DEFENSE AGAINST CACHE BASED MICRO-ARCHITECTURAL SIDE CHANNEL ATTACKS

by

Sahil Bhat

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

| | |
|---|---|
| _____ | Dr. Houman Homayoun, Thesis Director |
| _____ | Dr. Jens Peter Kaps, Committee Member |
| _____ | Dr. Jim Jones, Committee Member |
| _____ | Dr. Monson Hayes, Chairman, Department of Electrical and Computer Engineering |
| _____ | Dr. Kenneth Ball, Dean Volgenau School of Engineering |
| Date: _____ | Spring Semester 2019<br>George Mason University<br>Fairfax, VA |

Defense Against Cache Based Micro-architectural Side Channel Attacks

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Sahil Bhat
Master of Science
Savitribai Phule Pune University, 2016

Director: Dr. Houman Homayoun, Associate Professor
Department of Electrical and Computer Engineering

Spring Semester 2019
George Mason University
Fairfax, VA

# Dedication

I dedicate this thesis to my loving parents Mr. Vinod Kumar Bhat and Mrs. Veena Koul. I would also love to thank my brother Mr. Mohit Bhat and my friends for their constant love, support and encouragement.

# Acknowledgments

During my graduate study years, I have come across people who played a crucial role in my life. I would like to take this moment to thank them wholeheartedly for their guidance and having strong belief in me. I would especially thank Dr. Houman Homayoun and Dr. Sai Manoj PD, with your encouragement I have achieved all the results and accomplished my research. I would also thank my friend and my colleague Mr. Abhijitt Dhavlle for his motivation and all the help throughout the research. A very big thank goes to my family and my roommates.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

DEFENSE AGAINST CACHE BASED MICRO-ARCHITECTURAL SIDE CHANNEL ATTACKS

Sahil Bhat

George Mason University, 2019

Thesis Director: Dr. Houman Homayoun

To overcome the performance overheads 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 se-cure the systems. To classify benign and malicious applications, HMD primarily relies on the generated low-level micro-architectural events captured through Hardware Performance Counters (HPCs). Moreover, the hardware security domain in recent years has seen many state-of-the-art cache based side channel attacks (SCAs) which have posed and continue to pose threats to the integrity of our data. These attacks function by exploiting the side-channels which invariably leak important data during various operations of its (application) execution. These attacks have been successful to steal the private keys from RSA encryption by observing the sequence of operations. Shutting down the side channels is not a feasible approach due to various restrictions it would pose to system performance, hence it is necessary to reduce the entropy of the side channels to reduce the leakage and thus, thwart such attacks.

This work creates 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. To craft the attack, 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 similar to the adversarial predictor HPC trace. As the crafted adversarial sample generator application does not have any malicious operations, it is not detectable with traditional signature-based malware detection solutions. With the proposed attack, malware detection accuracy has been reduced to 18.04% from 82.76%. We also propose a method to minimize the side channel leakage thus thwarting the attack. A wrapper code adds perturbations to the data leaked by the victim application thereby reducing entropy which makes the data on the attacker's side resemble leaked secret data but with perturbations added which makes it arduous to retrieve the original secret data. The wrapper code 'Entropy Shield' can be implemented to protect any encryption algorithm with only a few tweaks.

# Introduction

### 1.0.1 Hardware Security

The ever-increasing complexity of modern computing systems result in the growth of security vulnerabilities, making such systems an appealing target for sophisticated attacks. The attackers take the advantage of existing vulnerabilities to compromise the systems and deploy malware. Malware, also known as malicious software, is a program or application designed by the attackers 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 (DoS) attack, and disrupting essential services to carry out financial fraud.

To overcome the shortcomings such as latency and computational complexity of traditional malware detection techniques including signature and semantics-based software-driven techniques [1, 2], hardware-assisted malware detection (HMD) approaches are proposed [3]. HMD refers to utilizing the low-level micro-architectural hardware events and logs for detecting and classifying the malware from benign applications. The HMD enjoys the benefit of reduced malware detection latency by orders of magnitude with smaller hardware cost [3]. Recent works [3–10] have shown that by deploying Machine Learning (ML) techniques fed with the low-level micro-architectural events (features) captured by Hardware Performance Counters (HPCs) can aid in differentiating benign and malware applications. The HPCs are a set of special-purpose registers built into modern microprocessors to capture the trace of hardware-related events such as LLC load misses, branch instructions, branch

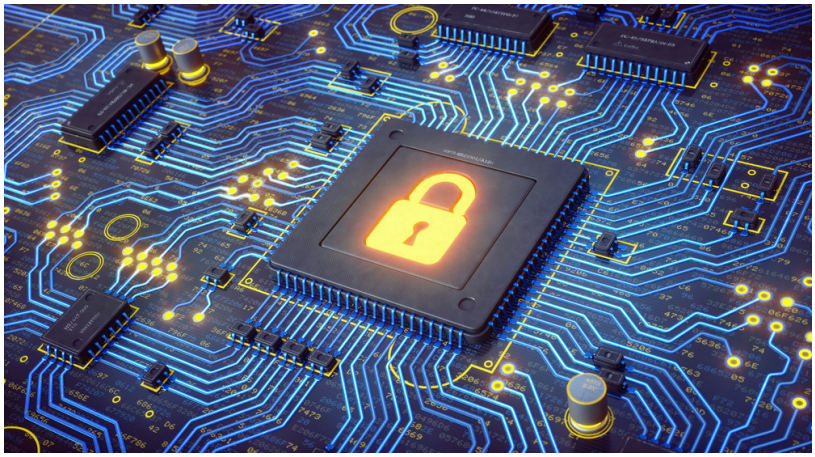


Figure 1.1: Hardware Security

misses, and executed instructions while executing an application (benign or malware).

The work in [3] was one of the preliminary works that has proposed to utilize the HPC data for malware detection and demonstrated the effectiveness of offline ML algorithms in malware classification. They showed high detection accuracy results for Android malware by applying multiple ML algorithms, namely Artificial Neural Network (ANN) and K-Nearest Neighbor (KNN). The researchers in [11] and [4] discussed the feasibility of employing unsupervised learning method on low-level features to detect Return-oriented programming (ROP) and buffer overflow attacks by finding an anomaly in the hardware performance counters' information. Although unsupervised algorithms are more effective in detecting new malware and attacker evolution, they are complex in nature demanding more sophisticated analysis and computational overheads. The work in [12] uses logistic regression to classify malware into multiple classes and train a specialized classifier for detecting malware class. They further used specialized ensemble learning to improve the accuracy of logistic regression. To enhance the performance, the work in [6, 12] proposes use of ensemble ML based solutions for effective malware detection using low-level micro-architectural features. These ML based malware detectors (HMD) can be implemented in microprocessor hardware

with significantly low overhead as compared to the software-based methods, as detection inside the hardware is very fast (few clock cycles) [2, 5]. As a whole, it can be seen that recently a large body of works have been dedicated to employ low-level micro-architectural events fed to ML classifiers to make the systems secure.

On the other hand, despite the ML classifiers being deployed in numerous applications and shown robustness against random noises, the exposed vulnerabilities have shown that the outcome of ML classifiers can be modified or controlled by adding specially crafted perturbations to the input data [13–16], often referred as *Adversarial samples*. A plethora of works on adversarial attacks exist, focusing specifically on computer vision applications [13–16], where the number of features are often large. Recently, a few works on crafting adversarial malware are as well proposed in [17]. However, the works such as [17] consider the application features in a binary format (feature exist or not) for showcasing the attack and defense. Though the application features (in binary format) are manipulated, traditional techniques such as semantic and signature analysis based methods can detect these adversaries [18]. Similarly, in [19] authors talks about the efficiency of detecting malware through HPCs. Though the presented experimental results in [19] are in-favor of efficient malware detection through HPCs, they claim that if HPC traces of malware and benign applications are similar, it is hard to detect malware. However, no details on crafting nor feasibility to create such malware is provided, which limits the efficacy. In contrast to the existing works, *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 furthermore, manipulation of HPCs is highly complex to perform despite employing techniques like code obfuscation for executing malware [3, 20].

Firstly, we assume the victim's defense system to be a black box 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 crafting of adversarial HPC generator is performed by employing a linear model that relates the HPC events and the parameters of the adversarial generator code. This adversarial HPC generator application is wrapped around the application that needs to be misclassified. To the best of our knowledge, *this is the first work that is capable of generating adversarial HPCs through a benign application and proposes a methodology how to craft such an application and obtain adversarial behavior.* The main focus of this work is 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: a) 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); b) the crafted application has no malicious features embedded, thus not detectable by ML malware detectors; and c) scalable and flexible i.e., the crafted application can generate events as required to generate powerful adversary. All the above propose work has been published in our DAC 2019 paper [21].

