Micro-architectural Threats to Modern Computing Systems

by

Mehmet Sinan İnci

A Dissertation

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in

Electrical and Computer Engineering
by

April 2019

APPROVED:

Associate Professor William Robertson Dissertation Committee Northeastern University	Assistant Professor Robert J. Walls Dissertation Committee CS Department
Professor Thomas Eisenbarth Dissertation Committee ECE Department	Professor Berk Sunar Dissertation Advisor ECE Department

Professor Ludwig Reinhold
Department Head
ECE Department

Abstract

With the abundance of cheap computing power and high-speed internet, cloud and mobile computing replaced traditional computers. As computing models evolved, newer CPUs were fitted with additional cores and larger caches to accommodate run multiple processes concurrently. In direct relation to these changes, shared hardware resources emerged and became a source of side-channel leakage. Although side-channel attacks have been known for a long time, these changes made them practical on shared hardware systems. In addition to side-channels, concurrent execution also opened the door to practical quality of service attacks (QoS).

The goal of this dissertation is to identify side-channel leakages and architectural bottlenecks on modern computing systems and introduce exploits. To that end, we introduce side-channel attacks on cloud systems to recover sensitive information such as code execution, software identity as well as cryptographic secrets. Moreover, we introduce a hard to detect QoS attack that can cause over 90+% slowdown. We demonstrate our attack by designing an Android app that causes degradation via memory bus locking.

While practical and quite powerful, mounting side-channel attacks is akin to listening on a private conversation in a crowded train station. Significant manual labor is required to de-noise and synchronizes the leakage trace and extract features. With this motivation, we apply machine learning (ML) to automate and scale the data analysis. We show that classical machine learning methods, as well as more complicated convolutional neural networks (CNN), can be trained to extract useful information from side-channel leakage trace.

Finally, we propose the DeepCloak framework as a countermeasure against side-channel attacks. We argue that by exploiting adversarial learning (AL), an inherent weakness of ML, as a defensive tool against side-channel attacks, we can cloak side-channel trace of a process. With DeepCloak, we show that it is possible to trick highly accurate (99+% accuracy) CNN classifiers. Moreover, we investigate defenses against AL to determine if an attacker can protect itself from DeepCloak by applying adversarial re-training and defensive distillation. We show that even in the presence of an intelligent adversary that employs such techniques, DeepCloak still succeeds.

Acknowledgments

The research that resulted in this dissertation is funded by the National Science Foundation grants CNS-1618837, CNS-1318919, and CNS-1314770. I would like to thank the National Science Foundation for their support.

I would like to extend my sincere gratitude to Professor Berk Sunar for his continuous support, guidance, and wisdom that made this dissertation possible. Through many setbacks and challenges during my Ph.D., his trust and the freedom he gave, enabled me to find a solution and make it work. His foresight, intelligence, and work discipline have resonated with me throughout my Ph.D. and will continue to do so in the future. He is truly an inspiration to find interesting research challenges of the future and tackle them.

I would like to thank my co-advisor Professor Thomas Eisenbarth for his support and guidance throughout my research. His unique and original perspective and attention to detail was an invaluable asset for me. It gave me a more wholesome view of things and made me a better researcher.

I would like to thank my dissertation committee members Professor Berk Sunar, Professor Thomas Eisenbarth, Professor Robert J. Walls, and Professor William Robertson. I am grateful for their guidance, feedback and their invaluable time spent for the preparation of this dissertation.

I am grateful for working with the amazing people that make up the Vernam Lab. In the past years, we have shared a lot and bonded as brothers (and sister!) in arms. All of us spent more time together than we did with our close friends and family outside of academia. In not any particular order, I would like to thank my dear friends Gorka Irazoki, Berk Gulmezoglu, Yarkin Doroz, Gizem Cetin, Michael Moukarzel, Cong Chen, Dai Wei, Marc Green, Saad Islam, Koksal Mus, Okan Seker, Abraham Fernandez and Aria Shahverdi for their friendship and support.

I would like to thank my parents and for their continuous support through my education, starting from the first grade up until the end of the Ph.D. My father always stressed the importance of a good education and imprinted that on me. I would also like to thank my uncle Ibrahim and my grandfather for their support, belief in me and wishes to see me as a doctor one day. They both passed away during my Ph.D., may they rest in peace.

Last but not least, I would like to thank my wife, Gulseren Gizem for being the incredible person that she is. Her support throughout the grad school and putting up with me -which is not an easy task- made it all possible. She is a caring wife, my friend, my other half. On top of that, I am eternally grateful to her for shouldering the raising our son Yunus Emre with a part-time father, and away from grandparents or any other close family to help her out. Not to mention the cold, harsh climate of the city of Worcester. This doctorate would not possible without her many sacrifices. As this Ph.D. journey reaches its conclusion, I intend to make up for the lost time and create infinite happy memories, and set sail to new adventures with her, as a family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1965, Gordon Moore famously predicted that the number transistors in integrated circuits would double every two years. For the most part, processors kept up with the Moore's Law and reached incredible transistor densities. With more transistors, came the opportunity for hardware designers to implement increasingly complex optimizations that would be otherwise impossible. Single-core CPUs evolved into 64-core, multi-threaded behemoths. Functionality that was previously maintained by separate chips got integrated into CPUs. Memory architecture, size and bandwidth radically changed, always towards faster and more efficient computing. While these improvements certainly increased compute capabilities, they were designed and implemented with the mindset and assumption that a hardware system would be used only by trusted parties. This assumption was the conclusion of thinking that if a potential attacker has access to run code on the machine, the system was already compromised. While certain isolation techniques were implemented, like the address space layout randomization (ASLR), to protect systems from untrusted code, these protections were aimed at software based threats and not necessarily affected side-channel attacks. Moreover, optimizations like the addition of multi-

