmful purposes such as stealing sensitive information, unauthorized data access, destroying files, running intrusive programs on devices to perform Denial-of-Service attack, and disrupting essential services.

A plethora of works focus on detecting malware, but the downside of using software-based approaches is the overhead, owing to computational complexity. Also, software-based detection utilizes signature-based detection that matches the behavior signature of the application to its database. This approach fails to recognize zero-day attacks, and signatures that do not match its database, given an outdated database. We focus on hardware-based detection approach.

To overcome shortcomings such as latency and computational complexity of traditional malware detection techniques, including signature and semantics-based software-driven techniques [111, 112], Hardware-Assisted Malware detection (HMD) approaches are proposed [27]. HMD refers to utilizing the low-level microarchitectural hardware events for detecting and classifying the malware from benign applications. Browsers, utility applications, text editors, C-based programs were some of the benign applications that we profiled as a part of the dataset. In contrast, applications embedded with Trojan, worm, virus, and other malwares were profiled as part of the malign traces. The HMD delivers reduced malware detection latency by orders of magnitude with smaller hardware cost [27].

This work proposes an adversarial attack on HMDs in which the adversarial samples are generated through a benign code that is wrapped around a benign or malware application to produce a desired output class from the embedded ML-based malware detector. One of the main challenges to address is that the attacker or user has no direct access to modify the HPC and manipulation of HPCs is highly complex to perform despite employing techniques like code obfuscation for executing malware [27]. First, we assume the victim's defense system is unknown and perform reverse engineering to mimic the embedded HMD's behavior and build a machine learning (ML) classifier. To determine the required number of HPCs to be generated through the application to be misclassified, we employ an '*adversarial sample predictor*' which predicts the number of HPC traces to be generated to misclassify an application by the HMD. by the attacker. To perturb the HPCs, we craft an '*adversarial HPC generator*' application (code) that generates the required number of HPCs. This adversarial HPC generator application is wrapped around the application that needs to be misclassified by perturbation.

This work discusses a novel way of crafting adversarial HPC traces through a benign application and proposes a methodology on how to craft such an application to obtain adversarial behavior. The main focus of this work is to generate false alarms (malware classified as benign and benign classified as malware) to weaken the trust on the embedded defenses, which increases the scope for attacks. The proposed work benefits from the following: a) no need to tamper or modify the source code of the application around which the proposed adversarial sample generator code is wrapped; b) the crafted application has no malicious features embedded, thus not detectable by malware detectors; c) scalable and flexible, i.e., the crafted application can generate events required to create a powerful adversary. With these adversarial attacks, the HMD delivers unacceptable performance. To make the HMD robust and resilient to such adversarial attacks, we propose to perform adversarial learning by training the HMD on adversarial samples. This method has proven successful for different types of adversarial attacks and can boost the HMD performance to classify malign applications from their benign counterparts reliably. We then present

hardware implementation of the ML classifiers used in the HMD for analysis purposes.

3.1 Hardware Assisted Malware Detection

Thus far, we have had a background of what adversaries are in context to the HMDs and discussed if microarchitectural features like those obtained from the performance counters can be used to classify malware from benign. In this section, we are going to take a look at how microarchitectural features can be used to train HMDs, what features will serve as the best means to classify a malign application from benign class and the overall view of the HMD detection process. Refer to Figure 4.17 for an overview of the detection process. The preliminary step in the training process requires to profile applications and generate a dataset that can be later used for training machine learning models - the heart of the HMD. The dataset comprises of captured features that describe the state of the hardware at different time instances for applications executing on the system. A variety of applications are executed in Linux containers to ensure safety and to prevent contamination of the OS, other applications and to prevent other measurements from being affected. Linux containers, unlike virtualization, allows sharing the operating system kernel and provide isolation to the application from the other system [113]. Perf tool [114] in Linux is employed to profile applications and collect all the available HPCs. This process is described in the Figure as the "Feature Extraction" block. These HPCs are used to build the dataset comprising of all the applications with the corresponding features (HPC values or traces). Not all features are useful in training the ML-models as some are irrelevant to the current context; also high dimensional dataset will lead to sluggish performance and difficult to establish in runtime. Hence, feature reduction techniques are employed to reduce the number of features to the most relevant ones as shown in the figure and will be discussed in the next subsection. We train and deploy multiple ML-models in the HMD to observe the model that delivers best performance in detecting malware. Adaboost and Ensemble techniques are utilized to render high detection accuracy with less number of features so the HMD could be used in runtime to classify applications and with high-performance gains. Next, we discuss feature

reduction and training in more detail.

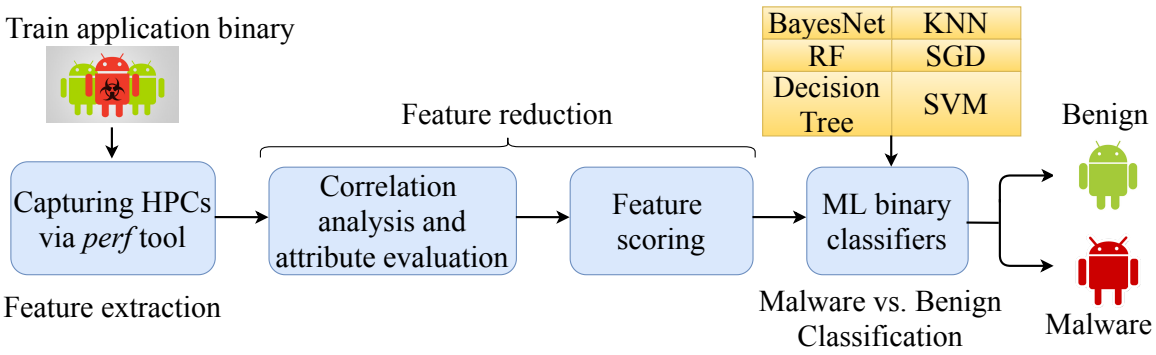


Figure 3.1: Overview of the proposed hardware-based malware detection process

This section briefly discusses the overview of hardware-assisted malware detectors, referring to Figure 3.1. The preliminary step in the training process requires profiling applications and generate a dataset that can be later used for training machine learning models - the heart of the HMD. The dataset comprises captured features that describe the hardware's microarchitectural state at different time instances for applications executing on the system.

The process of profiling of applications is described in the figure as the "Feature Extraction" block. These HPCs are used to build the dataset comprising all the applications with the corresponding features (performance counters).

We train and deploy multiple ML-models in the HMD to observe the model that delivers the best performance in detecting malware.

3.1.1 Feature Selection

As mentioned earlier, detecting malware using machine learning models requires representing programs at a low microarchitectural level. This process produces a very high dimension dataset. Running ML algorithms with large dataset would be complex and slow. Besides,

Table 3.1: Microarchitectural events important for runtime malware detection

Rank	Event name	Rank	Event name
1	Branch Instructions	5	dTLB_store_misses
2	Branch Loads	6	LLC_prefetch_misses
3	iTLB_load_misses	7	L1_dache_stores
4	dTLB_load_misses	8	cache_misses

incorporating irrelevant features would result in lower accuracy for the classifier [24]. Therefore, instead of accounting for all captured features, irrelevant data is identified and removed using a feature reduction algorithm, and a subset of captured traces is selected that includes the most important features for classification. The features are supplied to each learning algorithm, and the learning algorithm attempts to find a correlation between the feature values and the application behavior to predict the application type.

Table 3.2: Microarchitectural events of high priority for runtime malware detection

Rank	Event name	Rank	Event name
1	Branch Instructions	5	dTLB_store_misses
2	Branch Loads	6	LLC_prefetch_misses
3	iTLB_load_misses	7	L1_dache_stores
4	dTLB_load_misses	8	cache_misses

As discussed, the key aspect of building an accurate detector is finding the right features to characterize the input data. We collected 44 performance counters, as they were all the hardware counters our experimental setup allowed. As shown in Figure 4.17, after feature extraction, the feature reduction process reduces the number of features. We first

use Correlation Attribute Evaluation on our training set under WEKA [115] to monitor the most vital microarchitectural parameters to capture application characteristics. Next, the features are scored based on their importance and relevance to the target variable through the feature scoring process. By applying the feature reduction method, the eight most related hardware performance counters are determined and numbered in order of importance for malware detection. These HPCs are listed in Table 3.2. Most modern processors allow recording 4 or 8 events simultaneously. Hence, to suit the detection process given the hardware limitation on the number of events that can be collected, we constant the feature size to eight. These features are included in our prediction model as input parameters. The selected features include HPCs representing pipeline front-end, pipeline back-end, cache subsystem, and main memory behaviors influential in the performance of standard applications. It is to be noted that the eight HPCs selected are needed to determine the feature vector for feeding to the classifier. Hence, the dimension of the feature vector is 8x1 for a data sample.

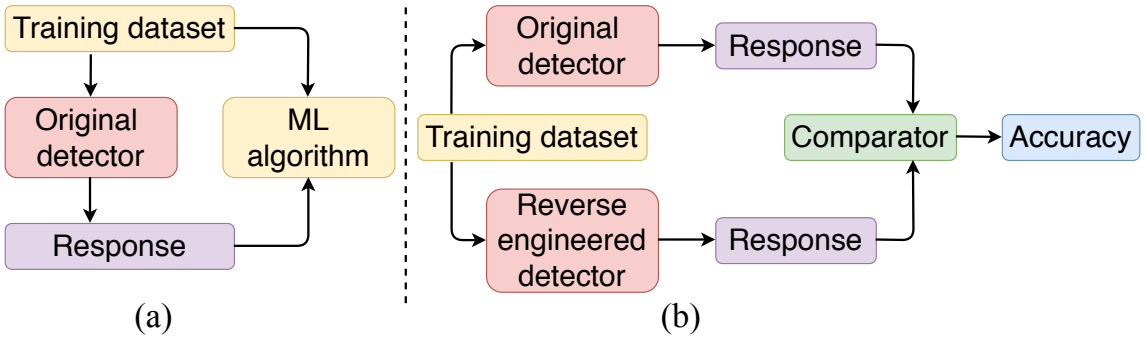


Figure 3.2: (a) Process of reverse engineering an HMD; (b) Testing Performance of Reverse-Engineered Detector

3.1.2 Training and Testing the Malware Detectors

Training involves profiling the incoming application with the Perf [114] tool under Linux and extracting low-level feature values for each training program, reducing the extracted features to the most vital performance counters, and developing a learning model from the training data. It is important to note that the input variables in our classifiers are the HPCs extracted every 10 ms interval from the running applications, and the output variable is the class (malware vs. benign) of an application. The HPC event sampling rate can be varied from a few milliseconds to few tens of microseconds. We sample the events at 10 ms interval to reduce overhead on the system performance. The sampling rate is a trade-off between performance and overhead. The chosen sampling rate satisfies the HPC event capture for the applications (malware and benign) used. For each ML classifier, we construct the general and ensemble models (AdaBoost and Bagging) to detect the malware. In order to validate each of the utilized ML classifiers, a standard 70%-30% dataset split for training and testing is followed. To ensure a non-biased splitting, 70% benign- 70% malware application for training (known applications) and 30% benign-30% malware applications for testing (unknown applications) are used. We perfom k-fold validation method, hence we did not break the dataset into train, test and validate.

3.2 Adversarial Sample Prediction

In this section, we discuss the adversarial attack on HMD, our proposed adversarial sample predictor, and the adversarial sample generator to degrade HMD performance.

3.2.1 Adversarial Attacks on Machine Learning Classifiers

In this work, the terms 'adversarial malware' and 'adversarial attack' are used interchangeably. Similarly, 'adversarial defense' and 'hardening' have been used interchangeably. Works in [116,117] describe the process of different adversarial attacks on ML classifiers. The fundamental idea is to perturb the inputs such that the performance of the classifier is degraded.

We propose to exploit the concept of input sample perturbation to attack the HMDs, as will be discussed further.

3.2.2 Reverse Engineering a Hardware-based Malware Detector

Under the assumption where the victim malware detector is unknown, we perform reverse engineering to mimic the functionality of the victim[1] HMD detector. Thus, as a first step to craft adversarial malware, we perform reverse engineering of the victim's HMD similar to that proposed in [118]. The performed reverse engineering is described in Figure 3.2.

We first create a training dataset that comprises benign and malware applications. Nearly 10,000 benign and 10,000 malware applications are used in the reverse engineering process. The victim's HMD (Original HMD) is fed with all the applications, and the responses are recorded. These responses are utilized for training different ML classifiers to mimic the functionality of the victim's HMD, as shown in Figure 3.2(a). Further, it is tested by comparing the outputs from the victim's HMD response and the reverse-engineered ML classifier's response, as shown in Figure 3.2(b). Reverse engineering is non-trivial as the adversaries generated on a closely functional model will be highly effective compared to a weakly developed adversary. To ensure reverse engineering is performed efficiently, we train multiple ML classifiers and choose the classifier that yields high performance, i.e., mimics the victim's HMD with high accuracy.

3.2.3 Process of Crafting the Adversarial Malware

Once the reverse-engineered HMD is built, such as MLP (or any victim defense classifier), the hyperparameters are determined. To launch and craft an adversarial malware, it is non-trivial to determine the level of perturbations that need to be injected into performance counter traces to get the applications misclassified. To determine the number of such HPC events to be generated, we deploy (offline) an adversarial sample predictor. As the ML classifiers are robust to random noises, perturbing the HPC patterns is sophisticated. To

[1]Victim HMD is the detection mechanism under the proposed adversarial attack

perturb HPC patterns, we employ a low-complex gradient loss-based approach, similar to Fast-Gradient Sign Method (FGSM) [119], which is widely employed in image processing.

To craft the adversarial perturbations, we consider the reverse engineered ML classifier with θ as the hyperparameter, x being the input to the model (HPC trace), and y is the output for a given input x, and $L(\theta, x, y)$ be the cost function used to train the reverse-engineered classifier. The perturbation required to misclassify the HPC trace is determined based on the cost function gradient of the chosen classifier. The adversarial perturbation generated based on the gradient loss, similar to the FGSM [116] is given by

$$x^{adv} = x + \epsilon sign(\nabla_x L(\theta, x, y)) \tag{3.1}$$

where ϵ is a scaling constant ranging between 0.0 to 1.0 is set to be very small such that the variation in x (δx) is undetectable. In the case of FGSM, the input x is perturbed along each dimension in the direction of the gradient by a perturbation magnitude of ϵ. Considering a small ϵ leads to well-disguised adversarial samples that successfully trick the machine learning model. In contrast to the images where the number of features are large, the number of features, i.e., HPCs are limited. Thus the perturbations need to be crafted carefully and also made sure that they can be generated during runtime by the applications.

In contrast to works that assume the application features to be binary, such as [120], this work aims to predict and determine the adversaries for the microarchitectural event patterns, i.e., HPCs, to generate during runtime with the aid of a benign code, which is one of the primary distinctions from existing works. It needs to be noted that determining the required perturbation for a given application is done offline. The process of crafting the adversarial application to generate the perturbations in the HPC trace during runtime is presented next.

3.3 Adversarial Hardware Performance Counter Trace Generation

To generate the required number of HPCs, we craft an application (benign) that wraps the victim application and generates additional performance counter traces that make the overall trace (of the victim application) similar to the predicted HPC trace. We discussed the adversarial HPC predictor previously. We do not know any works in the past that have employed the same approach as our work.

Algorithm 1 Pseudocode for generating adversarial HPCs

Require: Application 'App()'
Ensure: Adversarial microarchitectural events
1: **cache_miss_function()** {Sample pseudo code that generates required number of adversarial LLC misses}
2: #define array[n] % Size of array and loop define amount of variation
3: `load` i #0
4: *Loop 1:* `cmp` i #n {Compare i with n}
5: array[i]=i
6: `jump Loop1`
7: end
8: `load` i #0
9: *Loop 2:* `cmp` i #k {$k <= n$}
10: *ld rax &array[i] # load array address in register rax*
11: *cflush (rax)* {Instruction as a function of array size and loop size}
12: *jump* Loop2
13: end
14: **branch_misses_function()** {Code that generates required number of adversarial branch instructions and branch misses}
15: #define int a, b, c, d
16: a<b<c<d<n
17: *Loop 3:* `cmp` i #a { $\cdots$ function $\cdots$}
18: *Loop 4:* `cmp` i #b { $\cdots$ function $\cdots$}
19: *Loop 5:* `cmp` i #c { $\cdots$ function $\cdots$}
20: *Loop 6:* `cmp` i #d { $\cdots$ function $\cdots$}
21: *Loop 7:* `cmp` i #n { $\cdots$ function $\cdots$}
22: *jump* Loop 3; end ;
23: {Similar functions to generate other HPCs as predicted by adversarial sample predictor}
24: **APP()** {User/Attacker's application to be executed}

In Algorithm 1, we show the pseudo-code to create adversarial LLC load misses and branch misses. The LLC load misses, and branch misses are some of the pivotal microarchitectural events that malicious applications [112]

To generate LLC load misses, an array of size n is loaded from memory and flushed to

generate LLC load misses. This is outlined in Line 2-12 of Algorithm 1. The experiments are repeated multiple times with different array sizes (n) and the different number of elements flushed (k) to determine the number of LLC load misses generated. Further, a linear model is built to find the dependency of n and k on the number of LLC load misses. Once the adversarial sample predictor predicts the number of LLC load misses generated to craft an adversarial sample, the n and k are accordingly determined. We employ a linear model due to its low complexity and high accuracy (<3% error) to determine the dependency between n and k for our experiments.

Example: For instance, the crafted application similar to that depicted in Line 2-12 of Algorithm 1 with n and k set to 100K leads to an LLC load miss of 73K, whereas when n and k is set to 500K, around 287K LLC load misses are generated. The flushing of the data has been verified by executing attack code with and without flushing the cache lines - the execution time is around 1.5× when the data is flushed compared to the case when data is not flushed.

In a similar manner, branch misses and branch instructions are generated as shown in Line 15-22 of Algorithm 1. To increase the branch misses, a set of conditional statements, i.e., comparison statements, are embedded into the application to create branch misses. The branch instructions depend on the number of conditional statements evaluated. In the presented pseudo code, we have five conditional statements for generating branch-misses (Line 15-22). For the attack code on branch miss events, with a loop size of 20K and integer values assigned to a, b, c, and d, branch misses' value is around 255K. An increase in branch misses is observed by inserting not taken (not executed) dummy loops.

The process flow of adversarial sample predictor is shown in Figure 3.3(a), where the performance counters from the victim application are combined (offline) with the predicted samples. These are fed to the ML model to gauge its performance. Suppose the ML model classifies the malware and benign with high accuracy. In that case, the adversarial attack parameters are modified; else the predicted samples are utilized for adversarial sample generation during application execution, as shown in Figure 3.3(b). In Figure 3.3(b), the

overall performance counters seen by the system are a result of the original application and the adversarial code. The HMD profiles the applications in runtime. If the predicted HPC values are smaller than those generated by original applications, we insert delay elements to smoothen the HPC trace and reduce the HPC values. It needs to be noted that using this process we generate adversaries to force the HMD to misclassify benign as malware and malware as benign applications.

3.4 Hardening HMD Against Adversarial Malware

We have discussed how attacks are performed to trick the HMD and force misclassification. The adversarial malware is crafted to perturb the HPC patterns and hence trick the victim HMD. Although HMD-Hardener can be employed for different defense strategies to ensure hardening for best performance under different adversarial attack types, we keep the discussion limited to 'FGSM' type attack and 'Adversarial Training' (FGSM in this work) type defense for conciseness. Adversarial training is one of the initial solutions introduced as a way for ML classifiers and deep learning classifiers to battle against adversarial samples. The method of adversarial training focuses on having adversarial samples used to train the model/classifier. They obtain the adversarial information in the training stage itself and stay robust against such attacks. We retrained the HMD using adversarial samples, and it is observed to deliver robust performance under an adversarial attack. The assumption is that we know the type of attack that can happen and the attack parameters.

3.5 Results and Evaluation

In this section, we present the accuracy of HMD in classifying malware and benign applications. Further, we give the impact of an adversarial malware attack on the HMD and attack resiliency post hardening. Finally, we present hardware implementation results for the ML classifiers.

3.5.1 Experimental Setup and Data Collection

The applications (both malware and benign) are executed on an Intel Xeon X5550 machine running Ubuntu 18.04. We execute more than 3000 benign and malware applications for HPC data collection. Benign applications include MiBench benchmark suite [121], Linux system programs, browsers, text editors, and word processor. For malware applications, Linux malware is collected from virustotal.com [122] and virusshare.com [123]. Malware applications include five classes of malware comprising of 607 Backdoor, 532 Rootkit, 2739 Virus, 1264 Worm and 7221 Trojan samples. All the applications (malign/benign) are profiled in Linux Containers. The adversarial sample predictor is implemented in Python using the Cleverhans library. The linear model is derived using the traditional statistical curve fitting technique. The adversarial sample generator is implemented using C and executed on a Linux terminal.

3.5.2 HMD Classification Performance

Figure 3.4 shows a comprehensive accuracy comparison of various ML classifiers used for malware detection. We implemented six general ML classifiers. The accuracy of malware detection with two feature sizes (4 and 2) are reported. As seen in Figure 3.4, MLP, random forest (RF) and decision tree classifiers perform very well for both the 4HPC and 2HPC as feature sizes. High performance with fewer features enables the HMD-Hardener to classify applications and detect malware in runtime with less overhead on the system, making repeated calls to the *Perf* tool. For instance, as shown, MLP achieves close to 82% accuracy, 80% for random forest (RF), 79% for SVM, and so on with four HPCs. However, we observe that reducing the number of vital performance counters to 2 results in similar classifiers' accuracy. We also observe that the HMD achieves detection accuracy in the range of 84-90% with 16 and 8 features. The higher gain results in overhead; results not shown for conciseness. For this work, we will consider the accuracy with 4HPCs with which the classifiers such as MLP, RF, decision tree, and KNN perform well with around 82% detection accuracy on an average. Four HPCs are easily possible to be captured

in runtime to allow malware detection. We also evaluated the aforementioned classifiers' performance by observing the Precision, F1-Score, and Recall metrics. These metrics' values are approximately similar to the accuracy metric's values shown in Figure 3.4.

Table 3.3: Impact of adversarial attack on HMD

	Accuracy	Precision	F1-score	Recall
Before attack	82%	78.1%	78.1%	82.1%
After attack	18.1%	45.0%	10.0%	18.0%
After hardening	81.2%	80.1%	80.1%	81.2%

Table 3.4: Post synthesis hardware results of different ML classifiers (@100MHz) when deployed in HMD-Hardener

Classifier	Power (mW)	Energy (mJ)	Area (mm^2)
MLP	90.45	5.12	4.5
RF	40.64	2.35	2.25
SVM	45.63	2.79	1.81
Decision Tree	36.54	2.29	1.55
SGD	54.46	3.21	1.46
KNN	44.81	3.37	1.27

3.5.3 Impact of Adversarial Attack on HMD Detector

We depict the impact of adversarial sample generator on the performance counters in Figure 3.5 that shows the LLC load misses for a benign application (ISCAS'85). The adversarial pattern predicted by the adversarial sample predictor is shown in Figure 3.5(a). We observe that there exist some spikes in the pattern as marked in the figure. Figure 3.5(b) shows the HPC pattern generated when the application is integrated (wrapped) with the adversarial

HPC generator. On average, there is an error of 2.2% between the trace predicted by the adversarial sample predictor and the trace generated by the adversarial sample generator. This indicates the adversarial generator can efficiently generate the required number of perturbations in the HPC traces. The MLP classifier delivers 82% accuracy on average in detecting malware. Post adversarial attack, the accuracy of the MLP drops to 18.1%, indicating that the adversarial sample generator degrades the HMD performance. The results of the MLP classifier's accuracy, precision, F1, and recall metrics are presented in Table 3.3. We observe similar results, shown in Figure 3.5, for branch miss type adversarial perturbations.

3.5.4 Adversarial Learning - Hardening

For hardening the HMD, it needs to be trained on normal samples and adversarial samples as well. For MLP classifier, the accuracy is restored close to the original accuracy before the attack, as presented in Table 3.3. We observe similar results with other classifiers after hardening, thus verifying that the HMD can become resilient against an adversarial attack, provided it is trained on adversarial samples. The HMD is trained on a new dataset, containing the original and adversarial samples combined. As the HMD is robust against adversarial samples, it delivers high performance in detecting the malware and benign samples, as shown in Table 3.3. Hence, if the HMD deployed in a system is adversarially trained on the perturbed HPC traces generated by using the process discussed thus far, the HMD is hardened against adversarial malware samples. This ensures the HMD delivers high performance against the normal and adversarial attack samples in runtime.

3.5.5 ASIC Implementation of Classifiers in HMD-Hardener

We conduct comprehensive hardware implementation of the classifiers embedded into HMD on ASIC. All the experiments are implemented on a Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit, 28 nm SoC running at 1.5 GHz. The power, area, and energy values are reported at 100MHz. We used Design Compiler Graphical by Synopsys to obtain the area for the models. Power consumption is obtained using Synopsys Primetime PX. The

post-layout area, power, and energy are summarized in Table 3.4. Among all the classifiers, MLP consumes highest power, energy and area on-chip (Table 3.4). The post-layout energy numbers were almost 2x higher than the post-synthesis results. This increase in energy is mainly because of metal routing resulting in layout parasitics. As the tool uses different routing optimizations, the power, area, and energy values keep changing with the classifiers' composition and architecture.

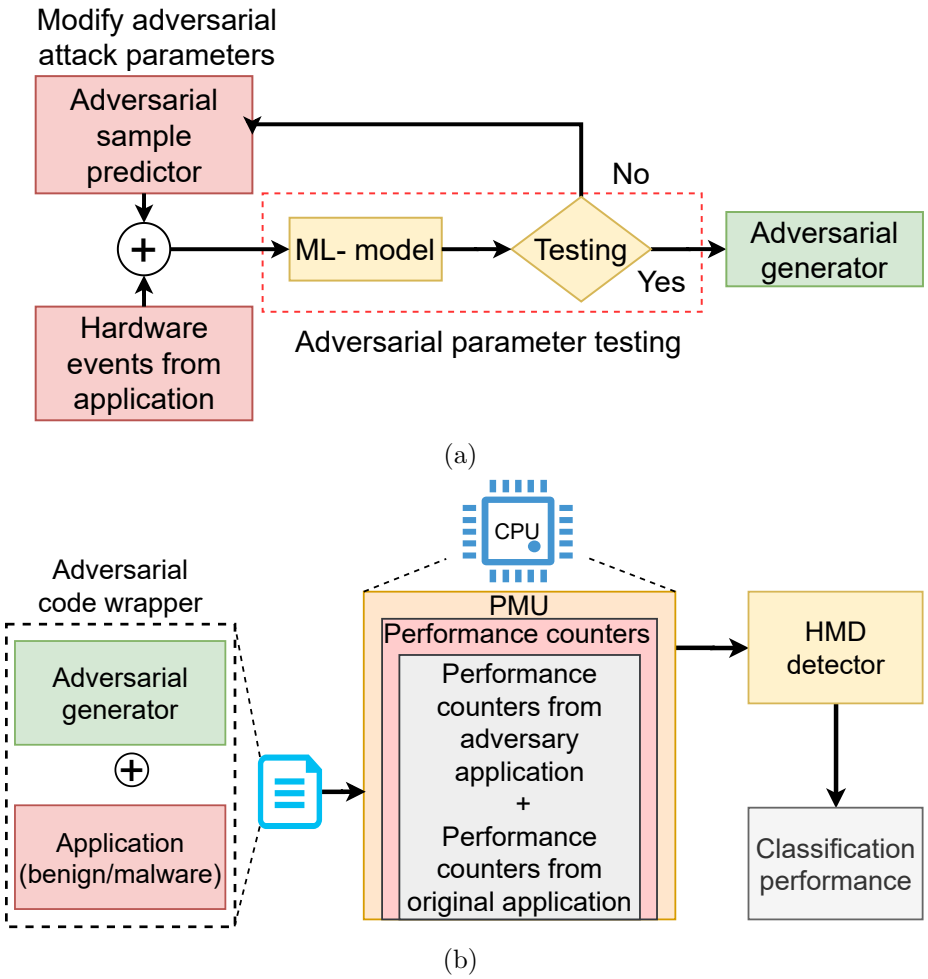


Figure 3.3: (a) Determining adversarial code generator parameters with the aid of adversarial HPC predictor; (b) Process of adversarial sample generation with adversarial code to force HMD performance degradation through misclassification of benign/malware applications

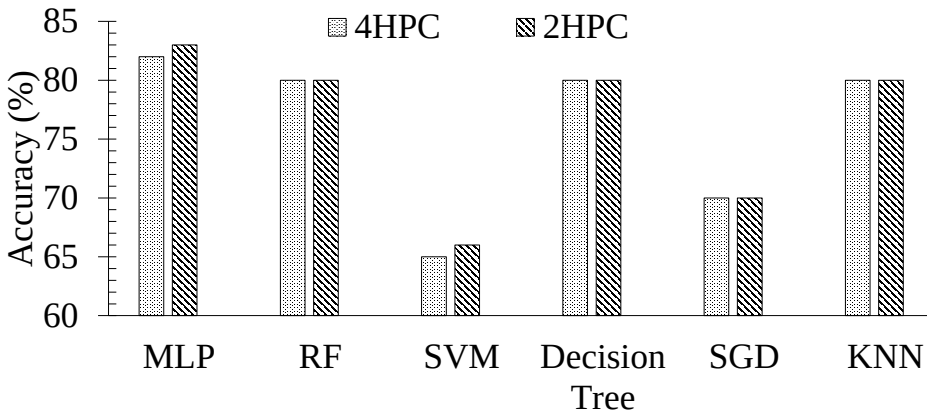
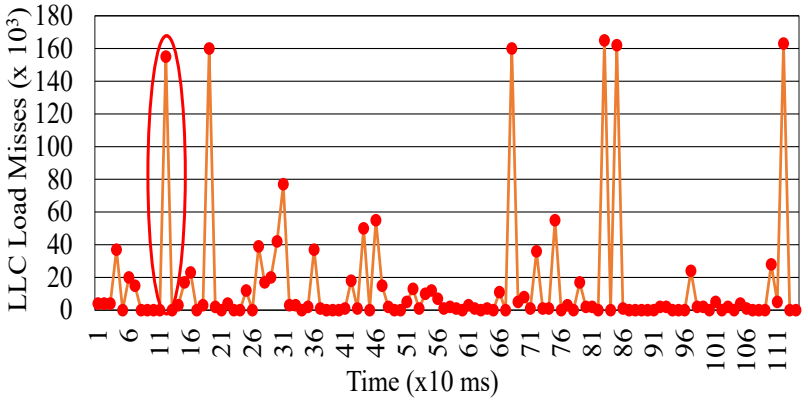
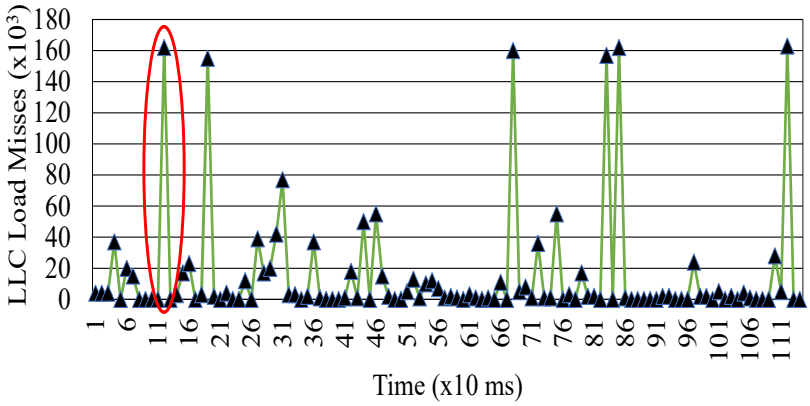


Figure 3.4: Accuracy results for various ML classifiers with feature size (HPCs) of two and four



(a)



(b)

Figure 3.5: (a)LLC load miss trace of the application predicted by adversarial sample predictor; (b) Generation of LLC load miss trace by adversarial sample generator

Chapter 4: Side-Channel Attack and Defense

4.1 CR-Spectre: Side-Channel Attack on ML-assisted Detectors

eThe insurgence of the side-channel attacks, attacks that exploit the inherent vulnerability in a system while trying to snoop on secure sensitive applications, at an alarming rate, is considered one of the pivotal issues. Spectre [6] is one such recently introduced powerful exploit that targets the vulnerability in modern branch predictors. Spectre 'mistrains' a branch predictor to perform legitimate operations initially, and later, it forces an erroneous speculative execution, which leaks sensitive data over a covert channel.

There exist detection mechanisms [124–128] to mitigate Spectre attacks by employing machine learning (ML)-based detectors, also known as Hardware-assisted Intrusion Detection (HIDs). The seminal theme of these works [124–128] is to train the ML-based detectors on the microarchitectural patterns[1] of the executing applications. The performance counters can extract different microarchitectural information regarding the application, such as cache-hits, cache-miss, total cycles, instruction count, etc. Spectre affects the branch predictor, cache, memory-related instructions' patterns during its execution [124, 125]. The existing defense techniques such as [124–127] utilize the affected performance counters' patterns to differentiate an attack and a benign application.

Traditionally, Spectre is launched as a standalone attack. However, in a system where an adversary does not have the permissions to execute a malicious binary as a standalone application, there is a need to evade the conventional launch process. Also, the HID detects

[1]Microarchitectural traces are obtained from the performance monitoring unit (PMU). These features are also known as hardware performance counters.

and protects the system by profiling the applications, thus guarding the system against side-channel attacks, such as Spectre.

On the other hand, there exists a genre of attacks known as *Code-reuse* attacks [129,130], which operate by subverting the control flow of the victim without directly manipulating the victim application or memory. Return-oriented programming (ROP) [130, 131] is one such example of a code-reuse attack. The methodology of the ROP attack is to target fragments of code, generically known as *gadgets* in the victim that end with *ret* (return) instruction. By chaining such gadgets together, an attacker can perform a Turing-complete manipulation to execute malicious instructions. Hence, the attack is also known as an ROP-chain attack. The attack redirects the program flow to the malicious code, thus, hijacking the control flow of the victim application. With the instruction code already existing in the memory, victim application can be forced by ROP attack to execute a malicious code, hence the term 'code-reuse.'

There exist techniques that can mitigate the ROP attack, such as Stack Canaries [132], and Address Space Layout Randomization (ASLR) [133]. ASLR works by randomizing the addresses in the memory. Although address randomization can deter an ROP-attack by randomizing the address space, the ASLR defense can be circumvented [134–137]. Stack Canaries [132] is memory protection that inserts a randomly chosen value in the stack between the local variables and return address. When a function call returns, it checks the value for any corruption. If the value is overwritten, the program exits without executing further. Similar to ASLR, Stack Canaries technique can also be evaded to launch a ROP attack.

e Yet another class of defenses for thwarting Spectre attack are InvisiSpec [138] and Context-Sensitive Fencing [139]. The former technique works by making speculative execution invisible to the system and other applications. It uses a *speculative buffer* to save data from load instruction until the load is deemed safe; later, the data is re-loaded to local caches, which also affects the microarchitecture. The latter defense employs a microcode customization mechanism allowing processors to insert fences into the dynamic instruction

stream to mitigate undesirable side-effects of speculative execution [139]. Both the defenses are employed at software-level, inducing overheads and require architecture level modifications [138, 139]. In contrast, this work targets system that demonstrate machine learning assisted mechanisms in detecting and mitigating Spectre. And, for injecting the attack, this work exploits buffer overflow vulnerability.

In this work, we exploit the ROP code-injection attack as a promising methodology to launch a malicious application, such as Spectre, that intends to steal sensitive information. The mechanism offers the benefit of attack injection without explicitly writing to victim's memory and using existing code in the memory. Mere integration of code-reuse attack with Spectre can be still vulnerable to the existing defense techniques [124–128]. To alleviate the detection, we pivot the proposed *code-reuse Spectre (CR-Spectre)* attack on the ROP injection and dynamic adaptation to keep the malicious behavior undetected by existing detection techniques. The dynamic adaptation in the CR-Spectre generates perturbations, thus contaminating the HPC generated by the host with the injected malicious application. These dynamic patterns intend to degrade HID performance, forcing misclassification of the attack.

The advantage of the proposed CR-Spectre compared to other Spectre variants [6, 140, 141] is its distributed nature and the capability to be a moving target for the defender, especially the ML-based solutions such as [124, 125, 127, 128].

Our proposed attack is capable of extracting secret information from an application while evading ML-based detection. This work evaluates the proposed attack on a Hardware-based intrusion detection system (HID) utilizing machine-learning (ML) [124–126, 142], that provides inference based on unique application traces, hardware performance counters (HPC), rendered by hardware performance monitoring unit (PMU).

In summary, the essential contributions of this work are:

1. Propose CR-Spectre attack, capable of executing under the cloak of a benign (white-listed) application as a launching mechanism.

2. A dynamic attack capable of modifying microarchitectural state to render the attack more robust against an HID is introduced.