### 1.0.2 Side Channel Attack

In addition to malware threats, the design complexity also attracted attackers and exposed some of the security vulnerabilities eventually leading to side channel attacks. Despite enhanced performance achieved with advanced features such as cache-sharing, speculative execution, they have been exploited for crafting security attacks. These security threats utilize side-channels to obtain secret information from the system and are passive in nature.

Side-channel attacks are a class of attacks that primarily exploit security of computing

systems based on the obtained side-channel information as a result of design vulnerabilities rather than the exploits in the application. Side-channels are inherent in any computing system and the foremost challenge in defending against side-channel attacks is that they cannot be completely terminated. The obtained side-channel information such as be timing information, power analysis, speculative executions, electromagnetic analysis, and cache-access patterns can be employed to craft side-channel attack. In the recent times, the cache-based side-channel attacks have gained attention in terms of crafting the attacks as well as defenses. This has been further exacerbated with the features introduced in modern computing systems such as memory-sharing, co-location of applications, which were introduced for an efficient resource management and higher throughput. However, a large number of cache-based side-channel attacks rely on the timing information to determine the cache-access (hit or miss) patterns in order to obtain the accessed address and eventually the secret key from the cache. For instance, Flush+Reload [22] depends on the assumption that the victim and the attacker share the same memory space and utilizes the cache-access timing information to retrieve the secret key from system. Attacks such as Prime+Probe [23] supersedes the Flush+Reload attack by not requiring any shared memory space with the victim to extract sensitive information.

To address the challenges of cache side-channel attacks, techniques such as static cache partitioning, partition locked cache, non-monopolizable (nomo) cache architectures are proposed. These techniques can tremendously reduce the interference between the attacker and victim's memory access, thus providing a better defense. However, adopting such techniques require alterations in the cache design and also leads to performance degradation. To overcome the limitations of the cache-partitioning, randomization of cache architectures are introduced. Conventional fully associate cache is one of the preliminary randomization based cache in which a memory line can be mapped to any of the existing cache lines, and similarly any of the cache lines can be evicted in random. Thus, preventing the leakage of cache-access information. Despite the achieved security benefits, this technique incurs large delays and is power hungry. In similar vein, random permutation cache, newcache,

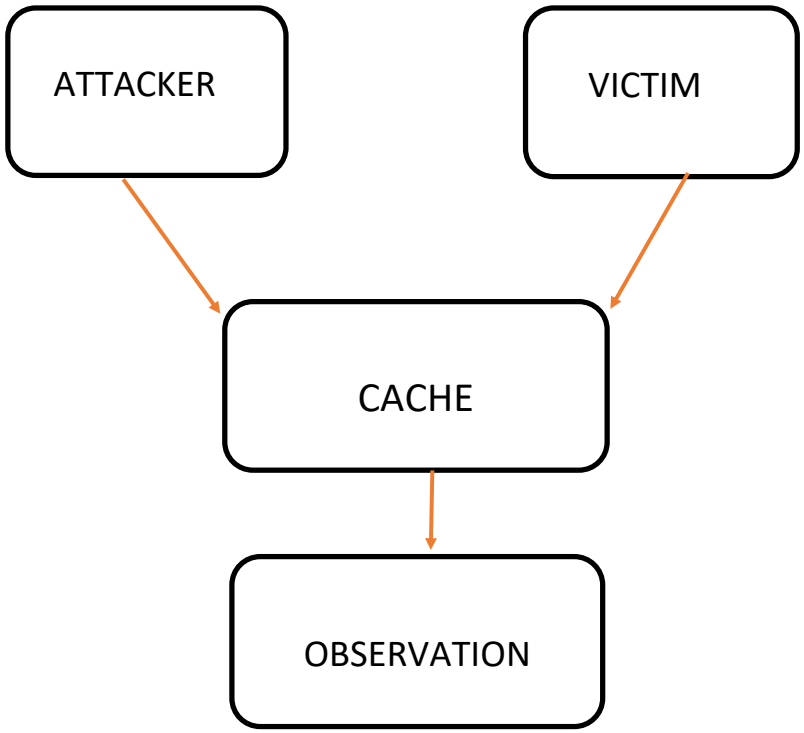



Figure 1.2: Cache Side Channel Attack

random fill cache, and random eviction cache strategies are implemented. Compared to the cache-partitioning, the randomization based solutions have shown higher robustness, yet the challenge of performance degradation is not addressed.

Hardware-assisted security has gained interest among industry as well as researchers due to its low overhead, faster and efficient attack detection capabilities. For instance, the cloud radar [24] discusses the side-channel attack detection by utilizing the on-chip hardware performance counters (HPCs) techniques to monitor different virtual machines (VMs) and further deploy defense through techniques such as VM relocation to thwart the attack. Despite the advancements in defending against the side-channel attacks, most of the aforementioned defenses require significant modifications to the hardware architecture or software such as disabling some of the features such as using 'clflush' instruction to dissuade

the attacker, which is not practical to be deployed and also lead to performance degradation. Previously proposed defenses are confined to a specific attack which makes it difficult to defend against a set of attacks. As a summary, the unsolved challenges and limitations of the existing defenses can be outlined as follows: a) side-channels are inevitable; b) hardware or software modifications can lead to enhanced security, but result in performance degradation and not practical to adapt; and c) assumptions on the attacker side (for example required timing information).

In this work, we introduce defense for timing based side-channel attacks such as Flush+Reload and Prime+Probe. In contrast to the existing techniques that focuses on architectural changes or perturbing cache lines, the proposed defence mechanism primarily focuses on minimizing the entropy of the side-channel information obtained by the attacker without interfering with the original functionality of the victim application. The original application is wrapped with a protective application that is able to facilitate the perturbation of the cache-access timing information obtained by the attacker under the constraints of the achieved information looking similar to the normal timing information, yet leading to a wrong key. The proposed method introduces perturbations in the sequence (timing information) by executing dummy functions that do not affect the result of the key for the victim, but scrambling patterns observed by the attacker thereby reducing entropy and dissuading the attack. Furthermore, we provide the shield application with multiple tuning knobs that provides the options of how many bits in the key needs to be perturbed and how frequently the perturbation across different executions of an application should happen. The proposed technique is evaluated against Flush+Reload and Prime+Probe with different keys and the technique is found to be successful in defending against the attacks without any assumptions on the attacker capabilities.

The contributions of this work are outlined in a three-fold manner as follows:

- an adversarial attack on microarchitectural event based malware detection systems i.e., HMD systems.These HMD systems utilize the underlying hardware performance

counters to capture the microarchitectural events and provide them to ML classifier for detecting and classifying malware

- The proposed defense against cache-based side-channel attacks by minimizing the entropy in the leaked side-channel information, yet ensuring the information is similar to a high entropy information.
- With the aid of proposed technique, users can determine the required level of security i.e., can control the perturbations performed for one execution [1] as well as determine the frequency at which the perturbation across multiple runs[2] can happen.
- The proposed technique here is independent of the victim application and can defend against timing-based side-channel attacks, and demonstrated against Flush+Reload attack with different victim applications in this work.

[1]For a key of $m$-bits, the user can determine how many bits ($k$) to perturb. For instance, for a 128-bit key, user can fix 10-bits to perturb and location will be random.

[2]The location of perturbed keys will change after every $n$ iterations of executing the application, thus introducing randomization within the key and across the keys.

# Background

A cache side-channel attack works by monitoring security critical operations such as AES T-table entry or modular exponentiation multiplicand accesses. Attacker is able to recover the secret key depending on the accesses made (or not made) by the victim, deducing the encryption key. Also, unlike some of the other side-channel attacks, this method does not create a fault in the ongoing cryptographic operation and is invisible to the victim. There exist numerous cache-based side-channel attacks, which utilizes the cache-access information to retrieve the secret key. Here, we describe some of those attacks on which we deploy the proposed defense as a proof-of-concept.

### 2.0.1 Hardware Malware Detectors (HMD)

Hardware based detectors offer fast online detection, efficiency in resource utilization, and invulnerability from getting infected by attackers which make them suitable for mitigating newer threats. However, there are several design challenges with hardware based detectors including having the capability of online monitoring of HPC, low false positives, small logic area and power overhead for implementation on processor, and small detection latency which includes reading HPC and running ML classifiers.They have recently been proposed as a defense against the proliferation of malware. These detectors use low-level features, that can be collected by the hardware performance monitoring units on modern CPUs to detect malware as a computational anomaly. Several aspects of the detector construction have been explored, leading to detectors with high accuracy.Detection of malicious software at the hardware level is emerging as an effective solution to increasing security threats.