ple levels of memory into the CPU, called caches added shared resources between threads, cores and processes. The purpose of these caches is to use temporal and spatial locality as a means to keep frequently used data in the CPU and improve performance. However, as we demonstrate in this dissertation, these optimizations towards performance open new attack surfaces to modern computing systems.

As computing systems became simultaneously more efficient, more capable and cheaper, computing paradigm shifted into mobile computing via smartphones and shared systems via cloud. This shift increased productivity while allowing software from multiple sources running side-by-side on a shared hardware. While side-channel attacks have been demonstrated to recover secret information from chips in the past, these attacks required the physical access to work and were considered out-of-scope of security evaluation of shared hardware resources. For instance, on cloud, there was always an implied trust that an attacker cannot have physical access to the system therefore cannot mount a side-channel attack.

In this dissertation, we show how to exploit shared hardware resources in cloud and mobile environments to perform quality of service attacks, privacy violations as well as cryptographic key recovery attacks. We show that micro-architectural threats including software side-channels leave modern computing systems vulnerable when the underlying hardware is shared.

To demonstrate the severity of the threat, we implement attacks such as Flush+Reload, Prime+Probe and Memory Bus locking. Further, we show all the necessary steps to perform such attacks on cloud and mobile environment. We show that it is possible to recover information from co-located virtual machines running on the same physical system, even though they are isolated by virtualization, ASLR and other isolation techniques.

Note that in order to perform these side-channel attacks on cloud environment,

the first challenge is to achieve co-location with a vulnerable target. While there are studies investigating cloud co-location, previously known methods are now outdated. Side-channel leaks signaling the co-location have been eliminated and new methods are required. For this purpose, we have developed 3 novel cloud co-location techniques that exploit micro-architectural features to detect and verify co-location under various scenarios.

After establishing that software side-channels pose a significant threat for the modern computing systems and demonstrating such attacks on the cloud, we tackle the problems of automation and scalability of such attacks. While practical and quite powerful, mounting side-channel attacks is still akin to listening on a private conversation in a crowded train station. The attacker needs to perform significant manual labor to extract important features, de-noise and synchronize the leakage trace. Here, we argue that there is a better alternative, the use of Machine Learning (ML) systems to automate and scale the attack process and data analysis. With the abundance of cheap computing power and the improvements made in implementation of ML algorithms, such automation is quite advantageous. We demonstrate that classical machine learning methods as well as more complicated convolutional neural networks (CNN) can be trained to extract useful information from side-channel leakage trace.

The use of ML, on the other hand, brings an inherent weakness, vulnerability to adversarial learning (AL) methods. Here, in contrast to the previous literature, we use this inherent weakness, as a defensive tool to cloak side-channel leakage. We argue that by running a sister process alongside the original, it is possible to cloak its side-channel trace from prying eyes. We demonstrate the viability of this approach by first training highly accurate (99+% accuracy) CNN and other ML models on side-channel leakage traces. We then test these classifiers against various

AL methods and show that we can cause misclassification very efficiently by carefully crafted adversarial samples.

Finally, we investigate defenses against AL to determine if an attacker can protect itself from DeepCloak by applying adversarial re-training and defensive distillation. We conclude that even in the presence of an intelligent adversary who employs such techniques, AL methods still manage to fool the attacker's model and can be used to cloak processes.

1.1 Contributions

This dissertation demonstrates the viability of software side-channel attacks and other micro-architectural threats to modern computing systems. In order to reliably show the underlying threat, we tackled multiple challenges ranging from identifying side-channel vulnerable software to finding bottlenecks in CPU pipelines.

Contributions can be summarized as follows;

- We show how to implement the Flush+Reload in native and virtualized systems to extract cryptographic keys, and recover information about the code execution. We show that it is in fact possible and practical to detect, in real-time, what software, library or a specific function is running on other virtual machines running on the same platform and how this information can be exploited.

- We implement Prime+Probe attack in native and virtualized systems to extract cryptographic keys across virtualization boundaries. In addition to that, we show that Prime+Probe on the CPU last level cache allows both covert channel communication and side-channel data extraction.

- We solve the co-location detection problem on the cloud by proposing 3 novel methods that rely on underlying shared hardware to reliably verify co-location with specific targets as well as recover target IP address. We demonstrate the viability of these methods by successfully implementing them on Amazon Web Services, Google Compute Engine and Microsoft Azure IaaS clouds.