3. A thorough evaluation of the performance of CR-Spectre under different scenarios are presented. Another aspect is to evaluate the overhead of the proposed attack.

We present our results and evaluation of the proposed attack and demonstrate that CR-Spectre can help degrade HID performance, thus misclassifying benign from an attack application. Experiments are performed using Spectre [6], and MiBench benchmark suite [143]. In our experiments, the HID performance degrades from 90% to 16%, indicating our CR-Spectre attack evades detection successfully.

4.1.1 Proposed CR-Spectre Attack

4.1.1.1 Threat Model

There exists an adversary that intends to steal secret data from an application that processes sensitive data. The adversary employs attack code to steal secret data from the target application (target). The exploited victim (host application) is the application into which the adversary injects the attack - the attack refers to CR-spectre - and the intention is to steal data from the target application (target). For the demonstration of the attack, we keep the *secret* as an array that is stored in the host application; the host never accesses the secret. The CR-Spectre attempts to read the *secret* in the array. Similar to [6], we assume that the adversary knows the address of the *secret* processed by target. The adversary has no special or root privileges to execute the attack. CR-Spectre is tested on HIDs that are inspired from the recent works presented in [124–126, 144], all of which utilize the HPC information for training the machine learning classifiers. CR-Spectre attempts to inject malicious code to steal secret information from the target while evading HID detection using perturbations. For an ROP-chain attack to function, there needs to be a mechanism to overflow the buffer and rewrite the stack contents. Hence, it is a prerequisite that the host application makes a write operation to the buffer, controlled by the adversary.

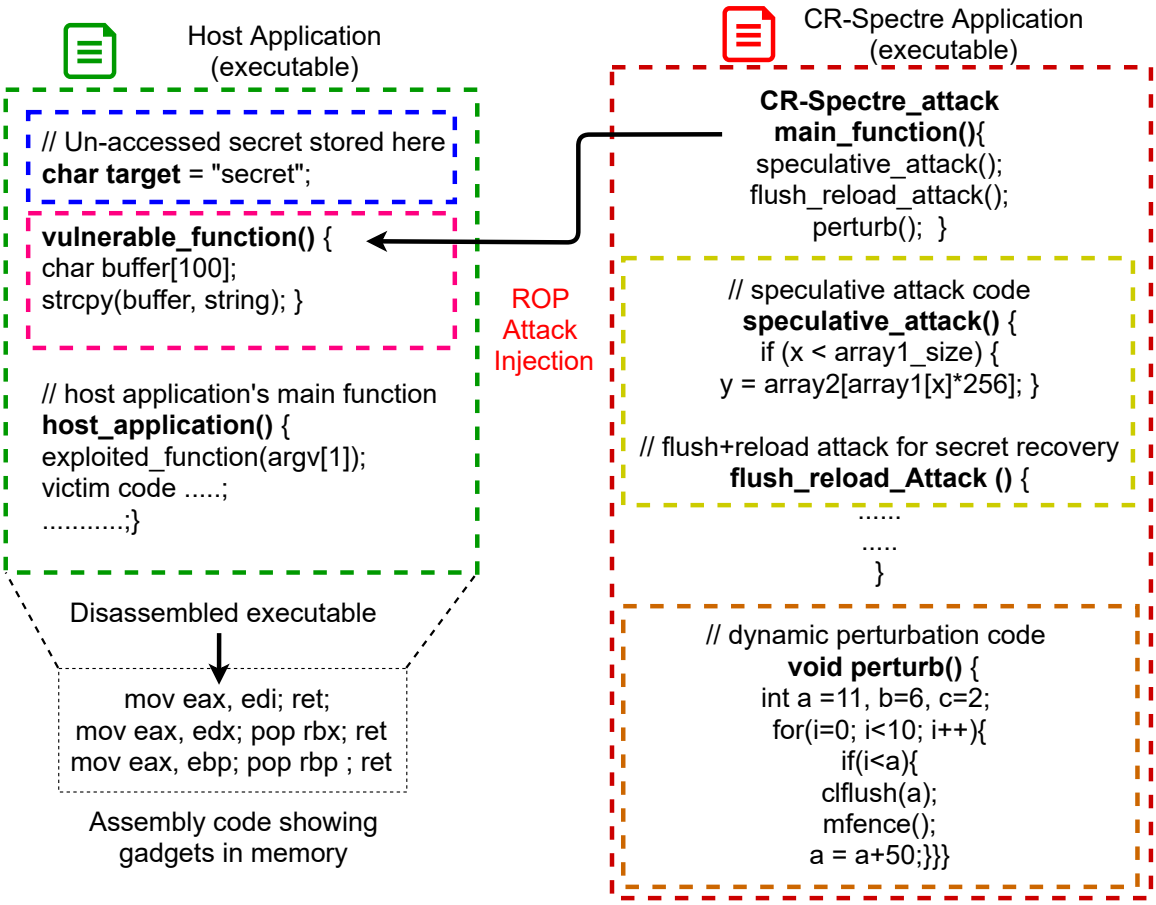


Figure 4.1: CR-Spectre program flow

4.1.1.2 Overview of the Proposed CR-Spectre

Here, we explain the overview of the attack injection, dynamic perturbations, and HID for detection. Figure 4.1 shows the attack process flow depicting various aspects of the attack. There are five aspects to CR-Spectre attack - host, vulnerable code fragment, target, speculative attack code, and dynamic perturbations. The host is the application to which the malicious code (Spectre) is injected. The adversary attempts to access the *secret* stored in the target application. The vulnerable code is the host application's code

fragment that serves as a point-of-entry for the ROP attack. The speculative attack code exploits the vulnerability in computing systems to access unauthorized memory locations. At the same time, the dynamic perturbation is proposed to contaminate the victim's (host) and speculative attack's HPCs to degrade HID performance. The secret (target) is stored in the same application as the host, sharing the memory space, but it can be contained as a standalone application. The proposed attack is initiated with the knowledge of the vulnerability in the host. In our case, we utilize buffer overflow vulnerability to launch the ROP attack.

The host expects a string of a certain length, and it is stored in a buffer. The ROP attack is deployed by passing to a host a string that exceeds the buffer's capacity/length. This overwrites the contents in the stack space, which corrupts the return address of the calling function - "victim_application()" in Algorithm 2. The string passed to the host also contains arguments that will be needed by the 'execve' as its argument, for example, the address of the malicious binary. Addresses of the ROP gadgets are also provided as the input string to the host. Hence, the host returns to a series of gadgets carefully chosen to make an 'execve' system call and inject the malicious binary. After injecting the malicious binary, speculative execution application, it will attempt to access the secret in the target. The address of the secret in the target is known to the adversary. The computing system is protected by HID, which samples the HPCs of applications executing on the system in runtime.

The HID can detect the speculative attack explained above with high accuracy. Hence, it becomes necessary to conceal the attack. We propose to conceal by introducing dynamic perturbations. The dynamic perturbations are generated by calling functions containing a couple of 'if' loops that execute based on the values of the attack parameters (variables in the loop). The *clflush* and the *mfence* instructions ensure that the data is flushed each time the function executes to cause variations in the HPC patterns. The perturbations can be modified dynamically by varying the parameters; thus, each generated variant producing a different HPC pattern. The dynamic perturbation is discussed in detail in Section

4.1.1.5 and Algorithm 4. With the CR-Spectre, the HPCs of both the host and attack are contaminated, leading to performance degradation of the HID, thus evading the defense.

4.1.1.3 CR-Spectre: Attack Methodology and Gadget Generation

Referring to Section 4.2, and for conciseness, we present our proposed CR-Spectre by considering a simple application as shown in Algorithm 2 that stores the string provided as an argument in the buffer. The *victim_application* is the *main* function of the host application. eFor our experiments, we utilize the MiBench suite as the host; any other application could be used as a host, as the proposed technique is not bound to host application. We load the compiled victim binary in the Linux Debugger (GDB) to search for all instructions that end in a *ret* instruction. We then carefully chose instructions such that by chaining them together, the ROP attack makes a system call, executing the malicious attack. Due to Data Execution Prevention (DEP), system-level memory protection that marks stack and heap as non-executable, we cannot write malicious code to the victim's memory; an ROP-chain attack utilizes existing code in the victim's memory to evade a DEP protection.

As the existing code in memory is marked executable, the aim is to setup the stack memory such that the sequential execution (chain) of gadgets executes a system call, "execve" in this case, which takes the path name of the CR-Spectre binary as an argument. The ROP attack exploits a vulnerable function, *exploited_function*, to serve as an entry point for the attack. However, the proposed attack with ROP-chain [129, 131] can be extended to any victim where a return address can be manipulated to execute a gadget.

In Algorithm 2, the buffer overflow manipulates the return address, replacing it with an address of a gadget. Likewise, all the addresses of the necessary gadgets are provided as arguments to the vulnerable function, thereby chaining them to execute the CR-Spectre binary using the system call. A binary compiled using GCC has various other libraries linked with it, thus providing more gadgets than available only with the host. With sufficient gadgets, there exist innumerable possibilities of what could be executed within the victim [131]. The content of the argument, as shown in Listing 4.2, is 108 bytes (0x6C) of random

Algorithm 2 Pseudocode for code reuse attack on victim

```
1: vulnerable_function(char* string) {
2:         char buffer[100];
3:         strcpy(buffer, string); // ROP attack injection with
                                   buffer overflow exploit }
4: host_application(int argc, char** argv) {
5:         exploited_function(argv[1]);
6:         victim code line 2 ...
7:         victim code line 3 ...
8:         victim code line 4 ...
9:         victim code line 5 ...
10:        return 0; }
```

data (all 'D's along with four bytes of 'FFFF' used to fill the buffer), followed by the address of *execve* function, followed by four bytes (ABCD), finally followed by the address of the Spectre binary.

Listing 4.1: Attack payload passed as argument for ROP attack

```
./victim_function "$(python -c 'print"D"*0x6C
+ "FFFF" + "address of system"
+ "ABCD" + "address of attack function"')"
```

For conciseness, we omit to show all the addresses of the gadgets accessed before finally making the system call. A working example of the code-reuse ROP attack is available on our anonymous repository[2]. The argument essentially fills in all the space in the buffer, shown in Algorithm 2, overwrites the return address to the address of the gadget in memory, address of the second gadget in the chain, and so on. Finally, it is followed by the "execve" gadget address and the address of the CR-Spectre binary executable, which is external to the host application. Hence, the host can execute a malicious code without writing to its memory, utilizing the gadgets already in the memory. The attack code is not contained in the host's code segment, instead it is injected in runtime; hence the HID cannot abort it by analyzing offline.

[2]https://github.com/hartanonymous3512/CR-Spectre

Algorithm 3 Pseudocode for generating dynamic perturbations for the CR-Spectre attack

```
1: void perturb() {
2:    int a =11, b=6;
3:    for(i=0;i<10;i++) {
4:       if(i < a) {
5:          cflush(address(a));
6:          mfence();
7:          a = a+50; }
8:
9:       if(i < b) {
10:         cflush(address(b));
11:         mfence();
12:         b = b+10;
13:         cflush(address(b));
14:         mfence();
15:         b = b-10; }
16:         .....More loops can be added here.....
17:                   }
18:               }
```

4.1.1.4 Attacking HID

Figure 4.2(a) shows how CR-Spectre attacks the HID. The CR-Spectre code is injected[3] into the host application. During execution, the application is profiled by the detector to record performance counters in runtime [124,125,127]. The HID monitors the recorded traces, and inference is provided - attack or benign. The decision is measured in terms of accuracy over time. The HID performance is discussed in Section 4.2.5. For an HID, a higher accuracy refers to distinguishing between benign and attack situations more accurately. The purpose of the proposed CR-Spectre is to degrade the performance of the HID to evade detection. Figure 4.2(b),(c) visually present the difference between traditional Spectre and CR-Spectre. In (b), the adversary exploits attack code to steal secret data from the target application. On the contrary, as shown in (c), CR-Spectre injects the malicious code in a host (benign) application and executes it under the umbrella of the host.

4.1.1.5 Defense-aware Dynamic Perturbation Generation

With the previously explained attack methodology, there can exist scenarios where the attack cannot evade the detection because the HID can learn, online learning or retraining, or

[3]Injection refers to the ROP attack that subverts the control of the host application forcing it to execute a malicious code.

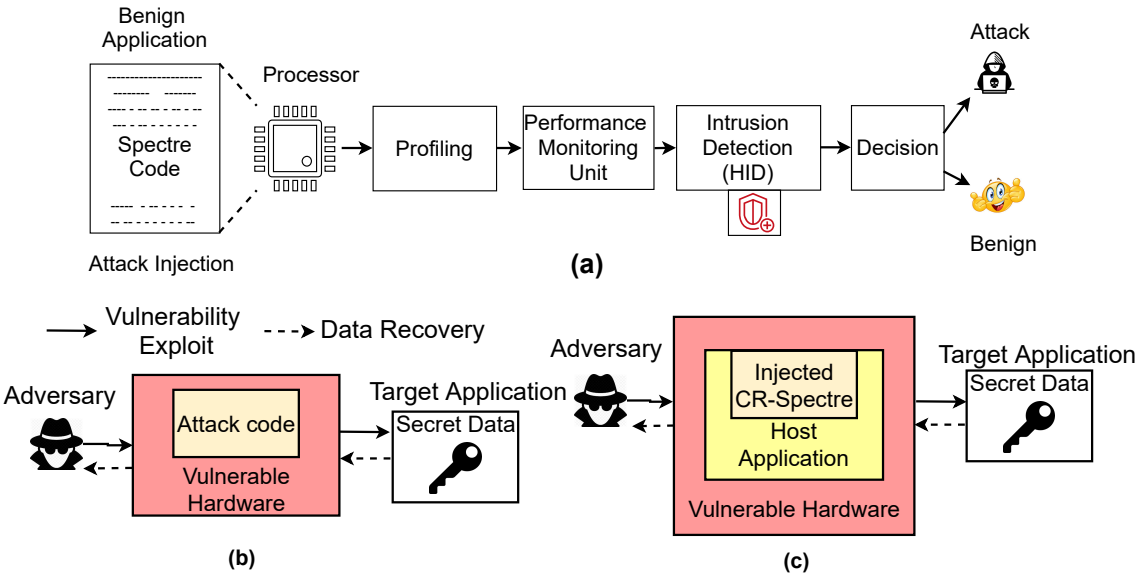


Figure 4.2: (a) Code injection of Spectre attack to evaluate HID performance, (b) Traditional Spectre attack strategy, (c) CR-Spectre attack strategy

the application can be tagged as an attack by the human-in-the-loop. Online learning type HIDs are retrained on the augmented dataset, the profiled HPC patterns of the applications during the runtime for robust threat detection. To add better evasion despite having online learning capable HIDs, we propose dynamicity in the CR-Spectre attack injected through ROP discussed in Section 4.1.1.3. This affects the microarchitectural behavior of the application such that the monitored information by the HID can be different from the traces on which the HID is trained.

Figure 4.3 shows the process of the attack and generation of perturbed variations. CR-Spectre generates a perturbed version of the speculative attack code and injects it into the host. The applications, benign and the CR-Spectre attack, are profiled to record performance counters (HPCs). The profiled traces are fed to the ML-based HID. The HID provides the inference with a certain accuracy, indicating if the attack is detected or not. For the attack to evade the HID detector, we consider accuracy of 55% or less. Suppose

the HID inference result accuracy is less than 55%. In that case, the attack successfully degrades the detector performance while the malicious attack steals secret data from the target. If the HID detects the attack with high accuracy (>80%), we consider that the attack was detected. The HID performs realtime profiling of the applications executing on the system [124, 125, 127].

Upon detecting the CR-Spectre attack, we modify the perturbation code's parameters to generate a variant, the HPC traces of which differ from the previous variant. A variant is generated by modifying the attack parameters like the loop count and operation variables, 'a' and 'b', as shown in Line 2 of Algorithm 4. The parameters are utilized in the algorithm as shown in Lines 4, 7, 9, 12, 15, 17, 20, and 23. The parameters affect the *clflush* instruction; hence it varies the HPC patterns as well. With different attack parameters, the generated HPC patterns are modified. The attack process is repeated to steal secrets from the target. The benign applications running on the system are also profiled and fed to the HID. This is necessary because, in a real-world situation, the system executes multiple applications. Hence, we profile applications like browsers, text editors, etc., and train the HID to emulate a practical situation. The code shown in Algorithm 4 is called from within the malicious code, Spectre.

The *cflush* on the arithmetic operation triggers a *cache_miss* and affects other hardware events such as those related to branch prediction, the number of instructions executed, and the cache access cycles. The *mfence* instruction ensures that the previous operation, clflush, completes before proceeding with the operation below. The data recovery process is elaborated in [6]. For conciseness, we only discussed situations where the generated perturbations (HPC) increase in magnitude. Nevertheless, we can use a delay loop to disperse generated perturbations, thus distributing them in time. In this manner, the generated HPC patterns can also reduce in magnitude.

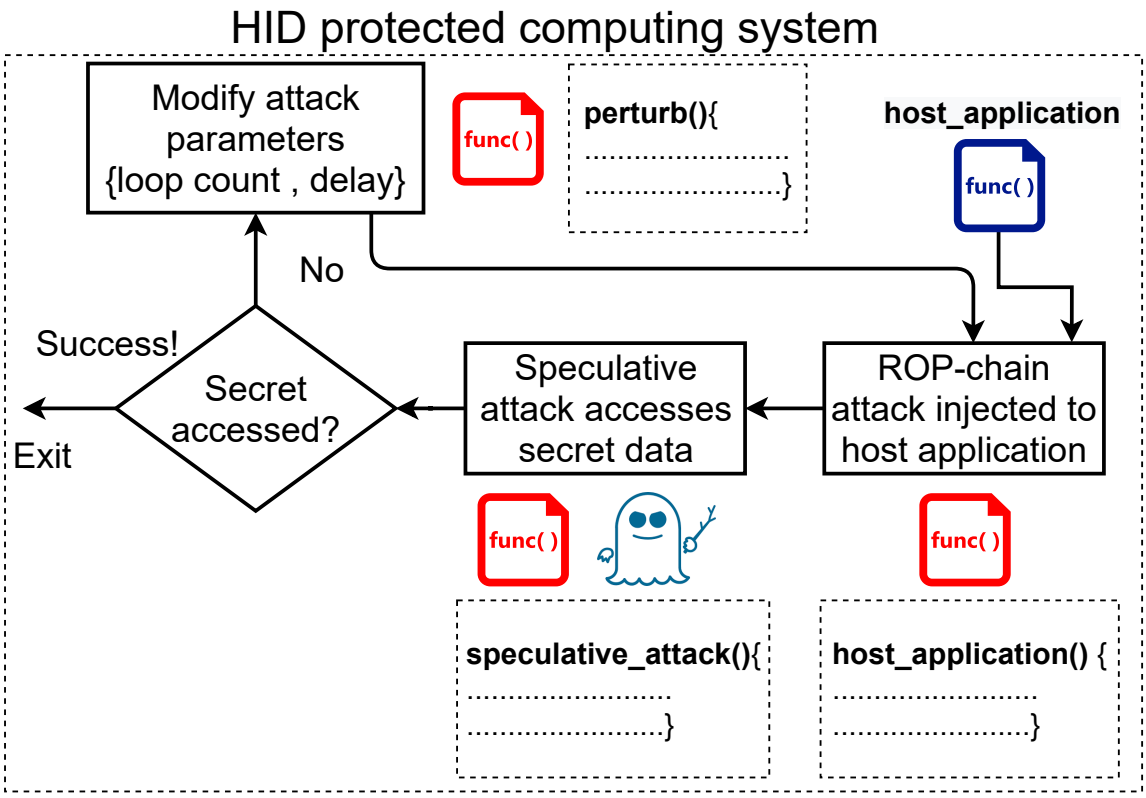


Figure 4.3: Process flow of the ROP attack and generation of perturbed code

4.1.2 Results and Evaluation

4.1.2.1 Experimental setup

MiBench [143], Spectre [6] are used as the host and malicious attack applications, respectively. The CR-Spectre attack is not limited to the applications reported here, but it can exploit other vulnerable applications, thus reading a specified unauthorized memory location in the system. PAPI-based profiling tool [127] is utilized for recording the performance monitoring unit's output, hardware performance counters (HPCs). All experiments, application profiling, ROP attacks are executed on Ubuntu 18.04 running on an Intel i5 with 16 GB RAM. We collect a total of 2000 samples for each class, CR-Spectre, and host; the scope

of applications profiled also includes the host and other benign applications like browsers, text editors, etc. The training and testing datasets are separated in a ratio of 70/30. e We evaluate HID performance on MLP (Sklearn) [124], Neural Network (NN) from Tensorflow [125, 126], Logistic Regression (LR) [124, 125], and a SVM [124, 125] classifier. The HIDs' parameters are as follows: the neural networks have 6-layers using 'Relu' activation; SVM classifier uses a linear kernel for classification; the MLP is 3-layer network-based classifier. The parameters for the hidden layers are determined experimentally. We choose the parameters that deliver high accuracy in detecting CR-Spectre from other applications profiled. Features used for the training are 'total cache misses', 'total cache accesses', 'total branch instructions', 'branch mispredictions', 'total number of instructions', and 'total cycles'. The first five features are affected by Spectre attack as presented in works [124, 125]. The last feature is utilized for the IPC metric for overhead analysis.

4.1.2.2 HID Performance on Spectre Detection

Figure 4.4 shows the performance (accuracy) of the HID in detecting/differentiating the benign (host and other applications) and Spectre applications. The HID is inspired from [124–126] and utilizes similar features for Spectre detection. The features fed to HID are the recorded performance events (HPCs). We collect a total of 56 performance events available on the system (offline). For real-time monitoring of the events, a limit is imposed on the number of events counted simultaneously. Hence, we present the results with multiple feature sizes (1, 2, 4, 8, and 16) to show the efficiency of the HID system deployed in this work for Spectre detection. Figure 4.4 shows the performance of the HID in differentiating MiBench and Spectre [6] applications. We experimented with different variants of the Spectre attack, discussed in [140, 141]. The accuracy shown in the figure is the average of the variants of Spectre. The legend `Spectre_1` indicates the performance in classification of Spectre - and other variants averaged - and MiBench application-1 (Math application as listed in Table 4.1), similarly we can interpret other legends. Performance with a few MiBench applications in Figure 4.4 are shown for conciseness. As seen, accuracy of more

than 80% for feature sizes 16, 8, 4, and 2 in Spectre detection irrespective of the MiBench application used. However, using only one feature for classification is inefficient due to its inability to capture the HPC variations in a single feature. To alleviate the monitoring and computational overheads, we consider utilizing 4 features in this work for Spectre detection that lead to >90% accuracy on average. For the rest of the article, we consider a feature size of 4 which can be recorded in runtime on modern processors [125].

4.1.2.3 Does CR-Spectre Evade HID?

Figures 4.5 and 4.6 present the HID performance under attack. The accuracy metric is plotted against the number of CR-Spectre attack attempts over time. We study two scenarios for the attack, offline and online learning HID. The offline learning HID is a static type that does not retrain itself (or retrained by the defender) during runtime, i.e., similar to the [142]. On the contrary, we deploy an online learning version of the HIDs which are retrained during runtime on newer traces to enhance attack detection capability on unseen data.

Figure 4.5 presents the offline-type HID performance for original Spectre and CR-Spectre with HID using different types of ML classifiers. In Figure 4.5(a), it is seen that the original Spectre attack is detected with high accuracy by the HID detector implemented with different ML classifiers. The accuracy variations are observed due to the variations in the recorded HPC traces during each attack attempt. Whereas, in Figure 4.5(b), the performance of HID degrades with perturbed instances of the attack. The accuracy shows a degrading trend as the offline HID is employed. It is to be noted that we do not generate dynamic perturbations for an offline-type HID. The reason being the offline-type does not 'learn' or retrain itself on newer traces. Hence, to save the overhead, CR-Spectre only generates one variation of perturbation but does not modify the attack parameters dynamically every time it attacks the HID.

Similarly, from Figure 4.6(a), it is observed that the HID detects Spectre with high accuracy. The patterns are leveled compared to Figure 4.5(a), as the online-type HID, by

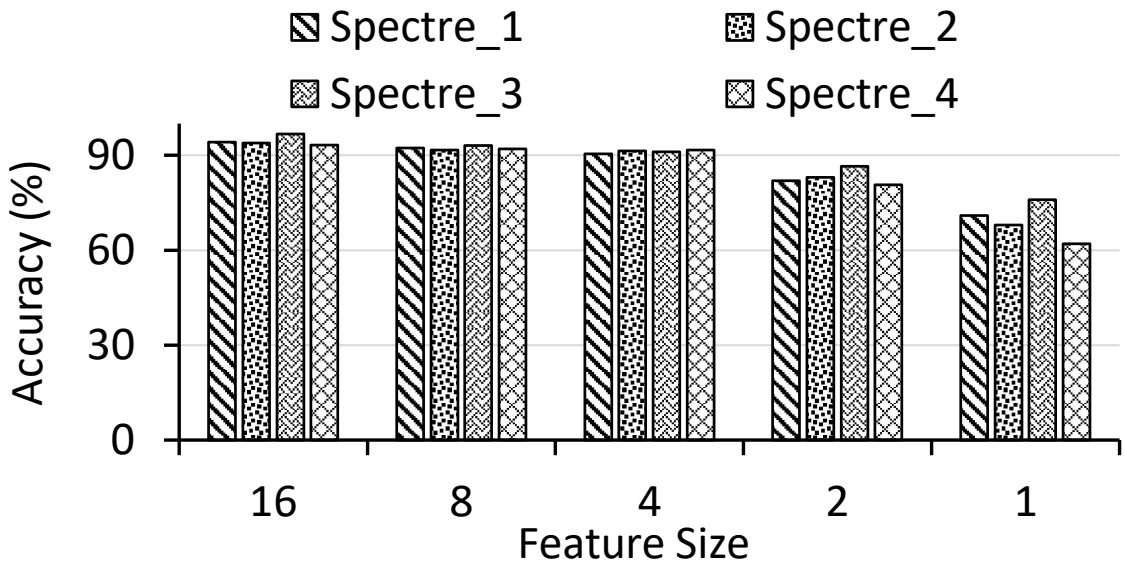


Figure 4.4: HID performance for four benign (host) applications and original Spectre attack studied for different feature sizes

retraining itself on new traces, hence becomes more robust to HPC trace variations during the recording phase. A degrading trend is again observed for the HID performance in Figure 4.6(b). The exception is that the HID attempts to boost the detection performance owing to retraining. Yet, with the introduced dynamic perturbations, the CR-Spectre performs well in degrading the HID detection accuracy to less than 55% to the lowest observed accuracy of 16% in our experiments. Under the cloak of such degraded performance, the speculative attack recovers the *secret* data from the target.

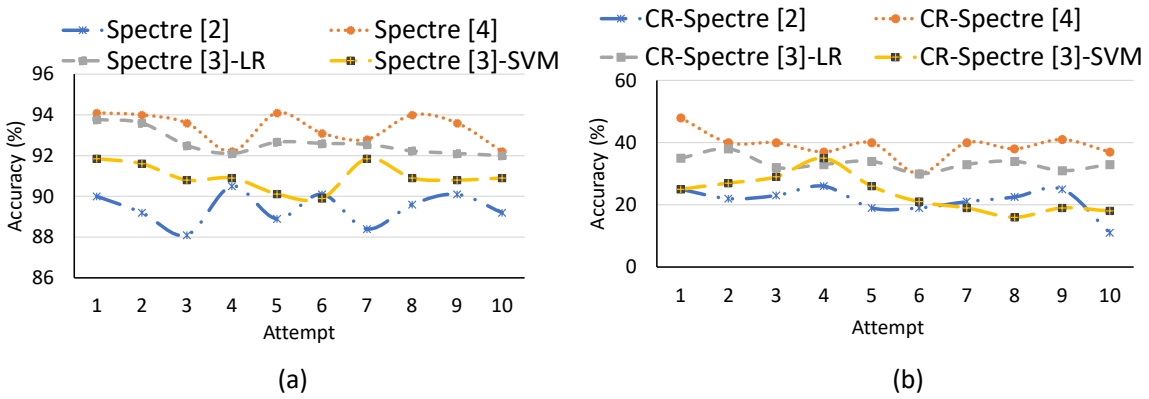


Figure 4.5: Comparison of offline-type HID performance with Spectre and CR-Spectre attack

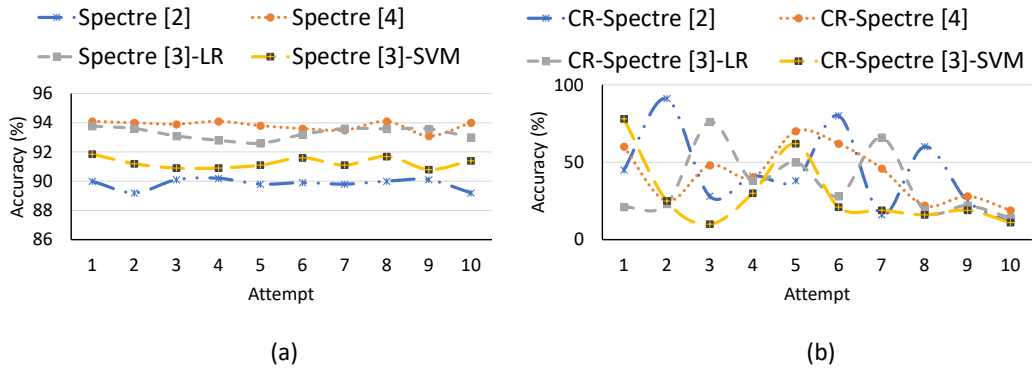


Figure 4.6: Comparison of online-type HID performance with Spectre and CR-Spectre attack

4.1.2.4 Overhead analysis

We perform overhead analysis of CR-Spectre by evaluating different applications in the MiBench suite. We select instructions per cycle (IPC) as an evaluation metric. Latency is also a metric other works [124–126, 142] have utilized. However, the latency metric could be

counterproductive due to system noise. The noise is caused by other applications and the operating system running in the background. Furthermore, IPC is also considered as a trait of the application in determining the presence of abnormalities or stalls in the application. We mitigate trace fluctuations by averaging the values by iterating the same application 100 times.

Table 4.1: Performance Overhead in Evaluated Benchmarks

Benchmark	Original Application (IPC)	CR-Spectre with offline-type HID (IPC)	CR-Spectre with online-type HID (IPC)
Math	1.9419	1.88	1.865
Bitcount 50M	3.041	3.05	3.031
Bitcount 100M	3.052	3.051	3.041
SHA_1	0.736	0.742	0.73
SHA_2	0.814	0.819	0.80

We report IPC values for the original application (without CR-Spectre), the offline execution of CR-Spectre, and the online execution of CR-Spectre. The aim is to deliver performance with negligible overhead. The overheads are reported in Table 4.1. For the Math application (math_small and math_large applications averaged), the IPCs observed are 1.94, 1.88, and 1.865 for the original, offline, and online execution. Similarly, for the Bitcount with 50M operations, the IPCs are 3.041, 3.05, and 3.031, respectively. And for the SHA cryptographic algorithm, it is 0.814, 0.819, and 0.818, respectively. Again, we average the values to cover for variations. The overhead average for the offline-type and online type is 0.6% and 1.1%, respectively, compared to the Spectre-only attack without dynamic perturbations and ROP attack injection.

4.1.3 Countermeasures

It is crucial to discuss the countermeasures for the proposed CR-Spectre attack to help mitigate potential security threats. Disable *clflush* and *mfence* instructions for non-privileged

processes, thus disabling dynamic perturbations; Accompanying automatic HID detection with manual inspection of processes that might be vulnerable to ROP/Buffer-overflow exploits; Using a shadow memory -only accessible to the operating system - to compare and correct when return address manipulation takes place. However, in-depth analysis and verification are needed to evaluate the robustness against the proposed CR-Spectre.

4.2 Covert-Enigma: Defense Against Side-Channel Attack on Crypto-Systems

Modern computing systems require designers to embed novel features to satisfy the ever exploding need for high performance and efficient systems. Regardless of evolved features, such as speculative execution, three-level cache architecture, memory sharing/deduplication, etc., the computing systems are vulnerable to security threats, also known as side-channel attacks (SCAs). A plethora of past research works has focused on the vulnerabilities in the systems. Some of the works on the vulnerabilities and their exploits are: malware [49, 145, 146], reverse engineering of hardware [147, 148]; attacks on machine learning-based malware detectors [11, 50, 149, 150], cache-based side-channel attacks [48, 151, 152]. SCAs exploit the architectural vulnerabilities, such as timing, power, frequency, etc., in the victim application. By exploiting such vulnerabilities, the SCAs attempt to steal confidential data from sensitive applications. There have been a rapid increase in the cache-targeted SCA [21, 22, 153]. Computing systems require cache to achieve performance gains. Hence, almost all the applications (sensitive or generic) utilize cache for storing recently accessed memory locations. For instance, cache-targeted SCAs rely cache-access patterns - hit or miss - to determine the recently accessed location(s) [7, 8, 154–156]. By studying such patterns that serve as a covert channel leaking sensitive information, the attacker can determine the recently accessed location, hence the secret information. To thwart such emerging threats, our work focuses on defending against cache targeted SCAs.

The unsolved challenges and limitations of the existing defenses can be outlined as follows: a) suggested hardware or software modifications might not be feasible to adapt; and b) VM[4] (virtual machine) migration -based mitigation are resource hungry strategies, and contribute to a significant timing overheads.

To overcome the limitations of previous works [3–5] and thwart SCAs, we introduce

[4]In a multi-tenanted cloud environment, the operating system (along with the victim application) is moved and executed on another physical hardware, disallowing the co-location of victim and the attacker OS.

Covert-Enigma, a defense for timing-based SCAs. In contrast to the previously mentioned existing works that focus on architectural changes, the proposed Covert-Enigma primarily focuses on maximizing the entropy[5] of the side-channel information obtained by the attacker without interfering with the original functionality of the victim application. In the Covert-Enigma, the original application is coupled with a protective application (wrapper) that induces *cognitive* perturbations in the cache-access information obtained by the attacker.

In contrast to the existing randomization techniques, proposed Covert-Enigma introduces randomization under the constraint that the archived information by attacker looks legit and similar to the normal information, yet leading to a wrong key. In Covert-Enigma, we induce cognitive perturbations in the (security-sensitive) applications' operations by executing dummy instructions that leave the victim's functionality unaltered yet scrambling the sequence observed by the attacker. These induced cognitive perturbations mislead the information retrieved by the attacker, thereby thwarting the attack. Our proposed Covert-Enigma tenders user-tunable parameters such as the length of successive bits to modify and the cycle frequency, where the number of cycles can be chosen after which the proposed method cognitively selects the other set of bits to perturb. This offers the user to adjust the level of complexity of the injected perturbations. Arbitrary and Cyclic are two operational modes that a user can select, and the details are discussed in other sections. The arbitrary mode offers one or more bits to be cognitively perturbed in the sequence of operations chosen at runtime, whereas the Cyclic mode chooses bit(s) and then keeps perturbing[6] same bits[7] for few executions as determined by the user, post which the position changes. The Cyclic mode is advantageous when the attacker suspects a defense mechanism is in place and tries to repeatedly execute the user application to ascertain the static part of the sequence - as seen by the attacker - which are added perturbation(s). We want to emphasize that, in this work, 'entropy-maximization' refers to a reduction in the useful information obtained by an

[5]We define Entropy as the amount of randomness in the obtained data. The less entropy information has, the easier it is to decrypt the data.

[6]Perturbation or cognitive calls refer to dummy cache accesses that leads to meaningful decryption, yet incorrect

[7]Bit here refers to the bit in the secret information. Bit position refers to the bit in the stream of secret information to be protected as observed by the adversary

attacker by increasing cognitive randomness over side-channels to decrypt the secret key. The proposed Covert-Enigma technique is thoroughly evaluated against active and passive cache-targeted SCAs with victim applications utilizing different keys.

The cardinal contributions of this work are:

- Contrary to the existing works, the proposed Covert-Enigma enforces security on the covert channel by injecting cognitively crafted perturbations that imitate legit operations yet mislead the attacker.
- Render the attack more time-consuming (in terms of the iterations it takes to break the defense) by providing two modes of operation, Arbitrary and Cyclic, thus offering more flexibility in terms of the defense.
- Evaluate and compare the benefits of the proposed Covert-Enigma in terms of overhead and performance based on the key size, mode of operation, user-tunable parameters, and the number of bits recovered post-attack on the victim.

The rest of the work is organized as follows. Section 4.2.1 provides the working principle of the flush+reload and flush+flush type side-channel attacks followed by the vulnerability in encryption application. Section 4.2.2 describes the proposed defense, threat model, generation of cognitive perturbations, and modes of operation of the proposed Covert-Enigma. This is followed by Section 4.2.5 which includes the validation process, recovery of sensitive data under SCA attacks (without the presence of Covert-Enigma), the behavior of Covert-Enigma under attack, the performance of Covert-Enigma and the overhead analysis. Section 4.2.6 describes the motivation supporting the proposed idea as a case study, followed by the state-of-the-art in Section 2.2.