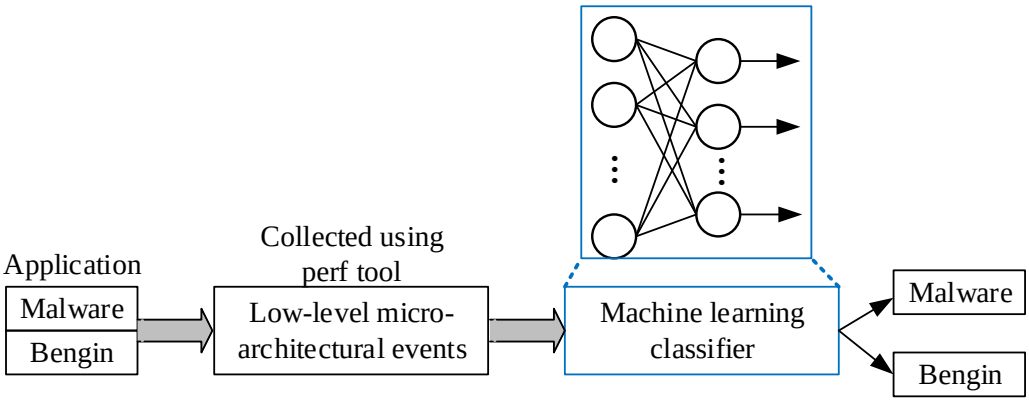



Figure 2.1: Process of detecting malware by employing low-level microarchitectural events

Hardware based detectors rely on Machine Learning(ML) classifiers to detect malware-like execution pattern based on Hardware Performance Counters(HPC) information at run-time. The effectiveness of these learning methods mainly relies on the information provided by expensive-to-implement limited number of HPC.

Hardware Performance Counters(HPC) are special purpose registers available in modern microprocessors which keep track of different micro-architectural events. The main purpose of HPC is to analyze and tune architectural level performance of running applications. While HPC are finding their ways in various processor platforms from high-performance to low power embedded, they are limited in the number of micro-architectural events that can be captured simultaneously. This is mainly due to limited number of physical registers on the processor chip which are expensive to implement.We are using HPC to collect execution traces for all available micro-architectural events by executing collected malware and benign applications in an isolated environment. If two different programs are executed on CPU, they generate different performance counter traces. In HMD, when an application is executed, the low-level micro-architectural events are captured with the aid of HPCs. These low-level micro-architectural events are utilized to train ML classifiers to classify the malware from benign applications.

For detecting malware during run time, the HPCs are collected and provided to the ML classifier to determine whether the executing application is malware or benign [3]. Similarly, [2] proposed a single-stage ML-based HMD and analyzed impact of different ML classifiers on area and power overheads. The work in [25] employed HPC values to construct support vector machine (SVM) detectors to identify malicious programs. Similar works are reported in [6, 12]. Figure 2.1 illustrates the process of using low-level micro-architectural events for malware detection and classification from benign applications.

### 2.0.2 Flush + Reload Attack

For a successful side-channel attack to happen, the attacker needs to monitor the victim application's operations, and is done mostly though shared resources such as library or data with the victim application. Flush+Reload []is one of the earliest cache-access based side-channel attacks that utilizes the cache-access timing information to retrieve the key. The process of Flush+Reload attack can be explained as follows:
*Step 1:* The attacker (spy) flushes a memory line in the (shared) cache. *Step 2:* Spy waits for a certain 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 retrieve the key. This can be inferred as follows: if there was a cache hit for the spy application, it indicates that cache line (data) was accessed (and fetched) by the victim application, else the data is not utilized by the victim. In this manner, the attacker can eavesdrop into security-critical operations of legitimate applications and steal the confidential data. In the Flush+Reload attack, depending on the cache hit/miss and the sequence of the Square, Reduce 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. The process of Flush+Reload attack is depicted in Figure 2.2

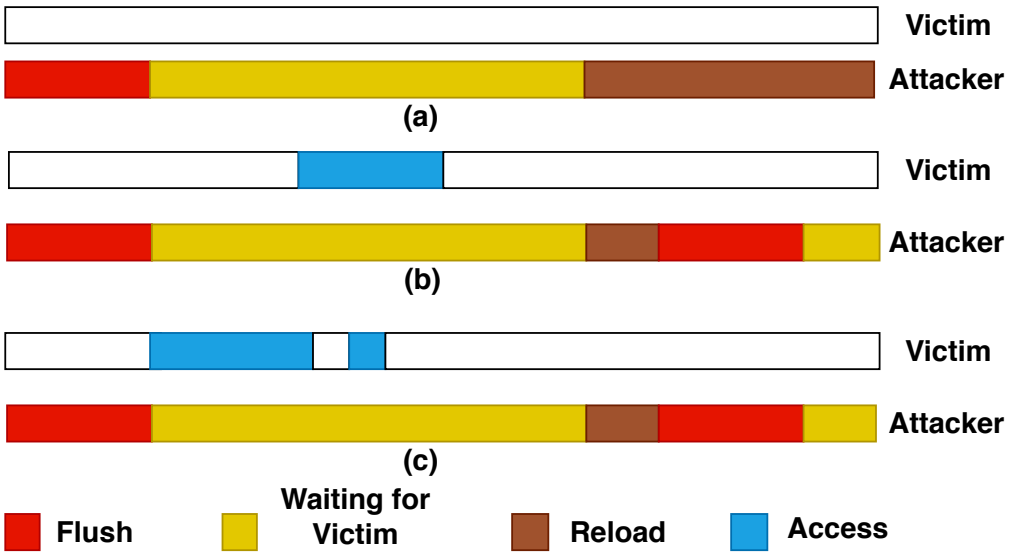



Figure 2.2: Flush+Reload Operation (a) Victim Does not Access; (b)Attack with Victim Access; (c) Victim multi-Access

### 2.0.3 Prime+Probe Attack

Prime+Probe [23] is another kind of cache-access based side-channel attack, In contrast to the Flush+Reload attack, to perform a side-channel attack, there is no need for the spy and victim applications to share the memory or library or data pages. In this attack, the spy application primes the cache i.e., loads a set of memory lines in the cache with it's own data and then waits for some time until the victim application executes. If the victim application happens to utilize the same memory lines of the cache that were previously primed by the attacker, the primed memory lines will be replaced by the victim's data. Further, the spy application tries to access it's own memory lines (primed previously), it will result in a cache miss if the victim utilizes and replaces the primed memory lines with the information required by the victim application and resulting in longer access time. In this manner, by utilizing the Prime+Probe attack, the secret key can be obtained by the attacker.

### 2.0.4 Flush+Flush Attack

The Flush+Flush attack is also a relatively new attack which supersedes the above mentioned cache based attacks both in terms of speed and stealth. Unlike the Flush+Reload attack, it works only by executing 'clflush' instruction in an infinite loop. Since it does not access any data - as Flush+Reload does- the number of cache misses thus created are zero. When the 'clflush' instruction is issued, data that is cached takes more time to be flushed out completely from all cache levels compared to non-cached data which takes less time. Based on the execution time the Flush+Flush concludes if the cache line was cached or not cached. The attack does not load any memory line in to the cache and hence if clflush takes more time to execute would imply that the victim accessed the data.

Referring to Figure 2.2 we have explained three different scenarios for the victim and attacker (spy) accesses, where Figure 2.2a shows spy application running without any victim accesses and hence the flush and reload action by the spy will result in cache misses since the victim does not access any probe cache line. Figure 2.2b shows victim access during the waiting phase of the spy where the spy allows some time expecting the victim to execute and access probed cache line and Figure 2.2c shows multiple accesses by the victim during the waiting phase of the spy where multiple accesses cannot be distinguished by the spy - hence the attacker probes functions or parts of code that are accessed frequently to increase the probability of detection and improve resolution of the attack. Another possible scenario would be the reload phase of the spy and the victim's access to memory are overlapped and in such a case the victim will benefit from the data already brought in to the cache by the spy. Based on the above mentioned Flush+Reload attack mechanism, we will see in later sections how the side-channel data is extrapolated to the actual bits in the secret-key.

All the aforementioned attacks have been successful in attacking AES, GnuPGs RSA, DSA and Elgamal, web browsers,etc.In this work, we demonstrate how the attacker exploits the cache-access time to determine the operations performed by the victim and deduce the corresponding key. We consider an example of GnuPG RSA as the victim application and

Flush+Reload as the spy application. The GnuPG RSA utilizes square, reduce and multiply operations to encrypt and decrypt the data. The attacker has the knowledge of locations of these operations in the victim application and probes them accordingly to determine the sequence of operations to decrypt the employed secret key.