- We introduce QoS degradation attacks using micro-architectural features and demonstrate them on both cloud and mobile environments. We design and implement an Android app named Degrader to mount such an attack on mobile systems. We show that through a background service, a malicious app can cause performance degradation up to 90% on the app running on the foreground. Moreover, we show that state-of-the-art malware detection tools including Google Play Store's bouncer fail to detect Degrader as malicious.

- We tackle the automation and scaling problems of side-channel attacks and propose machine learning as a solution. We show that machine learning and deep learning classifiers can be used to analyze side-channel leakage and extract meaningful information with high accuracy from high-dimensional side-channel leakage traces. Our classifiers achieve accuracy of over 99+% for the task of software detection from the system Hardware Performance Counter (HPC) trace.

- We introduce the DeepCloak framework to mask the side-channel leakage of security sensitive processes and protect against side-channel attacks. We take advantage of the inherent weakness of machine learning systems to adversarial samples and use it as a defensive tool against side-channel classifiers. We show that by profiling a sensitive process' HPC trace, we can craft an adversarial perturbation that will change the overall system HPC trace and cause

attacker's classifier to misclassify. Moreover, we show that these perturbations are lightweight and cause minimal overhead to the overall system. Finally, we investigate potential methods that an attacker can use to harden the classifier and show that even in the presence of adversarial re-training and defensive distillation, DeepCloak defense succeeds.

1.1.1 The publications resulted in this dissertation

The research described in this dissertation is the result and combination of the following peer-reviewed publications on various conferences and journals [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. The work presented in Chapter 4 are the result of collaborative work with Gorka Irazoqui and resulted in the publication [3]. The work presented in Chapter 5 is the result of collaborative work with Berk Gulmezoglu and resulted in the publication [6]. Finally, the publications [5, 9] are the result of the collaborative work with Berk Gulmezoglu and Gorka Irazoqui.

Chapter 2

Background

This dissertation implements and explains various attacks and defenses that use micro-architectural features of modern computing systems. In order to aid the reader understand the work, we provide the necessary background information on various subject such as the computer architecture, memory de-duplication, mobile computing, atomic operations, machine learning and adversarial learning.

2.1 Computer Architecture

This chapter provides background information on computer architecture, the memory hierarchy, CPU caches, and hardware performance counters (HPC).

2.1.1 Memory Hierarchy

Modern computing systems consist of volatile CPU caches, Dynamic Random Access Memory (DRAM) and non-volatile storage. As a program runs, its memory contents are first loaded from the persistent storage drive into the DRAM. After that, as the program executes, its code and data is loaded from the storage (slowest) to

DRAM and then into the CPU caches and registers (fastest). Also, since the faster memory elements are more costly than slower ones, their capacity is much smaller in comparison.

2.1.1.1 CPU Cache

The CPU cache is a small memory element that is located between the main memory and the CPU cores with the purpose of providing faster access to frequently used memory. Modern CPUs have multiple levels of low-latency Static Random Access Memory (SRAM) caches placed within the same silicon die as the CPUs. These caches, also referred to as L1, L2 and L3 cache, provide high-speed access to a small portion of an executable' s memory. Upon a memory request, if the data resides in the cache, a cache hit occurs and the data is loaded into the CPU registers rapidly. If the memory line is not found in the first level of cache, the L1 cache, a cache miss occurs and the data has to be fetched from lower level caches or the main memory.

Caches base their functionality on two main principles: the *temporal* and *spatial* locality. Former predicts that recently accessed data is likely to be accessed again, whereas the latter predicts that data in nearby memory locations to the accessed data are also likely to be requested soon. Thus, when a cache miss occurs, the memory controller fetches an entire memory block (cache line) containing both accessed and nearby memory locations. For instance, if a 32-bit unsigned integer is requested from memory, the whole cache line of 64 Bytes is loaded into the cache in participation of spatial locality.

2.1.2 Hardware Performance Counters

Hardware Performance Counters (HPCs), or Hardware Performance Events, are special purpose registers that provide low-level execution metrics directly from the

CPU. This low-level information is particularly useful during software development to detect and mitigate performance bottlenecks before deployment. For instance, the low number of cache hits and the high number of memory accesses can hint to an improperly ordered loop. By re-ordering certain operations, a developer can significantly improve the performance of the program. While there are many different HPCs, the availability of a specific counter depends on the CPU model. Moreover, the number of HPCs that can be monitored simultaneously depends both on the CPU model and the selected HPCs. Since some HPCs are derived from others, their use puts additional limitations to the monitoring process.

In addition to performance optimization, HPCs are also proven to be useful at providing system health check and anomaly detection, including malware such as ransomware and crypto-mining.

2.2 Memory De-duplication

Memory de-duplication is an optimization technique that is implemented in operating systems and virtual machine managers with the goal of utilizing the system memory more efficiently. The basic idea is to merge identical pages used by different processes or virtual machines into one shared page with a Copy-on-Write flag. After the merging, if one of the processes wants to modify this shared page, a duplicate page is created and assigned to this process and separating it from the shared page. The de-duplication is performed by first finding memory pages with matching hashes and then comparing these pages for an exact match. This is especially effective in virtualized environments where multiple guest OSs are co-located on the same physical machine and share hardware resources. Many variations of de-duplication techniques are now implemented in OSs and hypervisors e.g. Ker-

nel Samepage Merging (KSM) in Linux and Transparent Page Sharing (TPS) in VMware etc. Although these implementations have slight differences, the underlying principle is the same; merging duplicate pages in memory to save memory space.