4.2.1 Side-Channel Attacks: Background

This section will briefly introduce the SCAs on which the evaluation of proposed Covert-Enigma is performed along with some previous works.

4.2.1.1 Side-Channel Attacks

Flush+Reload Attack Flush+Reload is a prominent cache targeted SCAs that utilizes the cache-access timing information to retrieve the key. The process of Flush+Reload attack is performed in three steps, as follows: *Step 1:* The attacker (spy) flushes a memory line in the (shared) cache. *Step 2:* Spy waits for a certain amount of time (to let the victim access the cache). *Step 3:* After the timeout, the spy reloads the data into the cache and observes the access time to determine whether the cache line was accessed by the victim or not and in Figure 4.7(a).

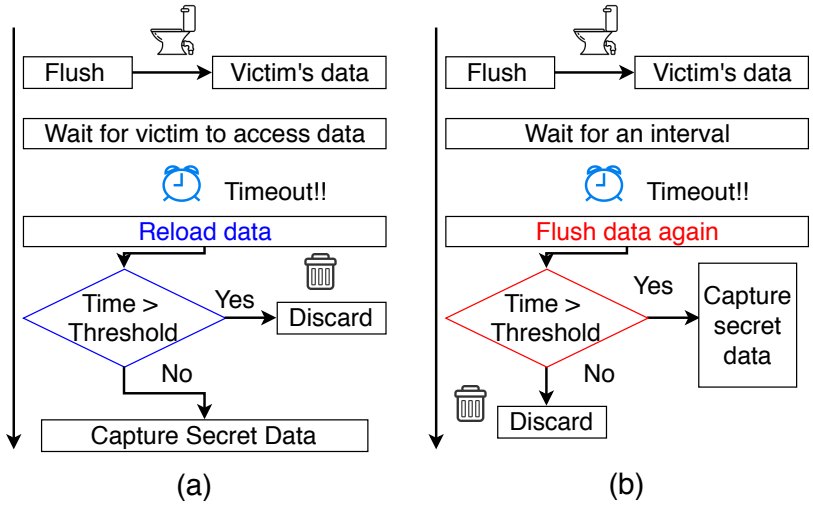


Figure 4.7: (a) Flush+Reload attack: the spy (attacker) flushes the data and waits to determine whether victim accessed the flushed line or not; (b) Flush+Flush attack: the spy (attacker) flushes victim data, waits for a short interval and re-flushes the same location to observe the time it takes to flush the data, thus, deciding if the data was accessed by the victim

Thus, the Flush+Reload attack can be inferred as follows: if there was a cache hit for the spy application indicates that the cache line (data) was accessed (and fetched) by the victim application, else the victim does not utilize the data. For instance, the encryption algorithms such as GnuPG's RSA encryption use a sequence of the square, reduce and

multiply operations to calculate the private key's exponent. Utilizing the Flush+Reload attack, depending on the cache hit/miss and the sequence of the Square, Modulo, and Multiply operations, the spy deduces if the bit in key was a logical '1' or '0'. By continuously repeating the above process, the attacker can retrieve the entire private key [7].

Flush+Flush Attack Flush+Flush attack [8] is a relatively advanced cache targeted attack that supersedes the above discussed Flush+Reload SCA both in terms of speed and stealthiness. The Flush+Flush attack is shown in Figure 4.7(b). Unlike the Flush+Reload attack, Flush+Flush is passive and works only by executing `clflush` instruction in an infinite loop. Unlike Flush+Reload, Flush+Flush attack does not access any data, the number of cache misses thus created are zero, and hence it becomes difficult to detect. When the `clflush` instruction is issued, data that is cached takes more time to be flushed out of the cache as it has to be evicted out across all cache levels completely as against non-cached data, which takes less time. Based on the execution time of the `clflush` the Flush+Flush attack concludes if the data was cached or not cached. The attack does not load any memory line into the cache, and hence if `clflush` takes more time to execute would imply that the victim accessed the data. Based on this strategy, the attack monitors the victim's activities by observing multiple cache lines or data of the victim.

4.2.1.2 GnuPG Encryption

In the previous subsection, we studied the SCAs, and here we describe briefly the victim application that we utilize as a case study to analyze the impact of proposed Covert-Enigma. GnuPG's public-key encryption (PKE) is a popular way of encrypting the data to maintain confidentiality and integrity. The PKE generates a pair of public and private keys, the width of which is decided by the user. Any document that is encrypted with a public key can only be decrypted with the corresponding private key. Let's say user A wants to send secret data to user B. In such a case, B will have its own public and private key, of which the public key will be made available to user A. User A will use user B's public key to encrypt the secret data and send the encrypted data to user B. To reveal the secret data, user B will

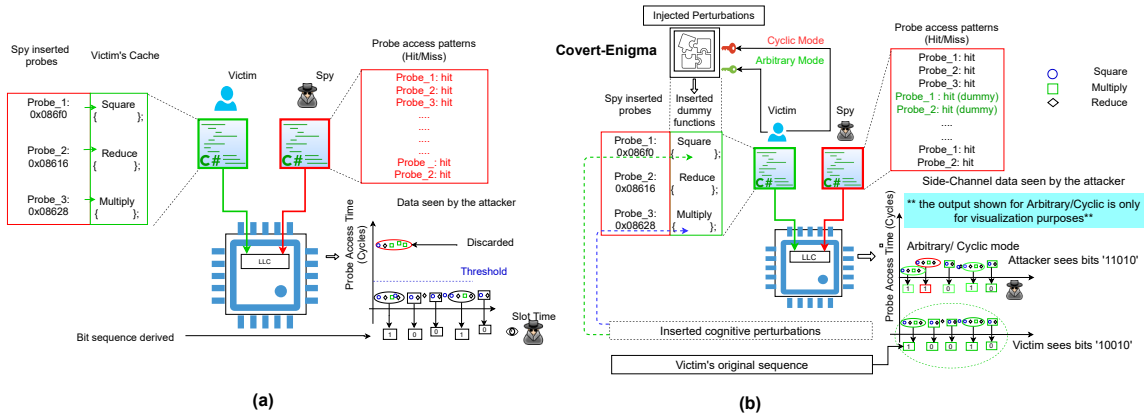


Figure 4.8: (a) Traditional side-channel attack on encryption algorithm where the data leaked via covert channel is accessible to the attacker; (b) Victim wrapped with Covert-Enigma that injects perturbation during run-time to perturb the sensitive information leaked thereby making SCA time-consuming. *the output shown is only for visualization purpose*

decrypt the file with its private key. The way the RSA algorithm is implemented, it is nearly impossible to brute force an encrypted file if the width of the secret keys is large enough and also due to the known fact that users (both legitimate and attackers) have no access to the RSA algorithm directly while it is in the process of encryption and decryption. This might have been true until a few years ago, but not anymore due to the state-of-the-art SCAs that have successfully broken the keys' secrecy, thereby rendering the PKEs vulnerable to attackers. We have discussed the GnuPG's implementation of the RSA and the DSA (with Elgamal) type encryption methods in this work. A series of complex calculations compute the private keys, and the exponent is what the attackers try to target. Once the exponent is captured over the covert channel, the algorithm can be easily broken. Work in [7] vividly describes how side-channel attack can be used to spy on victim's (RSA) operations and thus steal secret data.

4.2.2 Design and Implementation of Covert-Enigma

In this section, we will first discuss the challenges that need to be addressed to deploy a successful SCA defense. Further, we present the attack model used in many of the existing works.

Some of the cardinal challenges designers face while incubating any defense in place to protect the victim are: The defense mechanism should serve as a transparent shield and does not alter the victim application's functionality. Second, the attacker executes the application for a large number of times to reduce noise in the channel while trying to capture the desired secret information. In such a scenario, the defense mechanism must ensure that the victim application is guarded against such attack methodology while reducing useful information leaked to the attacker. Lastly, but crucial, the defense mechanism must not significantly add overhead to the system while trying to protect the victim application. Covert-Enigma draws inspiration from adversarial learning [157] where we introduce pixel-level perturbation for forcing misclassifications on the attacker's end. In Covert-Enigma we perturb the cache-access sequence by using cognitive operations to mislead the attacker.

```
function Square(){.......
        Probe 1 //  Address 0x086f0
        .......... }

function Multiply(){.......
        Probe 2 // Address 0x08628
        .......... }

function Modulo/Reduce(){.......
        Probe 3 // Address 0x08616
        .......... }
```

Listing 4.2: Spy inserts probes to monitor targeted vulnerable functions in the victim

4.2.2.1 The Attack Model

We assume an adversary whose intention is to steal the confidential data that the victim is processing. For the Flush+Reload and Flush+Flush attack to work, sharing the cache space with the victim is a prerequisite. The spy does not need access to privileged execution mode; instructions such as *clflush* are allowed for user-level processes. The Covert-Enigma does not require superuser privileges as well. It is realistic to assume that the spy knows the addresses to monitor the victim. The Covert-Enigma has similar knowledge of the same addresses of interest to shield the victim against the attacker [7]. The spy can execute on any core as the last-level cache (LLC) is shared across all the cores. Given the attack happens in a real-world setting, we assume that the adversary does not have the right/control to execute the victim at the same time as the adversary, but rather, the adversary can observe a part of the victim's execution during each run. Also, referring to [7], the adversary cannot capture successive victim cache accesses that happen before the next probe (monitored addresses) monitoring cycle.

4.2.2.2 Side-Channel Attack Without Covert-Enigma

Figure 4.17(a) shows the working methodology of traditional Flush+Reload attack to spy on a victim (encryption) application to reveal the secret key. The spy inserts probes at the function addresses of non-trivial functions such as square, modulo, and the multiply operations as these are repetitive and their sequence determine the data flow and reveals the secret key bits - this is how the existing cache-targeted SCAs [7,8] function. In the case of Flush+Reload attack, the spy (attacker) constantly flushes the addresses at probed locations and monitors if the victim accesses the flushed lines. The process of probing the square, multiply, and modulo/reduce encryption functions by the attacker is shown in Listing 4.2. Referring to Figure 4.17(a), the attacker is able to retrieve the secret information.

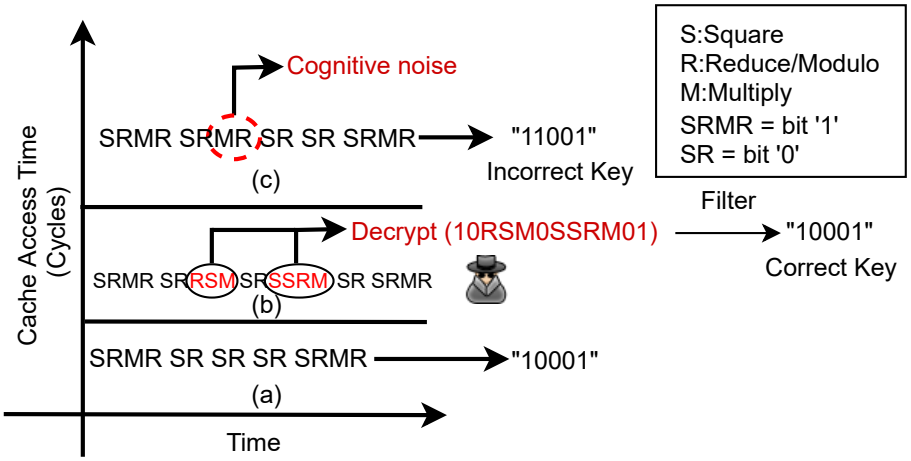


Figure 4.9: (a) Sequence of operations in RSA implementation that leaks secret data; (b) Random cache accesses [1–3]; (c) Cognitive perturbation injected in the observed side-channel data to secure the information

4.2.2.3 Covert-Enigma: Injecting Cognitive Perturbations

Introducing random operations[8] as in existing works to induce perturbations is not efficient as the attacker can filter out portion that does not contribute to the construction of secret data [1–3]. Figure 4.9 shows an example of cognitive perturbations injected during the victim's execution. Part (a) presents a sequence of operations that decodes to "10001". Part (b) shows random sequences injected, but these random calls do not make sense in the context of the secret data revealed by the victim. For example, adding a Reduce-Square-Multiply operation, as shown in 4.9(b), does make the secret data retrieval difficult, yet it does not force the attacker to translate a '0' bit to '1' or vice-versa. In other words, randomization does not lead to misinterpretation of the secret data and can be filtered out by the attacker. Hence, it is crucial to introduce perturbations cognitively that seem legit.

In this work, we consider RSA implementation [158] as the victim. For RSA, to induce

[8]By random, we mean that random injection of any fake/dummy cache access does not help much to mislead the attacker

cognitive perturbations, the Covert-Enigma makes cognitive calls to the code within the functions square, reduce and multiply, responsible for crypto-operation. For instance, a sequence of Square-Reduce operations corresponds to bit '0', whereas a sequence of Square-Reduce-Multiply-Reduce will correspond to bit '1' [7]. These operations are implemented as function calls in the GnuPG's encryption suite [158, 159]. Hence, by making a dummy function call to the Multiply followed by the Reduce function, the defense can pose as if the sequence corresponds to bit '1' whereas the actual secret bit was '0'. We term this phenomenon as the elevation of entropy. It is to be noted that the reverse operation holds true as well, injecting perturbations such that sequences corresponding to bit '1' are observed as a '0'. After cognitive perturbation, the series contains additional 'multiply' and 'reduce' operation, as shown in part (c). With these additional accesses, the sequence is deduced as "11001". The implementation of this technique does not modify the victim's original functionality. Referring to Figure 4.17(b), the attacker observes "11010" instead of the original sequence of "10010", given the cognitive perturbations.

The cognitive perturbations are dummy calls as they are not a part of the victim's original operations. Referring to Listing 4.3, the functions function_1 and function_2 are victim's original operations. In the function_main, the result of either function_1 or function_2 is fetched from the cache. We say the function call is a dummy call when the function's obtained result is discarded, meaning that the victim did not use the result. The cache access is only made to perturb the sequence of operations or cache accesses. The dummy calls are injected by Covert-Enigma as a part of the defense mechanism to trick the attacker into observing the sequence of operations the victim performs, including the injected dummy calls. The attacker depends on the cache access patterns, indicated by probe hit/miss, to steal secret information. Hence, by injecting dummy operations, the attacker observes the victim's original operations perturbed by dummy operations. These injected perturbations translate to misleading secret information different than the original information without injected perturbations.

4.2.2.4 Generation of Cognitive Perturbations

The generation of cognitive perturbation is intended to render the observance of sensitive information (over the side-channel) during the attack a time-consuming task. With cognitive perturbations, it becomes not only a time-consuming task for the attacker, but it also becomes difficult on the attacker's part to differentiate between the cache accesses caused by the defense in place versus those caused by the victim application. The adversary monitors the probed addresses. Therefore, the adversary is aware of the pattern of the victim's accesses when it executes the probed code lines. Our motivation in introducing cognitive perturbation is that if we can carefully craft cache accesses such that they would be considered legit by the adversary, it would be dummy operations that the victim makes to increase the entropy in the side-channel. Because these operations, though dummy in nature, make real cache accesses, they are considered by the adversary as an operation made by the victim while processing sensitive data. These cognitive operations need not be the replica of the functions found in a victim. Still, they could be simple lines of code that reload the same addresses[9] as are flushed by the adversary.

We build cognitive operations that execute (and access cache) similar to what the victim's original functions would do by simply reloading addresses in the memory corresponding to the lines of code in the victim's original operations. These cognitive operations are considered legit by the adversary application, as will be evident in Section 4.2.5, where we present and analyze the experimental results.

Addition of the cognitive perturbations might present the notion of additional power consumption, which may be used for other forms of side-channel attacks. However, the dummy function calls are limited in number, and the workload of these functions is miniature in nature. Thus, the amount of additional power or latency introduced by the Covert-Enigma is small. In addition, to perform a power-based SCA, the basic assumption would be that the attacker has power signatures for all the victim application(s) and can reliably compare the same with the golden power traces. However, the power trace collection involves

[9]The vulnerability in the victim application is known to the adversary.

uncertainties from different system components, which could be similar to additional power consumption by the introduced dummy operation of Covert-Enigma. Furthermore, the power SCAs are beyond the scope of this work

```
function_1 {
    a = cache_location(100);
    return a; }

function_2 {
    b = cache_location(170);
    return b; }

function_main {
    dummy_flag = 0;
    if(bit==0)
        result = function_1;

    else if(bit==1)
      result = function_2;

    dummy_flag = 1;
    if(bit==0)
        result = function_1;
        discard(result)
    else if(bit==1)
        result = function_2;
        discard(result)
                }
```

Listing 4.3: Example of a dummy operation

4.2.3 Covert-Enigma Modes of Operation

To enhance the robustness of the Covert-Enigma , the Covert-Enigma is equipped with two modes of operation - *arbitrary* and *cyclic*. Each mode can be set by the user to inject the corresponding level of perturbations. The tuning of the parameters in the mode refers to the reconfigurable aspect of Covert-Enigma. The reconfigurable parameters are "total bits

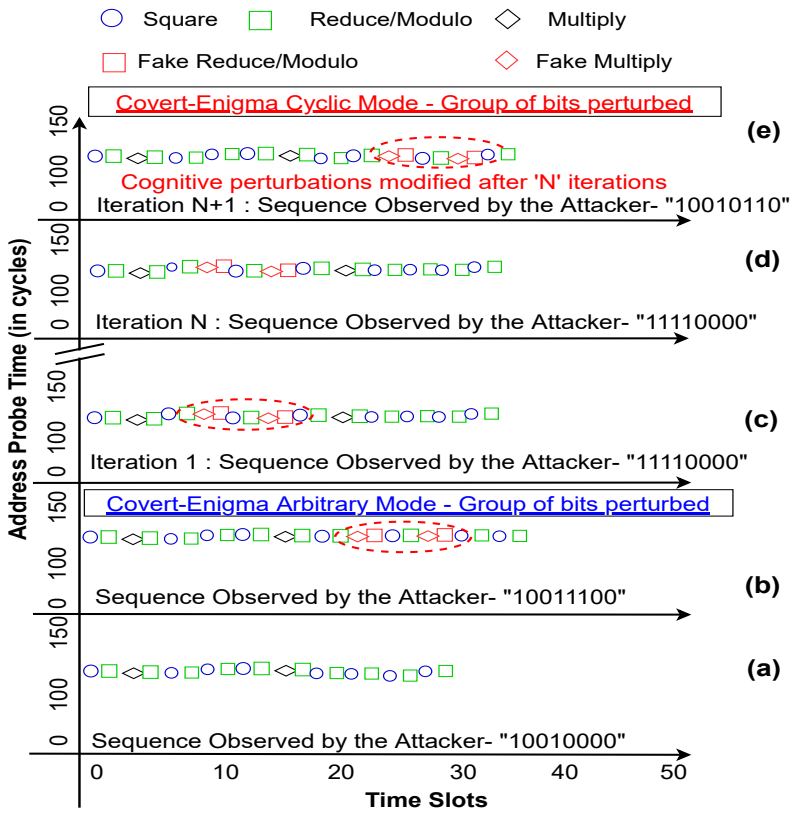


Figure 4.10: (a) Part of secret seen by both adversary and victim without Covert-Enigma (b) Sequence of bits seen by attacker when victim application is protected by Covert-Enigma arbitrary mode, where positions of the perturbed bits change each run; (c) Sequence seen by adversary with Covert-Enigma cyclic mode where position of group of perturbed bits remains same until iteration 'N'; (d) Bit positions from previous run remain same; (e)Bit positions have shifted randomly during new 'cycle' of same execution

perturbed" for the Arbitrary mode; and "total bits perturbed" and "Cycle Iterations (N)" for the Cyclic mode.

4.2.3.1 Arbitrary Mode

As in Figure 4.10(b), the *arbitrary* mode cognitively perturbs a group of cache operations by calling dummy operations to elevate the randomness. The *arbitrary* mode randomly selects positions to call dummy operations. To avoid keeping the number of successive dummy

operations static, arbitrary mode randomly groups cache operations (of the victim) and inserts dummy cache accesses[10] in between two successive victim cache access. The random bit position selection is explained in Section 4.2.3.3.

4.2.3.2 Cyclic Mode

As illustrated in Figure 4.10(c),(d) and (e), our Covert-Enigma supports *cyclic* mode of operation, where a group of bits is selected at random and cognitively perturbed, and the group selected stays the same for a few cycles (execution runs, in other words) determined by the user. Post the cycle count (details in the next subsection), a different set of bits is selected to insert dummy operations. In summary, the perturbed bit positions change every few cycles (denoted as 'N') selected by the user, and for new executions, i.e., at '$N+1$' cycle, new bits are selected, the position of which remains the same for another 'N' runs, as shown in Figure 4.10(d). The duration of the cycle is denoted as 'N' where N is an integer. After 'N' cycles, a new bit or set of bits are selected to perturb cognitively, and the sequence seen by the attacker changes; this is shown in Figure 4.10(e). The positions of these bits are random during every run and reduce the secret bits recovered during an attack by elevating the randomness in the side-channel.

4.2.3.3 Generation of Random Bit Positions

We introduced the two modes of operation previously. Both *arbitrary* and *cyclic* modes require random numbers to be generated to select bit position to inject cognitive perturbation(s). In such a case, Intel's *RDRAND* [160] and Linux's *'/dev/random'* [161] can be utilized. The true random number generator (TRNG) generates 'true' random numbers based on random, non-deterministic noise generated by the device drivers into an entropy pool, which returns random numbers. The Covert-Enigma utilizes these random numbers to generate cognitive noise.

[10]Cache is accessed but does not contribute to the functionality of the victim's operations

4.2.4 Summary of Covert-Enigma

Algorithm 4 outlines a high-level simplified view of the proposed Covert-Enigma along with a snippet of the victim and the attack code. Covert-Enigma part is presented in Algorithm 4 from Lines 15 to 39. The user needs to feed in the value of the size of the key. The *position_array, successive_bits_array* stores the values of the bit positions to inject dummy calls to and the successive bits to perturb, respectively. The values required for driving the *cyclic* mode are saved to a tamper-proof location that stores the current cycle count, along with the two arrays mentioned above. The Covert-Enigma and victim are synchronized using function calls. The arbitrary mode is shown in Lines 20-25. The *arbitrary* mode injects the perturbations until the *bit_count* in the *successive_bits_array*. In our implementation, the Covert-Enigma only injects a dummy multiply followed by a dummy reduce to give the notion of a bit '1' instead of a bit '0' - as a Square-Reduce-Multiply-Reduce sequence corresponds to bit '1' being processed by the RSA. The *cyclic* mode is shown in Lines 26-39. The *cyclic* mode operates similar to another mode. The major difference is that it does not keep injecting dummy operations with every new cycle of the victim application. To enable this, the Covert-Enigma accesses a tamper-proof location that stores the cycle count and the other two arrays mentioned previously. This helps to keep injected perturbations in the victim's cache access patterns 'static' for a user-selected number of cycles, specified by the value 'N'. If the victim has not completed the set number of cycles, the Covert-Enigma ensures that the same positions are selected to inject the dummy operations by reloading from the tamper-proof location. Otherwise, new random positions are generated. For the purpose of brevity, we limit the details of the attack, but interested readers can refer to [7] for details.

```
1 Attack {
2     Loop 1: clflush (Probe 1);
3     clflush (Probe 2);
4     clflush (Probe 3);
5     Reload Probe 1, Probe 2 and Probe 3;
6     wait for time = t_wait;
7     t= Measure Reloading time;
```

```
 8       jump Loop1 ;
 9  cmp (t, threshold time(th));
10  if( t > th) => Cache miss;
11  if( t < th) => Cache hit; }
```

Listing 4.4: Attack code to capture data

The attack code has been shown in Listing 4.4. The attack code is launched with the victim executing in parallel by the adversary to spy on the side-channel data. The probes 1,2, and 3 are inserted by the victim in the first few lines of code, which the attacker knows are called iteratively by the victim. These probes from the attack's point of view are simply the addresses of the lines in the victim code. The attack code then flushes these probed lines and waits for time *t_wait* for the victim to execute. If the victim executed and accessed the flushed cache line, the attacker, upon reloading the line, would see it as a cache hit - since the data was available and fetched quickly. Else, the attacker sees a cache miss. The cache hit/miss decisions are based on the threshold[11] value (slightly varies from system to system), which was '120' cycles for our experimental setup.

Entropy Maximization By the conventional entropy equation, $H = -\log(P_i)$, where P_i is the probability of bit i. As seen previously, Covert-Enigma increases randomization by injecting perturbations in the signal; hence, the probability that the attacker observes the correct/original key bit reduces dramatically. Hence, the lower the probability, the higher the entropy, which means more randomness in the retrieved information. Since a group of bits are selected to perturb the observed side-channel data, and the user can choose the position of these bits and their quantity, the total permutation of such sequence is huge if the attacker tries to observe the side-channel data after iterating the victim for thousands or even more number of times. With a 4096-bit key, the attacker would have to iterate it for ${}^{4096}P_2$=16.77*10^6 for 2 bits perturbed and ${}^{4096}P_6$=$4.7 * 10^{21}$ for 6 bits perturbed. Hence, by setting more number of bits to be cognitively perturbed or randomly choosing the number of bits to be cognitively perturbed, the user can render more resilience to SCAs, despite the

[11]The threshold is the probe access time in cycles

Algorithm 4 Pseudocode illustrating generation of perturbations with Covert-Enigma and the modes of operation

Require: Private Key
Ensure: Decoded Incorrect Key

```
1: Victim Program (Mode = Arbitrary or Cyclic)
   {// Performs secure-critical operations that leak data over covert channel}

2: func Square()
3:    { Probe 1 inserted here
4:    Do Square operation;
5:    Wait for the Covert-Enigma; }

6: func Reduce()
7:    { Probe 3 inserted here
8:    Do Reduce operation;
9:    Wait for the Covert-Enigma; }

10: func Multiply()
11:    { Probe 2 inserted here
12:    Do Multiply operation;
13:    Wait for the Covert-Enigma; }

14: Covert-Enigma (){
15:       position key_size = 1024/2048/3076 or 4096;
16:       position_array = true_random_generator();
17:       successive_bits_array = true_random_generator();
18:       tamper_proof_location = {N, position_array, successive_bits_array};
19:       bit_count=0;
20:       if (mode = Arbitrary(total_bits) ) then {
21:          for i in range(0 : sizeof(position_array)):
22:             if (current_position=position_array[i]) {
23:                do { Multiply(dummy);
24:                     Reduce(dummy);
25:                     } while(bit_count!= successive_bits_array[i]) }

26:       else if (mode = Cyclic(total_bits, N)) then {
27:          if (cycle_count != N;) then {
28:             reload tamper_proof_location = (N, position_array,
29:                                             successive_bits_array );
30:          else
31:             {refresh tamper_proof_location = (N, position_array,
32:                                             successive_bits_array );

33:       int N, cycle_count=0, bit_count; #N is selected by user

34:       for i in range(0 : sizeof(position_array)):
35:          if (current_position=position_array[i]) {
36:             do { Multiply(dummy);
37:                  Reduce(dummy);
38:                  tamper_proof_location++; }
39:                       while(bit_count!= successive_bits_array[i]) } }
40: end Victim Program;
```

attacker executing an attack a large number of times.

4.2.5 Experimental Evaluation

4.2.5.1 Validating the Attack and Covert-Enigma

Experimental Setup: We tested the proposed Covert-Enigma[12] on system with Intel-i7 core running Ubuntu 18.04 LTS OS with 16 GB RAM and GnuPG's [159] RSA [158] implementation. Flush+Reload [7] attack code can be found at [162].

Validation of Attack: Here, we evaluate the efficacy of the proposed defense to mitigate side-channel leakage to dissuade the adversary from stealing sensitive data. The cache size (last level cache) on our experimental setup was 2MB, with 16-way cache associativity. The cache map is a representation after one iteration of the victim. We present cache access maps (part of a cache that is of interest to investigate) in Figure 4.11 for scenarios where the victim is under attack and when our proposed Covert-Enigma shields the victim. The nuances of the colors shown in the figure demonstrate the relative accesses made to a particular location - darker shade signifies more frequent accesses. In comparison, a lighter shade signifies relatively less frequent accesses. As seen in Figure 4.11(a) and (b), probed functions for RSA victim are highlighted. These locations correspond to the cache locations attacked by the adversary.

Figure 4.11(a) shows access to the cache made by the victim and the adversary. Figure 4.11(b) shows the cache accesses made by the defense, adversary, and the victim, as the same cache is shared across. In 4.11(a) one can see tightly clustered dense regions of cache access. However, in 4.11(b), one can observe the dense areas spread across the whole map. Such a disaggregation leaves the attacker with more ambiguity. Further, some areas have shown an increase in access rate due to additional dummy operations introduced by Covert-Enigma. It is to be noted that the Figure 4.11 is a simplified illustration of cache accesses obtained from experiments to demonstrate the effectiveness of proposed Covert-Enigma.

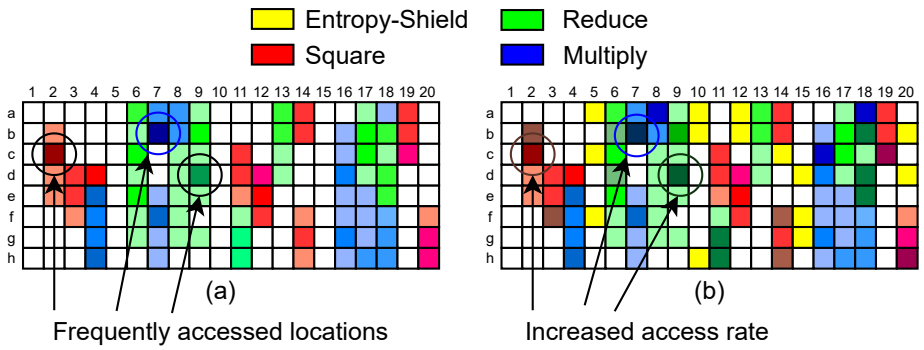


Figure 4.11: (a) Cache access map for operations observed when the victim is under attack; (b) Cache access map for operations observed when Covert-Enigma makes cognitive calls to the probed cache lines

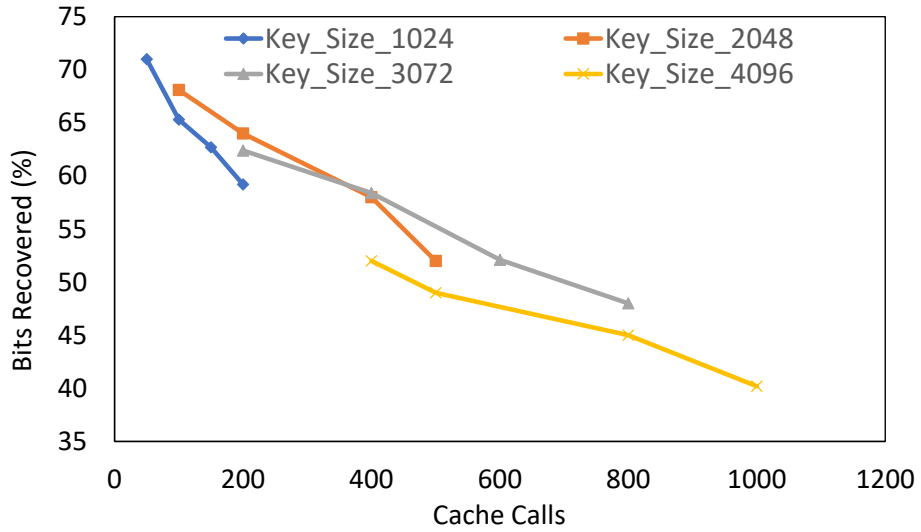


Figure 4.12: Bits recovered under SCA shown with different dummy cache accesses and for different key sizes. The victim application is wrapped by Covert-Enigma. The number of bits recovered (secret) reduces with increase in dummy cache calls made by Covert-Enigma

Table 4.2: Group of key bits perturbed by Covert-Enigma

Group of key bits perturbed by Covert-Enigma - Arbitrary mode							
Attack	Victim	Key	Original Key	Victim seen key	Key seen by the adversary		
Flush+Reload	RSA	key_1	100100001110	100100001110	100111001110		
	RSA	key_2	010110000111	010110000111	010110111111		
Group of key bits perturbed by Covert-Enigma - Cyclic mode							
Attack	Victim	Key	Original Key	Victim seen key	Key seen by the adversary		
					Iteration 1	Iteration Nth	Iteration (N+1)th
Flush+Reload	RSA	key_1	100100001110	100100001110	111100111110	111100111110	100111001111
	RSA	key_2	010110000111	010110000111	110111100111	110111100111	011110011111

4.2.5.2 Recovering Sensitive Data

We also evaluate the effectiveness of the proposed defense in terms of key extraction. In other words, we present the information regarding how many key-bits can be extracted by executing the RSA application under the Flush+Reload SCA with traditional randomization and proposed defense. We follow the procedure described in [7, 8] for key extraction. We execute the attack on the victim and recover as many bits of the secret data as possible. As described in our threat model, we place our experiments in a real-world setting where the adversary does not have control over the victim and can only observe a part of the victim's execution. This is realistic as the victim only executes for encryption/decryption operations only when required. Hence, it is imperative to mention that the adversary initiates the attack during such instances and observes a portion of the victim's execution. The results of the key recovery are presented in Table 4.5 and 4.7. The colored text highlights the cognitive perturbations that are injected. For Cyclic mode, the selected bit positions remain the same until 'N'th round ($N = 25$ in our experiments), followed by new bit positions selected. A part of the observed key is shown for conciseness. We evaluate the defense under different key sizes for crypto-operation.

Table 4.3 shows the key extraction for different key sizes with traditional randomization defense. We implement a randomization strategy similar to that in [3–5]. We do not

[12]https://github.com/hartanonymous3512/Covert-Enigma.git

claim an accurate replication of the defense in [3–5], yet consider a similar approach as a baseline for comparing the proposed methodology versus a similar defense where the cache is accessed randomly to mislead the attacker. The works in [3–5] are based on randomization but require hardware or software stack changes. We have replicated them without software stack or hardware architecture changes. Hence, to establish a baseline, we consider the above-mentioned works as a randomization-based defense generically, and compared our work with a similar version. Figure 4.12 presents the results for bit recovery with Covert-Enigma for different key sizes and dummy cache accesses. The number of calls made are for one complete execution of the victim. The results in Table 4.3 demonstrate that with a defense strategy like that presented by the work in [3–5], the recovery of the secret key/data ranges from 88% to 77%. We have presented the recovery rate with when the victim is protected by Covert-Enigma in Figure 4.12. We have compared Table 4.3 with Figure 4.12 indicating that with our proposed defense, the recovery rate reduces to 72-59% for key size of 1024, 68-52% for 2048, 62-48% for 3072, and 52-40% for a key size of 4096.

For the *arbitrary* mode, we obtain similar results as in Figure 4.12. The cyclic mode's advantage is in scenarios where the user happens to use a crypto operation that uses the same key for de-obfuscating different encrypted files on the system and where the adversary can obtain information from multiple executions of the victim. Another evaluation technique we have used in this work is by comparing the observed traces. The spy program is made to print the operations' sequence while the probed locations - probed functions Square, Reduce, and Multiply - are accessed by the victim. These sequences are compared against the victim's operations under Covert-Enigma. Table 4.4 presents the number of perturbations (additional cache calls) injected for a 1024-bit key. The table reports the differences seen in percentage. For instance, an 8% difference is observed while comparing the victim's operations without Covert-Enigma and with Covert-Enigma. Theoretically, 8% should have been 10% for 100 additional calls in a 1024-wide key. But, as explained previously, the spy cannot see successive cache operations, and hence, some operations are not observed, as the probe scan time is less than the cache access time.

Table 4.3: SCA on victim protected by traditional randomization and Covert-Enigma. Attacker recovered secret data

Key Size	1024	2048	3072	4096
Bits Recovered (traditional randomization) (%)	88.2	85.1	81.7	77.0
Bits Recovered (Covert-Enigma) (%)	60-70	53-68	48-62	40-52

Table 4.4: Percentage difference comparison of victim operations with and without Covert-Enigma

Amount of injected perturbations	100	300	500	600
Difference observed with perturbations(%)	8	26	48	55

4.2.5.3 Covert-Enigma with Flush+Reload Attack

We have chosen the Flush+Reload and Flush+Flush attack spying on RSA-RSA and DSA-Elgamal encryption algorithms with a secret key of 4096-bits, as implemented in the GnuPG. We will also present the outcome with different modes of operation - *Arbitrary* and *Cyclic*. We verified our proposed Covert-Enigma by examining the perturbations injected both on the victim and spy end. Figure 4.10 presents a pattern of the sequence of operations plotted against time slots versus the probe time as seen by the attacker/victim. Figure 4.10(a) shows the secret information observed by the victim and the attacker without the Covert-Enigma. Every Square-Modulo operation not followed by Multiply is translated as bit '0,' and every Square-Modulo-Multiply-Modulo operation as bit '1', as in [7]. In this case, the victim and the attacker both see the same information - the spy observes the channel's leaked information. Figure 4.10(b) shows the sequence of operations when the victim is being protected by the Covert-Enigma in the arbitrary mode -the victim observes the key as "10010000", the original key, while the attacker sees it as "10011100" since some

of the '0' bits are flipped to bit '1'. These perturbations are induced irrespective of the key, as shown in Table 4.5 with cognitively selected zeros converted to ones for the *key_1* for the RSA-RSA type encryption -victim sees the key as "100100001110", the attacker observes it as "100111001110". One needs to note that in Figure 4.10 all the bits are not shown to avoid congestion in the figure, and also, it was not possible to show all of the 4096-bits. Also, for the Table 4.5 and 4.7, a part of the large key has been shown to demonstrate the perturbation rather than the actual position in itself.