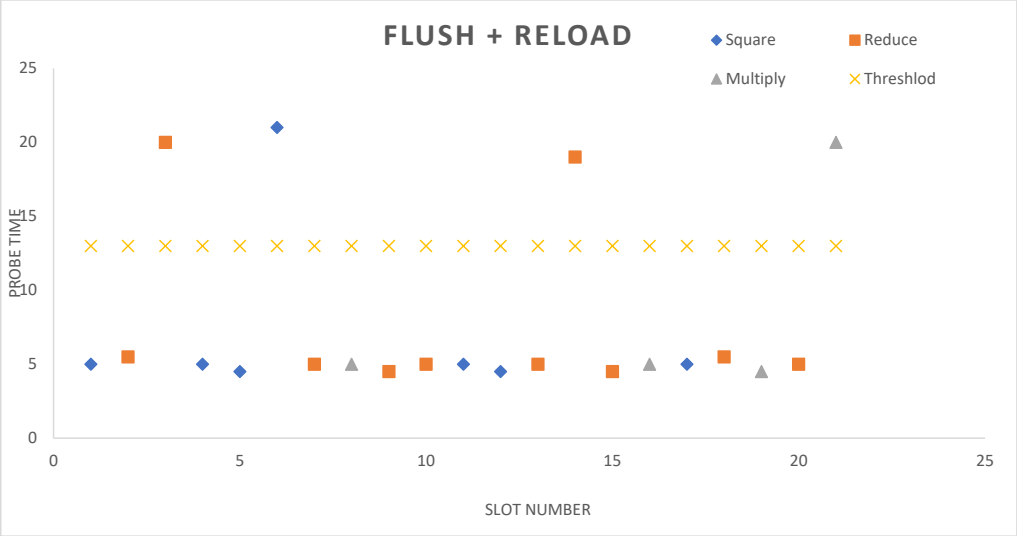



Figure 2.3: Execution time of operation with reference to threshold time

Figure 2.3 shows the sequence of operations that the victim makes along with the cache-hit and cache-miss during the reload phase, which is exploited by the spy application to deduce the sensitive information via the side-channel. As described in Flush + Reload [22], that the threshold value of particular system configuration and running application is a determining factor between the data hit and data miss for the the spy to infer the sequence of operations made by the victim application

For the performed experiment with Gnupg Elgamal key generation as victim and Flush + Reload attack as the spy application, one can observe from Figure 2.3, some operation are executed in less than threshold time,while some of them execute in longer time. All the sequence of operations corresponding to the region above the threshold values are inferred

as cache-miss for the spy i.e., the higher probe time indicates that the victim did not access the flushed data from the memory, eventually resulting in a long reload time for the spy. As seen from the Figure 2.3, operations (Square, Reduce and Multiply) above the threshold are not used to capture the secret key, whereas the operations below the threshold refers to those sequences captured by the spy that were accessed by the victim during the wait phase of the spy. Figure 2.3 also explains the sequence of operations based on which the Flush+Reload spy decides whether the private key bit was a logic '1' or '0'. The operations Square-Reduce-Multiply-Reduce that are below the threshold implying it was a logic '1' bit of the secret key and Square-Reduce not followed by Multiply indicates a logic '0' bit of the secret key.

# Recent Works

In order to secure the hardware systems against cache side channel attacks, various defense techniques have been proposed that use different strategies. They seek to provide built-in defenses against side-channel attacks. They are not limited to specific attacks and rule out many possible unknown attacks. We discuss the most prominent ones here:

Isolation by Cache Partitioning: Two processes that do not share a cache cannot snoop on each others cache activity. One approach is to assign to a sensitive operation its own cache set, and not to let any other programs share that part. As the mapping from to a cache set involves the physical memory address, this can be done by the operating system by organizing physical memory into non-overlapping cache set groups, also called colors, and enforcing an isolation policy.

Access Randomization: To randomize the side channel information, making the attack much harder, even impossible. It 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. It can also use software based Compiler assisted approach to transform applications to randomize its memory access patterns.

In addition to these general defense techniques, there are many recent works in progress to minimize the cache side channel attacks. For example Vladimir Kiriansky proposed DAWG in [26], Dynamically Allocated Way Guard , 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 Intels 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 DAWGs 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 has its limitations and additional techniques are required to block exfiltration channels different from the cache channel.

In [27] Oleksii Oleksenko proposed Varys, a system that protects unmodified programs running in SGX enclaves from cache timing and page table side-channel attacks. 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 todays 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. This approach has certain drawbacks: it requires the application to monitor the SSA value, thus increasing the overhead and it introduces a window of vulnerability.

Stephen Crane in [25] 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 dynamic control-flow diversity and diversifying transformations create binaries with randomized program traces, without requiring hardware assistance.

Chongxi Bao work in [28] shows 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 our cache design, the side-channel leakage is significantly reduced while still achieving performance gains over a conventional 2D system.

Xiaowan Dong in [29] presents indefenses 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 Intels Cache Allocation Technology (CAT) in concert with techniques that prevent physical memory sharing. Apparitions control over privileged hardware state can partition the LLC to defeat cache side-channel attacks. Their defense combines Intels 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.

Additionally, malware detection is an area that has attracted extensive research and commercial interest over the past decade. In general, malware detection techniques are either static (focusing on the structure of a program or system) or dynamic (analyzing the behavior during execution). Detection approaches are also classified as signature-based (looking for signatures of known malware) or anomaly-based (modeling the normal structure/behavior

of programs or systems and detecting deviations from this model). Static approaches including virus and spyware scanners are the first line of defense in malware detection. Originally, these scanners are operated using pattern matching to look for signatures of known malware. However, these approaches can be easily evaded using program obfuscation or simple code transformations that preserve the function of the malware but make it not match the patterns known to the scanner. More advanced detectors based on semantic signatures have been proposed, and significantly improved the performance of static scanners. Static approaches are limited and can be bypassed by sophisticated attackers. In particular, code obfuscation techniques (polymorphic malware), and malware encryption (packing or metamorphic malware) are both sufficient to hide even from these more advanced detectors. Dynamic detection observes the behavior of the program (or the system) as it runs and interacts with the environment. A large number of software malware detectors have been investigated that vary in terms of the monitored events, the normal behavior model, and the detection algorithm. The advantage of dynamic detection is that it is resilient to metamorphic and polymorphic malware; it can even detect previously unknown malware. However, disadvantages include a typically high false positive rate, and the high cost of monitoring during run-time. Moreover, since detection is a one time (or periodic) process, malware can evade detection either probabilistically or by recognizing that it is being observed and acting normally for that period. The software implementation is not an effective solution to detect malware at run-time, due to large latency to compute the complex algorithms.

# Our Work

### 4.0.1 Reverse Engineering of HMD

Considering the worst case scenario, where the victim malware detector (defense) is unknown, we perform a reverse engineering to mimic the functionality of the victim HMD. Thus, as a first step to craft adversarial malware, we perform reverse engineering of the victim's HMD similar to that proposed in [18]. The performed reverse engineering is described in Figure 4.1.

In order to reverse engineer the victim's HMD, we first create a training dataset that comprises of benign and malware applications. Nearly 12,000 benign and 12,000 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 to train different ML classifiers in order to mimic the functionality of the victim's HMD, as shown in Figure 4.1(a). Further, it is tested by comparing the outputs from victim's HMD response and the reverse engineered ML classifier's response, as shown in Figure 4.1(b). Reverse engineering is non-trivial as the adversaries generated on a closely functional model will be highly effective compared to a weakly generated adversary. To ensure the reverse engineering is performed in an efficient way, we train multiple ML classifiers and choose the classifier that yields high performance i.e., mimics the victim's HMD with high accuracy.

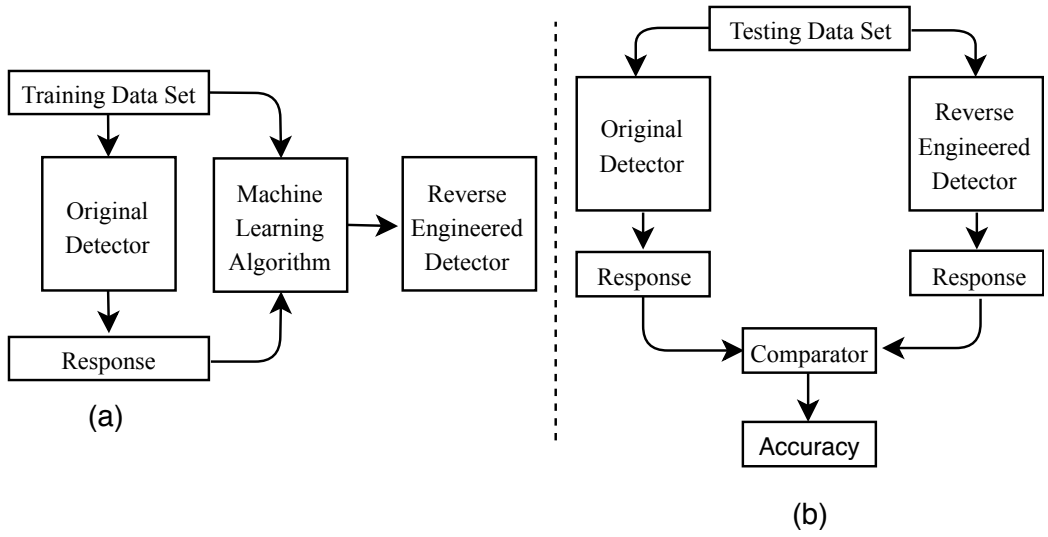



Figure 4.1: (a) Process of reverse engineering HMD; (b) Testing Performance of Reverse-Engineered Detector

### 4.0.2 Adversarial HPC Sample Prediction

Once the reverse engineered HMD is built i.e., neural network's hyper parameters 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 HPC patterns in order 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, one needs to perturb the HPC patterns in more sophisticated manner. To perturb the HPC patterns, we employ a low-complex gradient loss based approach, similar to Fast-Gradient Sign Method (FGSM) which is widely employed in image processing. The advantage of such an approach is its low complexity and low computational overheads. Additionally, it has been observed from our experiments that the HPC samples follow a continuous distribution, and as such a gradient loss based approach is feasible and beneficial to determine the required perturbation in HPC features to be misclassified.