Even though de-duplication is a great memory optimization technique, it opens a source of side-channel leakage between processes. This leakage can be exploited using the Flush+Reload method to extract sensitive information. While the memory contents cannot be modified by this attack, the knowledge of the memory lines that have been accessed can be exploited to gain access to secret information like cryptographic keys. An adversary can monitor cache and memory accesses to enable the recovery of such information. For this reason, albeit incredible memory savings it provides, the memory de-duplication is disabled by all major public cloud service providers e.g. Amazon EC2, Microsoft Azure, Google Compute Engine etc., and only used in private clouds.

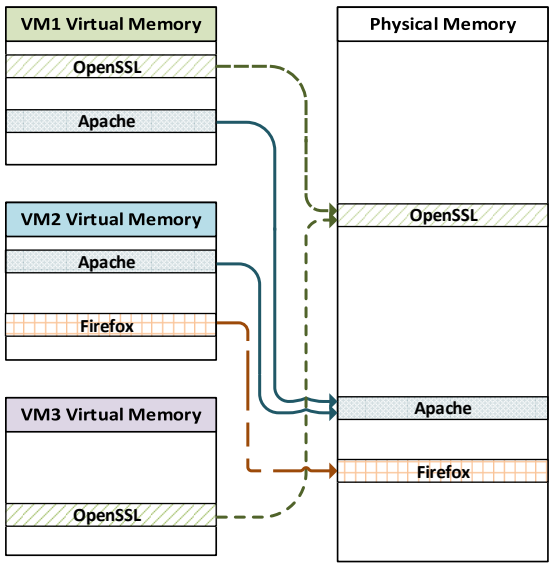


Figure 2.1: Memory De-duplication Feature

2.2.1 Kernel Same-page Merging (KSM)

KSM is a memory optimization feature that was first introduced in Linux kernel version 2.6.32 [14] aiming at removing redundant copies of pages in system memory [16]. KSM works as follows; when the KSM agent finds a candidate page to be shared, it creates a signature of this page and stores it in a de-duplication table. Each time KSM generates a new signature, it is compared against the signatures stored in the table. Whenever two pages with the same signature are found, KSM merges them. By default, KSM in KVM scans 100 pages in every 200 milliseconds. This is why any memory disclosure attack on KVM, has to wait for a certain time before the de-duplication takes effect upon which the attack can be performed [18, 19, 20].

In virtualized systems, KSM works in a similar way to the one described above. KVM, for example, uses the KSM mechanism for scanning the memory contents of VMs. In this case, KSM performs memory merging techniques among VMs instead of among processes. When a duplicate page from memory spaces of two VMs is detected, the page is de-duplicated if it is declared as shareable. Moreover, the client OS still performs KSM among the processes running inside it.

Using KSM, especially in a cloud environment translates into great memory savings. For example, experimental implementations [15] show that it is possible to run 50 Windows XP VMs with 1GB of RAM each on a physical machine with just 16GB of RAM. This in turn reduces the power consumption and system cost are significantly reduced for systems with multiple users. Using KSM, researchers demonstrated that 30% of the total pages are used by two 1024MB RAM VMs [17].

Finally, KVM is not the only hypervisor that implements de-duplication. VMware uses a similar technique called Transparent Page Sharing (TPS) with the goal of improving performance among their VMs via memory de-duplication [21].

2.3 Mobile Computing

With the advent of smartphones and mobile apps, computing systems evolved from PCs to smaller, portable, hand-held devices. Every smartphone has a motherboard, a CPU, DRAM, and I/O peripherals, just like a traditional computing system. They contain firmware, OS, software applications, networking stack, display, keyboard/touchscreen and all the security problems that come with it.

There are numerous mobile operating systems such as Android, iOS, Blackberry OS, Windows Mobile, Samsung Tizen, Symbian. However, in comparison to Android and iOS, the market share of the rest of the operating systems are negligible. As of now, Android has the largest market share and used throughout the world. In the following, we provide overview of the Android and the permission management system it employs to manage system resources.

2.3.1 Overview of Android Security

Android continues to dominate the smartphone market with an 82.8% share as of 2015 Q2 [22]. Unsurprisingly, mobile malware development almost exclusively focuses on the Android platform, with a combined total of 355 new families and variants of malware found in 2014 [23, 24]. Due to the large number of different resources such as sensors, camera, storage, GSM trans-receiver etc, Android malware is diverse. For instance, some malware like SmsSend, SmsSpy, and FakeInst send SMS to premium-rate numbers while others like Eropl exfiltrate personal data.

The Android platform is composed of three fundamental building blocks: the device hardware, the Android OS, and the Android application runtime, each sitting atop the previous. While Android is processor-agnostic, it can take advantage of hardware-specific security features such as ARM v6 eXecute-Never when present.