Similarly, for *key_2*, DSA-Elgamal type, some other random bits are perturbed, and the attacker observes a different pattern. For the *Cyclic* mode, as shown in Figure 4.10 and Table 4.7, the perturbed key remains the same for '$N = 25$' iterations, post which other random bits are perturbed in the sequence that begins with $(\mathrm{N}+1)^{th}$ iteration, which stays static until the end of the cycle which is $(N + N)^{th}$ iteration. As seen from Figure 4.10, the attacker observes the sequence as "11110000" which remains the same until the end of iteration 'N', post which it changes to "10010110" and the results for the same can be confirmed from Table 4.7.

Tables 4.6 and 4.8 demonstrate the results where randomly chosen (similar to the group perturbations) single bits are flipped/perturbed. The *successive_bits_array* value can be modified to choose single bit perturbation instead of grouped perturbation, where successive bits are perturbed. From Table 4.6, the victim sees the value as "100100001110" while the attacker observes it as "10011000110" for RSA type. Similarly, it can be seen from Table 4.8 how the observation is affected using the *Cyclic* mode. The single-bit perturbations can be chosen to perturb bits along the entire sequence of operations of cache accesses. The user can choose a single bit versus a group of bits considering the security-overhead trade-off.

4.2.5.4 Covert-Enigma with Flush+Flush Attack

We have evaluated our Covert-Enigma against Flush+Flush, whose key extraction results are presented in Table 4.5 and 4.7 for both the modes. Similar to the Flush+Reload, the induced perturbations can deceive the spy in both *arbitrary* and the *Cyclic* modes.

Table 4.5: Group of key bits perturbed by Covert-Enigma - Arbitrary mode

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	100111001110
	DSA-Elgamal	key_2	010110000111	010110000111	010110111111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111011100110
	DSA-Elgamal	key_4	100000110011	100000110011	100111110011

Table 4.6: Single key bit perturbed by Covert-Enigma - Arbitrary mode

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	100110001110
	DSA-Elgamal	key_2	010110000111	010110000111	010110100111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111001100110
	DSA-Elgamal	key_4	100000110011	100000110011	100001110011

Table 4.7: Group of key bits perturbed by Covert-Enigma- Cyclic mode

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker		
					Iteration 1	Iteration N^{th}	Iteration $(N+1)^{th}$
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	111100111110	111100111110	100111001111
	DSA-Elgamal	key_2	010110000111	010110000111	110111100111	110111100111	011110011111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111011111110	111011111110	111000111110
	DSA-Elgamal	key_4	100000110011	100000110011	100110111111	100110111111	111000110011

For instance, for the RSA type keys, in the *arbitrary* mode, the key gets translated from "111000100110" to "111011100110" whereas for the Cyclic mode it is observed as "111011111110" and "111000111110" during iteration-1 and iteration $(N+1)^{th}$ respectively. For our proposed defense to work even for Flush+Flush, it needs to ensure that the lines of code within the square, modulo, or multiply functions are cached, and only then the attacker can flush

Table 4.8: Single key bit perturbed by Covert-Enigma- Cyclic mode

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker		
					Iteration 1	Iteration N^{th}	Iteration $(N+1)^{th}$
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	101100011110	101100011110	100101001111
	DSA-Elgamal	key_2	010110000111	010110000111	110110100111	110110100111	011110010111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111010110110	111010110110	111000101110
	DSA-Elgamal	key_4	100000110011	100000110011	100100111011	100100111011	101000110011

a cache line within the code and consider that the encryption must have accessed the function/operation. Tables 4.6 and 4.8 present results for single bit perturbations for both the modes for Flush+Flush.

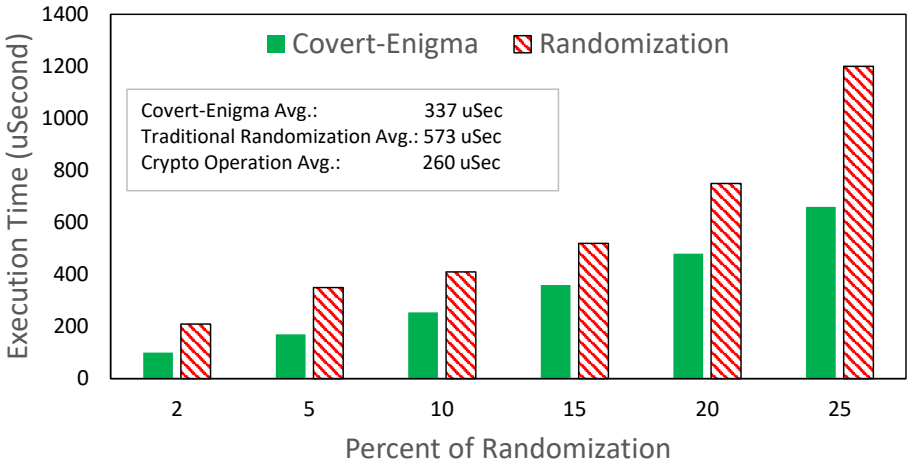


Figure 4.13: Overhead analysis for 4096-bit key with different amounts of randomization. Overhead compared with a close replication of random cache policy similar to [3–5]

4.2.5.5 Summary of the Implemented Results

Tables 4.5, 4.6, 4.7, 4.8 are ideal cases because while executing them on our machine we reduced the number of background activity. But, in actual scenarios, the OS and other

application activity will generate noise in the cache, making the attack more difficult. The attacker might not be able to see the key bits in consecutive order. Hence, as the keys seen by the attacker will be different every time and with such randomness, explained in detail in Subsection 4.2.4 it is very difficult for the attacker to retrieve the key knowing the fact that executing SCAs successfully is non-trivial when it comes to retrieving secret keys amid operating system noise and various cache operations. Our Covert-Enigma enhances security, but there is no single/unified mechanism to evaluate all the corner cases. A single solution does not address all the problems. Our proposed solution holds for the threat model described previously.

4.2.5.6 Overhead Analysis

The overhead analysis graph is shown in Figure 4.13. The figure compares the execution times of traditional randomization technique similar to [3–5] and Covert-Enigma. We consider the arbitrary mode for presenting the results. The trend is shown for different amounts of randomization added (along the X-axis) and the execution time in microseconds (along Y-axis). The average execution times across different percent of randomization is shown in Figure 4.13. The trend clearly explains that with our defense, the overhead is 50% less than a randomization technique that tries to insert random calls for every bit of the secret key. Our proposed defense inserts the calls cognitively, hence incurs less overhead - cache is accessed less frequently than traditional randomization techniques. We see this trade-off as a beam scale balance that weighs security and performance (in terms of execution time/cycles) on each of its scale pans. The user can determine the amount of perturbations by analyzing the overhead-security trade-off. Again, it is to be noted that the traditional randomization we compare our proposed methodology is not an exact reproduction of the work in [3–5], but it is similar in a manner that we allow injecting random perturbations in the cache in software; no hardware modifications are required.

How the cognitive perturbations differ from traditional randomization is explained further referring to Figure 4.13. If each bit of the key is perturbed, meaning the cache is

accessed in a dummy fashion (randomly), the overhead is significantly higher. From our experiments, if X is the base execution time for the victim, then 2.19X is the overhead with traditional randomization, while it is 1.29X with Covert-Enigma's arbitrary mode. All the overheads are averaged for simplicity. A 4096-bit key is used for crypto operations, and by varying the number of cognitive perturbations, higher security can be offered. This comes at the expense of some overhead. With 10% injected perturbations, the overhead is 25% less with Covert-Enigma compared to randomization only. With 25% injected perturbations, we observe a 50% less overhead against the randomization only method as mentioned above. Hence, perturbing each bit is not a solution owing to large infeasible overhead. With traditional randomization, the overhead can be feasible, but the attack can break the defense much earlier than it can when Covert-Enigma wraps the victim. Moreover, the overhead of Covert-Enigma is less than the other technique.

4.2.6 The attack phase and the Covert-Enigma: A case-study

In this section, we will briefly discuss the motivation supporting the proposed Covert-Enigma which is presented as a case study.

Figure 4.14 shows different scenarios of the victim's access to the cache memory and the attacker's access and how the attacker exploits this information deducing the secret key. Figure 4.14(a) shows a scenario where the attacker tries to flush the victim's data, then waiting for a predefined time before reloading the same data. As can be seen, since the victim did not access the data, the attacker experiences a cache-miss when it tries to reload the data, which is discarded. Figure 4.14(b) visually describes the victim's access while the attacker was waiting for the victim to execute. Since the victim accessed the data, the attacker experiences a cache-hit during the reloading phase, thus deducing the data accessed by the victim. Figure 4.14(c) presents a scenario when the victim accesses the cache multiple times within the same 'wait' window of the attacker. But the attacker can spy on only the recent chunk of data accessed by the victim, and it will never know what locations the victim accessed preceding the recent access. In summary, irrespective of

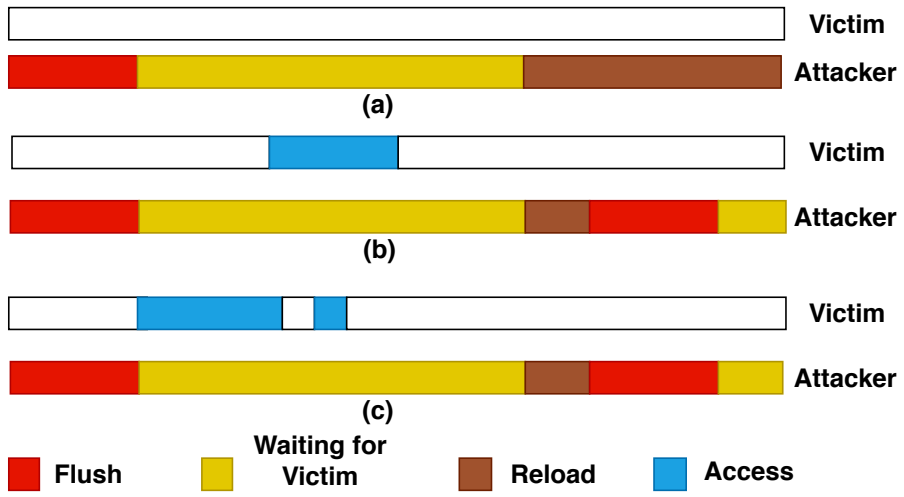


Figure 4.14: Timing diagram depicting different scenarios where a victim and/or an attacker may access the cache ;(a) Victim Does not Access; (b)Attack with Victim Access; (c) Victim multi-Access

the scenario, the attacker can still spy on the location accessed by the victim.

In addition to this, referring to Figure 4.15, we can get an idea of how the attacker spies on the crypto application while executing secure critical operations. As explained in Section 4.2.1, the GnuPG uses different operations to encode/decode user data or secret data where the sequence of these operations can leak the secret data shown in Figure 4.15. Part (a) presents a sequence of operations that decodes to "10001" as discussed previously. If only some noise could be added to these sequence traces, the SCAs could be thwarted with little effort. Let's consider a defense mechanism that adds noise to the traces observed by the attacker. This might dissuade the attacker from decoding the secret data as due to noise, deducing the key would seem difficult. Still, in case of a persistent attack on the system, the attacker can break the defense wall by observing a large sample of the observed data and filtering the noise. Hence, merely adding noise to the operations' sequence will not suffice and is not a robust solution. We introduced cognitive noise to the sequence of operations that looks legit to the attacker yet leads to an incorrect deduction of the secret

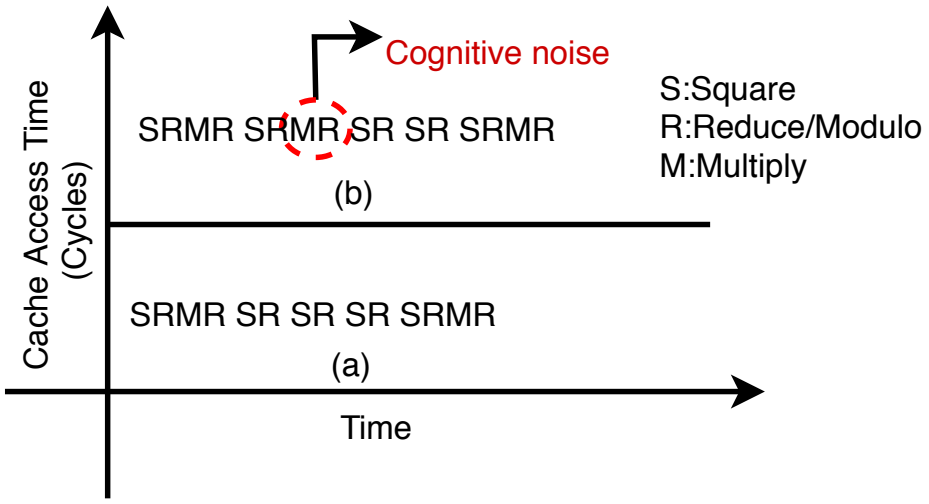


Figure 4.15: (a) Sequence of operations in a crypto system that potentially leaks secret data; (b) Cognitive noise injected in the victim's covert channel data to protect the information while tricking the attacker

key. Figure 4.15(b) shows the same sequence of operations as part (a) but with intelligently crafted noise injected in the sequence. This crafted noise concerning the sequences makes sense to the attacker. As can be seen, the multiply and reduce operations are dummy called succeeding the square and reduce operations (original operations called by the victim), which, when observed by the attacker, will translate to bit-1 instead of bit-0 which tricks the attacker. Since the injected operations are dummy, they do not harm the victim's crypto operations. This has been our motivation behind the proposed work. We have discussed different modes of operation of the proposed method to render the defense more robust and resilient to attacks.

Extending to other victims: It needs to be noted that our proposed Covert-Enigma could be extended to any victim that repeatedly calls for lines of code, where the sequence of accesses to the cache is important. For example, suppose the victim application is Advances Encryption Standard (AES). In that case, the user can decide to randomize the cache accesses AES makes for reading tables used for the crypto-operations. The adversary

times the access to the locations of the tables to conclude which ones were accessed recently. The user can employ Covert-Enigma to introduce cognitive perturbations, which seems legit (cache access), but the sequence is scrambled to camouflage the sequence of cache operations. Thus, it becomes possible to hide the real accesses made by AES, yet leading the attacker to consider cache accesses made by Covert-Enigma.

4.3 A Novel Adversarial Attack-assisted Hardware Vulnerability Exploit on Deep Learning Networks

Modern computing systems are gaining superior performance than ever before owing to the development of sophisticated machine learning services such as deep learning. The deep learning networks (DNN) are becoming ubiquitous in almost all domains in the real-world due to their unparalleled performance in tasks such as malware detection, autonomous cars, image recognition, object detection, medical industry, security critical applications and much more. Considering the crucial role deep learning systems play in such mission critical applications, maintaining a secure environment is mandated. Nevertheless, the performance of a deep learning network can be severely degraded when exposed to certain vulnerabilities.

Recent studies [163–166] have demonstrated that DNNs are vulnerable to adversarial attacks; these attacks are implemented by adding well-crafted perturbations to the input images. A carefully crafted perturbation can mislead the DNN thus misclassifying a legit input image. Thus far, previous works have focused on offline poisoning of the legit input images. Meaning, the adversarial perturbations are added to the images in an offline fashion. The poisoned data is later consumed by a DNN model for inference. The crafted perturbations result in loss of DNN performance. Here, the underlying assumption is that the adversary has access to the input image(s). On the other hand, there are instances where internal vulnerabilities in a DNN are exploited as well [167]. A DNN provides its classification or inference based on the weights that are designated at training time. Work in [168] have demonstrated that the a DNN is very sensitive to its weight parameters. Injected perturbations or alterations in the model weight(s) can lead to degradation of DNN performance; an intelligent perturbation of the weights can dramatically degrade DNN performance thus leading to misclassification of the input data.

We observe that the works in the past have focused either on the software side of the attack or hardware side of the attack. This work focuses on a novel attack methodology

that targets the victim DNN from both the software and hardware aspects. This unique approach makes it harder to defend. We intend to propose this approach to demonstrate the resiliency of a DNN network to a novel attack. This will contribute to improvise DNN design or environments which DNNs utilize for their operation.

In this work, we propose a novel attack, Adv-Exploit , that utilizes a two-prong approach in attacking a DNN. One aspect of the attack is by perturbing the pixels of the legit input images using vulnerabilities in the underlying hardware - say, using a Rowhammer attack [169]. Because, there may be scenarios where the adversary may not have complete access to the image dataset to inject perturbations. Hence, a methodology that can poison the images in-memory is required. This is novel to our paper in contrast to the previous adversarial attacks [163] that poison the input images by adding perturbations to the images stored offline. Our proposed attack manipulates the bits representing the input image pixel values using a bit flipping strategy. Another principal feature is that we utilize a powerful adversarial attack-assisted data manipulation, Min-invasive . The perturbation of the image pixels is carried out by our proposed adversarial attack that imposes group sparsity on adversarial perturbations by extracting structures from the inputs. Our proposed adversarial attack identifies minimally sufficient regions that make attacks successful, but without incurring extra pixel-level perturbation power. Our bitflip hardware attack is inspired from the work in [167]. We propose an enhanced version of the bitflip attack by driving it using our adversarial attack.

The second aspect of the attack is to poison the DNN model weight parameters using a gradient search algorithm. The weight parameters crucial to DNNs inference performance. Incorrectly trained weights or perturbed weight parameters could lead to degradation of the DNN inference. Also, random weight parameters cannot be manipulated as it may not lead to significant loss. Hence, similar to an adversarial attack, the weight poisoning attack also needs to be intelligently crafted. Hence, a small number but the most sensitive bits must be flipped to affect the DNN performance. Compared to the popular C&W attack [163], our proposed Adv-Exploit is far more efficient in terms of the pixels perturbed and

naturally the total bit flips required to attack the DNN model. A real-world application of our proposed attack could be, say, an autonomous vehicle. An autonomous vehicle utilizes a sophisticated DNN model trained on input images derived from a continuous stream of data (video). Our proposed attack installed as a third-party application on the system could internally perturb the input images in memory and the model weight parameters to degrade DNN performance, thus leading to misclassification scenarios - a stop sign misclassified as a green signal. Such a vulnerability exploit could be very catastrophic, causing damage to property up until loss of human life. Hence, we propose to evaluate DNN resiliency against our Adv-Exploit attack in different testing scenarios.

We evaluate our attack on different architectures and corresponding popular datasets, MNIST, ImageNet, and CIFAR-10. It is evident from the evaluation, as discussed in Section 4.2.5, that our proposed attack technique requires 16% and 24% less bit perturbations compared to the Carlini attack [163] for MNIST and CIFAR-10 dataset, respectively. We also demonstrate that the DNN accuracy under attack degrades to less than 30%, which is worse than a random guess of 50% accuracy. Using different DNN architectures and datasets, it is demonstrated that our proposed vulnerability exploit attack is both model architecture and dataset agnostic. In other words, the proposed methodology should work on other architectures and dataset as well.

In summary, the essential contributions of this work are:

- Construct a novel adversarially driven hardware vulnerability attack targeting a Deep Neural Network (DNN). The intent is to evaluate the resiliency of the DNN against such vulnerabilities. The proposed attack is a two-pronged approach that targets the software and hardware aspects. The attack is architecture agnostic and is validated for MNIST, CIFAR-10, and ImageNet datasets.

- For the software aspect of the attack proposing an adversarial attack that is less expensive in regards to the bits needed to perturb an input as compared to the popular C&W attack.

- As for the hardware aspect, proposing an enhanced bit flipping that is driven based on the perturbation matrix provided by adversarial attack. Also, enhanced bit flipping is used to manipulate weight parameters of the DNN model to degrade model accuracy.

The rest of the paper is organized as: Related works are discussed in Section 4.3.1, followed by the design methodology of the proposed attack in Section 4.2.2; Evaluation is presented in Section 4.2.5. The adversarial attack, Min-Invasive, described in the proposed attack section and its results in evaluation section is credited to the team at North Eastern (NEU) University [170]; the adversarial work is done in collaboration with the team at NEU university. The attack in [170] is adapted and enhanced for the proposed attack in this section.

4.3.1 Related Works

Adversarial Attacks.

Many adversarial attacks are constrained by ℓ_p norms. For example, FGM [165] and IFGSM [164] attacks try to maximize the classification error with a ℓ_1-norm constraint. Moreover, L-BFGS [166] and C&W [163] attacks minimize the ℓ_2-norm distortion while achieving mis-classifications. Besides, JSMA [171] and one-pixel [172] attacks try to perturb the minimum number of pixels, i.e., minimizing the ℓ_0-norm of adversarial perturbations. Auto-attack [173] forms a parameter-free, computationally affordable and user-independent ensemble of attacks to test adversarial robustness

In the aforementioned adversarial attacks with norm-ball constraints, there are two opposite principles: C&W attack (or ℓ_1 attacks) modifies all pixels with minimal pixel perturbations; one-pixel attack only perturbs a few pixels with more significant pixel-level distortions. Both attacking principles might lead to a high noise visibility due to perturbing too many pixels or perturbing a few pixels too much. In this work, we explore a more effective attack to achieve a tradeoff between the perturbation strength and the number of perturbed pixels. We show that the proposed Min-invasive can identify sparse perturbed regions (rather than perturbing every pixel) with negligible pixel-level perturbations to

achieve successful attacks. Note that one-pixel attack usually has much lower attack success rate on ImageNet than C&W attack and Min-invasive .

Defense against Adversarial Attacks. Many defense works against adversarial attacks have been proposed such as defensive distillation [174] to distill the original DNN and introduce temperature into the softmax layer, and robust adversarial training [175, 176] to incorporate the min-max optimization. It is commonly known that the robust adversarial training method ensures the strongest defense performance against adversarial attacks on MNIST and CIFAR-10. Due to the large training efforts of adversarial training, a line of works [177–179] try to implement adversarial training more efficiently with less computation complexity. In this work, we evaluate the effectiveness of Min-invasive to three defense methods: defensive distillation [174], adversarial training via data augmentation [180], and robust adversarial training [175].

Visualization of Adversarial Examples. Despite the increasing popularity of adversarial attack and defense in research, the visual explanation on adversarial perturbations is less explored since the minimal perturbation is hard to recognize by human eyes. The work [181] investigates how the internal representations of DNNs are affected by adversarial examples. However, it only considers an ensemble-based attack, which fails to explore different ℓ_p-norm constrained adversarial attacks. Unlike [181], we employ adversarial saliency map (ASM) [171] and class activation map (CAM) [182] as interpretability tools to demonstrate the effectiveness of different attacks. CAM can localize class-specific image discriminative regions [183] and ASM measures the sensitivity of pixel-level perturbation on label classification. We show that the sparse adversarial pattern obtained by Min-invasive offers a great interpretability through ASM and CAM compared with other norm-ball constrained attacks.

4.3.2 Proposed Attack

This section discusses the proposed Adv-Exploit , overview of the proposed attack flow, and details about the adversarial attack and adversarial attack-assisted data manipulation. As

discussed earlier, our proposed attack targets to 1) Predict the pixels or bits to perturb and then manipulate the corresponding locations in-memory, and 2) Find the most sensitive bits of the DNN weight parameters and flip them. To achieve this, we propose and utilize our adversarial attack-assisted data manipulation (ADM) and a DNN weight sensitivity mechanism. We further explain the two mentioned mechanisms.

4.3.2.1 Proposed Min-invasive Attack

In the section, the concept of *Min-invasive* is introduced [170]. It works by dividing an image into sub-groups of pixels and then penalize the corresponding group-wise sparsity. The resulting sparse groups introduce the least amount of adversarial perturbations on the local structures of the original images.

Let $\mathbf{\Delta} \in \mathbb{R}^{W \times H \times C}$ be an adversarial perturbation added to an original image $\mathbf{X}_0$, where $W \times H$ gives the spatial region, and C is the depth, e.g., $C = 3$ for RGB images. To characterize the local structures of $\mathbf{\Delta}$, we introduce a *sliding mask* $\mathcal{M}$ with stride S and size $r \times r \times C$. When $S = 1$, the mask moves one pixel at a time; When $S = 2$, the mask jumps 2 pixels at a time while sliding [170].

By adjusting the stride S and the mask size r, different group splitting schemes can be obtained. If $S < r$, the resulting groups will contain *overlapping* pixels. By contrast, groups will become *non-overlapped* when $S = r$. A sliding mask $\mathcal{M}$ finally divides $\mathbf{\Delta}$ into a set of groups $\{\mathbf{\Delta}_{\mathcal{G}_{p,q}}\}$ for $p \in [P]$ and $q \in [Q]$, where $P = (W - r)/S + 1$, $Q = (H - r)/S + 1$, and $[n]$ denotes the integer set $\{1, 2, \ldots, n\}$. Given the groups $\{\mathbf{\Delta}_{\mathcal{G}_{p,q}}\}$, the group sparsity can be characterized through the following sparsity-inducing function [184–186], motivated by the problem of group Lasso [184]:

$$g(\mathbf{\Delta}) = \sum_{p=1}^{P} \sum_{q=1}^{Q} \|\mathbf{\Delta}_{\mathcal{G}_{p,q}}\|_2, \tag{4.1}$$

where $\mathbf{\Delta}_{\mathcal{G}_{p,q}}$ denotes the set of pixels of $\mathbf{\Delta}$ indexed by $\mathcal{G}_{p,q}$, and $\|\cdot\|_2$ is the ℓ_2 norm.

4.3.2.1.1 Min-invasive Attack with ADMM This section discussed a framework where the attacker relies only on the loss function's gradient given the input to the DNN model, the input being the images.

The model takes both the proposed group-sparsity regularization and adversarial distortion metrics that encodes spacial structures in attacks. It is shown that the process of generating structured adversarial examples leads to an optimization problem that is difficult to solve using the existing optimizers Adam (for C&W attack) and FISTA (for EAD attack) [163,170,187]. To address this, we utilize alternating direction method of multipliers (ADMM) for efficient optimization .

Given an original image $\mathbf{x}_0 \in \mathbb{R}^n$, the motive is to design the optimal adversarial perturbation $\boldsymbol{\delta} \in \mathbb{R}^n$ such that the adversarial example $(\mathbf{x}_0 + \boldsymbol{\delta})$ misleads DNNs trained on natural images. Throughout this paper, we use vector representations of the adversarial perturbation $\boldsymbol{\Delta}$ and the original image $\mathbf{X}_0$ without loss of generality. A well designed perturbation $\boldsymbol{\delta}$ can be obtained by solving optimization problems of the following form,

$$\begin{aligned} \underset{\boldsymbol{\delta}}{\text{minimize}} \quad & f(\mathbf{x}_0 + \boldsymbol{\delta}, t) + \gamma D(\boldsymbol{\delta}) + \tau g(\boldsymbol{\delta}) \\ \text{subject to} \quad & (\mathbf{x}_0 + \boldsymbol{\delta}) \in [0,1]^n, \ \|\boldsymbol{\delta}\|_\infty \leq \epsilon, \end{aligned} \tag{4.2}$$

where $f(\mathbf{x}, t)$ denotes the loss function for crafting adversarial example given a target class t, $D(\boldsymbol{\delta})$ is a distortion function that controls the perceptual similarity between a natural image and a perturbed image, $g(\boldsymbol{\delta}) = \sum_{p=1}^{P} \sum_{q=1}^{Q} \|\boldsymbol{\delta}_{\mathcal{G}_{p,q}}\|_2$ is given by (4.1), and $\|\cdot\|_p$ signifies the ℓ_p norm. In Equation (4.2), the 'hard' constraints ensure the validness of created adversarial examples with ϵ-tolerant perturbed pixel values. And the non-negative regularization parameters γ and τ place our emphasis on the distortion of an adversarial example (to an original image) and group sparsity of adversarial perturbation.

Equation (4.2) gives a general formulation for design of adversarial examples. If we remove the group-sparsity regularizer $g(\boldsymbol{\delta})$ and the ℓ_∞ constraint, problem (4.2) becomes the same as the C&W attack [163]. More specifically, if we further set the distortion function

$D(\boldsymbol{\delta})$ to the form of ℓ_0, ℓ_2 or ℓ_∞ norm, then we obtain C&W ℓ_0, ℓ_2 or ℓ_∞ attack. If $D(\boldsymbol{\delta})$ is specified by the elastic-net regularizer, then problem (4.2) becomes the formulation of EAD attack [187].

In this paper, we specify the loss function of problem (4.2) as below, which yields the best known performance of adversaries [163],

$$f(\mathbf{x}_0 + \boldsymbol{\delta}, t) = c \cdot \max\{\max_{j \neq t} Z(\mathbf{x}_0 + \boldsymbol{\delta})_j - Z(\mathbf{x}_0 + \boldsymbol{\delta})_t, -\kappa\}, \tag{4.3}$$

where $Z(\mathbf{x})_j$ is the jth element of logits $Z(\mathbf{x})$, representing the output before the last softmax layer in DNNs, and κ is a confidence parameter

that is usually set to zero if the attack transferability is not much cared.

We choose $D(\boldsymbol{\delta}) = \|\boldsymbol{\delta}\|_2^2$ for a fair comparison with the C&W ℓ_2 adversarial attack. In this section, we assume that $\{\mathcal{G}_{p,q}\}$ are non-overlapping groups, i.e., $\mathcal{G}_{p,q} \cap \mathcal{G}_{p',q'} = \emptyset$ for $q \neq q'$ or $p \neq p'$. The overlapping case will be studied in the next section.

The presence of *multiple* non-smooth regularizers and 'hard' constraints make the existing optimizers Adam and FISTA [163, 187–189] inefficient for solving problem (4.2). First, the subgradient of the objective function of problem (4.2) is difficult to obtain especially when $\{\mathcal{G}_{p,q}\}$ are overlapping groups. Second, it is impossible to compute the proximal operations required for FISTA with respect to all non-smooth regularizers and 'hard' constraints. Different from the existing work, we show that ADMM, a first-order operator splitting method, helps us to split the original complex problem (4.2) into a sequence of subproblems, each of which can be solved *analytically*.

We reformulate problem (4.2) in a way that lends itself to the application of ADMM,

$$\begin{aligned} \underset{\boldsymbol{\delta},\mathbf{z},\mathbf{w},\mathbf{y}}{\text{minimize}} \quad & f(\mathbf{z} + \mathbf{x}_0) + \gamma D(\boldsymbol{\delta}) + \tau \textstyle\sum_{i=1}^{PQ} \|\mathbf{y}_{\mathcal{D}_i}\|_2 + h(\mathbf{w}) \\ \text{subject to} \quad & \mathbf{z} = \boldsymbol{\delta},\ \mathbf{z} = \mathbf{y},\ \mathbf{z} = \mathbf{w}, \end{aligned} \tag{4.4}$$

where $\mathbf{z}$, $\mathbf{y}$ and $\mathbf{w}$ are newly introduced variables, for ease of notation let $\mathcal{D}_{(q-1)P+p} = \mathcal{G}_{p,q}$,

and $h(\mathbf{w})$ is an indicator function with respect to the constraints of problem (4.2),

$$h(\mathbf{w}) = \begin{cases} 0 & \text{if } (\mathbf{x}_0 + \mathbf{w}) \in [0,1]^n,\ \|\mathbf{w}\|_\infty \leq \epsilon, \\ \infty & \text{otherwise.} \end{cases} \tag{4.5}$$

ADMM is performed by minimizing the augmented Lagrangian of problem (4.4),

$$\begin{aligned} L(\mathbf{z}, \boldsymbol{\delta}, \mathbf{y}, \mathbf{w}, \mathbf{u}, \mathbf{v}, \mathbf{s}) = f(\mathbf{z} + \mathbf{x}_0) + \gamma D(\boldsymbol{\delta}) + \tau \textstyle\sum_{i=1}^{PQ} \|\mathbf{y}_{\mathcal{D}_i}\|_2 + h(\mathbf{w}) + \mathbf{u}^T(\boldsymbol{\delta} - \mathbf{z}) \\ + \mathbf{v}^T(\mathbf{y} - \mathbf{z}) + \mathbf{s}^T(\mathbf{w} - \mathbf{z}) + \tfrac{\rho}{2}\|\boldsymbol{\delta} - \mathbf{z}\|_2^2 + \tfrac{\rho}{2}\|\mathbf{y} - \mathbf{z}\|_2^2 + \tfrac{\rho}{2}\|\mathbf{w} - \mathbf{z}\|_2^2 \end{aligned} \tag{4.6}$$

where $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{s}$ are Lagrangian multipliers, and $\rho > 0$ is a given penalty parameter. ADMM splits all of optimization variables into *two* blocks and adopts the following iterative scheme,

$$\{\boldsymbol{\delta}^{k+1}, \mathbf{w}^{k+1}, \mathbf{y}^{k+1}\} = \arg\min_{\boldsymbol{\delta}, \mathbf{w}, \mathbf{y}} L(\boldsymbol{\delta}, \mathbf{z}^k, \mathbf{w}, \mathbf{y}, \mathbf{u}^k, \mathbf{v}^k, \mathbf{s}^k), \tag{4.7}$$

$$\mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} L(\boldsymbol{\delta}^{k+1}, \mathbf{z}, \mathbf{w}^{k+1}, \mathbf{y}^{k+1}, \mathbf{u}^k, \mathbf{v}^k, \mathbf{s}^k), \tag{4.8}$$

$$\begin{cases} \mathbf{u}^{k+1} = \mathbf{u}^k + \rho(\boldsymbol{\delta}^{k+1} - \mathbf{z}^{k+1}), \\ \mathbf{v}^{k+1} = \mathbf{v}^k + \rho(\mathbf{y}^{k+1} - \mathbf{z}^{k+1}), \\ \mathbf{s}^{k+1} = \mathbf{s}^k + \rho(\mathbf{w}^{k+1} - \mathbf{z}^{k+1}), \end{cases} \tag{4.9}$$

where k is the iteration index, steps (4.7)-(4.8) are used for updating primal variables, and the last step (4.9) is known as the dual update step. We emphasize that the crucial property of the proposed ADMM approach is that, as we demonstrate in Proposition 1, the solution to problem (4.7) can be found in parallel and exactly.

Proposition 1 *When $D(\boldsymbol{\delta}) = \|\boldsymbol{\delta}\|_2^2$, the solution to problem (4.7) is given by*

$$\boldsymbol{\delta}^{k+1} = \tfrac{\rho}{\rho+2\gamma}\mathbf{a}, \tag{4.10}$$

$$[\mathbf{w}^{k+1}]_i = \begin{cases} \min\{1-[\mathbf{x}_0]_i, \epsilon\} & b_i > \min\{1-[\mathbf{x}_0]_i, \epsilon\} \\ \max\{-[\mathbf{x}_0]_i, -\epsilon\} & b_i < \max\{-[\mathbf{x}_0]_i, -\epsilon\} \\ b_i & \textit{otherwise}, \end{cases} \quad \textit{for } i \in [n], \tag{4.11}$$

$$[\mathbf{y}^{k+1}]_{\mathcal{D}_i} = \left(1 - \tfrac{\tau}{\rho\|[\mathbf{c}]_{\mathcal{D}_i}\|_2}\right)_+ [\mathbf{c}]_{\mathcal{D}_i},\ i \in [PQ], \tag{4.12}$$

where ***a***

Definition 4.3.1 $z^k - \mathbf{u}^k/\rho$, $\mathbf{b}$

Definition 4.3.2 $z^k - \mathbf{s}^k/\rho$, $\mathbf{c}$

Definition 4.3.3 $z^k - \mathbf{v}^k/\rho$, $(x)_+ = x$ *if* $x \geq 0$ *and* 0 *otherwise,* $[\mathbf{x}]_i$ *denotes the* i*th element of* $\mathbf{x}$*, and* $[\mathbf{x}]_{\mathcal{D}_i}$ *denotes the sub-vector of* $\mathbf{x}$ *indexed by* $\mathcal{D}_i$*.*

It is clear from Proposition 1 that introducing auxiliary variables does not increase the computational complexity of ADMM since (4.10)-(4.12) can be solved in parallel. Moreover, if another distortion metric (different from $D(\boldsymbol{\delta}) = \|\boldsymbol{\delta}\|_2^2$) is used, then ADMM only changes at the $\boldsymbol{\delta}$-step (4.10).