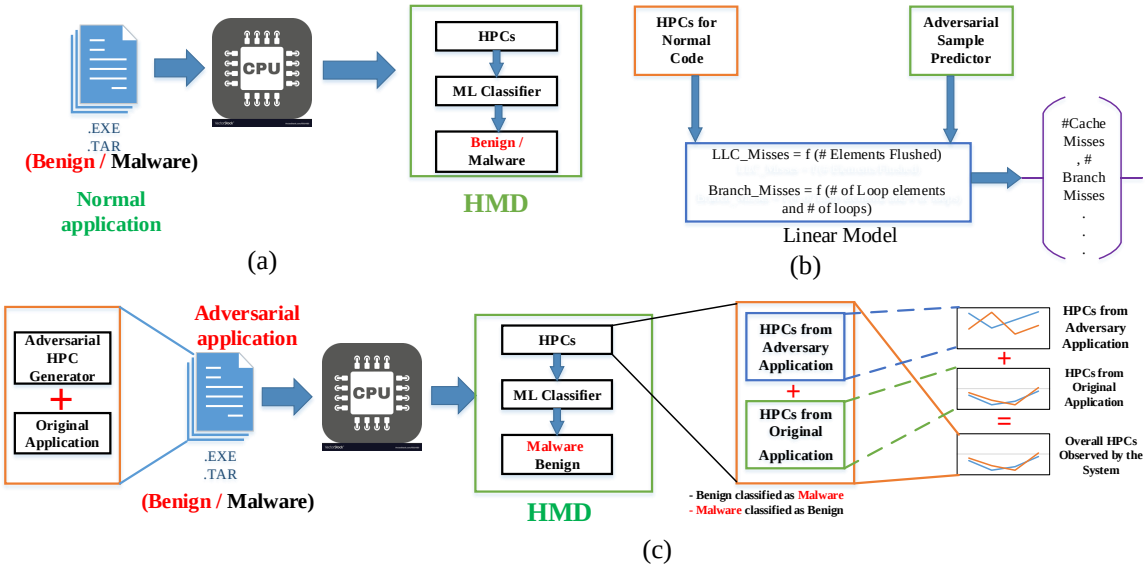



Figure 4.2: (a) Process of utilizing low-level micro-architectural events (HPCs) with ML for malware detection; (b) Determining parameter of adversarial code generator with the aid of adversarial HPC predictor; (c) process of adversarial HPC generator embedded into original application yet spawned as separate thread leading to adversarial output (misclassification)

In order to craft the adversarial perturbations, we consider the reverse engineered ML classifier i.e., neural network with $\theta$ as the hyper parameters, $x$ 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 neural network. Then the perturbation required to misclassify the HPC trace is determined based on the cost function gradient of the neural network (in this case). The adversarial perturbation generated based on the gradient loss, similar to the FGSM [14] is given by

$$x^{adv} = x + \epsilon sign(\nabla_x L(\theta, x, y)) \tag{4.1}$$

where $\epsilon$ is a scaling constant ranging between 0.0 to 1.0 is set to be very small such that the variation in $x$ ($\delta x$) is undetectable. In case of FGSM the input $x$ is perturbed along each dimension in the direction of gradient by a perturbation magnitude of $\epsilon$.Considering a

small $\epsilon$ leads to well-disguised adversarial samples that successfully fool 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 be made sure it can be generated during runtime by the applications. For instance, a HPC of value '$-1$' cannot be generated by an application. Hence, we provided lower bound on the adversary values that can be predicted.

In contrast to works that assume the application features to be binary such as [17], this work aims to predict and determine the adversaries for the low-level microarchitectural event patterns i.e., HPC patterns 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 in the following section.

### 4.0.3 Adversarial HPC Generator

In order to generate the required number of HPCs, we craft an application (benign) that spawns as a separate thread and generates the additional number of HPC events that makes the overall HPC count similar to the predicted HPC count by the adversarial HPC predictor discussed previously.

#### Adversarial HPC Generation

A pseudocode depicting the process of creating adversarial HPC is shown in Algorithm 1. 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 micro-architectural events that malicious applications [2] or even side-channel attacks affect.Hence, we showcase a simple example of perturbing those in Algorithm 1, however, other events can also be perturbed.

**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 in HPCs
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)* {Clflush 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 order to generate LLC load misses, an array of size $n$ is initially loaded from the 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 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 number of LLC load misses. As such, once the adversarial sample predictor predicts the number of LLC load misses to be generated to craft an adversarial sample, the $n$ and $k$ are accordingly determined. The rationale to employ a linear model is its low complexity, yet yielding high accuracy (<3% error) to determine the dependency between $n$ and $k$ for our experiments. It needs

to be noted that as the LLC misses are dependent on the system, and random in nature, hence, we execute the application multiple times (100) with same $n$, and $k$ and average the obtained LLC load misses to alleviate any errors caused.

*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 experiment is performed on Intel Core i7-8700K running Ubuntu 18.4, having GCC 7.3 version. The *Perf* tool available on Linux is utilized to obtain the HPC events. The flushing of the data has been verified by executing the 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 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, as the number of branch instructions depend on the number of conditions to be checked. In the presented pseudo code, we have five conditional statements for generating branch-misses as show in (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 based on the number of loops, as in Line 15-22 of Algorithm 1, the number of branch misses is around 255K. An increase in number of branch misses is observed with the addition of dummy loops that are designed not to satisfy the condition.

All these adversarial sample generators are spawned as separate threads along with the user or attacker's application that needs to be misclassified. In this manner, the adversarial HPC generator does not interfere with the original application's source code, yet is able to mislead the embedded defense mechanism. Figure 4.2(a) shows the HPC trace of a normal application, and the HPC trace predicted by the adversarial sample predictor to misclassify the ML classifier is depicted in Figure 4.2(b). The process of adversarial HPC generation

during runtime is depicted in Figure 4.2(c). If the predicted HPC values are smaller than that generated by original applications, we insert the delay elements to smoothen the HPC trace and reduce the HPC values. It needs to be noted using this process, we generate adversaries to classify benign as malware as well as malware as benign applications.

**Summary**

The proposed adversarial attack on micro-architectural events comprises of three phases. Firstly, we perform reverse engineering to build a ML classifier that mimics the functionality of the victim's HMD or malware detectors. Further, with the aid of adversarial sample predictor, the required number of HPC events required to misclassify the applications is determined. To determine the parameters of adversarial generator application, a linear model relating different features of the application and the HPC events is built. Thus, based on the derived linear model and the required number of adversarial HPCs, the parameters of the adversarial HPC generator application (for instance variables $i$,$k$,$n$ in Line 4 and Line 2 of Algorithm 1) are determined. Lastly, this crafted HPC generator application is spawned as separate thread together with normal (malware or benign) application, leading to overall HPCs generated by the modified application close to those predicted by the adversarial sample predictor, eventually leading to misclassification.

# Entropy Shield

Cache based side-channel attacks generically work on the fundamental principle of observing cache lines of the victim with what they call as probes. To determine if the victim was/is using a particular data from the cache, the attacker/spy inserts probe in to the cache space, which is shared with the victim as is the case for Flush+Reload- and then flushes the data and waits for the victim to execute. If the victim accessed the data, the spy gets a cache hit during the reload phase and cache miss otherwise. The Flush+Reload attack [22] based on the sequences of operations deduces the bits of the secret information. But only a few bits can be captured with few runs of the spy and victim; only by repeatedly executing the attack, all the bits of the secret information can be captured. There are a plethora of works that have been successful in detecting and mitigating SCAs, but, all of them have their respective downsides which makes them difficult to deploy. Hence, we here propose a defense mechanism that protects the victim application by reducing the entropy of the side-channel. We support our claim by giving a detailed analysis of the results obtained. The basic assumption in successfully probing the victim's critical operations is the attacker has knowledge of the addresses of the code sections which carry out sensitive operations. In order to protect sensitive information, we designed Entropy-Shield. The Entropy-Shield will be deployed as a wrapper that encompasses the victim without modifying the victim. The shield has similar knowledge as the attacker where it knows which sections of the victim code need to be monitored and protected.