The Android OS, built on top of the Linux kernel, inherits its robust security capabilities. This includes the user-based permission model, process isolation, and an extensible mechanism for secure inter-process communication (IPC). The Linux kernel is responsible for executing core system services like process management, network management, and memory access. The device resources, such as GPS, telephony, and the camera, are all accessed through the OS [25].

In addition to inheriting Linux's identity management system, Android also inherits the copy-on-write (COW) capability in its process creation mechanism. The root process, called Zygote, pre-loads core resources that all Android processes will use. COW allows Android to spawn new processes without memory duplication; since these core resources are read-only, the libraries common to all processes will only be stored in one location. This significantly reduces an application's memory footprint.

Prior to Android 5.0, the Dalvik VM and Android's core native services and libraries comprised the Android application runtime. Dalvik was replaced by Android Runtime (ART) in Android 5.0. Android applications are most commonly written in Java, which is compiled to bytecode for the Java Virtual Machine (JVM), and then translated to Dalvik bytecode. Native applications and libraries, such as those managing audio, SSL, and graphics, are written in C and C++ to interface with the OS more closely and achieve better performance. The Android application runtime is contained within the Application Sandbox, a security environment which prevents applications from interfering with each other and system services. The Application Sandbox is implemented at the kernel level, and thus encapsulates and protects all components sitting above the kernel.

The Application Sandbox is realized through two fundamental Linux features: user-based access-control and memory isolation. The former implements standard

file access rights by allowing each user to control who is allowed to read, write, and execute their files. The Application Sandbox leverages this by assigning each application it's own unique user identifier (UID), thus preventing one application from accessing the files of another. Android requires each application to be signed by its developer before being uploaded to the Play Store. Applications that are signed with the same key are assigned the same Linux UID within Android, bypassing the isolation model provided by the Application Sandbox. This allows applications to share data and application components. Since there is no central Certificate Authority (CA) for developer private keys, it is the developer's responsibility to keep her private key secure. If she fails, adversaries can abuse the shared UIDs to access sensitive information.

Further, system components are run as root, so no user-level application can access or modify them. In Linux, each process is given its own virtual address space as a form of memory isolation. Android takes advantage of this by running each application as a separate process so no application can interfere with the memory space of another. Theoretically, since all applications are sandboxed at the OS level, memory corruption errors cannot compromise the device; they will only allow code execution with the permissions of the vulnerable application. To break out of the application sandbox, adversaries must compromise the kernel.

2.3.2 Android Permission System

Android employs permission management system to give apps access to various hardware resources, sensors and data. While some of these permissions require user consent, others do not. Before Android 6.0 (API 23), the permission consent was set to be given at install time. Granting permissions at the install time meant that apps listed all the permissions that they might require during execution and that

the users raced to the install button without checking them. After Android 6.0, permission requests are presented when the app needs that specific permission for the first time. When the user is prompted, he/she can still deny the permission even though the app is already installed and running. In cases when the user declines to give the requested permission, some applications may keep working while others crash outright or prompt the user until the permission is granted. Moreover, not all permissions require user consent. Depending on the importance of information or service that an app needs to access, permission might be granted automatically without prompting the user. In the following, we explain these different types of permissions and how they are handled by the Android operating system.

Normal Permissions: Normal permissions are used when apps require access to data and resources that are not deemed sensitive by Android. For instance, setting an alarm requires only Normal Permission and does require user consent since there is no serious danger of a privacy leak by setting up an alarm. Because of this, Normal Permissions do not require explicit user consent and are automatically granted by the system. Note that the user can always review which permissions an app uses, normal or not. While deemed safe and trusted, normal permissions can also affect the operation of other apps. For instance, the `KILL_BACKGROUND_PROCESSES` permission allows an app to shut down other apps using only their package name. While this permission does not allow access to any sensitive data, it gives crucial control over other apps in the system.

Dangerous Permissions: As explained in [26], any permission that is needed to access a sensitive, private data of the user or service of the device, is classified as Dangerous Permission. The data or the service that these permissions allow access to are sensitive, and therefore require explicit user consent. For instance, using the device camera requires Dangerous Permission since an app can access the

camera and take unauthorized photos without the user's consent. In comparison to Normal Permissions, these type of permissions clearly carry a higher risk of privacy violation. The following permission groups are considered dangerous permission by the Android and require explicit user consent; Calendar, Camera, Contacts, Location, Microphone, Phone, Sensors, SMS, and Storage.

Signature Level Permissions: These permissions are granted by the system to apps only if the app requesting the permission has the same signature as the app that declared that permission. If these signatures match, the system grants the permission without prompting the user. Signature Level Permissions are vendor dependent and are generally closed by hardware vendors.

2.4 Atomic Operations

Atomic operations are defined as indivisible, uninterrupted operations that appear to the rest of the system as instant. While operating directly on memory or cache, atomic operation prevents any other processor or I/O device from reading or writing to the operating address. This isolation ensures computational correctness and prevents race conditions. While instructions on single thread systems are automatically atomic, there is no guarantee of atomicity for regular instructions in multi-threaded systems. In these systems, an instruction can be interrupted or postponed in favor of another task. During this interrupt, the data can be modified by another thread. Hence the atomic operations are especially useful for multi-threaded systems where multiple threads are running in parallel. ADD, AND, CMPXCHG and XOR are some of the instructions defined in x86 architecture that can be executed atomically with a lock prefix. Also, XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use.