We next focus on the $\mathbf{z}$*-minimization step (4.8), which can be equivalently transformed into*

$$\underset{\mathbf{z}}{minimize} \quad f(\mathbf{x}_0+\mathbf{z}) + \frac{\rho}{2}\|\mathbf{z}-\mathbf{a}'\|_2^2 + \frac{\rho}{2}\|\mathbf{z}-\mathbf{b}'\|_2^2 + \frac{\rho}{2}\|\mathbf{z}-\mathbf{c}'\|_2^2, \tag{4.13}$$

where $\mathbf{a}'$

Definition 4.3.4 $\boldsymbol{\delta}^{k+1} + \mathbf{u}^k/\rho$, $\mathbf{b}'$

Definition 4.3.5 $w^{k+1} + \mathbf{s}^k/\rho$, *and* $\mathbf{c}'$

Definition 4.3.6 $y^{k+1} + \mathbf{v}^k/\rho$.

We recall that attacks studied in this paper belongs to 'first-order' adversaries [175], which only have access to gradients of the loss function f. Spurred by that, we solve problem (4.13) via a linearization technique that is commonly used in stochastic/online ADMM Specifically, we replace the function f with its first-order Taylor expansion at the point $\mathbf{z}^k$ by adding a Bregman divergence term $(\eta_k/2)\|\mathbf{z}-\mathbf{z}^k\|_2^2$. As a result, problem (4.13) becomes

$$\begin{aligned}\underset{\mathbf{z}}{\text{minimize}} \quad & (\nabla f(\mathbf{z}^k + \mathbf{x}_0))^T(\mathbf{z} - \mathbf{z}^k) + \frac{\eta_k}{2}\|\mathbf{z} - \mathbf{z}^k\|_2^2 + \frac{\rho}{2}\|\mathbf{z} - \mathbf{a}'\|_2^2 \\ & + \frac{\rho}{2}\|\mathbf{z} - \mathbf{b}'\|_2^2 + \frac{\rho}{2}\|\mathbf{z} - \mathbf{c}'\|_2^2,\end{aligned} \tag{4.14}$$

where $1/\eta_k > 0$ is a given decaying parameter, e.g., $\eta_k = \alpha\sqrt{k}$ for some $\alpha > 0$, and the Bregman divergence term stabilizes the convergence of $\mathbf{z}$-minimization step. It is clear that problem (4.14) yields a quadratic program with the closed-form solution

$$\mathbf{z}^{k+1} = (1/(\eta_k + 3\rho))\left(\eta_k\mathbf{z}^k + \rho\mathbf{a} + \rho\mathbf{b} + \rho\mathbf{c} - \nabla f(\mathbf{z}^k + \mathbf{x}_0)\right). \tag{4.15}$$

In summary, the proposed ADMM algorithm alternatively updates (4.7)-(4.9), which yield closed-form solutions given by (4.10)-(4.12) and (4.15). The convergence of linearized ADMM for nonconvex optimization was recently proved by [190], and thus provides theoretical validity of our approach. Compared to the existing solver for generation of adversarial examples [163, 171], our algorithm offers two main benefits, *efficiency* and *generality*. That is, the computations for every update step are efficiently carried out, and our approach can be applicable to a wide class of attack formulations.

4.3.2.2 Enhanced Bit Flip

Earlier, a novel structural adversarial attack, Min-invasive , was discussed. This section discusses our enhanced version of the bit perturbation attack.

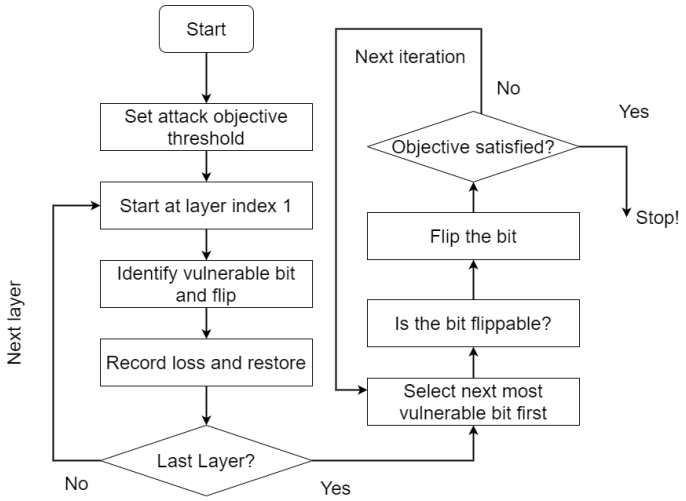


Figure 4.16: Flow of the enhanced bit flip algorithm

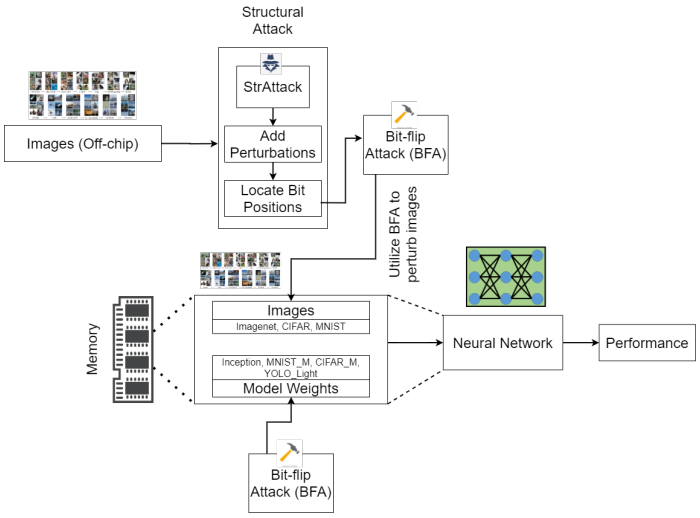


Figure 4.17: Overview of the proposed Adv-Exploit attack

4.3.2.2.1 Rowhammer Attack Rowhammer is a fault injection attack that exploits a DRAM by forcing bit flips. Previous work in [169] demonstrated that frequent accesses to a DRAM row causes voltage toggling on its word line. Frequent access to a row causes the earlier discharge of the capacitor, thus leading to state toggling in the neighboring row. The target of the attack is to discharge the capacitor sufficiently before the next refresh cycle causing the memory cell to lose its state resulting in a bit flip.

4.3.2.2.2 Enhanced Bit Flip Our proposed enhanced bit flip performs data perturbation via bit flipping strategy. Data (bit) flipping in-memory is a non-trivial task. It is not advisable nor feasible to randomly flip bits expecting to degrade the DNN performance, as that could prove counterproductive to the goal of the attack. Hence, we propose to do a two-rung strategy to successfully perturb the data. First, we utilize a loss-based bit ranking that grades the most vulnerable bits of the weight parameters of the DNN. It is as crucial to evaluate if the ranked vulnerable bits could actually be perturbed in the memory as it is to find the vulnerable bits. Hence, our second step is to generate an array of vulnerable bits that could be perturbed by modeling the system constraints.

Loss-based bit Grading: Figure 4.16 shows the flow of our enhanced bit flip attack. The attack initiates with an objective threshold that is the percentage loss expected during the attack. The attack will stop after the threshold is achieved. The attack then initiates a sensitivity analysis where it takes a weight parameter and flips the vulnerable bits. The vulnerable bits are those that result in DNN accuracy degradation when perturbed. After the vulnerable bits are identified at a particular layer, the attack starts to flip the bits one at a time and record the resulting loss during inferencing; all the vulnerable bit positions are recorded along with the loss. The attack then ranks these loss values corresponding to the vulnerable bits.

Flipping based on System Constraints It is interesting to note that not all bits in the memory can be controlled by the attacker, or in other words, cannot be rowhammered. Because some bits may not be perturbed and hence it is necessary to know the system constraints before marking the bit as a vulnerable one. For the system constraints, the

attack red flags bits that cannot be perturbed. Once the loss-based bit grading is complete, the attack chooses the next most vulnerable bit in the list and flips it as per the system constraints. If a bit flip cannot provide sufficient loss, multiple bits are flipped to achieve the targeted loss threshold. DNN accuracy is again recorded for the total bits perturbed to ensure the loss.

4.3.2.2.3 Adversarial Attack-assisted Input Data Perturbation Thus far, we discussed about our proposed adversarial attack, Min-invasive , and the enhanced bit perturbation attack, enhanced bit flip . The Min-invasive capable of generating perturbations to misclassify input inference data, which are the input images to DNN. The aim is to force the DNN to misclassify an input. Similarly, with enhanced bit flip we can force bit flipping of the weight data parameters such that DNN accuracy is degraded. We utilize the Min-invasive generated perturbations to drive the bit flipping attack to perturb image data in memory. This is in contrast to the previous perturbation works like [163] which generate perturbation in the images offline - the attacker has access to the dataset. Our proposed Adv-Exploit is capable of introducing perturbations in the input image data by rowhammering the bits in memory. But, instead of the enhanced bit flip attack explained above, it takes the generated perturbations to perform rowhammering to introduce faults. After Min-invasive generates the perturbations, the attack takes a note of the pixel offsets in the images. These offsets are then provided to the enhanced bit flip to flip bits of the input data in memory. In this case, the loss-based bit grading is not utilized as the vulnerable bit positions are already generated by the adversarial attack, Min-invasive . This renders a more cognitive attack strategy where the perturbations in the inference data could be prepared offline and the fault injection using enhanced bit flip can be performed online during process execution.

4.3.2.3 Integrating all the Parts

The ultimate goal of the proposed Adv-Exploit is to challenge DNN resiliency by launching two-pronged approach attack. One, by injecting perturbations in the input inference data

in memory using enhanced bit flip driven by the Min-invasive adversarial attack; Second, by utilizing enhanced bit flip with loss-based bit grading to inject faults in the weight parameters of the DNN. Below if a process flow of the integrated Adv-Exploit ; also refer to Figure 4.17:

- The Adv-Exploit generates perturbations on the input images using the Min-invasive attack.
- The offsets of the pixel values with injected faults are recorded by the attack.
- The offsets are used by the enhanced bit flip to perform rowhammering to perturb bits of the image(s).
- Similarly, the enhanced bit flip with loss-based bit grading is used to inject faults in the weight parameters of the DNN model.
- Both the perturbed images and weight parameters will cause accuracy degradation of the DNN model.

The novelty of the proposed Adv-Exploit is that with its two-pronged attack approach the DNN performance is degraded to below a random guess, which is unacceptable given the mission critical deployment of the such DNNs in modern world.

4.3.3 Results

4.3.3.1 Performance of the Proposed Min-invasive

We evaluate the performance of the proposed Min-invasive on three image classification datasets, MNIST , CIFAR-10 and ImageNet. To make fair comparison with the C&W ℓ_2 attack [163], we use ℓ_2 norm as the distortion function $D(\boldsymbol{\delta}) = \|\boldsymbol{\delta}\|_2^2$. And we also compare with FGM [165] and IFGSM ℓ_2 attacks [164] as a reference. We evaluate attack success rate (ASR)[13] as well as ℓ_p distortion metrics for $p \in \{0, 1, 2, \infty\}$.

[13]The percentage of adversarial examples that successfully fool DNNs.

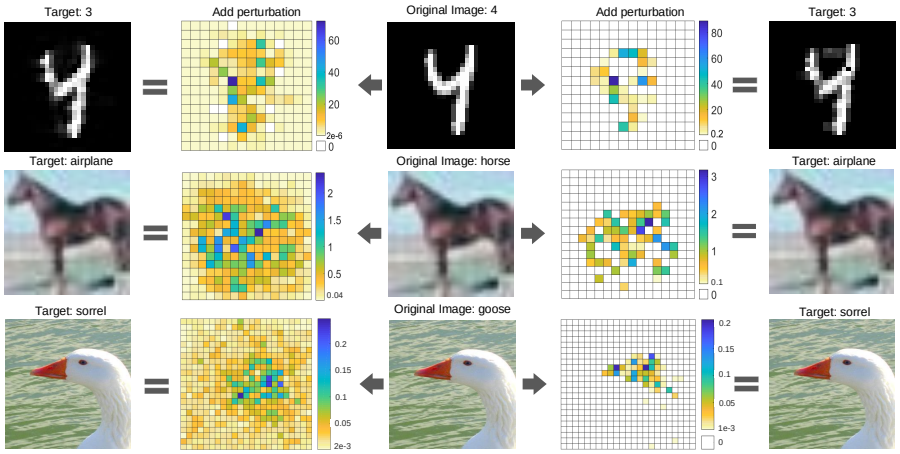


Figure 4.18: C&W attack vs Min-invasive . Here each grid cell represents a 2×2, 2×2, and 13×13 small region in MNIST, CIFAR-10 and ImageNet, respectively. The group sparsity of perturbation is represented by heatmap. The colors on heatmap represent average absolute value of distortion scale to $[0, 255]$. The left two columns correspond to results of using C&W attack. The right two columns show results of Min-invasive attack

For each attack method on MNIST or CIFAR-10, we choose 1000 original images from the test dataset as source and each image has 9 target labels. So a total of 9000 adversarial examples are generated for each attack method. On ImageNet, each attack method tries to craft 900 adversarial examples with 100 random images from the test dataset and 9 random target labels for each image.

Fig. 4.18 compares adversarial examples generated by Min-invasive and C&W attack on each dataset. We observe that the perturbation of the C&W attack has poor group sparsity, i.e., many non-zeros groups with small magnitudes. However, the attack success rate (ASR) of the C&W attack is quite sensitive to these small perturbations. As applying a threshold to have the same ℓ_0 norm as our attack, we find that only 6.7% of adversarial examples generated from C&W attack remain valid. By contrast, Min-invasive is able to highlight the most important group structures (local regions) of adversarial perturbations without attacking other pixels. For example, Min-invasive misclassifies a natural image

(4 in MNIST) as an incorrect label 3. That is because the pixels that appears in the structure of 3 are more significantly perturbed by our attack; see the top right plots of Fig. 4.18. Furthermore, the 'goose-sorrel' example shows that misclassification occurs when we just perturb a small number of non-sparse group regions on goose's head, which is more consistent with human perception.

By qualitative analysis, we report ℓ_p norms and ASR in Table 4.9 for $p \in \{0, 1, 2, \infty\}$. We show that Min-invasive perturbs much fewer pixels (smaller ℓ_0 norm), but it is comparable to or even better than other attacks in terms of ℓ_1, ℓ_2, and ℓ_∞ norms. Specifically, the FGM attack yields the worst performance in both ASR and ℓ_p distortion. On MNIST and CIFAR-10, Min-invasive outperforms other attacks in ℓ_0, ℓ_1 and ℓ_∞ distortion. On ImageNet, Min-invasive outperforms C&W attack in ℓ_0 and ℓ_1 distortion. Since the C&W attacking loss directly penalizes the ℓ_2 norm, it often causes smaller ℓ_2 distortion than Min-invasive . We also observe that the overlapping case leads to the adversarial perturbation of less sparsity (in terms of ℓ_0 norm) compared to the non-overlapping case. This is not surprising, since the sparsity of the overlapping region is controlled by at least two groups. However, compared to C&W attack, the use of overlapping groups in Min-invasive still yields sparser perturbations. Unless specified otherwise, we focus on the case of non-overlapping groups to generate the most sparse adversarial perturbations. We highlight that although a so-called one-pixel attack [172] also yields very small ℓ_0 norm, it is at the cost of very large ℓ_∞ distortion. Unlike one-pixel attack, Min-invasive achieves the sparsity *without losing* the performance of ℓ_∞, ℓ_1 and ℓ_2 distortion.

$$\begin{aligned} \underset{\boldsymbol{\delta}}{\text{minimize}} \quad & f(\mathbf{x}_0 + \boldsymbol{\delta}) + \gamma D(\boldsymbol{\delta}) \\ \text{subject to} \quad & (\mathbf{x}_0 + \boldsymbol{\delta}) \in [0,1]^n,\ \|\boldsymbol{\delta}\|_\infty \leq \epsilon \\ & \delta_i = 0,\ \text{if } i \in \mathcal{S}_\sigma, \end{aligned} \tag{4.16}$$

Furthermore, we compare the performance of Min-invasive with the C&W ℓ_∞ attack and IFGSM while attacking the robust model [175] on MNIST. We remark that all the considered

Table 4.9: Adversarial attack success rate (ASR) and ℓ_p distortion values for various attacks

Data Set	Attack Method	Best Case					Average Case					Worst Case				
		ASR	ℓ_0	ℓ_1	ℓ_2	ℓ_∞	ASR	ℓ_0	ℓ_1	ℓ_2	ℓ_∞	ASR	ℓ_0	ℓ_1	ℓ_2	ℓ_∞
MNIST	FGM	99.3	456.5	28.2	2.32	0.57	35.8	466	39.4	3.17	0.717	0	N.A.	N.A.	N.A.	N.A.
	IFGSM	100	549.5	18.3	1.57	0.4	100	588	30.9	2.41	0.566	99.8	640.4	50.98	3.742	0.784
	C&W	100	479.8	13.3	1.35	0.397	100	493.4	21.3	**1.9**	0.528	99.7	524.3	29.9	**2.45**	0.664
	Min-invasive	100	**73.2**	10.9	1.51	**0.384**	100	**119.4**	18.05	2.16	**0.47**	100	**182.0**	26.9	2.81	**0.5**
	+overlap	100	84.4	**9.2**	**1.32**	0.401	100	157.4	**16.2**	1.95	0.508	100	260.9	**22.9**	2.501	0.653
CIFAR-10	FGM	98.5	3049	12.9	0.389	0.046	44.1	3048	34.2	0.989	0.113	0.2	3071	61.3	1.76	0.194
	IFGSM	100	3051	6.22	0.182	0.02	100	3051	13.7	0.391	0.0433	100	3060	22.9	0.655	0.075
	C&W	100	2954	6.03	0.178	**0.019**	100	2956	12.1	0.347	**0.0364**	99.9	3070	16.8	**0.481**	**0.0536**
	Min-invasive	100	**264**	**3.33**	0.204	0.031	100	**487**	7.13	0.353	0.050	100	**772**	**12.5**	0.563	0.075
	+overlap	100	295	3.35	**0.169**	0.029	100	562	**7.05**	**0.328**	0.047	100	920	12.9	0.502	0.063
ImageNet	FGM	12	264917	152	0.477	**0.0157**	2	263585	51.3	**0.18**	**0.00614**	0	N.A.	N.A.	N.A.	N.A.
	IFGSM	100	267079	299.32	0.9086	0.02964	100	267293	723	2.2	0.0792	98	267581	1378	4.22	0.158
	C&W	100	267916	127	**0.471**	0.016	100	263140	198	0.679	0.03	100	265212	268	**0.852**	**0.041**
	Min-invasive	100	**14462**	**55.2**	0.719	0.058	100	**52328**	**152**	1.06	0.075	100	**80722**	**197**	1.35	0.122

attack methods are performed under the same ℓ_∞-norm based distortion constraint with an upper bound $\epsilon \in \{0.1, 0.2, 0.3, 0.4\}$. Here we obtain a (refined) Min-invasive subject to $\|\boldsymbol{\delta}\|_\infty \leq \epsilon$ by solving problem (4.16) at $\gamma = 0$. In Table 4.10, we demonstrate the ASR and the number of perturbed pixels for various attacks over 5000 (untargeted) adversarial examples. The ASR define as the proportion of the final perturbation results less than given $\epsilon \in \{0.1, 0.2, 0.3, 0.4\}$ bound over number of test images. Here an successful attack is defined by an attack that can fool DNNs and meets the ℓ_∞ distortion constraint. As we can see, Min-invasive can achieve the similar ASR compared to other attack methods, however, it perturbs a much less number of pixels.

Table 4.10: Attack success rate (ASR) and ℓ_0 norm of adversarial perturbations for various attacks against robust adversarial training based defense on MNIST.

	ASR at $\epsilon = 0.1$	ASR at $\epsilon = 0.2$	ASR at $\epsilon = 0.3$	ASR at $\epsilon = 0.4$	ℓ_0
IFGSM	0.01	0.02	0.09	0.94	654
C&W ℓ_∞ attack	0.01	0.02	0.10	0.96	723
Min-invasive	0.01	0.02	0.10	0.99	279

4.3.3.2 Performance of the Proposed Adv-Exploit

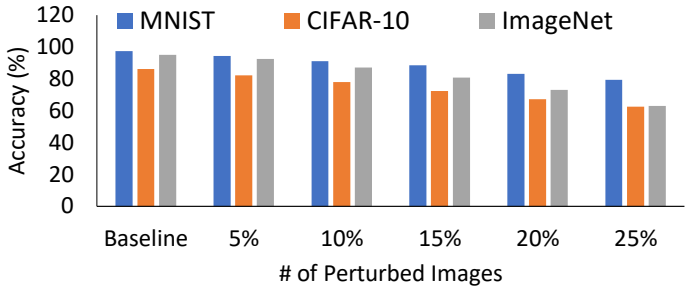


Figure 4.19: Performance of the DNN under Adversarial Attack

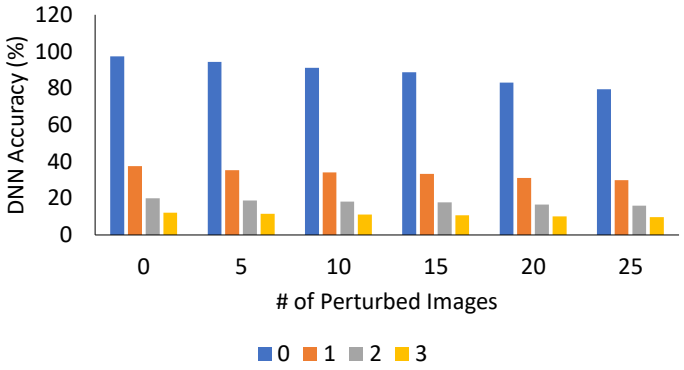


Figure 4.20: Overall performance of the proposed Adv-Exploit

In this section, we discuss the performance of the proposed Adv-Exploit as a whole. The Adv-Exploit integrates a sophisticated adversarial attack, Min-invasive , which requires least number of perturbation bits in the input image, our adversarial attack driven bit perturbation attack, enhanced bit flip , and the bit perturbation attack for model weight

parameters by fault injection using rowhammer. Figure 4.19 presents the DNN accuracy results under the Min-invasive adversarial attack. The figure shows the DNN accuracy of the baseline and increasing levels of perturbations. The baseline is the DNN accuracy without any adversarial perturbations. The results are presented for MNIST, CIFAR-10, and ImageNet datasets. It is observed that the DNN classification accuracy reduces as we increase the amount of perturbed images fed to the DNN classifier. The adversary can decide the amount of perturbations injected while processing to reduce the DNN accuracy.

Figure 4.20 presents the overall accuracy of the DNN model under both the Min-invasive and enhanced bit flip attacks. In other words, the figure shows results when both the images and model weight parameters are perturbed. It can be observed that the accuracy reduces as the amount of perturbed images and number of bit flips are increased. The 0,1,2, and 3 are the number of hardware bit flips done to the model weight parameters.

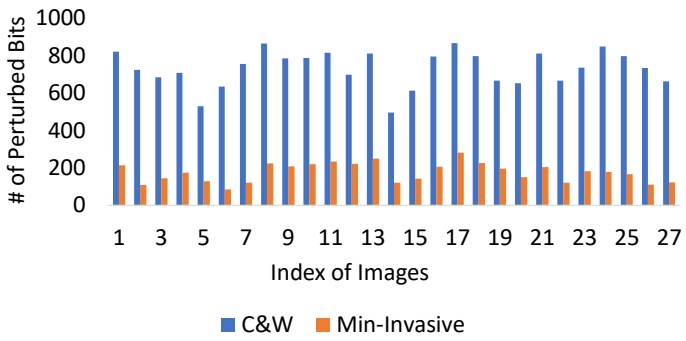


Figure 4.21: Comparison of Carlini & Wagner (C&W) attack and Min-invasive attack for CIFAR-10

4.3.3.3 Evaluation of the Bit Flip Cost Per Image

It is crucial to evaluate the cost of perturbing image data. As discussed earlier, we need the adversarial attack to perturb as less bits as possible to reduce the bit perturbation cost per

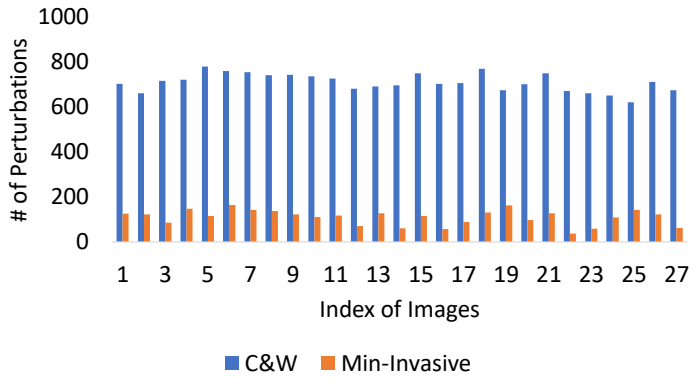


Figure 4.22: Comparison of Carlini & Wagner (C&W) attack and Min-invasive attack for MNIST

image. Hence, in this section the evaluation is made to compare bit perturbations required for the C%W attack and proposed Min-invasive adversarial attack. The C%W attack is considered here as it the state-of-the-art adversarial attack requiring less amount of bits to be perturbed per image. Figures 4.21 and 4.22 present the number of bits perturbed - corresponding to image pixels - for a sample set of images. The X-axis is for the index of images and not the number of images per tick. It can be observed that the C%W attack requires significantly more bits compared to the Min-invasive attack for both the CIFAR-10 and MNIST datasets. A similar pattern is observed with the ImageNet dataset. On an average, our proposed attack requires 16% less bits compared to the C%W for MNIST, and 24% less bits on the CIFAR-10 dataset.

4.3.3.4 Potential Mitigation

This section discusses the potential mitigation strategies that may be used to thwart the proposed Adv-Exploit . These is a brief overview of the potential solutions and it needs further investigation to evaluate their efficacy.

1. Disable “Rowhammer” effect: Allow only a certain number of accesses before the next

refresh cycle. Or bring memory cell contents to a cache-like memory, that are resilient against rowhammer attacks.

2. Adversarial training of the DNN model: Adversarial training may render the DNN model robust against such attacks. The DNN model could be trained on adversarially generated images. The downside of such a strategy is they may still be vulnerable to zero day attacks.

3. Execute a DNN model in a secure space: Executing the model in a secure environment. Secure environment should ensure that the memory of the DNN classification process is not visible to other processes. The downside is that owing to large weight parameters required for the modern DNN models it may not be feasible to reserve a large separate memory. Such an overhead may not be acceptable. To address this, designers could perform a sensitivity analysis and protect only those bits that contribute most to the DNN inferencing. In other words, protecting the most sensitive bits.

Chapter 5: Machine Learning-Assisted Hardware Trojan Attack and Defense Against Network-on-Chips

5.1 Machine Learning-Assisted Hardware Trojan Attack with a Sophisticated Attack Model Equipped with Data Augmentation

The ever increasing need for computing power in the modern digital world requires powerful platforms, such as, multi-core processors or blade servers, and embedded platforms with multiple processing nodes. Interconnection networks such as a Network-on-Chip (NoC) [191] that connects processing units to memory and peripherals impact the capabilities of the current systems, as efficiency of the data movement plays a pivotal role. Due to the crucial role of the NoC as a communication fabric, it forms one of the largest surface areas for attack in the system. On the other hand, to alleviate the operating costs, many chip vendors are becoming fabless. Modern System-on-chips (SoCs) use Third Party IPs (3PIPs) to minimize the time-to-market and design costs; these 3PIPs may be procured from untrusted organizations which is a potential security concern. An adversary either at the foundry or at the 3PIP design house can introduce a malicious circuitry, to jeopardize an SoC, which is known as a Hardware Trojan (HT) [192,193]. HTs can be utilized for various malicious purposes, including information leakage, functionality subversion and battery exhaustion [192, 194, 195].

Considering the critical role played by the NoCs, embedding a HT that exploits the interconnection backbone can reveal the communication patterns in the system. Leakage of such crucial information to a remote attacker can reveal important information pertaining to the application suites running on the system, thus compromising the user profile. Further,

such information leakage can enable more severe attacks on the systems on which compromised NoCs are deployed. For instance, an adversary obtaining secure military information through a HT deployed in a switch can subvert the military backbone, thus posing a severe threat to national security [28].

In this work, we first introduce a lightweight NoC-based HT, which, in its simplistic form, is a counter; the HT when inserted in one or a few switches of the NoC can count the number of packets traversing the specific switches over a time window. The HT periodically packetizes this count information and sends it to an external attacker program for payload analysis. This packetized count, which is the HT payload, can be subsequently analyzed by the external attacker using data processing techniques to infer information about application suites from the retrieved HT payload data containing packet traversal frequencies through specific switches.

The attacker maps the packet traversal frequencies at the switches to the application suites by analyzing the retrieved information using sophisticated Machine Learning (ML) algorithms. In this work, we demonstrate that the application suites running in the system can be detected with only 4 or 8 counter-based HTs with more than 95% accuracy using ML techniques. In this baseline attacker model, the packet count data is shared with a remote attacker over a long observation window by the HT. Hence, the attacker has access to a large volume of packet count data from the embedded HTs.

Our previous work in [196] discusses controlled random routing for an attacker that has access to a large observation window. Such an attack model requires plethora of data samples, which may not be feasible in all scenarios; the attacker may not have access to a considerably sized observation window. Hence, this would render scarce data for training the machine learning model, leading to less accuracy in detecting applications running on the system. On the contrary, this work not only includes the work described and demonstrated in [196] but also considers a sophisticated adversary which is equipped with data augmentation for data paucity scenarios. This work also augments the work in [196] by presenting the results with pseudo-adaptive west-first routing, comparison of the

baseline and advanced attacker model, comparison of other machine learning models with artificial neural network (ANN) model, and the efficacy of the proposed controlled random routing on the baseline and advanced attacker models.

In this work, we discuss scenarios where an attacker may not have access to an uniformly long observation window to collect packet count data leaked by the embedded HT. In such situations, the attacker model, deployed by utilizing ML methods, would not deliver sufficient accuracy in regards to the deciphered user profile information. Hence, we utilize data augmentation methodologies to strengthen the attacker model; we demonstrate that such techniques could render the attack more robust thus compensating for the loss occurring during the observation window. We also demonstrate that such advanced attacker model is capable of detecting the application suites running on a multi/many processor system with 98% accuracy. This is possible because specific routing protocols are proposed for these particular system configurations, which, when adopted, result in an application-specific traffic patterns. Therefore, analyzing the traffic patterns with the help of the HTs and machine learning can enable inferring the application(s) being executed in the system, thus compromising user confidentiality.

We propose a novel Simulated Annealing (SA)-based randomized routing algorithm to defend the NoC against such a HT threat. Simulated Annealing is a type of genetic algorithm that allows sub-optimal traversal of the search space for optimization to avoid being stuck in local optima [29]. Random packet routing over the interconnection can severely degrade performance of the system due to the fact that packets may not be routed over the shortest paths. Therefore, to achieve a desired trade-off between the defense and loss in performance, we propose a parameterized SA-based approach instead of simply adopting random routing.

For the SA-based random routing, the path for each packet is unpredictable and therefore, makes the mapping of packet traversal frequency through specific switches and corresponding applications unreliable. We demonstrate using cycle-accurate simulations that the proposed SA-based randomized routing can reduce the effectiveness of the attack for

both the baseline and advanced attacker model.

To the best of our knowledge, this work is a first of its kind that demonstrates that by monitoring traffic patterns in a Network-on-Chip through Hardware Trojan the user profile can be compromised. We also propose to defend the system against such attack with controlled random routing.

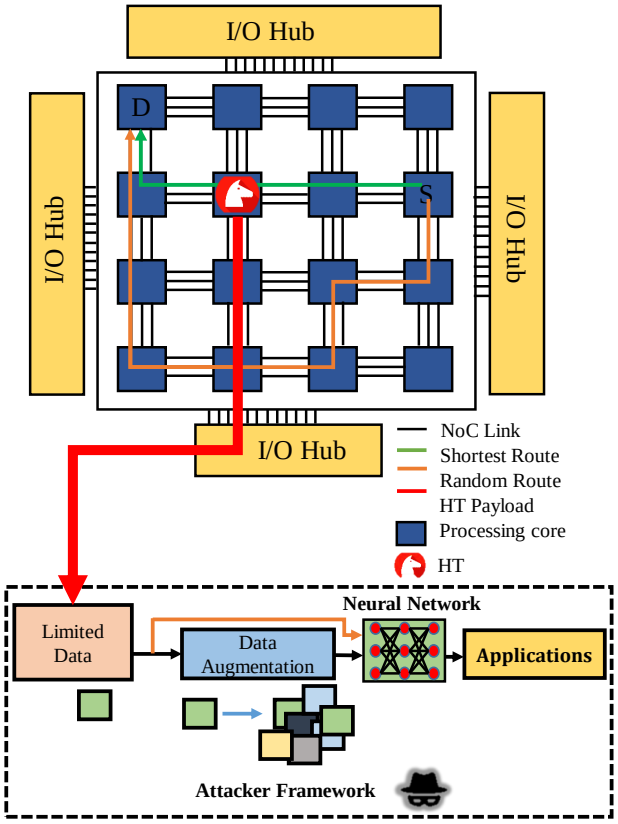


Figure 5.1: Multi-core NoC with proposed threat model

5.1.1 Threat Model

We consider a multi-tenanted server or data center with multi-core processors connected with a NoC. The NoC (3PIP) may be procured from a third-party organization that is different from the system integration designer, one or more HTs could be inserted in the switches of the NoC during the design and fabrication process. These deployed HTs could be an uncomplicated counter that counts the number of packets traveling the switch over an observation window and forwarding that information to an external adversary for off-chip analysis. A sufficiently motivated attacker can then employ large-scale compute capabilities and algorithms, such as Machine Learning (ML), to perform a traffic analysis attack on the packet count from one or more switches. This kind of attack can reveal the applications running on the system, because the traffic interaction in multi-core processors is always application dependent, and thus, compromise user privacy by revealing the applications that are being executed. This constitutes what we term as our baseline model. Further, there may be scenarios where the attacker is unable to observe the packet count samples to get a comprehensive view of the data. In such situations, the attacker needs to utilize data augmentation strategies to compensate for the lost data during observation. The data augmentation techniques can generate synthetic data samples similar to the actual observed data. This renders a robust attacker model. We denote this as advanced attacker threat model. Figure 5.14 shows both the baseline and advanced attack model discussed here. The key difference between the baseline and the advanced attacker model is the presence of data augmentation.

5.1.2 Hardware Trojan (HT) Design

In this section, we describe various aspects of the baseline and advanced attack model described earlier. The Trojan is a counter capable of counting the number of packets routed by the switch where the Trojan is inserted. We assume the HT is inserted inside the routing block of the NoC or interconnection switch as shown in Figure 5.15. There may be one or multiple HT embedded switches in the system. The functionality of the HT is that it will

count up when a new packet accesses the switch to get routed to its next destination. After counting for a pre-determined time window, the HT packetizes the count information by appending a destination address and other header information; The HT then inserts the packet into the NoC as the HT payload. Such processing systems are typically multi-user platforms where different parts of the processor are virtualized and allocated to various applications from a multitude of users. Because the attacker can impersonate as one of the users of this shared processor that hosts multiple users simultaneously, the destination of the payload will be a legitimate I/O port where the adversary is hosted. Therefore, the packet is unlikely to be flagged by any security measure in the system, as it does not exhibit any anomaly as compared to other packets in the system. The payload is then analyzed by the external attacker.

The HT is a 16-bit counter that counts the number of packets being routed through the switch. This count is packetized every five thousand cycles and forwarded to the external adversary. Another 16-bit down counter functioning as a timer maintains the observation window; after the timer expires the payload is launched by the HT. As the packets consist of multiple flits [197], taking several clock cycles to be routed through a switch [197], the maximum packet count will be less than the duration of the counting and hence the 16-bit counter is sufficient. This particular HT also does not alter the data path of the legitimate packets getting routed in the NoC as it is not sequential to the routing logic and the counting happens in parallel to the routing. Therefore, timing analysis can not detect the embedded HT(s). Moreover, even with a few such HTs of moderate sizes they remain undetectable in the large multi/many-core processors owing to its less area overhead and power consumption. A single NoC switch with a size of ∼30-40K gates [197] is orders of magnitude more complex compared to the HT.

5.1.2.1 Hardware Trojan Trigger Design

We consider a multi-user environment like cloud computing where different threads can be co-scheduled to leverage the same underlying processing elements and interconnection

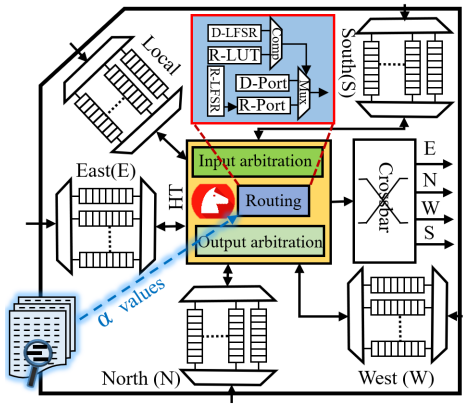


Figure 5.2: NoC switch components with inserted HT.

architecture. In such multi-user platform, due to the huge variety of the user applications, an attacker can easily launch an accomplice thread to trigger HT inserted inside NoC fabric.