The spy interprets the key bits based on the sequence of the operations executed by the victim. In the encryption algorithm, the secret key bits are computed by the sequence of

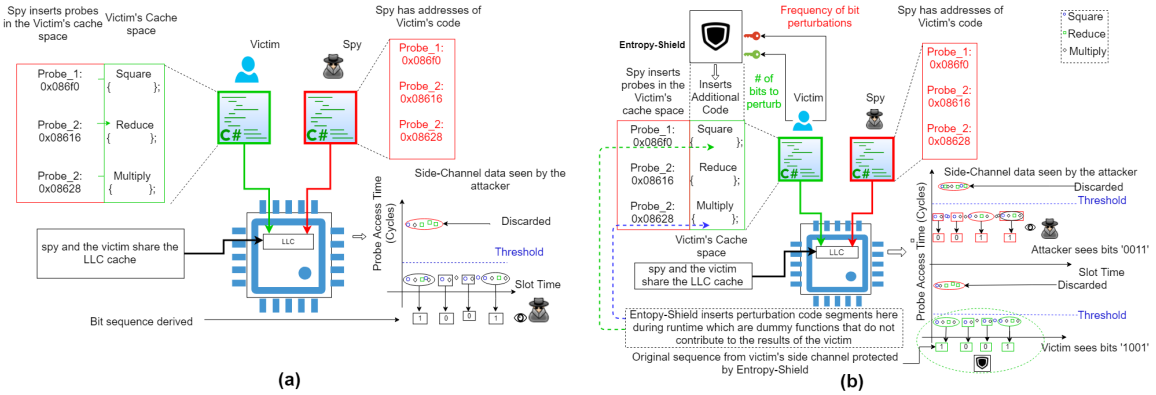



Figure 5.1: (a) Traditional Flush+Reload attack on encryption algorithm where all the data leaked via side-channel is accessible to the attacker; (b) Victim is wrapped with Entropy-Shield that injects perturbation code in the victim during run-time to reduce and perturb the sensitive information leaked thereby making SCAs laborious and time-consuming.

square, multiply and reduce operations respectively. If the victim executes the operation in the sequence: square reduce and multiply reduce, then the key computes to be '1', and if the sequence is square reduce and not followed by multiply reduce, the key comes out to be '0'. The spy insert probes in the functions of the victim code respectively, to obtain the sequence of operation. We created a wrapper code around the victim code in such a way that it does not alter the secret key computation operation on the victim side but tricks the spy in showing how the key was calculated(as shown in the psuedocode).

In the Algorithm 2 (representing standard Flush + Reload Attack) psuedocode, the functions are well defined and based on the sequence of their execution the probes inside the functions get triggered. The attacker, while the victim executes the operations, flushes the address(every probe) from the cache memory, waits for the victim to execute and the reloads the address. The attacker calculates the time victim takes to reload the data and then compares it with the threshold time($th$). If the reloading time t is less than '$th$' (Cache Hit), that implies that data was earlier used by the victim while the spy was waiting for it to execute, and if t is greater '$th$' (Cache Miss), that means while reloading the data

**Algorithm 2** Pseudocode illustrating generation and stealing of private key(Original Flush + Reload)

**Require:** Private Encryption Key
**Ensure:** Decoded Original Encryption Key

**Victim Program()** {Victim pseudo code that generates encryption secret key}
Square func()
    { - - - - - - - - - - -
      **Probe1**
    - - - - - - - - - }
Multiply func()
    { - - - - - - - - - - -
      **Probe2**
    - - - - - - - - - }
Reduce func()
    { - - - - - - - - - - -
      **Probe3**
    - - - - - - - - - }
**Attacker Program()** {Sample pseudo code that decodes the secret key}
    *Loop 1:* load i ¡ n
    clflush**(Probe 1)**
    clflush**(Probe 2)**
    clflush**(Probe 3)**
    `Reloading time(t)`
    `jump Loop1`
end
  `cmp` t # threshold time(th)
  if( t more than th ) = Cache miss
  if( t less than th ) = Cache hit
Based on sequence of Cache hit operation, Secret Key is Deduced

was called from main memory, and was not in the cache. Performing such flush and reload operation multiple times, the spy is able to deduce which address were used by the victim to calculate the key.

In the algorithm 3, most of operation remains the same with a minor but significant change, that leads to incorrect key value interpretation by spy. Our wrapper creates a dummy function inside an already existing standard function, for example a dummy square function inside a multiply function. From the victim end, this dummy function calls and executes the square function but does not include its output while calculating the key, whereas the spy while tracing the sequence of operations, gets tricked into believing that the square

**Algorithm 3** Pseudocode illustrating generation and stealing of private key

**Require:** Private Encryption Key
**Ensure:** Decoded Incorrect Encryption Key

**Victim Program()** {Victim pseudo code that steals incorrect secret key}
Square func()
    { - - - - - - - - - - -
      **Probe 1**
    - - - - - - - - - }
Multiply func()
    { - - - - - - - - - - -
      **Probe 2**
    - - **Dummy call to Square func()** - -
    - - - - - - - - - }
Reduce func()
    { - - - - - - - - - - -
      **Probe 3**
    - - - - - - - - - }
**Attacker Program()** {Sample pseudo code that decodes the secret key}
    *Loop 1:* load i ¡ n
    clflush **(Probe 1)**
    clflush **(Probe 2)**
    clflush **(Probe 1)**
    `Reloading time(t)`
    `jump Loop1`
end
  `cmp` t # threshold time(th)
  if( t more than th) = Cache miss
  if( t less than th) = Cache hit
  Based on **perturbed sequence** of Cache hit operation, **Incorrect Secret Key** is Deduced

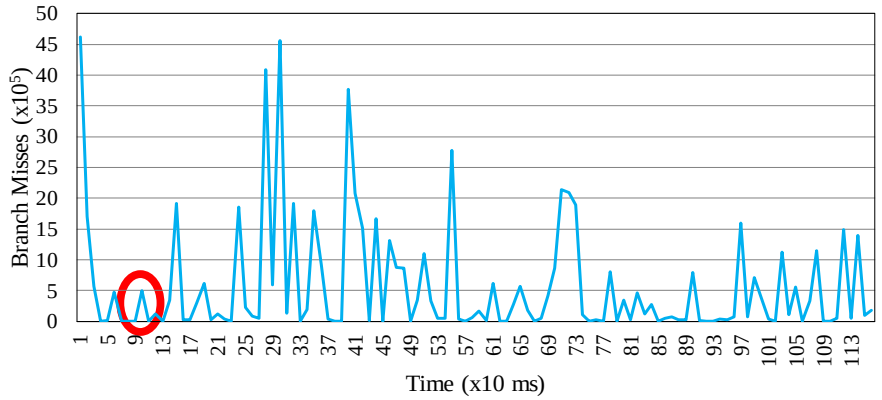
function was actually expected and therefore interprets the key incorrectly.

The two further advancements that can be made in this procedure is controlling the frequency and location of perturbation induced. For example how many times can we call the dummy functions. Based on the frequency of the dummy function being called, the key interpretation by spy changes significantly. The other important point to keep in the mind is that where do we want to place the dummy loop, particularly in which function. Depending on its placement, whether in one function or in multiple, the key interpretation by spy is affected