Various different mechanism are implemented in memory controllers and CPUs to ensure atomicity. For instance, in older x86 systems, the processor locks the memory bus completely until the atomic operation finishes, whether the data resides in the cache or in the memory. While ensuring atomicity, the process also results in a significant performance penalty to the system. In newer systems prior to Intel Nehalem and AMD K8, memory bus locking was modified to reduce this penalty. In these systems, if the data resides in cache, only the cache line that the processed data resides on is locked. This, 'cache lock' results in a very insignificant system overhead compared to the performance penalty of memory bus locking. However, when the data surpasses cache line boundaries and resides in two separate cache lines, the memory controller cannot guarantee atomicity by locking a single cache line. Instead, the memory bus is locked to ensure that no memory modifications can be performed until the atomic operation completes.

After Intel Nehalem and AMD K8, shared memory bus was replaced with multiple buses with non-uniform memory access bridge between them. While the system gets rid of the memory bottleneck for multiprocessor systems, it also invalidates memory bus locking. When a multi-line atomic cache operation needs to be performed, all CPUs has to coordinate and flush their ongoing memory transactions. This emulation of memory bus locking results in a significant performance hit.

On ARM processors, there are atomic instructions available in userspace as well. Prior to ARM v6, SWP instruction was used to provide atomic read and writes. Later, ARM v6k and ARM v7 introduced the `LDREX` and `STREX` instructions to split the atomic memory update into two pieces and ensure atomicity [27]. When an atomic memory update has to be executed, first the `LDREX` instruction is called to load a word from the memory and tag the memory location as exclusive. This operation immediately notifies the `exclusive monitor`, a simple state machine with

two states; open and exclusive [28]. After the memory location is tagged as exclusive, only the parties allowed by the `exclusive monitor` can store data to this location. If any other process/user attempts to store data to the location, the request is denied and an error state is returned. After the data is updated outside of the memory and the updated data needs to be stored, the `STREX` instruction is called to conditionally store to the memory location, the condition being the right to store to the location.

2.5 Cache Side-Channel Attacks

Side-channel attacks are defined as attacks that target the implementation of hardware or software rather than an algorithm itself. For instance, the power consumption of an SoC can leak information about the code running on the device. And if the SoC is performing cryptographic operations, it would reveal valuable secrets leading to a malicious recovery of secret keys.

Over the last decade, there has been a surge of micro-architectural attacks. Low-level hardware bottlenecks and performance optimizations have shown to allow processes running on shared hardware to influence and retrieve information about one another. For instance, cache side-channel attacks like Prime+Probe and Flush+Reload exploit the cache and memory access time difference to recover fine-grain secret information [29, 30, 31, 32]. In fact, researchers have shown that through this leakage, it is possible to recover cryptographic keys [33, 34, 1, 9, 35, 36, 37, 38]. In these works, the attacker exploits micro-architectural leakages stemming from data access time variations, e.g. when the data is retrieved from small but faster caches as opposed to slower DRAM memory.

The timing differences between cache and memory accesses can be exploited by a malicious co-located user to mount cache based side-channel attacks. Gaining

information about the memory lines accessed by a process may leak sensitive data dependent information leading to an unauthorized user discovering the secret key used in symmetric encryption or guessing the plaintext sent by an independent user. In general, all memory-dependent cryptographic algorithm implementations present a potentially exploitable side-channel leakage.

Cache side-channel attacks have been categorized into three main groups: *time driven*, *trace driven*, and *access driven* cache attacks. The classification is done based on the capabilities that the attacker obtains. The most restrictive ones are the time driven attacks, in which the attacker only can observe the aggregated time profile of a process. On the other hand, access-driven attacks assume only to know which sets of the cache have been accessed during an execution of the victim's code. Finally, trace-driven attacks are assumed to be able to collect the whole cache profile when the targeted program is running.

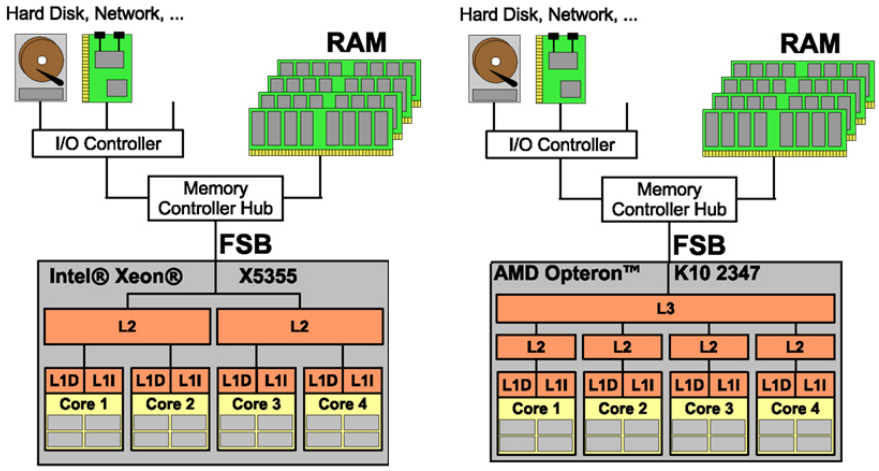


Figure 2.2: Cache hierarchy of Intel Xeon x5355 processor (left) and AMD Opteron K10 2347 processor (right).