To trigger the inserted HT , the accomplice thread transmits a multicast/broadcast traffic pattern similar to any coherence and system-level control messages with specific data sequence in its header flit. The Trojan inserted in the NoC switch works as a state machine consisting of four states that are shown in the Figure 5.3. In the *dormant state*, the HT snoops the header flit to identify the sequence. Once it finds the particular sequence, it proceeds to the *wake-up state*. Once an HT wakes up, it sends an ACK signal with another specific data sequence to the core hosting the accomplice thread to establish a covert communication channel between them. The HT changes its state to *count state* where it starts counting the number of packets traversed through the compromised (Trojan embedded) switch for a particular time window. After this time interval, it is proceeds to the *payload state* where it sends the packet count as payload to the accomplice thread and wait for the next instruction. Based on the next instruction, the HT can either go back to the count state to repeat the process or can return to the dormant state.

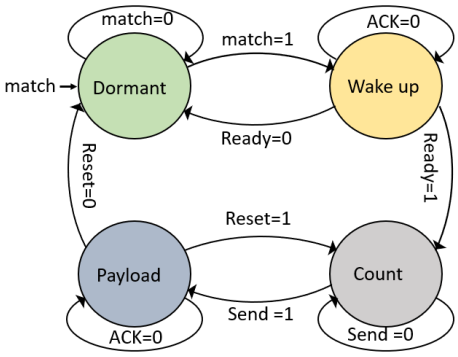


Figure 5.3: Hardware Trojan state diagram.

5.1.2.2 Off-Chip Hardware Trojan Payload Analysis

We craft an HT payload analyzer using machine learning techniques to demonstrate the extent of the threat such an HT based attack can pose. To deploy the payload analyzer, we select an Artificial Neural Network (ANN) due to its ability to map complex patterns efficiently as well as its resilience to variations. We initially construct a dataset consisting of eighty features (payloads) obtained from simulating a system with 64 cores and 16 memory controllers; the dataset has permutations of twelve different applications with not more than three applications running simultaneously on the system . We consider a maximum of three applications signifying up to three independent users hosted on the system executing three applications simultaneously, although a larger number of users can be easily accommodated only requiring creating a training dataset with that assumption. The data is created in the same fashion as an actual attacker who may be able to create using an emulator or a simulator of the real system. Specifically, we use a cycle-accurate simulator described in Section 5.1.4 that monitors the movement of packets broken down into flits in a NoC. Various permutations of the applications from a common parallel benchmark suites [198] are executed on the simulator to create traffic traces that are visible to the HTs; the traffic traces consisting of the number of packets traversing the HT embedded switches are used

as the training dataset.

As the traffic patterns (packet count) depend on the executing applications, the on-chip traffic, as well as the mapping of the applications to the cores, the observed patterns will not be constant for a given set of applications. Furthermore, the insignificant number of packets generated by the HTs to share the packet count data with an accomplice thread will also contribute to the number of packets traversing a NoC switch.

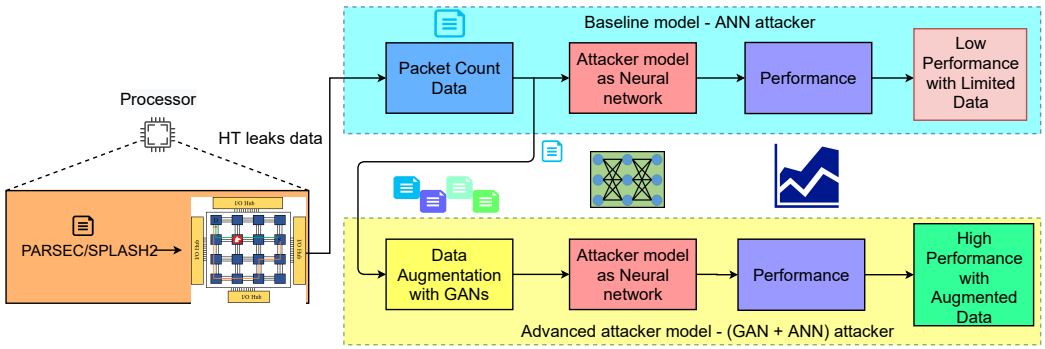


Figure 5.4: Attacker Model with Data Augmentation using GANs

To capture the variations due to network congestion, we add a random noise following a negative binomial distribution with our simulation data as a part of data processing; The ANN is made robust to such variations by training on the noise added dataset. For generating the random noise, we consider a M/M/1 queuing model for the NoC switches and a Poisson arrival process for the incoming packets in each cycle. For such queuing model, the number of packets in the switch buffers at every simulation cycle follows a geometric distribution and the total number of packets in the buffers over the simulation period can be computed as the convolution of the geometric distributions resulting in a negative binomial distribution [199], which is used for the random noise generation. We train our ANN with all the features available. This yields high accuracy in predicting the application(s) running on the system. However, the caveat here is that although the ANN could be trained on a

large set of features, it is not feasible to expect HT to snoop on all switches simultaneously. Doing so would contribute to a significant latency and overhead, eventually leading to HT detection. Thus, to reflect a real-world scenario, the attacker intends to insert a HT which focuses on the dominant features that still enable it to predict with high accuracy. We translated this approach by using correlation-based feature selection i.e., determine the switches in which the HTs need to be embedded (performed offline) that reduces latency without compromising the performance.

We deploy a 5 layer ANN (N-800-500-200-64-12 neurons; N represents number of features at input) with ReLU as activation function for hidden layers and softmax as the activation function for the output layer. There are twelve final outputs for the ANN, which are the number of individual applications executed on the system. To represent simultaneous execution of multiple applications, one-hot encoding is utilized. A 5-fold cross-validation is utilized to analyze performance, determined based on grid-search. As the ANN is deployed off-chip, the area and other overheads are not of concern, except the accuracy, precision, recall and F1-score that represent the performance of the ANN attacker model. In this work, the ANN model is consistent for both our baseline and advanced attacker model. The advanced attacker model is discussed next.

For a neural network to deliver better performance in predicting the application suite running on the system, a considerable number of samples (data) is needed. It may so happen that the attacker does not have access to uniformly long observation window of system under attack in all operating conditions - for all application combinations. Not having enough training samples for applications (or a mix of them) results in degradation of performance. Hence, in order to improve the attacker model's accuracy, the attacker may need a strategy for improving the performance. In scenarios where the attacker has access to limited data only, data augmentation strategies are required. Generative Adversarial Network (GANs) [200–203] is an unsupervised learning algorithm in machine learning that involves automatically discovering and learning the regularities or patterns in input data and generate new plausible samples. GANs comprise of a generator and a discriminator

model. The generator model is trained on the dataset to learn the patterns or distribution. Later, the generator model generates new samples. The discriminator model discriminates between original samples (from the dataset) and the generated samples. The GANs are extensively utilized in the image domain to generate new images or modify existing images. *To the best of our knowledge, this is the first work to have repurposed the GANs to augment the packet count data to build a robust adversary.*

Table 5.1: Architectural details of the discriminator, generator and artificial neural network (ANN) model

Parameter	**Discriminator**	**Generator**	**Neural Network**
Input image size	80x80x1		
Hidden layer	Convolution (two layers)	Dense, Conv2DTranspose, Conv2DTranspose	Five Dense Layers
Activation function	LeakyReLu	LeakyReLu	Relu (four layers)
Dropout	0.4		
Output layer	Dense with sigmoid activation	Conv2D with tanh activation	Softmax (output layer)
Optimizer	Adam with learning rate=0.0002 and beta_1=0.5		Adam
Loss function	Binary crossentropy		Categorical crossentropy

The entire process demonstrating the baseline and advanced attack model is presented in Figure 5.4. Our dataset consists of packet count data recorded by executing different application suites (SPLASH2 and PARSEC). The data is fed to the attacker model implemented as a neural network for training the network. Post training, new unseen packet count data samples are fed to the neural network for inference. For the baseline attack model, the performance of the neural network with entire dataset, without missing data, is shown in Figures 5.6 and 5.7. The accuracy with 16 features (number of HTs) is greater than 95%. But, for the advanced attacker model, the size of the data is small as the data instances are missing; we name this data as *limited data*. The attacker improves the robustness by augmenting this limited data using a GAN model. A GAN model, in its conventional utilization, is designed to function with images [200–206]. Hence, we converted the packet data to images by aligning the rows (samples) and columns (features) as a matrix to build a representation similar to images. The data points are scaled to represent grayscale images.

'Robust scaling' is used to scale the data points before feeding to the GAN model. Standard scaling is another option but outliers in data points can skew calculated mean and standard deviation, thus making it unfit for our application. The robust scaling technique ignores the outliers from calculation and then scale the variable post calculation. The resulting data has zero mean and median and a standard deviation of 1. The outliers are still a part of the data and depict same relative relationship to other data points. The generator is trained on the images; post generator training, the discriminator functions to bifurcate generated images from the original images.

The generator post training is utilized to generate new images (augmented data). The numerical data from the generated images is appended to the original data - to add more samples to the original data which compensates for the missing data samples as explained earlier. All the cumulative dataset is utilized to train the neural network, which represents the attacker, attempting to learn the application suites based on the input packet count data. The size of each image is 80×80 - given the fact that there are 80 switches; the data is organized such that the numerical data is represented as an image to be fed to the GANs. The original data used as training set to the GANs contains 50K images. We generate additional 50K images with augmentation, totaling to 100K images for ANN model evaluation. The generated images where each pixel is represented by a number is translated back to numerical data, which is later used for ANN performance evaluation. The performance in presence of data augmentation using GANs is restored to an acceptable level. Later in the results section, it is demonstrated that with data augmentation, the attacker can render the attack model more robust against missing data. This also serves as our motivation for deploying data augmentation using GANs for this work. The discriminator, generator and the ANN model's parameters are shown in Table 5.1.

5.1.3 Proposed Random Routing

In this section, we discuss the proposed live-lock and deadlock free routing methodology that ensures in-order packet delivery. The proposed random routing not only increases the

NoC resiliency against aforementioned and similar attacks, but also ensures lower latency for latency-sensitive messages.

5.1.3.1 Routing Methodology

To defend a NoC against the HT-based traffic analysis attack, the proposed routing mechanism introduces controlled randomness in selecting the routes a packet will take to reach the destination to obfuscate the traffic analysis mechanism. This is because, in deterministic routing, the application or the architectural parameters of the system can be used to determine the number of packets traversing through a particular NoC switch. Conversely, if the routing is based on a random walk, the number of packets through any particular switch is essentially random, losing predictable correlation with underlying system parameters. However, complete randomness in the route selection can negatively impact the overall system performance as such random routing mechanisms are shown to increase the NoC packet latency. Hence, as a trade-off between these opposing goals, in this work, we propose a distributed random routing mechanism for NoC architectures based on SA heuristics. Such SA heuristics uses an iterative approach to approximate the solutions of an optimization problem. During the initial iterations, the probability of accepting a random non-optimal solution is higher. This acceptance probability is a function of a temperature parameter which decreases with each iterations of the algorithm. Thus, the acceptance probability also decreases with the iteration, decreasing the chances of accepting random non-optimal solutions. Similar to this approach, in our proposed random routing algorithm, initially the probability of selecting a random output port for each new injected packet in the NoC fabric is high. However, over the lifetime of the packet in the NoC this probability decreases. Therefore, with progression of time, the routing decisions are constrained to be optimal (shortest-path) with only occasional random sub-optimal decisions. The difference between the shortest path (in green) and a SA-based random path (in orange) is shown in Figure 5.14 for a particular Source, (S)-Destination, (D) pair. Following the Metropolis criterion

in SA, the probability of choosing a random output port for the next hop is defined as

$$R_i = e^{\alpha(T_{inj} - T_i)} \tag{5.1}$$

Where T_{inj} is the packet injection time and T_i is the current time in clock cycles. This time difference, ΔT is embedded in the packet header and incremented in each clock cycle. α is a designer parameter that controls the degree of randomness in the routing and will be described in later sections. Hence, the proposed random routing mechanism also provides the designers with a choice for controlling the degree of the randomness.

To implement this random routing in the NoC switches, the NoC switches are equipped with a Linear Feedback Shift Register, (D-LFSR) to generate the random numbers denoting the probability of taking the shortest path. If this number is less than the probability stored in the LUT, (R-LUT) based on equation 5.2 and the current ΔT value, then the proposed routing chooses any of the ports of the switch that leads to non-optimal path with equal probability. Another LFSR, (R-LFSR) is used to generate this non-optimal random port, (R-Port). When the generated number is greater than the probability stored in an LUT, the optimal port is chosen by the switch depending on the adopted shortest path algorithm. The switch architecture implementing this random routing mechanism is shown in Figure 5.15.

According to equation 5.2, the probability of taking non-optimal, random paths decreases with the current time, T_i. Therefore, with elapsed time, each packet is more likely to take the optimal routing decision determined by the adopted optimal routing algorithm in the system. This guarantees that each packet will reach its destination as the time progresses. Hence, the proposed SA-based random routing is guaranteed to be live-lock free. In this paper, the shortest paths for each source and destination pair is calculated using Dijkstra's algorithm as such shortest path algorithm is applicable to multiple NoC topologies such as trees, meshes and random topologies. Each switch is equipped with a forwarding table containing the port number to reach the next switch in the shortest path for each

destination. This forwarding table will be pre-populated at design time and hence reduces the routing delays that incurs due to the route computation for every packet at a switch.

To ensure application performance, latency sensitive packets should experience less randomness in routing compared to other messages. As mentioned earlier, the parameter α in equation 5.2 can be used to control the degree of randomness in routing and a higher α value is preferred for latency critical messages as it rapidly diminishes the probability of taking non-minimal paths. Due to this, the latency critical messages then follow the shortest path determined by adopted shortest path algorithm in the system. Alternatively, for latency-tolerant messages, a smaller α value can be used to have more randomness in their routing and ensure better threat resiliency. We consider α to be a tunable parameter to be determined by considering the security-performance trade-off, applications, and run environment.

5.1.3.2 Deadlock Avoidance and In-order Packet Delivery

In the proposed routing mechanism, initially, each newly injected packet will take a random path. However, as the time elapses, the packets will eventually follows the shortest path (as determined by the Dijkstra's algorithm). As Dijkstra's routing is inherently cyclic dependency free, the proposed routing will be deadlock free when the packets follows the shortest path. Hence, to ensure complete deadlock freedom for the proposed routing, we should avoid deadlocks when random paths are chosen for a packet.

To avoid deadlock, we consider the Virtual Channel (VC) occupancy of the output ports before routing an incoming packet. If the VC occupancy of an output port is less than a pre-determined threshold, the packet is not routed immediately and waits in the input VCs to be rerouted. This ensures that a heavily utilized port causing a deadlock is avoided. Algorithm 5 describes this mechanism by using a pseudo-code for the proposed routing algorithm. Additionally, due to the exponential decaying probability of taking random paths, a packet is more likely to be routed toward the deadlock free port following the Dijkstra path as elapsed time increases.

As each packet follows a different path, such random routing also need to ensure in-order packet reception for each message in the system. We adopt the in-order packet delivery mechanism described in [207], where a lookup table entry is compared with the packet identifier of a message at the re-convergent switches. If the identifiers match, the packet is granted arbitration and the look-up table identifier value is incremented and thus in-order packet reception is ensured without significant re-order buffer overheads.

Algorithm 5 Pseudo code for the proposed random routing

Input: α, T_{inj}, freeVCThreshold
Variable: $\Delta\ T$, xyProb, randomProb, T_i, randRoute
Function: checkVCStatus(outPort)
Output: routingDecision

```
 1: foreach packet in switch do:
 2:     ΔT = T_inj − T_i
 3:     xyProb = rand(0,1)
 4:     randomProb = e^(αΔT)
 5:     if(xyProb < randomProb) then
 6:         routingDecision = random
 7:         if(checkVCStatus(outPort) ≥ freeVCThreshold) then
 8:             randRoute = True
 9:         else
10:             randRoute = False
11:             go to line 2
12:         endif
13:     else
14:         routingDecision = xy
15:     endif
16: endfor
```

5.1.4 Evaluation

In this section, we evaluate the average packet latency for the proposed SA-based random routing mechanism with different α values as discussed in section 5.2.4.1. We also evaluate the proposed attacker models by measuring its accuracy to infer the application suites and the effectiveness of the proposed random routing against such attack models; in this work we have sometimes used *performance* and *accuracy* interchangeably in regards to model evaluation.

Table 5.2: Component configuration for simulation

Component	**Configuration**
System size	64 cores, Out-of-Order, 16cores/chip
Cache	32KB (private L1), 512KB (shared L2), MOESI
NoC switch	3 stage pipelined 5 ports,0.078pJ/bit
Total VC	4, each 8 flits deep, 64 bits/flit
Wired NoC links	64-bit flits, single cycle latency, 0.2pJ/bit/mm
Technology	65nm, 1V supply, 1GHz clock

5.1.4.1 Experimental Setup

We consider a 64 core system arranged in a 8x8 mesh NoC fabric with 4 in-package memory modules. The core configurations shown in the Table 5.8 are used to extract the core-to-memory and cache coherency traffic for PARSEC and SPLASH2 benchmark suites executed until completion using SynFull [198]. To map these traffic patterns to the 64 core NoC environment, we have considered multiple threads of the same application kernel running on the system where each processing core executes a single thread and the memory stacks are shared among threads. We also vary the NoC configurations such as number of VCs, topology, cache distribution/access using Uniform Memory Access, (UMA) and Non-Uniform Memory Access, (NUMA) to analyze corresponding ML results. Table 5.9 lists the NoC architectures considered in this paper. We use ASIC design flows with Synopsys Design Compiler with 65nm CMP standard cell libraries (https://mycmp.fr/) to synthesize the NoC switches. The delay and energy dissipation on the NoC links are obtained through Cadence simulations considering the specific lengths of each link based on the NoC topology assuming 20mm×20mm chips. The power and delay overheads of the NoC switches and NoC links are considered during simulation. In order to train the ANN classifier, a cycle accurate simulator based on NoXim [208] is modeled to implement the proposed SA-based random routing and track the number of packets passing through each switch.

Table 5.3: Architectures for evaluation

Architecture name	**Topology**	**VCs**	**Cache access**
Mesh	Mesh	4	UMA
V2Mesh	Mesh	2	UMA
DMesh	Mesh	4	NUMA
Torus	Torus	4	UMA
FTorus	Folded Torus	4	UMA

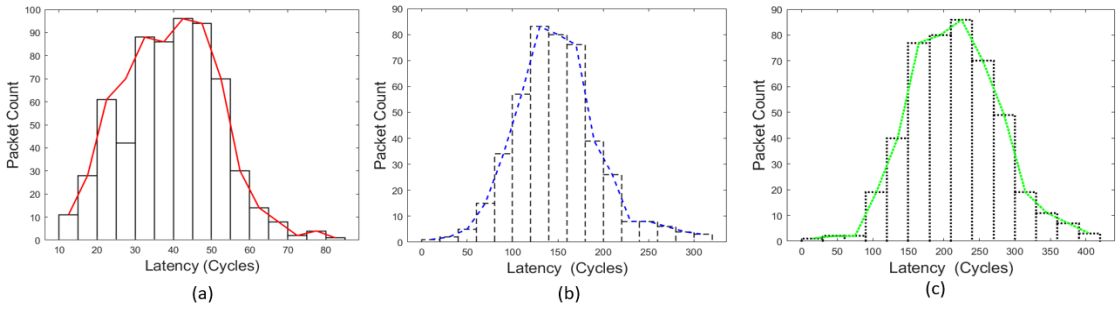


Figure 5.5: Latency variation for (a) $\alpha = 100$ (Deterministic) (b) $\alpha = 0.01$ and (c) $\alpha = 0.005$.

5.1.4.2 Average Packet Latency with α Variation

We consider the degree of randomness, α, to be a designer's parameter; in this section, we study the average packet latency (μ) variation for different values of α. We consider α to be 100, 0.01 and 0.005 and study the corresponding effect on a 8x8 mesh NoC running Fast Fourier Transform, (FFT) traffic traces obtained from SynFull [198]. Following Equation (1), higher α values result in a lower probability of taking more random paths which diminishes rapidly with elapsed time. Therefore with a large value of α, say 100, the average packet latency is same as the average packet latency achieved using deterministic routing as it represents no randomness in the routing. On the contrary, smaller values of α increases the probability of taking more random paths and hence increases average packet latency. Figure 5.5 presents the probability distribution of packet latency with change in α. The average packet latency for $\alpha = 100$, $\alpha = 0.01$, and $\alpha = 0.005$ were found to be 37.13, 154.75, 226.15 cycles, respectively, for FFT traffic. This latency penalty could be considered as the cost of improved security provided by the proposed random routing. Keeping this security-performance trade off in mind, designers can choose α values that can provide improved security as well as meet system performance and security requirements making such a NoC routing secure-by-design. It is also interesting to note that with smaller α, the standard deviation (σ) in latency also increases. The 99^{th} percentile of the latency distribution is at 76.77, 320.1 and 433.18 cycles for deterministic routing, $\alpha = 0.01$ and α= 0.005 values respectively providing designers upper bounds of latency for their design guidelines.

5.1.4.3 Evaluation of ML Models for an Efficient Attacker Design

Multiple ML classifiers such as ANN, support vector machine (SVM), K-nearest neighbor (KNN), and DT (decision trees) were investigated to evaluate their accuracy for application detection to determine an efficient computing model for the proposed attacker. A linear kernel based SVM has been utilized for application detection. Similarly, experiment was also done with k-nearest neighbors with k=12 and 78 in this work. As k=12 and k=78 provide similar performance, we enlisted only the performance for k=12 in both Table

5.4 and Table 5.5 as it provides less delay and computation overhead. We considered 12 application-specific traffic obtained through SynFull [198] from PARSEC and SPLASH2 benchmark suites with maximum of 2 applications running simultaneously. For each of the ML model, a 16 feature input was considered signifying a maximum of 16 HTs to be inserted in the system. From the performance metrics listed in Table 5.4, it can be observed that the ANN, KNN, and DT classifiers have similar performance in detecting application suites running within a NoC using deterministic routing. Therefore, it can be argued that any of these classifiers can be used to design an efficient attacker to launch proposed traffic analysis attack.

However, apart from the 12 base classes (for 12 applications), considering a maximum of 2 simultaneous running applications, the KNN, SVM, and DT classifiers required additional ${}^{12}C_2 = 66$ classes to be created during training to enable detection of 2 simultaneously running applications. Therefore, the complexity of training and computation overhead of those ML classifiers increase as the attacker intends to detect larger number of simultaneously running applications. For example, for a similar SoC, to detect a maximum of 4 simultaneously running applications, the attacker needs to create ${}^{12}C_4 = 495$ additional classes to train those classifiers. The number of required classes also increases as the number of baseline applications which is 12 in our case, increases. Due to the increased training complexity and the huge time required by the classifiers like KNN and DT, an attacker with such ML classifiers does not represent an efficient attacker as it is not scalable in terms of detecting multiple applications in a typical ever increasing multi-user environment. On the other hand, due to the softmax activation at the output layer and one-hot encoding, ANN can train on and detect multiple applications, along with their probability distribution, without any additional complexity and hence best suites the attacker purpose for the proposed traffic analysis attack.

Moreover, from Table 5.5 it can also be observed that for a NoC employing SA-based routing, the ANN classifier provides higher application detection accuracy compared to other ML classifiers. As from attacker's perspective it is desired that the ML classifier

Table 5.4: Performance of the ML classifiers for deterministic routing

ML classifier	Accuracy (%)	Precision	Recall	F-score
ANN	98.18	0.98	0.98	0.98
SVM	74.3	0.72	0.72	0.74
KNN	97.8	0.97	0.97	0.97
DT	98	0.98	0.98	0.98

Table 5.5: Performance of the ML classifiers for SA-based routing

ML classifier	Accuracy (%)	Precision	Recall	F-score
ANN	17.42	0.10	0.17	0.11
SVM	6.41	0.01	0.01	0.06
KNN	6.41	0.03	0.02	0.06
DT	1.28	0.003	0.001	0.01

should provide a better accuracy irrespective of the routing algorithm used in a NoC; from Table 5.5 it can be concluded that an attacker equipped with ANN model can provide better application detection accuracy compared to other ML classifiers even in presence of novel routing algorithms like SA-based routing proposed in this work. We consider this ANN based attacker model without any data augmentation capability to be the baseline attacker model.

In summary, due to low complexity in detecting multiple simultaneous applications in a large multi-user environment and better accuracy in presence of novel routing algorithms, the ANN classifier best serves the purpose of an attacker targeting such traffic analysis attack on a NoC-enabled SoC. Although it can also be argued that various other experiments can be done and the existing classifiers can be optimized to come up with an ultimate ML classifier that best suites the purpose of the attacker, however, it is necessary to understand here that the objective of this work is to present the vision of a novel HT-enabled traffic analysis attack and a probable solution, not optimization of attacker efficiency; it will be explored in the future work.

5.1.4.4 ML Performance with Deterministic Routing

We evaluate the effectiveness of our baseline attack model for the architectures considered in Table 5.9 using Dijkstra's shortest path routing only. We use this routing as it can be generalized to all topologies and in the case of mesh topologies it is identical to the dimension-order routing yielding same latency performance. The inserted HT leaks the packet count as payload to the ML-based attacker in a periodic interval of 5000 cycles. Therefore, as attack efficiency metric we consider the accuracy, F1-score, recall and precision of the ANN model placed on the attacker side. We also alter the number of observed features (number of inserted HTs) in the system and measure the change in the performance metrics. As shown in Fig. 5.6, for a 64 core system, the attack efficiency increases with the increase in feature size. Even with a feature size of 4 (4 HTs embedded) the attack accuracy varies between 85-98% for all the architectures considered in Table 5.9. Similarly, Figure

5.7 presents the ANN attack model performance for mix of three applications running simultaneously. We observe similar trends for the other metrics such as F1, recall, and precision with different feature sizes and are shown in Table 5.6 for mix of two applications. Thus, it can be concluded that the proposed attacker can efficiently interpret the user profile with less number of HTs inserted, thus resulting in small footprint/overhead within the chip.

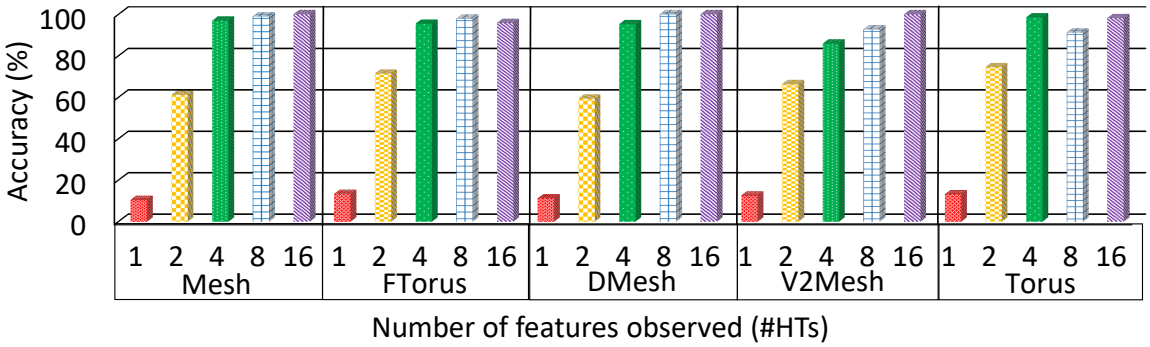


Figure 5.6: Accuracy for deterministic routing with different features (number of HTs observed).

5.1.4.5 ML Performance with Proposed SA-based Random Routing

We evaluate the performance of the baseline attack model with the proposed SA-based random routing. As the degree of randomness for the proposed random routing can vary depending on α, we consider $\alpha = 100$ (deterministic), $\alpha = 0.01$, and $\alpha = 0.005$ and analyze the accuracy of the attacker with different feature sizes. In this work, Alpha (α) is interchangeability denoted as 'R' as well. Figure 5.18 shows the accuracy of the attacker on folded torus architecture for deterministic and random routing. It can be observed from Figure 5.18 that due to the higher routing obfuscation introduced by decreasing α values in the proposed SA-based random routing, the accuracy of the attacker reduces significantly

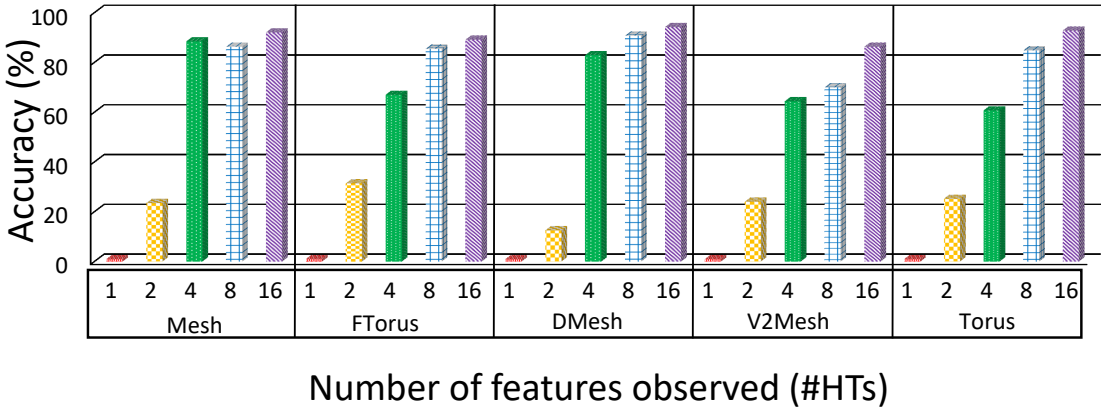


Figure 5.7: Attack accuracy for deterministic routing with 3 applications

Table 5.6: Attacker model performance results with deterministic routing

Deterministic routing with two applications											
Architecture	Features	Accuracy	F1	Recall	Precision	Architecture	Features	Accuracy	F1	Recall	Precision
Mesh-4VC	16	98.18	98.18	98.19	98.18	Mesh-2VC	16	95.83	95.84	95.83	96.03
	8	91.17	91.23	91.17	91.16		8	97.88	97.86	97.87	98.08
	4	98.44	98.43	98.43	98.50		4	95.45	95.48	95.44	95.64
	2	74.52	74.46	74.54	75.37		2	71.38	71.24	71.39	72.18
	1	13.13	13.13	13.12	13.13		1	13.27	3.76	13.17	2.19
FoldedTorus-4VC	16	99.99	99.99	99.89	99.97	Torus-4VC	16	99.98	99.95	99.96	99.95
	8	92.65	92.62	92.64	92.83		8	98.96	98.96	98.96	98.98
	4	85.86	86.28	85.86	85.86		4	96.86	96.85	96.85	96.86
	2	66.23	65.50	66.24	66.19		2	61.14	60.26	61.14	62.63
	1	12.50	3.6	12.5	2.14		1	10.42	2.92	10.48	1.7
Dcache-Mesh-4VC	16	99.99	99.99	99.99	99.99	Mesh-6VC	16	97.05	97.04	97.05	97.16
	8	99.98	99.98	99.98	99.98		8	93.86	93.88	93.86	94.21
	4	95.19	95.19	95.44	95.44		4	91.79	91.75	91.80	91.89
	2	59.29	57.59	59.30	58.42		2	76.07	75.98	76.12	76.68
	1	11.14	3.19	11.11	1.86		1	15.26	4.3	15.26	2.55

(<15%) compared to deterministic routing for smaller α values with higher feature sizes. Also, from Figure 5.18, it can be discerned that for smaller α values increasing feature size does not increase the accuracy significantly and thus represents the robustness of the proposed random routing. If we combine Figure 5.18 and Figure 5.5, it is interesting to note that the attacker accuracy is similar for both $\alpha = 0.01$ and $\alpha = 0.005$, however, using

Table 5.7: Attacker model performance results with proposed random routing

Random routing with two applications													
Architecture	Features	Alpha	Accuracy	F1	Recall	Precision	Architecture	Features	Alpha	Accuracy	F1	Recall	Precision
Mesh-4VC	16	0.01	14.39	10.94	14.39	9.1	Mesh-2VC	16	0.01	10.60	7.93	10.60	7.47
		0.005	17.42	11.96	17.42	10.36			0.005	17.42	10.31	17.42	8.52
	8	0.01	13.63	5.34	13.63	6.57		8	0.01	18.18	12.68	18.18	11.38
		0.005	13.63	6.08	13.63	11.25			0.005	13.63	7.10	13.63	5.37
	4	0.01	17.42	10.68	17.42	14.74		4	0.01	6.06	6.94	6.06	13.73
		0.005	13.63	0.07	13.63	20.05			0.005	9.09	5.72	9.09	4.50
	2	0.01	14.39	9.12	14.39	10.9		2	0.01	12.87	12.21	12.87	11.78
		0.005	15.90	8.77	15.90	7.14			0.005	9.84	8.98	9.84	9.72
	1	0.01	13.63	3.89	13.63	2.27		1	0.01	12.63	3.89	13.63	2.27
		0.005	13.63	3.89	13.63	2.27			0.005	13.64	3.89	13.63	2.27
FoldedTorus-4VC	16	0.01	10.6	3.74	10.60	2.42	Torus-4VC	16	0.01	11.36	6.53	11.36	7.33
		0.005	9.0	3.49	9.0	2.32			0.005	10.60	5.75	10.60	7.88
	8	0.01	12.12	6.77	12.12	6.06		8	0.01	12.87	4.83	12.87	3.32
		0.005	12.12	6.61	12.12	5.69			0.005	12.87	4.83	12.87	2.21
	4	0.01	15.90	6.32	15.9	4.05		4	0.01	9.84	3.68	9.84	2.31
		0.005	14.39	5.86	14.39	4.04			0.005	12.12	4.7	12.12	3.05
	2	0.01	11.36	5.35	11.36	5.0		2	0.01	12.87	8.75	12.87	8.63
		0.005	10.6	4.62	10.6	3.7			0.005	13.63	10.39	13.63	9.95
	1	0.01	12.87	3.6	12.8	2.14		1	0.01	10.60	3.03	10.60	1.76
		0.005	12.87	3.6	12.87	2.14			0.005	10.60	3.03	10.60	1.76
Dcache-Mesh-4VC	16	0.01	20.46	16.68	20.46	16.92	Mesh-6VC	16	0.01	15.15	7.89	15.15	5.72
		0.005	18.93	11.47	18.93	8.37			0.005	12.87	6.47	12.87	4.83
	8	0.01	13.63	8.06	13.63	12.92		8	0.01	15.15	6.46	15.15	4.73
		0.005	13.63	8.8	13.63	7.03			0.005	13.63	4.68	13.63	3.13
	4	0.01	12.12	6.6	12.12	6.16		4	0.01	9.09	3.67	9.09	2.49
		0.005	11.6	7.48	12.8	5.8			0.005	9.84	4.4	9.8	3.07
	2	0.01	12.87	3.6	12.87	2.14		2	0.01	13.63	6.0	13.63	4.58
		0.005	12.87	3.67	12.87	2.14			0.005	17.42	11.28	17.42	19.08
	1	0.01	11.36	3.24	11.36	1.8		1	0.01	15.90	4.54	15.90	2.65
		0.005	11.36	3.2	11.36	1.89			0.005	15.90	4.54	15.90	2.65

$\alpha = 0.005$ increases average system latency by 46.13% compared to $\alpha = 0.01$. Similarly, Figure 5.8b presents the results for the proposed SA-based routing for three applications running simultaneously; detailed results for other performance metrics are shown in Table 5.7. Therefore, it can be concluded that, reducing α further to zero will realize a completely random routing, but the increase in obfuscation will be marginal while resulting in significant increase in average packet latency, which is not desirable.

5.1.4.6 ML Performance with Pseudo-adaptive Routing

So far, we have considered that the baseline SoC uses a deterministic routing and the attacker trains its ANN accordingly to detect the applications running in such systems. However, the use of pseudo-adaptive routing algorithms to avoid congestion while ensuring deadlock freedom is very common in modern day SoCs. Similarly, the attacker could be intelligent enough to train its ANN with such pseudo-adaptive routing to intercept the traffic running in the system. As representative of such pseudo-adaptive routing algorithms, in

this section, we choose west-first routing and evaluate the attacker's efficiency in predicting various application specific traffic as shown in Figure 5.9. We vary the number of features and evaluate the attacker's accuracy in predicting application suites using west-first routing (WFR) and SA-based routing with two different α values. The limited randomness offered by the WFR or any other turn model routing depends on the relative positions of the source and destination nodes. Depending on the location of the destination nodes the routing can be still very deterministic and therefore, from Figure 5.9, it can be observed that the even for SoCs using WFR, the underlying applications can be detected with more than 95% accuracy. On the other hand, the proposed SA-based routing being a controlled random routing scheme reduces the attacker detection accuracy below 15% for both α values even when the attacker is trained with pseudo-adaptive WFR.