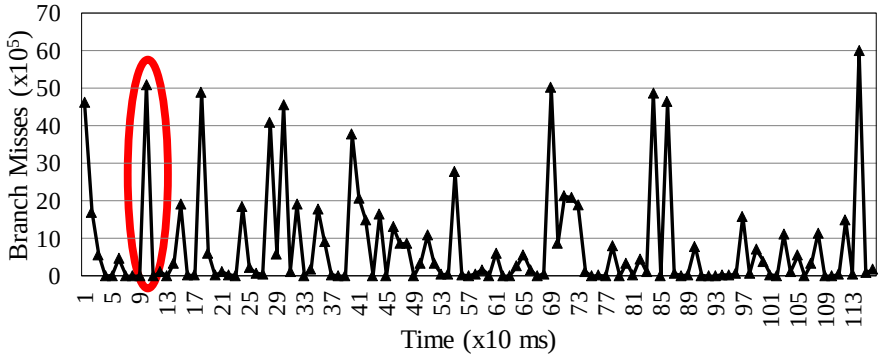
# Evaluation and Results

### 6.0.1 Experimental Setup and Data Collection

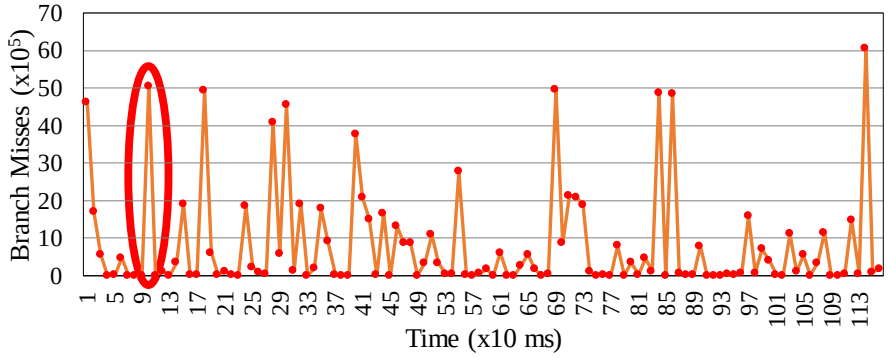
This section provides the details of the experimental setup and data collection process. The applications (both malware and benign) are executed on an Intel Xeon X5550 machine running Ubuntu 14.04 with Linux 4.4 Kernel. In order to extract the HPC information, we used *Perf* tool available under Linux. *Perf* provides rich generalized abstractions over hardware specific capabilities. It exploits *perf-event-open* function call in the background which can measure multiple events simultaneously. We executed more than 3000 benign and malware applications for HPC data collection. Benign applications include MiBench benchmark suite [30], Linux system programs, browsers, text editors, and word processor. For malware applications, Linux malware is collected from virustotal.com [31] and virusshare.com [32]. Malware applications include five classes of malware comprising 607 Backdoor, 532 Rootkit, 2739 Virus, 1264 Worm and 7221 Trojan samples. 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 as a shell script that facilitates to execute the user/attacker's application in parallel. The hyper parameters of the neural network mimicing the victim's HMD or security defense and the parameters used for adversarial sample predictor are outlined in Table 6.1.




(a) HPC trace of branch misses from a normal program




(b) HPC trace predicted by adversarial sample generator




(c) HPC trace from crafted adversarial code

Figure 6.1: (a)LLC load miss HPC trace of an application; (b) LLC load miss HPC trace of the application predicted by adversarial sample predictor; and (c) LLC load miss HPC trace of the application predicted by adversarial sample generator

Table 6.1: Architectural details of HMD

| Parameters of ML classifier in HMD | | | |
|---|---|---|---|
| Input | 16 features | Optimization | ADAM |
| # hidden layers | 1 | Batch size | 128 |
| Hidden layer 1 (ReLu) | 250 neurons | Epochs | 100 |
| Dropout | 0.2 | Learning rate | 0.001 |
| Adversarial Sample Predictor Parameters | | | |
| Attack type | | FGSM | |
| Adversarial perturbation | | 0.3 | |

### 6.0.2 Impact of Adversarial Attack on HPCs

We depict the impact of adversarial sample generator (application) on the generated HPC events in Figure 6.1 and 6.2. Figure 6.1 shows the LLC load misses of a benign application (notepad++). The Figure 6.1(a) shows the LLC load misses in normal case. For this HPC pattern, the adversarial HPC pattern predicted by the adversarial sample predictor (implemented in Python) is shown in Figure 6.1(b). One can observe that there exist some spikes in the pattern compared to the normal HPC pattern, as marked by circle. Figure 6.1(c) shows the HPC pattern generated when the application is integrated with the adversarial HPC generator. On an average, there is an error of 2.23% between the trace predicted by the adversarial sample predictor and the trace generated by the adversarial sample generator.

In a similar manner, we depict the branch misses in Figure 6.2. Figure 6.2(a) shows the HPC pattern of branch misses for a normal application (notepad++). The adversarial pattern predicted and generated by adversarial sample generator for branch misses is shown in Figure 6.2(a), and 6.2(b) respectively. One can observe that pattern predicted by the adversarial sample predictor and generator are similar. An average error of 2.15% is observed for branch misses, and 0.91% for branch instructions. A 2.23% error is observed for

branch miss instruction. This indicates that adversarial generator can efficiently generate the required number of HPCs without being detected by the malware detectors.

The neural network based HMD achieves an accuracy of 82.76% with normal samples. However, when the applications are integrated with the proposed adversarial sample generator application, the accuracy reduces to 18.04%. Similarly, a drastic reduction in precision, F1-score and recall are observed with the proposed attack on different applications. This is outlined in Table 6.2.

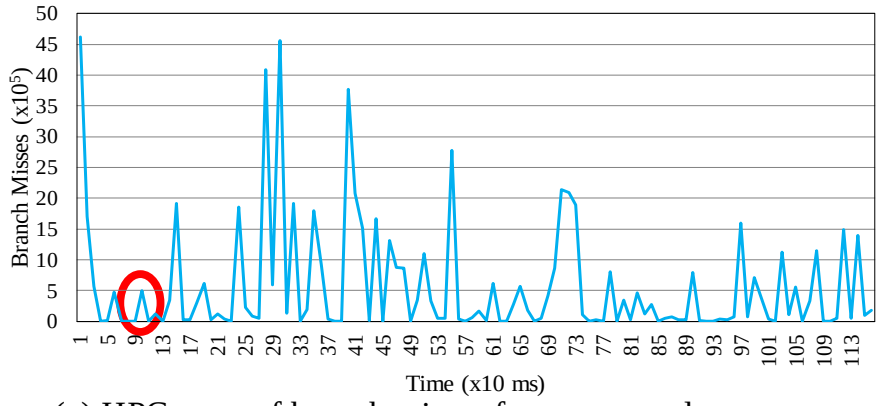
Table 6.2: Impact of adversarial attack on HMD

| | Accuracy | Precision | F1-score | Recall |
|---|---|---|---|---|
| Before | 82.7% | 80.0% | 80.0% | 83.0% |
| After | 18.3% | 45.0% | 10.0% | 18.0% |

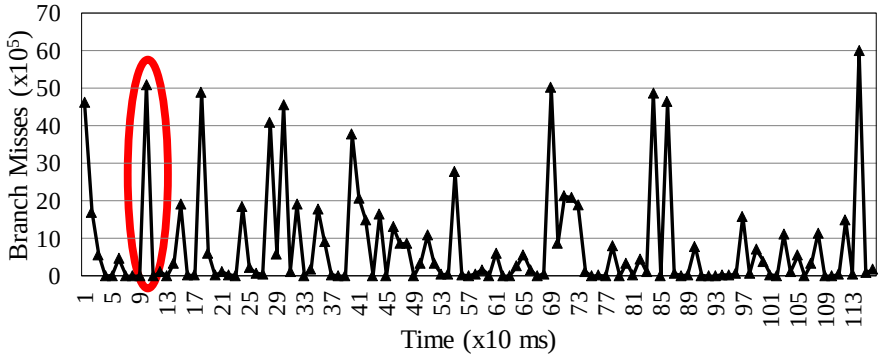
### 6.0.3 Transferability Analysis

Though the reverse engineering results in building ML classifier that mimics the victim's HMD, they might not be same. For instance, victim's HMD might be using a logistic regression (LR) and the reverse engineered solution is a neural network. To showcase the robustness of proposed adversarial malware crafting, we perform a transferability analysis. As stated in [18], LR and neural network achieves good performance. Hence, we perform the transferability analysis of the generated adversarial malware on the LR based HMD.

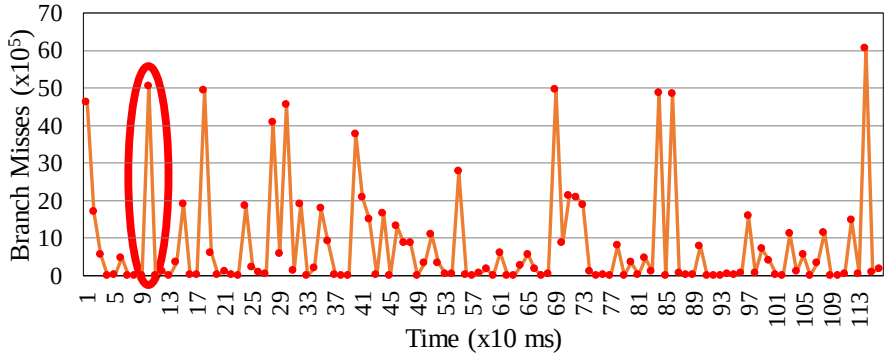
Thus, adversarial malware generated is applied to a HMD using logistic regression, whose functionality is mimicked through reverse engineering. The results show that the malware detection accuracy falls to 5.10% with prevision, F1-score, and recall to 16.0%, 7.0% and 5.0% respectively with the adversarial malware. This indicates that ML classifier used to craft adversarial malware is transferable to other systems until we can mimic the victim's malware detector functionality.