Covert Channel vs Side-channel Attacks: The difference between a side-channel attack and a covert channel is that former requires no collaboration from

the victim while the latter implies that the two parties are willing to communicate. What is common among the two is that the information is passed through not logical channels but rather through side-channels e.g. CPU cache, DRAM, EM emanations etc.

2.5.1 Flush+Reload (F+R) Attack

Flush+Reload is a powerful cache-based side-channel first studied in [39] where Gullasch et al. demonstrated that it can be used to recover security sensitive information such as AES secret keys. Later in 2013, Yarom et al. used it to recover RSA encryption keys and named the attack Flush+Reload [31].

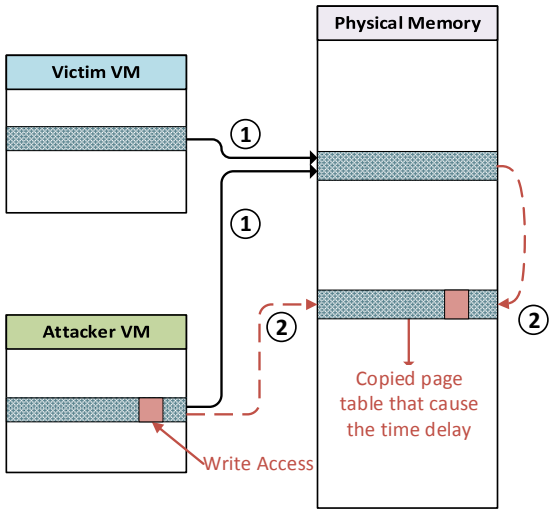


Figure 2.3: Copy-on-Write Method

	Shared Memory	Non-shared Memory
Read	Cached data read time*	Physical I/O read time**
Write/Modify	Copy time+cache write time	Cache write time

The attack vector of the Flush+Reload is the memory de-duplication where multiple processes share the physical memory and point to same memory pages. When the same memory page is mapped to multiple processes, a malicious process can

monitor the access of the others in two ways. First way is exploiting the difference between memory and cache access times. The second is the access time difference caused by the Copy-on-Write (CoW) command which copies the shared memory space upon any modification by one of the users. The CoW operation in this case will cause a longer access time due to the overhead of the copy and leak information about the execution.

In short, the Flush+Reload attack can be summarized in three stages:

- **Flushing stage**: In this stage, the spy process flushes the targeted memory lines from the cache hierarchies of all the cores using the `clflush` command. This means that after this stage, the monitored memory lines will not be present in any cache level of any core in the system. Instead, they will reside in the main memory.

- **Victim process run**: After the monitored lines are flushed out of cache, the attacker triggers the execution of victim code. If the victim program uses any of the monitored lines, these lines will be loaded back into the corresponding CPU core's cache hierarchy, from the last level of cache to the first one. However, if the monitored memory lines are not accessed by the target program, they will remain in the slower DRAM memory.

- **Reloading stage**: In this stage, the attacker checks the victim access to the monitored lines by reloading them. If the lines are loaded fast i.e. are coming from CPU cache, the attacker knows that these lines have been accessed by the victim. On the other hand, if target lines reside in the main memory, the reload time is going to be longer. Note that since the last level cache is shared across cores, a spy process working in a different core can perform this attack while running on a different core than the victim.

2.5.1.1 Flush+Flush (F+F) Attack

F+F attack is a variant of the F+R attack that has been proposed by Gruss et al. in [38]. Just like the F+R, the F+F attack also requires memory de-duplication and shared memory between the attacker and the victim. F+R works as follows;

- **Flushing stage**: The attacker flushes the targeted memory lines from the cache hierarchies of all the cores by using the `clflush` command.
- **Target process stage**: The attacker waits for a select period of time to allow the victim to make accesses to the monitored memory lines. If the victim accesses any of the monitored lines, the targeted data will be loaded from DRAM memory to CPU caches. However, if none of the monitored memory lines are accessed by the victim, they will remain in DRAM memory only.
- **Re-flush stage**: The attacker flushes the targeted memory lines and measures the time of the flushing operation. If the monitored data/code reside in the cache, the flush time is small. If it resides in the main memory i.e. it is not in any of the CPU caches, then the re-flush takes longer. By measuring the re-flush time, the attacker determines whether or not the victim made an access to the monitored memory lines.

2.5.2 Prime+Probe (P+P) Attack

In modern computer systems, the physical memory is protected and not available to the userspace applications. Instead, non-root users and processes can only access their virtual addresses. In order to map these virtual addresses to physical addresses, a memory address translation stage is required. On the other hand, this virtual to physical address translation only applies to a part of the address and not all bits of

the address are modified. For instance, the least significant p_{low} bits of $2^{p_{low}}$ sized memory pages are not translated and remain constant. These are called the *page offset*, while the remaining part of the address is called the *page frame number* and their combination makes the physical address.