5.1.4.7 Proposed Defense Performance with Advanced Attacker Model

The data augmentation technique generates synthetic data samples thereby compensating for the missing data. Also, as mentioned earlier, GANs have been used to generate synthetic images given a real image dataset. In this work, the data available to the attacker is the packet count data leaked by the hardware Trojan. In such a scenario, evaluating the authenticity of the generated synthetic samples is crucial to the correct implementation of the advanced attacker model. As such, we base the credibility of the GAN-based attacker model by analyzing the loss plots of the GAN. Figure 5.10 shows the loss plots of the discriminator, working on real and fake data, and the generator. It can be seen that the D-real and D-fake plots are close to each other, which is an expected behavior; while the generator loss plot spikes initially and later settles close to the discriminator plots. In summary, the loss between the generator and discriminator is close which proves the credibility of the generated synthetic data samples. In a real-world situation, an attacker would also need to analyze the loss plots while training the GAN to ensure the GAN was able to learn the data distribution and reliably generate synthetic data. The generated

synthetic data is then utilized to augment the packet count dataset which is later fed to the attacker neural network model for inference.

We discussed earlier in Section 5.2.2 that *It may so happen that the attacker does not have access to uniformly long observation window of system under attack in all operating conditions.* The performance degradation owing to lack of observed samples is shown in Figure 5.11. The left bars (ANN) show the ANN attacker model performance for different scenarios. In each scenario, the attacker is unable to observe a mentioned amount of data, meaning, missing instances of application classes; missing data refers to the lack of training data available to the attacker. Hence, the accuracy of the ANN attacker reduces owing to the missing data compared to the baseline accuracy as shown. In Figure 5.11, the performance of the ANN model corresponding to missing instances (data) of one application (ANN-1D) is 89.2%, while for missing two application data (ANN-2D) is 82.6%, and finally for missing four application data (ANN-4D) it is 73.2%; all of the plots show a decreasing trend as more and more data becomes unobservant to the attacker. From Figure 5.11, it is seen that the accuracy of the model is dramatically enhanced; attacker model performance with data augmentation for one missing application instances (GAN-1D), two missing application instances (GAN-2D), and four missing application instances (GAN-4D) are restored to 96%, 95.5% and 93.2%, respectively. This proves that with data augmentation the attacker can render the attack model more robust against missing data.

Figure 5.12 presents the accuracy of the advanced attacker model with and without data augmentation for a varying number of feature sizes. The feature size is the number of HTs considered for the attack. Referring to Figure 5.6 we see that the accuracy for varying feature sizes decreased owing to the missing data not observed by the attacker. On the contrary, in Figure 5.12, with data augmentation, we observe that the accuracy increases, relative to accuracy without data augmentation, with an increase in the number of HTs observed. This is due to the fact that data augmentation compensates for missing data samples by generating synthetic samples. More the available samples for training the ML model, better its capability in detecting the applications running on the system.

We compare the performance of the attacker model with and without the presence of the GAN-based data augmentation technique for our proposed SA-based random routing in Figure 5.13; Presence of GANs is denoted by a suffixed 'G'. In this scenario, the attacker is unable to observe the entire execution window and hence the performance without GANs for deterministic data (original leaked HT data) suffers degradation. Figure shows the comparison between deterministic data and random routing data samples with two α factors of 0.01 and 0.005 for five different architectures. With data augmentation, the accuracy of the attacker model is improved significantly. For instance, for a Mesh architecture, the accuracy is 90% for deterministic data without augmentation, which increased to 98% with augmentation; we see a similar pattern for accuracy comparison between the random routing with and without augmentation for two α factors. It is to be noted that the number of HTs observed for the figure is 16; with less number of features the accuracy for deterministic data follows a similar pattern as shown in Figure 5.12, and the accuracy of the model for random routing data degrades even further.

In summary, for our *baseline attacker model*, Figures 5.6 and 5.7 demonstrated the success percentage in terms of accuracy in revealing the applications executing in the system. Later, Figures 5.18 and 5.8b presented that the attacker accuracy degrades with our proposed SA-based random routing methodology, thus thwarting the attack. Figure 5.9 presents results with a different routing strategy (west-first) considered to evaluate ANN performance on the proposed random routing when attacker model is trained using west-first routing. Further, we consider the *advanced attacker model* that is equipped with data augmentation using GANs for scenarios where limited data is available for launching an attack. Figure 5.10 shows the credibility of the GANs model in augmenting original leaked HT data by generating synthetic samples. Figures 5.11, 5.12, and 5.13 demonstrated the efficacy of using data augmentation; Figure 5.12 presented the accuracy of the attacker model for different architectures with varying number of HTs observed for both cases - with and without data augmentation; and, Figure 5.13 shows our obfuscation with SA-based random routing holds true for the advanced attacker model with data augmentation technique.

5.1.4.8 Routing and HT Overheads

The increased security offered by the proposed SA-based random routing comes with additional overheads in routing logic as shown in the inset of Fig 5.15. We consider the D-LFSR to be 8 bit wide whereas the R-LUT had 300 entries with each having 8 bits for each of the α values. Considering a system having three α values, the routing logic consumes additional $7685.07um^2$ of area, $1.82uW$ of power and $0.93ns$ of delay in 65nm technology node. Moreover, each HT takes $1551.6um^2$ area, $125.88nW$ power and $0.2ns$ delay for its hardware realization in same technology node.

5.2 Parameterizable α-Based Defense Against Hardware Trojan Data Leakage Attack on Network-on-Chips

Many chip vendors are becoming fabless to alleviate costs. Further, to minimize the time-to-market and design costs, modern System-on-chips (SoCs) use Third Party Intellectual Properties (3PIPs), which maybe procured from untrusted organizations. An adversary either at the foundry or at the 3PIP design house can introduce a malicious circuitry to jeopardize an SoC, which is known as a Hardware Trojan (HT) [192]. HTs can be used for various malicious purposes, including information leakage, functionality subversion, and battery exhaustion [192, 194, 195].

Interconnection networks in multi/many-core processors such as a Network-on-Chip (NoC) [191] that connects processing units to memory and peripherals play a pivotal role in these systems as efficiency of data movement is often the main bottleneck. Due to this important role played by the NoC, they form one of the largest surface areas for attack in the system both physically and functionally. Considering the critical role played by the NoCs, embedding an HT that exploits the interconnection backbone can reveal the communication patterns in the system. This information when leaked to a malicious external attacker can reveal important information regarding the application suites running on the system, thereby compromising the user profile. This information in turn, can enable further more severe attacks not just on the multi/many-core processor infected with the HT, but on the systems on which they are deployed. For instance, an adversary obtaining secure military information through a HT deployed in a router can subvert the military backbone, thus leading to a compromise of the national security [28].

In this paper, we define such an HT that is instrumental in carrying out an attack in conjunction with an external agent as a Remote Access Hardware Trojan (RAHT). We first discuss one possible incarnation of a lightweight NoC-based RAHT, which, in its simplistic form, is a simple counter, which, when inserted in one or a few switches of the NoC can count the number of packets traversing the specific switches over a time window. The HT

can then periodically packetize this count and send it to an external attacker for payload analysis, severely compromising user profile confidentiality.

This packetized count, which is the HT payload, can be analyzed by the external attacker using machine learning (ML) techniques to infer the applications running on the system or reverse engineering architectural IP of the system. To analyze the retrieved information, the attacker trains an ML algorithm, that can map packet traversal frequencies at specific switches to the application suites or reveal confidential information about the architecture. Training samples for such a data analysis tool can be built using a system emulator or simulator such as [198] and [208].

Using a RAHT and an external data-analysis attacker, we demonstrate that the application suites running in the system can be detected with high accuracy. This is possible because specific routing protocols are proposed for these particular system configurations, which, when adopted, result in application-specific traffic patterns. We also show that it is possible to use the same technique to infer architectural information of the system such as the NoC topology to aid in architectural reverse engineering of the system enabling industrial espionage. In the end, we discuss our vision of an adaptive controlled routing mechanism that can be tuned based on external threat perception to implement a defensive routing measure that achieves a user-defined trade-off between security and performance of the NoC.

5.2.1 Threat Model

While many different types of HTs are possible to envision each with a different payload and trigger mechanism, in this paper, we focus on a RAHT that has extremely low footprint and can actually cause an attack by granting remote access to an external attacker. In our threat model, we envision a multi-user or multi-tenant server or data center where the processing engines are multi-core processors connected with a NoC. Since the NoC may be procured from an organization different from the system integration designer, one or more HTs can be inserted in the routers of the NoC during the design and fabrication process. These HTs

can be simple in functionality such as counting the number of packets traversing the switch over an observation window and sending that information to an external attacker. In this way the actual RAHT can maintain a negligible area, power and timing footprint in order to thwart detection using traditional HT detection methods. The attacker can then employ large-scale compute capabilities and algorithms to perform a traffic analysis attack on the packet count from one or more routers. This kind of attack can reveal the applications running on the system, since the traffic interaction in multi/many-core processors is always application dependent, and thus, compromise user privacy by revealing the applications that is being executed.

Additionally, it can reveal information regarding the micro-architecture of the processor platform, such as cache organization, NoC topologies or router buffer sizes. As these micro-architectural decisions determine various performance metrics of the platforms, these are proprietary and constitute IP of the designers. Therefore, if such information is revealed, it can lead to reverse-engineering of the processing platform. Figure 5.14 shows the attack model.

5.2.2 RAHT Based Attacker Design

In this section, we describe various aspects of the particular attack model that we have studied. The RAHT in particular is a counter that is capable of counting the number of packets getting routed by the switch where it is inserted. We assume that the RAHT is inserted inside the routing block of the NoC or interconnection switch as shown in Fig. 5.15. There maybe one or more such infected switches in the system. The functionality of this counter is that it will count up whenever a new packet accesses the router of the switch to get routed to its next destination. After counting for a pre-determined duration, it packetizes this count if it is non-zero, by appending a destination address and other header information and inserts this packet into the NoC as the HT payload. Such processing systems are typically multi-user platforms where different parts of the processor are virtualized and allocated to various applications from myriad of users. As the attacker can be disguised as one of the multiple users of this shared virtualized multi/many-core processor hosting

multiple users simultaneously, the destination of the payload will be a legitimate I/O port where the malicious program is hosted. Therefore, the packet is unlikely to be flagged by any other security measure in the system, as it does nothing anomalous compared to other packets in the system. By sending only non-zero counts the RAHT also avoids detection in quiescent operation or sleep mode. The payload then gets analyzed by the external attacker. We consider a 16-bit counter as the RAHT for our evaluations here.

We envision that the proposed HT does not lie in the data-path of the packet transfer mechanism over the NoC. It recognizes the flit-type and in case of a header, it increments a counter. This happens in parallel with the function of the routing block. Due to no impact on delay, the HT is difficult to detect based on timing analysis methods alone. Moreover, due to the nature of the payload as described below, the area and power footprint of the HT is negligible compared to the NoC or the entire processor. Therefore, the HTs can remain in always-on state without the need for a sophisticated trigger.

5.2.3 Off-Chip RAHT Payload Analysis

We craft a RAHT payload analyzer using ML techniques to utilize the information leaked by the RAHT. We chose artificial neural network (ANN) in this work due to its ability to map complex patterns efficiently as well as its resilience to variations. We initially built a dataset consisting of eighty different features (payloads) obtained from simulating a system with 64 cores and 16 memory controllers, with permutations of twelve different applications with no more than three running on the system simultaneously signifying up to three independent users hosted on the system executing three applications running simultaneously. We create this data in the same manner as an actual attacker who may be able to create using an emulator or a simulator of the real system.

We consider a 64 core system arranged in a regular 8x8 mesh NoC fabric with 4 in-package memory modules. The core configurations in Table 5.8 have been used to extract the core-to-memory and cache coherency traffic for PARSEC and SPLASH2 benchmark suites when they were executed until completion using SynFull [198]. In order to map these

traffic patterns to the 64 core NoC environment, we consider multiple threads of the same application kernel running on the system where each processing core executes a single thread and the memory stacks are shared among threads. We also vary the NoC configurations such as number of VCs, topology, cache distribution/access using Uniform Memory Access, (UMA) and Non-Uniform Memory Access, (NUMA) to analyze corresponding ML results. Table 5.9 lists the NoC architectures considered in this paper for inference on NoC topology IP reverse-engineering.

We deploy a 5 layer ANN (N-800-500-200-64-12 neurons for application detection and N-800-500-300-100-50-5 for topology detection; N represents number of features at input) with ReLU as activation functions for hidden layers and softmax as the activation function for the output layer. The architecture is proposed for the ANN that supports application detection for both two and three concurrent applications. There are twelve final outputs for the ANN, which are the number of individual applications used to build the dataset and five different architectures are experimented. To represent simultaneous execution of multiple applications, one-hot encoding is utilized. A 5-fold cross-validation is utilized to analyze performance, determined based on grid-search. As the ANN is deployed off-chip, the area and other overheads are not of concern, thus we evaluate only the detection accuracy.

As shown in Fig. 5.16a and 5.16b, the attack efficiency increases with the increase in feature size. Even with a feature size of 4 i.e., with 4 RAHTs, the classification accuracy varies between 80-98% and 60-85% for the application detection, simultaneously 2 and 3 applications running in the 64-core processor connected with a Mesh NoC. Thus, it can be concluded that the proposed attacker can efficiently interpret the user profile with very small footprint/overhead within the chip. The result of NoC topology inference is presented in conjunction with the protective routing in Section VIC.

5.2.4 Protection from the RAHT

In this section we discuss a possible method to protect against the RAHT. We investigate a controlled random routing which can confuse the external payload analyzer and also provides

a design knob that can be adjusted to react to perceived threat levels in environment. This will enable achieving a dynamically varying trade-off between security and performance in the system. The external threat perception will be determined by a data-mining engine operating either at a separate system in the user environment or by the design house of the NoC-based processor.

5.2.4.1 Protective Routing Methodology

The defense mechanism against the HT-based traffic analysis attack, is based on obfuscating the traffic analysis mechanism by introducing controlled randomness in the routing. The underlying principle is that with deterministic routing, the number of packets through a particular router is highly correlated to the application or the architecture of the system while, with random routing decisions, that correlation will be reduced. A Simulated Annealing (SA) based random routing methodology enable designers to have a degree of controllability over the randomness of the routing decisions. According to SA heuristic, at the beginning of the annealing schedule, the probability of taking random non-optimal decisions are higher. However, the probability of accepting random non-optimal solution decreases as temperature is reduced according to the annealing schedule. Similar to the Metropolis criterion in SA, the probability of taking a random output port for the next hop is defined as

$$R_i = e^{\alpha(T_{inj} - T_i)} \tag{5.2}$$

In our case, T_{inj} is the packet injection time and T_i is the current time in clock cycles. This time difference, ΔT is embedded in the packet header and incremented in each clock cycle. α is the control parameter that controls the degree of randomness in the routing and will be described in later sections. As shown in Fig. 5.15, any switch in the routing path of the packet generates a random number using a Linear Feedback Shift Register, (D-LFSR) which represents the probability of taking the shortest path. If this number is less than the probability stored in a Lookup Table, (R-LUT) based on equation 5.2 and current ΔT value, then the proposed routing chooses any of the ports of the switch that leads to non-optimal path with equal probability. The non-optimal random port, (R-Port)

is generated by another LFSR, (R-LFSR). Otherwise, the optimal port is chosen by the router depending on the adopted shortest path algorithm.

5.2.4.2 Data-mining to Inform the Routing Mechanism

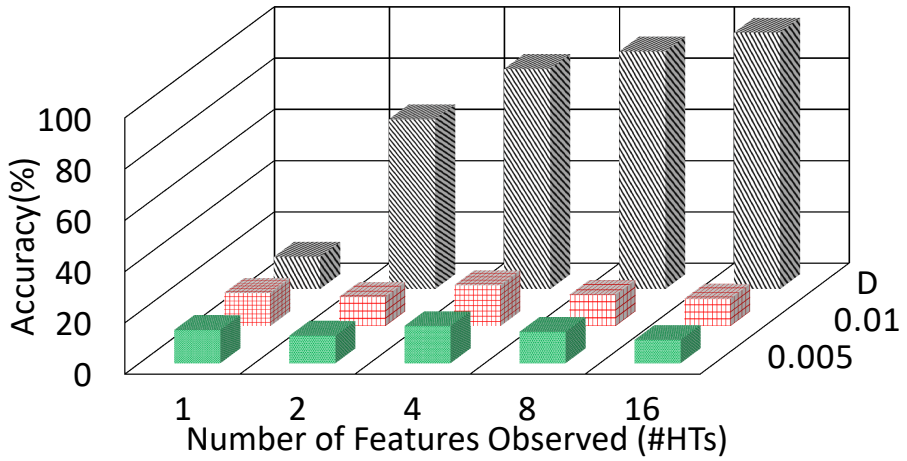
We envision the routing mechanism to be informed by a data-mining based approached [209] that creates a consciousness in the user and/or the design house to choose the value of the parameter α in the routing algorithm based on the threat perception. This is because increasing randomness by choosing lower values of α causes worsening of NoC latency due to sub-optimal paths repeatedly chosen by packets [210]. We anticipate a data corpus of hardware Trojan documents (conference and journal publications as well as documents from NIST database) that describes multiple Trojans and its impacts on the system. These documents will be mined to extract the design features of the HTs and their impacts to automatically extract the features and analyze the level of threat. For instance, an HT that is active and leading to data leakage along with power consumption and timing violations will be considered as high-threat compared to mere timing violating delays. Depending on the threat-level, the value of α will be fine-tuned. This design is shown in Fig. 5.15. More detailed analysis of data-mining approach will be performed in our future works.

5.2.4.3 Effectiveness of Protective Routing Method

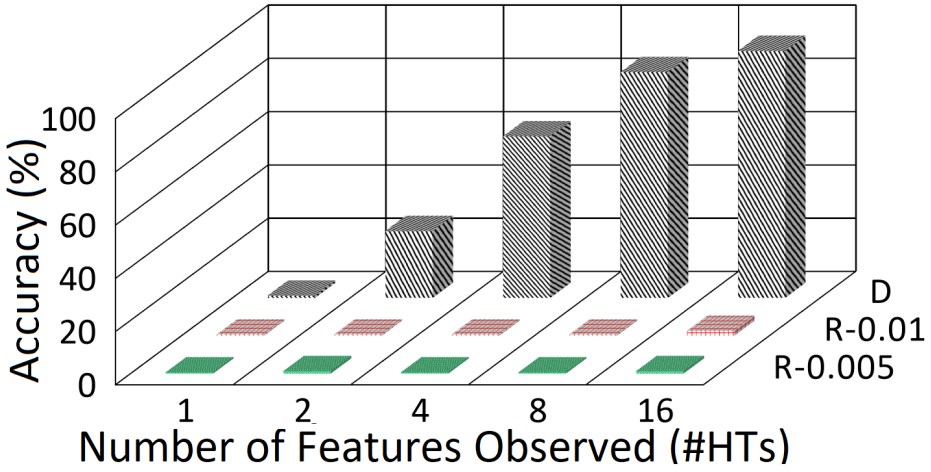
We evaluate the effectiveness of the proposed SA-based random routing in protecting against the RAHT attack. As the degree of randomness for the proposed random routing can vary depending on α, we consider D (deterministic routing), $\alpha = 0.01$, $\alpha = 0.005$, and analyze the accuracy of the attacker with different feature sizes. Figure 5.17a and 5.17b show the accuracy of the attacker on folded torus architecture for deterministic and random routing with 2 and 3 applications respectively. It can be observed that due to the higher routing obfuscation introduced by decreasing α values in the proposed SA-based random routing, the accuracy of the attacker falls down significantly ($<$15%) compared to deterministic routing for smaller α values with higher feature sizes. In the case of 3 applications, the

accuracy drops even more drastically due to a higher confusion created by the routing. Moreover, reducing α decreases the attack accuracy even more while causing even more randomness, indicating the need for the threat perception consciousness enabled by the data-mining approach to not cause unnecessary degradation in latency.

The same 64 core processor was considered to be connected using the NoC topologies listed in Table 5.9. In Fig. 5.18 it is shown that with deterministic routing strategies, the RAHT along with the external attacker can infer the topology of the NoC in the system with over 80% accuracy with 4 or more RAHTs. Moreover, with the SA-based random routing the attacker accuracy drops to less than 20% in most cases demonstrating that the SA-based routing can protect against attacks on NoC topology reverse-engineering.



(a) Attack accuracy for random routing for $\alpha = 0.01, 0.005$ and Deterministic routing (D)



(b) Attack accuracy for random routing with three applications for $\alpha = 0.01, 0.005$ and Deterministic routing (D)

Figure 5.8: Baseline attacker model accuracy for detecting applications with deterministic routing and SA-based random routing with mix of two and three applications

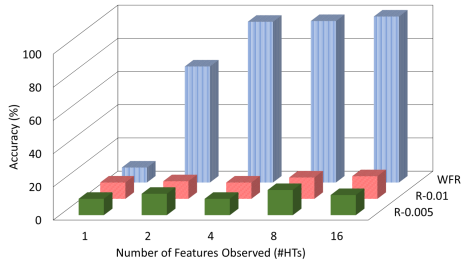


Figure 5.9: Attack accuracy for pseudo-adaptive west-first routing

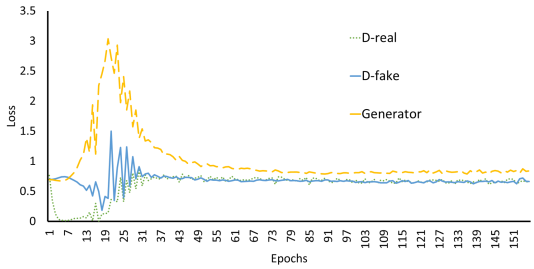


Figure 5.10: Evaluation of GAN using the loss plot

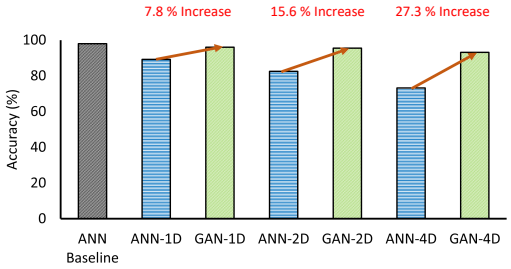


Figure 5.11: Comparison of ANN performance with baseline and advanced attacker model for different missing data scenarios

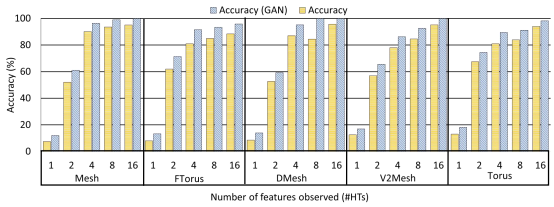


Figure 5.12: Comparison of ANN performance with the baseline attack model and advanced attack model shown for different feature sizes (number of HTs observed)

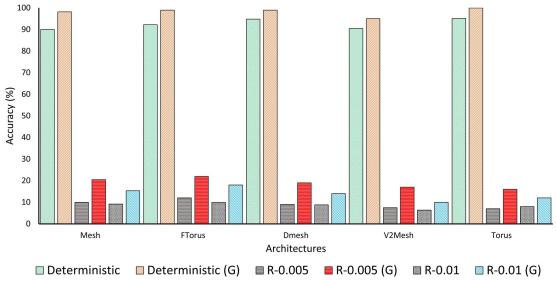


Figure 5.13: Comparison of the ANN performance with the baseline attack model and advanced attack model on the proposed random routing technique

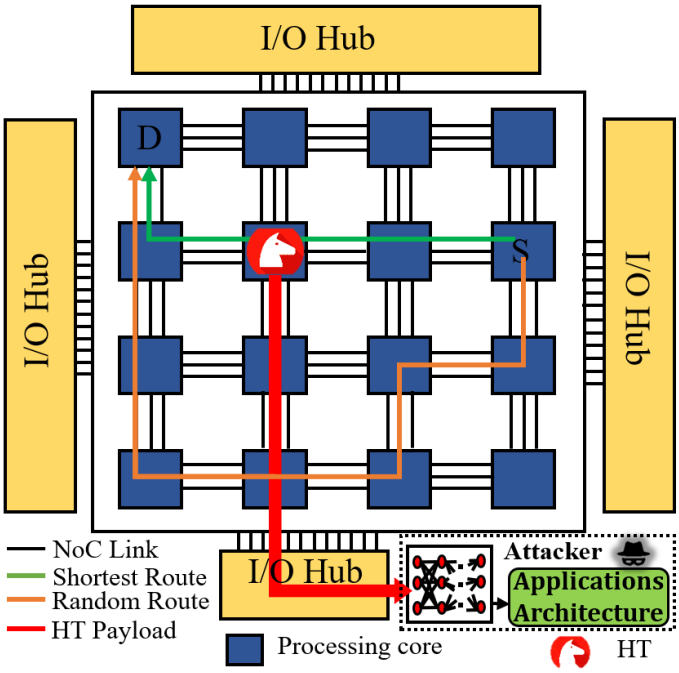


Figure 5.14: Multi-core NoC with proposed threat model

Table 5.8: Component configuration for simulation

Component	**Configuration**
System size	64 cores, Out-of-Order, 16cores/chip
Cache	32KB (private L1), 512KB (shared L2), MOESI
NoC router	3 stage pipelined 5 ports,0.078pJ/bit
Total VC	4, each 8 flits deep, 64 bits/flit
Wired NoC links	64-bit flits, single cycle latency, 0.2pJ/bit/mm
Technology	65nm, 1V supply, 1GHz clock

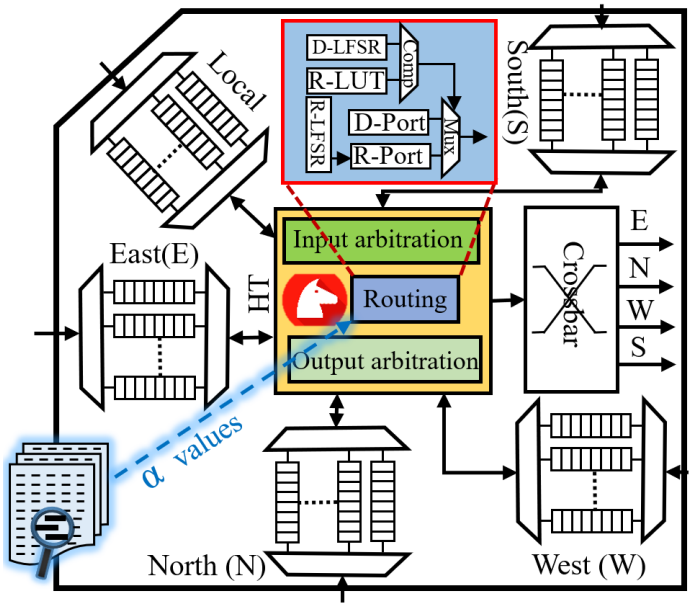
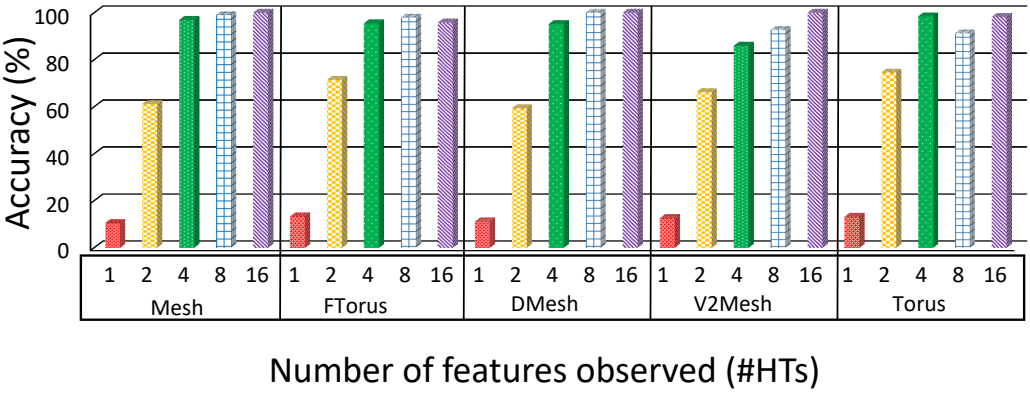


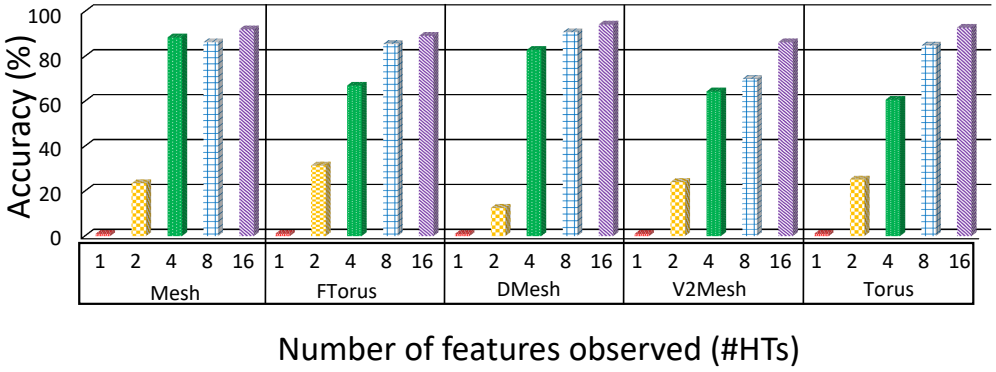
Figure 5.15: NoC switch components with inserted HT

Table 5.9: Architectures for evaluation

Architecture name	**Topology**	**VCs**	**Cache access**
Mesh	Mesh	4	UMA
V2Mesh	Mesh	2	UMA
DMesh	Mesh	4	NUMA
Torus	Torus	4	UMA
FTorus	Folded Torus	4	UMA

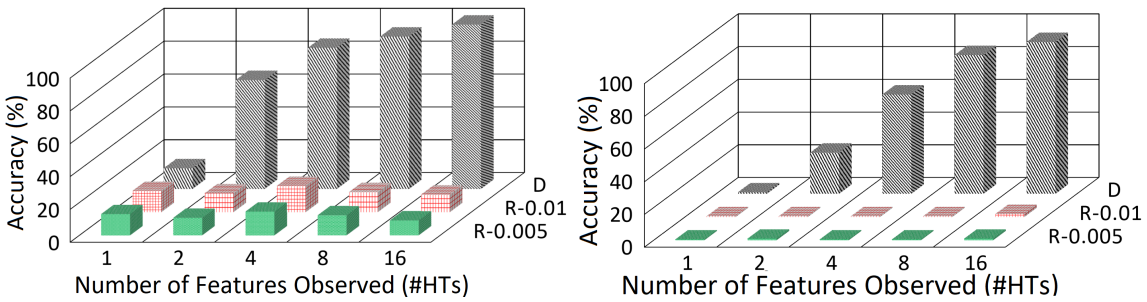


(a) 2 concurrent applications



(b) 3 concurrent applications

Figure 5.16: ANN accuracy for detecting applications with deterministic routing



(a) 2 concurrent applications (b) 3 concurrent applications

Figure 5.17: Attack accuracy with the proposed routing for application detection

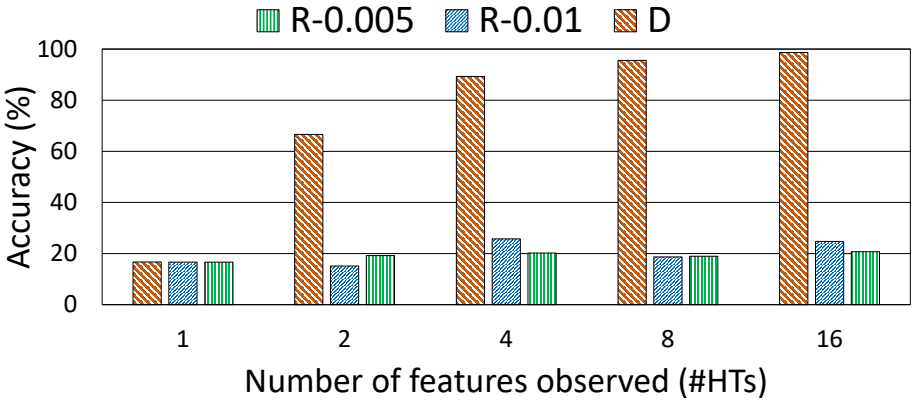


Figure 5.18: Attack accuracy with proposed routing for topology detection

Chapter 6: Conclusion and Future Works

6.1 Conclusion

In the discussion thus far we discussed that Computing systems since their origin have evolved and become pervasive in different fields of technology, extensively. The computing systems have evolved in regards to speed, performance, optimization, and security. Yet, in recent decade, the computing system discipline has witnessed a diverse range of attacks, targeting both software and hardware, exploiting the underlying vulnerabilities in the systems. The dissertation discussed the threats caused by malware, cache-based side-channel attacks, and hardware-based Trojan. The research presented to mitigate these threats incurred less overhead, required low/no modifications to the hardware architecture and can be extended to a variety of victim applications. The research presented is my approach to mitigate the threats. But, as the computing systems are evolving to accommodate the ever increasing demands of the technology, the scope of the attacks is increasing manifold. Hence, the field of defenses against hardware security threats will be ever evolving to protect us and our systems from adversaries.

6.2 Completed Works

This section presents in brief the projects that are completed[1].

1. **Malware detection ***: Malware detection, as discussed earlier, is a non-trivial task and Chapter 3 already presented how hardware attributes, also known as hardware performance counters, can be utilized with ML/DNN techniques to categorize benign and malware application. To render further resiliency to the hardware-based malware

[1] An * marks completed project

detector (HMD), we will be integrating the hardware attributes with packet count information captured when each profiled application accesses the internet. This enables us to capture even more data for providing resiliency to the detection process. The concept will be tested with a wide range of ML/DNN techniques to evaluate the performance gains.

2. **Network-on-chip Hardware Trojan ***: Chapter 5 presented how a simple Hardware Trojan (HT) can leak out confidential data and violate security boundaries. It also discussed a defense technique to mitigate the threat. The dataset utilized in this work is captured on a real-world NOC simulator system. It is non-trivial to generate a large dataset on the existing environment, nor it is feasible to cover all the corner cases of actual packet counts observed while executing the applications on the simulator. The future work in this domain would be to augment the data using neural networks. The augmented dataset can be utilized to evaluate the DNN performance and present analysis supporting the same.

3. **Side-Channel Domain ***: Machine learning techniques are deployed in a wide range of domains. The ML/DNN techniques offer superior performance. Hence, by degrading these implementations, the adversary can degrade the system behavior. Currently, there is a long list of research papers that discuss how certain exploits could be utilized to attack ML/DNN techniques. Chapter 4 presented a mitigation strategy for a cache-based SCA attack. My future work in this domain would be to present a mitigation strategy to protect the deep learning applications from side-channel attack threat.

Appendix A: List of publications

A.1 Published Papers

- [211] Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar, Sahil Bhat, Abhijitt Dhavlle, Hossein Sayadi, Avesta Sasan, Houman Homayoun, Setareh Rafatirad, "Adversarial attack on microarchitectural events based malware detectors", Design Automation Conference (DAC), 2019.

- [212] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, Houman Homayoun, "Advances and throwbacks in hardware-assisted security: Special session", Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems(CASES), 2018.

- [213] M Meraj Ahmed, Abhijitt Dhavlle, Naseef Mansoor, Purab Sutradhar, Sai Manoj Pudukotai Dinakarrao, Kanad Basu, Amlan Ganguly, "Defense against on-chip trojans enabling traffic analysis attacks", Asian Hardware Oriented Security and Trust Symposium (AsianHOST), 2020.

- [214] Abhijitt Dhavlle, Setareh Rafatirad, Khaled Khasawneh, Houman Homayoun, Sai Manoj Pudukotai Dinakarrao, "Imitating functional operations for mitigating side-channel leakage", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2021.

- [215] Abhijitt Dhavlle, Sanket Shukla, Setareh Rafatirad, Houman Homayoun, Sai Manoj Pudukotai Dinakarrao, "Hmd-hardener: Adversarially robust and efficient hardware-assisted runtime malware detection", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021.

- [152] Abhijitt Dhavlle, Raj Mehta, Setareh Rafatirad, Houman Homayoun, Sai Manoj Pudukotai Dinakarrao, "Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks", International Symposium on Quality Electronic Design (ISQED), 2020.

- [216] Abhijitt Dhavlle, Sai Manoj Pudukotai Dinakarrao, "A comprehensive review of ml-based time-series and signal processing techniques and their hardware implementations", International Green and Sustainable Computing Workshops (IGSC), 2020.

- [217] Abhijitt Dhavlle, Rakibul Hassan, Manideep Mittapalli, Sai Manoj Pudukotai Dinakarrao, "Design of Hardware Trojans and its Impact on CPS Systems: A Comprehensive Survey", International Symposium on Circuits and Systems (ISCAS), 2021.

- [218] M Meraj Ahmed, Abhijitt Dhavlle, Naseef Mansoor, Sai Manoj Pudukotai Dinakarrao, Kanad Basu, Amlan Ganguly, "What Can a Remote Access Hardware Trojan do to a Network-on-Chip?", International Symposium on Circuits and Systems (ISCAS), 2021.

- [219] Abhijitt Dhavlle, Setareh Rafatirad, Houman Homayoun, and Sai Manoj Pudukotai Dinakarrao, "CR-Spectre: Defense-aware ROP InjectedCode-Reuse based Dynamic Spectre", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022.