(a) HPC trace of branch misses from a normal program




(b) HPC trace predicted by adversarial sample generator




(c) HPC trace from crafted adversarial code

Figure 6.2: (a) Branch miss HPC trace of an application; (b) Branch miss HPC trace of the application predicted by adversarial sample predictor; and (c) Branch miss HPC trace of the application predicted by adversarial sample generator

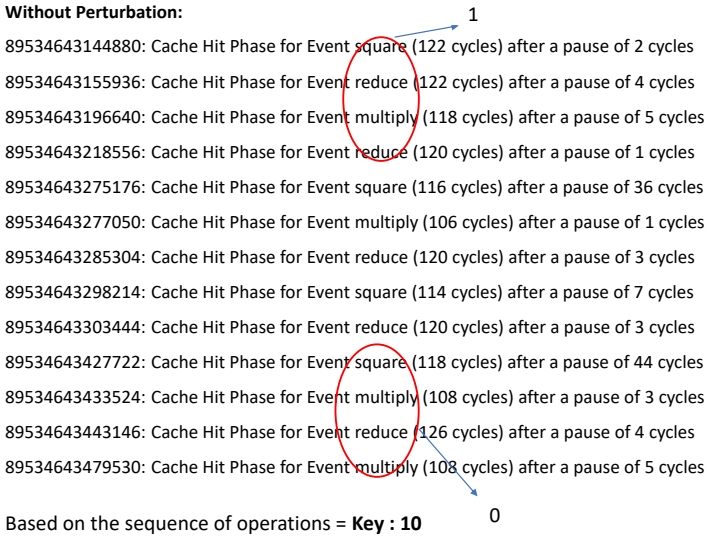



Figure 6.3: Key interpretation without perturbation

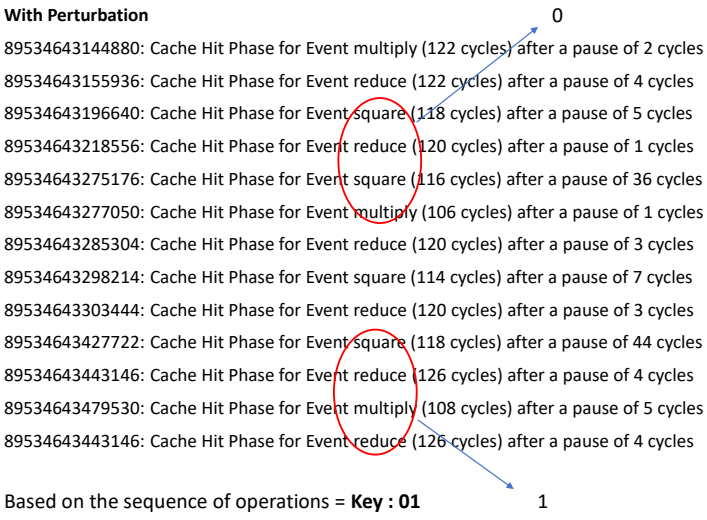



Figure 6.4: Key interpretation with perturbation

### 6.0.4 Perturbation Analysis

For the perturbation, we executed the applications (spy and victim) on Intel core i7 8th Gen processor, Ubuntu 14.04 with Linux 4.4 Kernel. Using the wrapper technique as discussed above we found that its possible to fool the spy in decoding the key by misleading it while interpreting the sequence of operations as shown in Figure 6.3 and Figure 6.4. We did the experiment using Flush + Reload cache side channel attack as a spy and different combinations of Gnupg generated public and private keys as victim as shown in Table 6.3. During the normal execution i.e without any perturbation in the code, almost 95 percent of the key were interpreted by the spy correctly as observed by the original Flush + Reload authors. With the wrapper added around the victim code, the key interpreted by the spy comes out to be incorrect (based on the incorrect sequence of the operation interpreted by the spy). Table 6.3 also represents the difference in key observation (from spy side) with and without adding the perturbation to the victim. The modification shown is just for the 32-bit key print of the actual 1024 bit or 4096 bit key.

Table 6.3: Key Perturbation Table

| Flush+Reload Attack | Without Perturbation | With Perturbation |
|---|---|---|
| RSA and RSA | F9D2EDC5 | F1D2ADC7 |
| DSA and Elgamal | 3DA77005 | 3FA7710D |

# Conclusion

In this work, we propose an adversarial attack on micro-architectural event based malware detection systems i.e., HMD systems. These HMD systems utilize the underlying hardware performance counters to capture the micro-architectural events and provide them to ML classifier for detecting and classifying malware. This work employs an adversarial sample predictor to determine the HPC count required to get misclassified. Post determining the required number of HPC count, using the proposed adversarial sample generator the required number of additional HPC count is generated without intervening with the original application and eventually leading to misclassification. An error of $< 3\%$ in predicted and generated HPC events to create adversary is observed. Furthermore, the malware detection accuracy is reduced from 82.7% to 18.04%. We also evaluated how timing based side channel attacks can be minimized by reducing the entropy using a wrapper code around victim. For the future work, we will be working on changing the perturbation in a much more controlled way, that can lead to further wrong key interpretation by the spy, thus providing enhanced security against side channel attacks.

# Bibliography


[1] 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.

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

[3] 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.

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

[5] X. Wang and et al., "ConFirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *IEEE/ACM International Conference on Computer-Aided Design*, 2015.

[6] 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.

[7] ——, "Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning," in *ACM Computing Frontiers*, 2018.

[8] F. Brasser and et al., "Advances and throwbacks in hardware-assisted security: Special session," in *Int. Conf. on CASES*, 2018.

[9] S. Dinakarrao and et al., "Lightweight node-level malware detection and network-level malware confinement in iot networks," in *Design Automation and Test Con. in Europe*, 2019.

[10] H. Sayadi and et al., "2SMaRT: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *Design Automation and Test Con. in Europe*, 2019.

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

[12] K. Khasawneh and et al., "EnsembleHMD: Accurate hardware malware detectors with specialized ensemble classifiers," *IEEE Trans. on Dependable and Secure Computing*, 2018.

[13] C. Szegedy and et al., "Intriguing properties of neural networks," in *Int. Conf. on Learning Representations*, 2014.

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

[15] N. Papernot and et al., "The limitations of deep learning in adversarial settings," in *IEEE European Symp. on Security and Privacy*, 2016.

[16] Y. Liu, X. Chen, C. Liu, and D. Song, "Delving into transferable adversarial examples and black-box attacks," in *Int. Conf. on Learning Representations*, 2017.

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

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

[19] B. Zhou and et al., "Hardware performance counters can detect malware: Myth or fact?" ser. ACM Asia Conf. on Computer and Communications Security, 2018.

[20] 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.

[21] S. B. A. D. H. S. A. S. H. H. Sai Manoj Pudukotai Dinakarrao, Sairaj Amberkar and S. Rafatirad, "Adversarial attack on microarchitectural events based malwaredetectors," in *Design Automation Conference*, 2019.

[22] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: https://doi.org/10.1109/SP.2015.43

[24] S. Bleikertz, C. Vogel, and T. Groß, "Cloud radar: Near real-time detection of security failures in dynamic virtualized infrastructures," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664274

[25] M. B. Bahador, M. Abadi, and A. Tajoddin, "HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *Int. Conf. on Computer and Knowledge Engineering*, 2014.

[26] V. Kiriansky, H. Xu, M. Rinard, and S. P. Amarasinghe, "Cimple: instruction and memory level parallelism: a DSL for uncovering ILP and MLP," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques,*

*PACT 2018, Limassol, Cyprus, November 01-04, 2018*, 2018, pp. 30:1–30:16. [Online]. Available: https://doi.org/10.1145/3243176.3243185

[27] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 227–240. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oleksenko

[28] Y. Liu, Y. Xie, C. Bao, and A. Srivastava, "A combined optimization-theoretic and side-channel approach for attacking strong physical unclonable functions," *IEEE Trans. VLSI Syst.*, vol. 26, no. 1, pp. 73–81, 2018.

[29] C. W. Fletcher, R. Harding, O. Khan, and S. Devadas, "A low-overhead dynamic optimization framework for multicores," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 467–468. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370899

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

[31] (2019) Virustotal intelligence service. Last accessed: 04-May-2019. [Online]. Available: www.virustotal.com/intelligence

[32] (2019) Virusshare team. Last accessed: 04-May-2019. [Online]. Available: www.virusshare.com

# Curriculum Vitae

SAHIL BHAT is a computer engineering masters student at George Mason University in Electrical andComputer Engineering department. He graduated from Pune University in 2016. His research interest are in hardware security and side channel attacks.