- [220] Sathwika Bavikadi, Abhijitt Dhavlle, ,Amlan Ganguly, Anand Haridass, Hagar Hendy, Cory Merkel,Vijay Janapa Reddi, Purab Ranjan Sutradhar, Arun Joseph, and Sai Manoj Pudukotai Dinakarrao,"A Survey on Machine Learning Accelerators and Evolutionary Hardware Platforms," in IEEE Design & Test, vol. 39, no. 3, pp. 91-116, June 2022.

A.2 Papers Under Review

- Abhijitt Dhavlle, Ahmed Meraj, Mansoor Naseef, Kanad Basu, Amlan Ganguly, and Sai Manoj Pudukotai Dinakarrao, "Defense against On-Chip Trojans Enabling Traffic Analysis Attacks based on Machine Learning and Data Augmentation", IEEE Transactions on Computers, 2022.

- Thomas Mountford, Abhijitt Dhavlle, Andrew Tevebaugh, Naseef Mansoor, Sai Manoj and Amlan Ganguly, "Address Obfuscation to Protect Against Hardware Trojans in Network-on-Chips", NoCArc 2022: Network on Chip Architectures, 2022.

- Sanket Shukla, Abhijitt Dhavlle, Houman Homayoun, Setareh Rafatirad and Sai Manoj Pudukotai Dinakarrao, "Securing IoT Networked Systems at Runtime by Network and Device Characteristics to Confine Malware Epidemics", ICCD, 2022.

Bibliography

Bibliography

[1] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the ISCA*, 2007.

[2] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the IEEE/ACM*, 2017.

[3] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[4] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[5] S. Crane, A. Homescu, and et al., "Thwarting cache side-channel attacks through dynamic software diversity," in *In Network and Distributed System Security Symposium*, 2015.

[6] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy (SP)*, 2019.

[7] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Conference on Security*, 2014.

[8] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[9] S. Hossein and et al., "Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning," in *ACM Computing Frontiers*, 2018.

[10] H. Sayadi, N. Patel, S. M. PD, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *Design Automation Conference (DAC)*, 2018.

[11] H. Sayadi, H. M. Makrani, S. M. P. Dinakarrao, T. Mohsenin, A. Sasan, S. Rafatirad, and H. Homayoun, "2smart: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection." in *DATE*, 2019.

[12] S. M. P. Dinakarrao, S. Amberkar, S. Bhat, A. Dhavlle, H. Sayadi, A. Sasan, H. Homayoun, and S. Rafatirad, "Adversarial attack on microarchitectural events based malware detectors," in *Design Automation Conference (DAC)*, 2019, pp. 1–6.

[13] J. Clements and Y. Lao, "Hardware trojan design on neural networks," in *Int. Symp. on Circuits and Syst.*, 2019.

[14] X. Hu, Y. Zhao, L. Deng, L. Liang, P. Zuo, J. Ye, Y. Lin, and Y. Xie, "Practical attacks on deep neural networks by memory trojaning," *Trans. on CAD of Integrated Circuits and Syst.*, 2020.

[15] W. Yu and Y. Wen, "Malicious attacks on physical unclonable function sensors of internet of things," in *North Atlantic Test W. (NATW)*, 2019.

[16] S. Guo, J. Wang, Z. Chen, Y. Li, and Z. Lu, "Securing iot space via hardware trojan detection," *IEEE IoT Journal*, vol. 7, no. 11, pp. 11 115–11 122, 2020.

[17] S. Bhasin, J. Danger, S. Guilley, X. T. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic IP cores," in *W. on Fault Diagnosis and Tolerance in Cryptography*, 2013.

[18] X. Wang, S. Narasimhan, A. Krishna, T. Mal-Sarkar, and S. Bhunia, "Sequential hardware trojan: Side-channel aware design and placement," in *Int. Conf. on Computer Design*, 2011.

[19] S. Kelly, X. Zhang, M. Tehranipoor, and A. Ferraiuolo, "Detecting hardware trojans using on-chip sensors in an ASIC design," *Journal of electronic testing*, vol. 31, no. 1, pp. 11–26, 2015.

[20] A. Ferraiuolo, X. Zhang, and M. Tehranipoor, "Experimental analysis of a ring oscillator network for hardware trojan detection in a 90nm ASIC," in *ICCAD*, 2012.

[21] *https://www.symantec.com/security-center/threat-report.*

[22] *https://lp.skyboxsecurity.com/rs/440-MPQ-510/images/Skybox_Report_Vulnerability_and_Threat_Tre*

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[24] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, pp. 34 – 39, Jun 2008.

[25] *https://spectrum.ieee.org/computing/hardware/counterfeit-chips-on-the-rise*, accessed: 13-Nov-2020.

[26] Kaspersky, "Advanced threat defense and targeted attack risk migration," *White Paper*, pp. 1–12, 2017, https://media.kaspersky.com/en/business-security/enterprise/KL_KATA_Whitepaper_OG.pdf.

[27] J. Demme and et al., "On the feasibility of online malware detection with performance counters," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun 2013.

[28] J. Cruz, F. Farahmandi, A. Ahmed, and P. Mishra, "Hardware trojan detection using atpg and model checking," in *Int. Conf. on VLSI Design*, 2018.

[29] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[30] A. Garcia-Serrano, "Anomaly detection for malware identification using hardware performance counters," *CoRR*, vol. abs/1508.07482, 2015.

[31] A. Tang, S. Sethumadhavan, and S. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses*, 2014.

[32] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *ACM/EDAA/IEEE Design Automation Conference*, 2017.

[33] H. Sayadi and et al., "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *Design Automation Conference*, 2018.

[34] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[35] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 296–310.

[36] M. B. Bahador, M. Abadi, and A. Tajoddin, "Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2014.

[37] H. Sayadi, H. Mohammadi Makrani, O. Randive, S. M. P D, S. Rafatirad, and H. Homayoun, "Customized machine learning-based hardware-assisted malware detection in embedded devices," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*, 2018.

[38] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.

[39] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 35:1–35:21, Jan. 2012.

[40] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 01 2006.

[41] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *ACM Workshop on Computer Security Architectures*, 2008.

[42] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008.

[43] V. Kiriansky, I. A. Lebedev, and et al., "DAWG: A defense against cache timing attacks in speculative execution processors," in *MICRO*, 2018.

[44] O. Oleksenko, B. Trach, and et al., "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX*, 2018.

[45] C. Bao and A. Srivastava, "3d integration: New opportunities in defense against cache-timing side-channel attacks," *IEEE (ICCD)*, 2015.

[46] X. Dong, Z. Shen, and st al., "Shielding software from privileged side-channel attacks," in *USENIX Security Symposium*, 2018.

[47] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[48] F. Brasser, L. Davi, and et. al., "Advances and throwbacks in hardware-assisted security: Special session," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2018.

[49] S. Shukla, G. Kolhe, S. M. P. Dinakarrao, and S. Rafatirad, "Stealthy malware detection using rnn-based automated localized feature extraction and classifier," in *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2019.

[50] S. Shukla and et.al., "Microarchitectural events and image processing-based hybrid approach for robust malware detection: work-in-progress," in *Embedded Systems Week*, 2019.

[51] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[52] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[53] P. Li, D. Gao, and M. K. Reiter, "Stopwatch: A cloud architecture for timing channel mitigation," *ACM Trans. Inf. Syst. Secur.*, vol. 17, Nov. 2014.

[54] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, 2011.

[55] S. Skorobogatov, *Physical Attacks and Tamper Resistance*, 2012.

[56] E. Dubrova, M. Näslund, and G. Selander, "Secure and efficient lbist for feedback shift register-based cryptographic systems," in *2014 19th IEEE European Test Symposium (ETS)*, 2014, pp. 1–6.

[57] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[58] M. K. J.Y.V., A. K. Swain, S. Kumar, S. R. Sahoo, and K. Mahapatra, "Run time mitigation of performance degradation hardware trojan attacks in network on chip," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 738–743.

[59] T. Boraten and A. K. Kodi, "Packet security with path sensitization for nocs," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1136–1139.

[60] V. Y. Raparti and S. Pasricha, "Lightweight mitigation of hardware trojan attacks in noc-based manycore computing," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[61] R. Manju, A. Das, J. Jose, and P. Mishra, "Sectar: Secure noc using trojan aware routing," in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2020, pp. 1–8.

[62] D. M. Ancajas, K. Chakraborty, and S. Roy, "Fort-nocs: Mitigating the threat of a compromised noc," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[63] R. Fernandes, C. Marcon, R. Cataldo, J. Silveira, G. Sigl, and J. Sepúlveda, "A security aware routing approach for noc-based mpsocs," in *Symp. on Integrated Circuits and Systems Design*, 2016.

[64] L. S. Indrusiak, J. Harbin, and M. J. Sepulveda, "Side-channel attack resilience through route randomisation in secure real-time networks-on-chip," in *Int. Symp. on Reconfigurable Communication-centric SoC (ReCoSoC)*, 2017.

[65] T. H. Boraten and A. K. Kodi, "Securing nocs against timing attacks with non-interference based adaptive routing," in *IEEE/ACM Int. Symp. on Networks-on-Chip (NOCS)*, 2018.

[66] *https://www.darkreading.com/vulnerabilities—threats/researchers-develop-new-side-channel-attacks-on-intel-cpus/d/d-id/1337287*.

[67] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 1, May 2016.

[68] J. Rajendran, E. Gavas, J. Jimenez, V. Padman, and R. Karri, "Towards a comprehensive and systematic classification of hardware trojans," in *Int. Symp. on Circuits and Syst.*, 2010.

[69] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[70] R. Hassan, G. Kolhe, S. Rafatirad, H. Homayoun, and S. M. PD, "A cognitive sat to sat-hard clause translation-based logic obfuscation," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021.

[71] Z. Chen, X. Guo, R. Nagesh, A. Reddy, M. Gora, and A. Maiti, "Hardware trojan designs on basys fpga board."

[72] M. N. I. Khan, K. Nagarajan, and S. Ghosh, "Hardware trojans in emerging non-volatile memories," in *DATE*, 2019.

[73] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A Case Study in Hardware Trojan Design and Implementation," *Int. J. Inf. Secur.*, vol. 10, no. 1, p. 1–14, Feb. 2011.

[74] J. Zhang and Q. Xu, "On hardware Trojan design and implementation at register-transfer level," in *Int. Symp. on HOST*, 2013.

[75] T. Kumaki, M. Yoshikawa, and T. Fujino, "Cipher-destroying and secret-key-emitting hardware Trojan against AES core," in *Int. Midwest Symp. on CAS*, 2013.

[76] R. Elnaggar, K. Chakrabarty, and M. B. Tahoori, "Hardware trojan detection using changepoint-based anomaly detection techniques," *Trans. on Very Large Scale Integration (VLSI) Syst.*, vol. 27, no. 12, pp. 2706–2719, 2019.

[77] M. M. Ahmed, A. Dhavlle, N. Mansoor, P. Sutradhar, S. M. P. Dinakarrao, K. Basu, and A. Ganguly, "Defense against on-chip trojans enabling traffic analysis attacks," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020.

[78] A. Vashist, A. Keats, S. M. Pudukotai Dinakarrao, and A. Ganguly, "Securing a wireless network-on-chip against jamming-based denial-of-service and eavesdropping attacks," *Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2781–2791, 2019.

[79] ——, "Securing a wireless network-on-chip against jamming based denial-of-service attacks," in *ISVLSI*, 2019.

[80] F. Brasser, L. Davi, A. Dhavlle, T. Frassetto, S. M. P. Dinakarrao, S. Rafatirad, A.-R. Sadeghi, A. Sasan, H. Sayadi, S. Zeitouni, and H. Homayoun, "Advances and throwbacks in hardware-assisted security: Special session," in *CASES '18*, 2018.

[81] K. Basu, S. M. Saeed, C. Pilato, M. Ashraf, M. T. Nabeel, K. Chakrabarty, and R. Karri, "CAD-Base: An Attack Vector into the Electronics Supply Chain," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 4, Apr. 2019.

[82] C. Pilato, K. Basu, F. Regazzoni, and R. Karri, "Black-hat high-level synthesis: Myth or reality?" *Trans. on Very Large Scale Integration (VLSI) Syst.*, vol. 27, no. 4, pp. 913–926, 2019.

[83] S. Ghandali, G. Becker, D. Holcomb, and C. Paar, "A design methodology for stealthy parametric trojans and its application to bug attacks," in *Cryptographic Hardware and Embedded Systems*, 2016.

[84] S. Kaji, M. Kinugawa, D. Fujimoto, and Y. Hayashi, "Data injection attack against electronic devices with locally weakened immunity using a hardware trojan," *Trans. on EM Compatibility*, vol. 61, no. 4, pp. 1115–1121, 2019.

[85] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian, "Parametric trojans for fault-injection attacks on cryptographic hardware," in *W. on Fault Diagnosis and Tolerance in Cryptography*, 2014.

[86] M. M. Ahmed, A. Vashist, S. M. P. D, and A. Ganguly, "Architecting a Secure Wireless Interconnect for Multichip Communication: An ML Approach," in *AsianHOST*, 2020.

[87] M. Xue, R. Bian, W. Liu, and J. Wang, "Defeating untrustworthy testing parties: A novel hybrid clustering ensemble based golden models-free hardware trojan detection method," *IEEE Access*, vol. 7, 2019.

[88] M. Yasin, O. Sinanoglu, and J. Rajendran, "Testing the Trustworthiness of IC Testing: An Oracle-Less Attack on IC Camouflaging," *Trans. on Info. Forensics and Security*, vol. 12, no. 11, pp. 2668–2682, 2017.

[89] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware trojan insertion by direct modification of fpga configuration bitstream," *IEEE Design and Test*, vol. 30, no. 2, pp. 45–54, 2013.

[90] J. Breier and W. He, "Multiple fault attack on present with a hardware trojan implementation in fpga," in *Int. W. on Secure IoT*, 2015.

[91] K. Nagarajan, M. N. I. Khan, and S. Ghosh, "Entt: A family of emerging nvm-based trojan triggers," in *Int. Symp. on Hardware Oriented Security and Trust*, 2019.

[92] C. Krieg, C. Wolf, and A. Jantsch, "Malicious lut: A stealthy fpga trojan injected and triggered by the design flow," in *ICCAD*, 2016.

[93] G. Becker, F. Regazzoni, C. Paar, and W. Burleson, "Stealthy dopant-level hardware trojans: Extended version," *Journal of Cryptographic Engineering*, vol. 4, pp. 19–31, 04 2014.

[94] M. Xue, C. Gu, W. Liu, S. Yu, and M. O'Neill, "Ten years of hardware trojans: a survey from the attacker's perspective," *IET Computers Digital Techniques*, vol. 14, no. 6, pp. 231–246, 2020.

[95] X. Wang, T. Mal-Sarkar, A. Krishna, S. Narasimhan, and S. Bhunia, "Software exploitable hardware trojans in embedded processor," in *Int. Symp.on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2012.

[96] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*, 2013.

[97] J. Clements and Y. Lao, "Hardware trojan attacks on neural networks," *CoRR*, vol. abs/1806.05768, 2018.

[98] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "Fpga trojans through detecting and weakening of cryptographic primitives," *IEEE Trans. on CAD of Integrated Circuits and Syst.*, vol. 34, no. 8, pp. 1236–1249, 2015.

[99] K. Nagarajan, A. De, M. N. I. Khan, and S. Ghosh, "Trapped: Dram trojan designs for information leakage and fault injection attacks," *arXiv preprint arXiv:2001.00856*, 2020.

[100] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice—hardware trojan against a high-security usb flash drive," *Journal of Cryptographic Engineering*, vol. 7, no. 3, pp. 199–211, 2017.

[101] A. De, M. N. I. Khan, K. Nagarajan, and S. Ghosh, "Hartbleed: Using hardware trojans for data leakage exploits," *Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 968–979, 2020.

[102] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *S&P*, 2016.

[103] I. H. Abbassi, F. Khalid, S. Rehman, A. M. Kamboh, A. Jantsch, S. Garg, and M. Shafique, "Trojanzero: Switching activity-aware design of undetectable hardware trojans with zero power and area footprint," in *DATE*, 2019.

[104] T. Reece and W. H. Robinson, "Analysis of data-leak hardware trojans in aes cryptographic circuits," in *Int. Conf. on Tech. for Homeland Security*, 2013.

[105] M. Muehlberghuber, F. K. Gürkaynak, T. Korak, P. Dunst, and M. Hutter, "Red team vs. blue team hardware trojan analysis: detection of a hardware trojan on an actual ASIC," in *Int. W. on Hardware and Architectural Support for Security and Privacy*, 2013.

[106] T. Hoque, X. Wang, A. Basak, R. Karam, and S. Bhunia, "Hardware trojan attacks in embedded memory," in *VLSI Test Symp.*, 2018.

[107] S. M. Nair, R. Bishnoi, A. Vijayan, and M. B. Tahoori, "Dynamic faults based hardware trojan design in stt-mram," in *DATE*, 2020.

[108] C. Liu, P. Cronin, and C. Yang, "A mutual auditing framework to protect iot against hardware trojans," in *Asia and South Pacific DAC*, 2016.

[109] B. Shanyour and S. Tragoudas, "Detection of low power trojans in standard cell designs using built-in current sensors," in *Int. Test Conf.*, 2018.

[110] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas," in *Security Symposium (USENIX)*, 2020, pp. 1803–1819.

[111] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, vol. 4, no. 3, pp. 251–266, Aug 2008.

[112] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *Design Automation Conf.*, 2017.

[113] *https://www.redhat.com/en/topics/containers/whats-a-linux-container.*

[114] *https://perf.wiki.kernel.org/index.php/Tutorial*.

[115] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham, "Weka: Practical machine learning tools and techniques with java implementations," 1999.

[116] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[117] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *CoRR*, 2016.

[118] K. Khasawneh and et al., "RHMD: Evasion-resilient hardware malware detectors," in *IEEE/ACM Int. Symp. on Microarchitecture*, 2017.

[119] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014.

[120] A. Huang and et al., "Adversarial deep learning for robust detection of binary encoded malware," *CoRR*, vol. abs/1801.02950, 2018.

[121] M. R. Guthaus and et al., "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE Int. W. on Workload Characterization*, 2001.

[122] (2019) Virustotal intelligence service. Last accessed: 04-May-2019. [Online]. Available: www.virustotal.com/intelligence

[123] (2019) Virusshare team. Last accessed: 04-May-2019. [Online]. Available: www.virusshare.com

[124] C. Li and J. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *Computer Architecture and High Performance Computing*, 2018.

[125] B. A. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," 2020.

[126] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," *Advanced Microkernel Operating Systems*, p. 75, 2018.

[127] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Appl. Soft Comput.*, vol. 49, 2016.

[128] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," Cryptology ePrint Archive, Report 2017/564, 2017.

[129] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Conf. on Computer and Communications Security*, 2007.

[130] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre," *SIGSAC Conference on Computer and Communications Security*, Nov 2019.

[131] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," vol. 15, no. 1, 2012.

[132] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Symposium on Information, Computer and Communications Security*, 2015.

[133] P. Bania, "Security mitigations for return-oriented programming attacks," *CoRR*, 2010.

[134] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *2009 Annual Computer Security Applications Conference*, 2009, pp. 60–69.

[135] A. Sotirov and M. Dowd, "Bypassing browser memory protections," in *In Proceedings of BlackHat*, 2008.

[136] *https://www.computerworld.com/article/2516793/hacker-busts-ie8-on-windows-7-in-2-minutes.html*.

[137] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[138] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *(MICRO)*, 2018.

[139] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019, p. 395–410.

[140] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *WOOT*, 2018.

[141] V. Kiriansky and C. A. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *CoRR*, 2018.

[142] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[143] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite." USA: IEEE Computer Society, 2001.

[144] C. Li and J. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," in *COMPSAC*, 2019.

[145] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Trans. Comput.*, vol. 65, p. 3332–3344, Nov. 2016.

[146] S. Shukla and et.al., "Rnn-based classifier to detect stealthy malware using localized features and complex symbolic sequence," in *International Conference on Machine Learning and Applications*, 2019.

[147] G. Kolhe and et.al., "Security and complexity analysis of lut-based obfuscation: From blueprint to reality," in *Int. Conference On Computer Aided Design*, 2019.

[148] Z. Chen, G. Kolhe, and et.al, "Estimating the circuit deobfuscating runtime based on graph deep learning," in *Design, Automation and Test in Europe Conference (DATE)*, 2020.

[149] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemblehmd: Accurate hardware malware detectors with specialized ensemble classifiers," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 3, pp. 620–633, 2020.

[150] S. M. P. Dinakarrao, S. Amberkar, S. Bhat, A. Dhavlle, H. Sayadi, A. Sasan, H. Homayoun, and S. Rafatirad, "Adversarial attack on microarchitectural events based malware detectors," in *Design Automation Conf.*, 2019.

[151] A. Dhavlle, S. Bhat, S. Rafatirad, H. Homayoun, and S. M. P. D, "Work-in-progress: Sequence-crafter: Side-channel entropy minimization to thwart timing-based side-channel attacks," in *International Conference on Compliers, Architectures and Synthesis for Embedded Systems (CASES)*, 2019.

[152] A. Dhavlle, R. Mehta, S. Rafatirad, H. Homayoun, and S. M. P. Dinakarrao, "Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks," in *International Symposium on Quality Electronic Design (ISQED)*, 2020.

[153] *https://blog.sonicwall.com/en-us/2019/03/new-spoiler-side-channel-attack-threatens-processors-mitigated-by-sonicwall-rtdmi/.*

[154] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*, 2006.

[155] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM Conf. on CCS*, 2012.

[156] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security Privacy*, vol. 8, no. 6, pp. 40–47, Nov 2010.

[157] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *ACM Workshop on Security and Artificial Intelligence*, 2011.

[158] R. L. Rivest and et al., "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, p. 120–126, Feb. 1978.

[159] *https://www.gnupg.org/.*

[160] *https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html.*

[161] *http://man7.org/linux/man-pages/man4/random.4.html*.

[162] T. Hornby. (2016) Flush+reload attack. Last accessed: 1-July-2019. [Online]. Available: https://github.com/defuse/flush-reload-attacks

[163] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1608.04644

[164] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," 2016.

[165] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2014.

[166] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations, ICLR*, Y. Bengio and Y. LeCun, Eds., 2014.

[167] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1211–1220.

[168] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *USENIX Security Symposium*, 2019, pp. 497–514.

[169] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[170] K. Xu, S. Liu, P. Zhao, P.-Y. Chen, H. Zhang, Q. Fan, D. Erdogmus, Y. Wang, and X. Lin, "Structured adversarial attack: Towards general implementation and better interpretability," in *International Conference on Learning Representations*, 2019.

[171] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 372–387.

[172] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, oct 2019.

[173] F. Croce and M. Hein, "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks," in *International conference on machine learning*. PMLR, 2020, pp. 2206–2216.

[174] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 582–597.

[175] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[176] A. Sinha, H. Namkoong, R. Volpi, and J. Duchi, "Certifying some distributional robustness with principled adversarial training," 2017. [Online]. Available: https://arxiv.org/abs/1710.10571

[177] A. Shafahi, M. Najibi, M. A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, "Adversarial training for free!" *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[178] E. Wong, L. Rice, and J. Z. Kolter, "Fast is better than free: Revisiting adversarial training," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=BJx040EFvH

[179] M. Andriushchenko and N. Flammarion, "Understanding and improving fast adversarial training," *Advances in Neural Information Processing Systems*, vol. 33, pp. 16 048–16 059, 2020.

[180] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," 2017. [Online]. Available: https://arxiv.org/abs/1705.07204

[181] Y. Dong, H. Su, J. Zhu, and F. Bao, "Towards interpretable deep neural networks by leveraging adversarial examples," 2017.

[182] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," 2016.

[183] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song, "Spatially transformed adversarial examples," 2018.

[184] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 68, no. 1, pp. 49–67, 2006.

[185] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Optimization with sparsity-inducing penalties," 2011.

[186] S. Liu, S. Kar, M. Fardad, and P. K. Varshney, "Sparsity-aware sensor collaboration for linear coherent estimation," *IEEE Transactions on Signal Processing*, vol. 63, no. 10, pp. 2582–2596, may 2015.

[187] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: Elastic-net attacks to deep neural networks via adversarial examples," 2017. [Online]. Available: https://arxiv.org/abs/1709.04114

[188] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[189] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM Journal on Imaging Sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[190] Q. Liu, X. Shen, and Y. Gu, "Linearized admm for non-convex non-smooth optimization with convergence analysis," 2017.

[191] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conf.*, 2001.

[192] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.

[193] A. Dhavlle, R. Hassan, M. Mittapalli, and S. M. P. Dinakarrao, "Design of hardware trojans and its impact on cps systems: A comprehensive survey," in *International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[194] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *IEEE Int. W. on Hardware-Oriented Security and Trust*, 2009.

[195] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *IEEE Int. High Level Design Validation and Test W.*, 2009.

[196] M. Meraj Ahmed, A. Dhavlle, N. Mansoor, P. Sutradhar, S. M. Pudukotai Dinakarrao, K. Basu, and A. Ganguly, "Defense against on-chip trojans enabling traffic analysis attacks," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.

[197] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. on Computers*, vol. 54, no. 8, pp. 1025–1040, 2005.

[198] M. Badr and N. E. Jerger, "Synfull: Synthetic traffic models capturing cache coherent behaviour," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 109–120, 2014.

[199] R. V. Hogg, J. McKean, and A. T. Craig, *Introduction to mathematical statistics*. Pearson Education, 2005.

[200] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[201] K. Wang, C. Gou, Y. Duan, Y. Lin, X. Zheng, and F.-Y. Wang, "Generative adversarial networks: introduction and outlook," *IEEE/CAA Journal of Automatica Sinica*, vol. 4, no. 4, pp. 588–598, 2017.

[202] B. Vega-Márquez, C. Rubio-Escudero, J. C. Riquelme, and I. Nepomuceno-Chamorro, "Creation of synthetic data with conditional generative adversarial networks," in *14th International Conference on Soft Computing Models in Industrial and Environmental*

Applications (SOCO 2019), F. Martínez Álvarez, A. Troncoso Lora, J. A. Sáez Muñoz, H. Quintián, and E. Corchado, Eds. Springer International Publishing, 2020, pp. 231–240.

[203] C. Bowles, L. Chen, R. Guerrero, P. Bentley, R. Gunn, A. Hammers, D. A. Dickie, M. V. Hernández, J. Wardlaw, and D. Rueckert, "Gan augmentation: Augmenting training data using generative adversarial networks," 2018.

[204] X. Yi, E. Walia, and P. Babyn, "Generative adversarial network in medical imaging: A review," *Medical Image Analysis*, vol. 58, p. 101552, 2019.

[205] P. Teterwak, A. Sarna, D. Krishnan, A. Maschinot, D. Belanger, C. Liu, and W. T. Freeman, "Boundless: Generative adversarial networks for image extension," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[206] Y. Chen, Y.-K. Lai, and Y.-J. Liu, "Cartoongan: Generative adversarial networks for photo cartoonization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[207] S. Murali, D. Atienza, L. Benini, and G. De Micheli, "A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip," in *Design Automation Conf.*, 2006.

[208] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *IEEE ASAP*, 2015.

[209] Z. Zhu and T. Dumitraş, "Featuresmith: Automatically engineering features for malware detection by mining the security literature," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[210] P. Bogdan and R. Marculescu, "Hitting time analysis for fault-tolerant communication at nanoscale in future multiprocessor platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1197–1210, 2011.

[211] S. M. P. Dinakarrao, S. Amberkar, S. Bhat, A. Dhavlle, H. Sayadi, A. Sasan, H. Homayoun, and S. Rafatirad, "Adversarial attack on microarchitectural events based malware detectors," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019.

[212] F. Brasser, L. Davi, A. Dhavlle, T. Frassetto, S. M. P. Dinakarrao, S. Rafatirad, A.-R. Sadeghi, A. Sasan, H. Sayadi, S. Zeitouni, and H. Homayoun, "Advances and throwbacks in hardware-assisted security: Special session," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2018.

[213] M. Meraj Ahmed, A. Dhavlle, N. Mansoor, P. Sutradhar, S. M. Pudukotai Dinakarrao, K. Basu, and A. Ganguly, "Defense against on-chip trojans enabling traffic analysis attacks," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.

[214] A. Dhavlle, S. Rafatirad, K. Khasawneh, H. Homayoun, and S. M. P. Dinakarrao, "Imitating functional operations for mitigating side-channel leakage," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[215] A. Dhavlle, S. Shukla, S. Rafatirad, H. Homayoun, and S. M. Pudukotai Dinakarrao, "Hmd-hardener: Adversarially robust and efficient hardware-assisted runtime malware detection," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1769–1774.

[216] S. M. P. Dinakarrao, "Self-aware power management for multi-core microprocessors," in *IEEE International Green Sustainable Conference (IGSC)*, 2020.

[217] A. Dhavlle, R. Hassan, M. Mittapalli, and S. M. P. Dinakarrao, "Design of hardware trojans and its impact on cps systems: A comprehensive survey," in *International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[218] M. M. Ahmed, A. Dhavlle, N. Mansoor, S. M. P. Dinakarrao, K. Basu, and A. Ganguly, "What can a remote access hardware trojan do to a network-on-chip?" in *International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[219] A. Dhavlle, S. Rafatirad, H. Homayoun, and S. M. P. Dinakarrao, "Cr-spectre: Defense-aware rop injected code-reuse based dynamic spectre," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022, pp. 508–513.

[220] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V. J. Reddi, P. R. Sutradhar, A. Joseph, and S. M. Pudukotai Dinakarrao, "A survey on machine learning accelerators and evolutionary hardware platforms," *IEEE Design Test*, vol. 39, no. 3, pp. 91–116, 2022.

Curriculum Vitae

Abhijitt Dhavlle pursued his diploma in industrial electronics from Fr.Agnel polytechnic, India, followed by Bachelor of Engineering in Electronics and Telecommunication from Mumbai University, India; Later, Abhijitt pursued his Master of Science in Computer Engineering from George Mason University, Virginia. Abhijitt interned at University of Southern California Information Sciences Institute, Arlington, and later interned at Xilinx, Inc., San Jose. Abhijitt's research interests are side-channel attacks, physical side-channel attacks, machine learning and deep learning, and hardware accelerators for machine learning.

Abhijitt Dhavlle

Electrical and Computer Engineering, George Mason University, Fairfax, VA, USA 22031
571-320-7417 | adhavlle@gmu.edu | LinkedIn Profile

Computer engineering graduate with over 4 years of professional and internship experience. Strong background in Applied Deep Learning and Machine Learning, Python, EDA analysis, Embedded Systems, Digital System Design, Hardware Security, and Computer Architecture. Experience in GUI designing, sensor interfacing and monitoring, MATLAB scripting, and Deep Learning Toolbox.

Education

George Mason University, Fairfax, Virginia **Expected August 2022**
Ph.D. in Computer Engineering - Ph.D. Candidate

George Mason University, Fairfax, Virginia **May 2022**
Master of Science in Computer Engineering

University of Mumbai, India **May 2014**
Bachelor of Engineering in Electronics and Telecommunication Engineering

Experience

Application Test Development Intern, Radiation Effects Team, Xilinx Inc. **May 2021 – August 2021**

Design and Integration of Radiation Tolerant Testbench for New Gen. (Versal) FPGAs

- Developed a code for continuous logging of Deep Learning Processing Unit (DPU) Inference on ImageNet dataset for testing.
- Integrated Error Correction Code IP for mitigating radiation induced errors.
- Testing of Fault Aware Training (FAT) model for radiation tolerant testbench.

Technologies: Xilinx Vivado, Xilinx Vitis-AI, Python, Deep Learning

Graduate Research Assistant, George Mason University, Fairfax **January 2019 – Present**

Machine Learning for Side-Channel Attack Evaluation

- Evaluated Neural Network, MLP, Logistic Regression, SVM, and Random Forest ML models on a novel side-channel attack.
- Developed a Python code for automating the attack and Hardware Intrusion Detection system -based defense interaction.
- Automated the generation of perturbed instances of the attack for defense evasion and evaluation of attack performance.

Technologies: Machine Learning, Deep Learning, Python, Pandas, Numpy, Scipy, Sklearn

ML-assisted Off-Chip Hardware Trojan Payload Analysis

- Developed a Artificial Neural Network (ANN) based model to detect applications running on the processor based on the communication packets observed by a hardware Trojan in the Network-on-Chip (NoC).
- Developed a Python code for utilizing custom one-hot encoded inputs and outputs for ANN model.
- Developed a Artificial Neural Network (ANN) model to detect NOC architecture and applications running on a processor from the perspective of an inserted hardware Trojan.

Technologies: Deep Learning, Machine Learning, Python, Pandas, Scikit-learn, Numpy, Seaborn

Machine Learning for Malware Detection using Performance Counters and Network Data

- Utilized Machine Learning models for developing a hardware-based intrusion detector for malware classification.
- Developed the process flow on a IoT device running malware and benign application suite. The ML-based intrusion detector utilized network traffic data and hardware attributes for classification.

Technologies: Machine Learning, Regression, Ensemble Learning, Python, Pandas, Scikit-learn, Numpy, Seaborn

Machine learning assisted Hardware-based Malware Detection

- Extracted hardware performance counters corresponding to various hardware events on Intel-based system running plethora of malware and benign applications.
- Performed exploratory data analysis and component selection to determine predominant events for different classes of malware family.
- Implemented decision tree, multi-layer perceptron, random forest, support vector machine, K-nearest neighbor, and neural network to classify malware and benign applications.
- Implemented classifiers in RTL for FPGA inference. Reported power and area analysis of the implemented classifiers.

Technologies: Deep Learning, Machine Learning, Python, Pandas, Scikit-learn, Numpy, Xilinx Vivado, MATLAB

Hardware Research Intern at University of Southern California - Information Sciences Institute **May 2020 – August 2020**

Power Analysis Side-Channel Attack for Breaking FPGA Bitstream Encryption

- Development and analysis of side-channel attack techniques for Virtex Series FPGAs to decipher secret key to allow bitstream readback for reprogrammability and portability.
- Contributed as a beta tester for verifying WBSTAR attack on FPGA to retrieve encrypted bitstream.

Technologies: Jupyter Notebook, Python, Microsoft Excel, VHDL, Xilinx Vivado, *ChipWhisperer* for Side-Channel Analysis

Technical Skills

Languages: Python, C/C++, MATLAB, Verilog, VHDL
Hardware platforms: Digilent Zedboard, Zybo, Baysy3; Zynq UltraScale MPSoC ZCU102, Raspberry Pi 3/4
ML libraries and frameworks: TensorFlow, Keras, WEKA, scikit-learn, numpy, pandas, matplotlib, scipy
IDE and Tools: Docker, Mobaxterm, PyQt, PyCharm, Jupyter notebook, MS-Office suite, MS-Excel, Simulink, Deep Learning Toolbox
Design, Synthesis and Simulation Tools: VHDL, Verilog, Synopsys Design Compiler, Prime Time, IC Compiler, Xilinx ISE, Vivado HLS
Miscellaneous Tools: Digital oscilloscopes, Logic analyzers, Protocol Analyzers, Power supplies, PCB design tools, Autodesk Fusion 360 CAD tool; USB, I2C, SPI, UART protocols; Soldering; Linux

Relevant Coursework

Digital System Design with VHDL, Computer Architecture, Operating Systems, Advanced Microprocessor & Microcontrollers, Applied Machine Learning, Object-Oriented Programming; Instructed VHDL, IoT Design with Raspberry Pi, H/w Accelerators for Machine Learning.

Selected Publications

1. **Abhijitt Dhavlle**, S. Shukla, S. Rafatirad, H. Homayoun, and S. M. PD. - "HMD-hardener: Adversarially robust and efficient hardware-assisted runtime malware detection" - Design, Automation Test in Europe Conference Exhibition (DATE), 2021
2. **Abhijitt Dhavlle** , Rakibul Hassan, Manideep Mittapalli, and Sai Manoj PD - "Design of Hardware Trojans and its Impact on CPS Systems: A Comprehensive Survey" - International Symposium on Circuits and Systems (ISCAS), 2021
3. **Abhijitt Dhavlle** and Sai Manoj Pudukotai Dinakarrao - "A Comprehensive Review of ML-based Time-Series and Signal Processing Techniques and their Hardware Implementations" - International Green and sustainable computing Conference (IGSC), 2020
4. M Meraj Ahmed, **Abhijitt Dhavlle**, Naseef Mansoor, Purab Sutradhar, Sai Manoj Pudukotai Dinakarrao, Kanad Basu and Amlan Ganguly -"Defense against On-Chip Trojans Enabling Traffic Analysis Attacks" - AsianHOST, 2020
5. Sai Manoj PD, Sairaj Amberkar, Sahil Bhat, **Abhijitt Dhavlle** et al. - "Adversarial Attack on Microarchitectural Events based Malware Detectors" - Design Automation Conference (DAC), 2019