The location of a memory block in the cache is determined by its physical address. Usually, the physical address is divided into three different sections to access *n-way* caches: the byte field, the set field, and the tag field. The length of the byte and set fields are determined by the cache line size and the number of sets in the cache, respectively. The more sets a cache has, the more bits are needed from the *page frame number* to select the set that a memory block will occupy in the cache.

The Prime+Probe attack is a cache based spy process first described 10 years ago by Osvik et al. [40]. The attack procedure consists of three main stages:

- **Cache priming:** In the priming stage, the attacker uses an eviction set and fills all the ways of a cache set with own data, essentially priming it.

- **Waiting for victims access:** In the second stage, the attacker waits for a certain time to let the victim use the previously primed cache. Victim's access, evicts attacker's data from the primed set and pushes them to a lower levels cache.

- **Probing the primed blocks:** In this stage, the attacker loads the previously primed memory blocks. Some of the blocks (the ones that the victim did not evict) will still reside in the cache, while the other ones will be evicted to a lower level cache or the memory. The location of the probed data in the memory hierarchy can be deduced by measuring its access time for each data block. This is possible because the lower level cache and DRAM accesses are slower than L1 cache accesses.

2.5.2.1 Prime+Probe in the LLC

The Prime+Probe attack has been widely studied in upper level caches [34, 41], but was first introduced for the LLC in [42, 43] with the use of **hugepages**. Unlike regular memory pages that reveal only 12 bits of the physical address, hugepages reveal 21 bits, allowing the LLC monitoring. Indeed, one of the main reasons why Prime+Probe was not applied to LLC earlier is that regular (4KB) memory pages limit the known physical address to 12 bits. This limitation makes the creation of eviction sets much harder for large LLCs that consist of thousands of sets.

Although Prime+Probe has been known for many years, it was not until one year ago when it was applied to the LLC. Some of the reasons why it was not trivial to modify the Prime+Probe attack to the LLC are:

- **Large cache**: The LLC is usually in the order of MBs making it impractical to prime the whole set.
- **Unknown physical bits**: Due to larger size of the LLC, the location of the memory blocks in the LLC is unknown. With small caches (like the L1 cache), the page offset provides enough information to infer the location in the cache, as demonstrated in [40]. However, more sets the cache has, the more bits from the pfn are needed to specify a location.
- **LLC slices**: In order handle several concurrent LLC accesses, Intel processors usually divide their LLC in slices, with an unpublished slice selection algorithm distributing the memory blocks among them. This means that even if it is possible to calculate the cache set of an address, the cache slide still needs to be located.

Indeed all of these complications can be handled to mount a LLC Prime+Probe attack, as demonstrated in [42, 43]. The first issue can be solved by monitoring only

the target sets where the targeted memory block location is known. The second issue can be solved by using hugepages as described in 2.5.2.2. Hugepages are usually in the order of MBs, making the p_o larger than the usual 12 bits that is obtained with regular pages. With 2MB pages, 21 bits of the page offset is visible and sufficiently large to target modern LLC. Figure 2.4 represents the number of bits known to an attacker using regular pages (12 bits) vs the number of bits known if he uses huge size pages (21 bits).

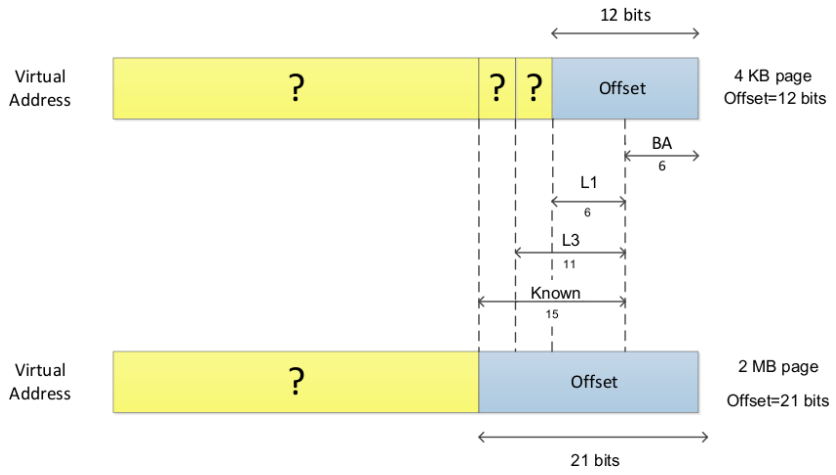


Figure 2.4: Virtual to physical address mapping of an Intel x86 processor for regular and hugepages

For CPUs with non-linear slice selection algorithms, it is harder to find the exact location of the target code in the cache. For instance, with a 10 core machine with 20 MBs of cache, even if the last 6 bits of the set number is known, there are still 5 unknown bits. In addition, the data can be located in non-linearly or linearly addressed slices of the cache. Hence, there are total $2^5 \times 10$ possible set-slice pairs for a given virtual address. To find the correct set-slice pair, all 320 possibilities must be profiled before an actual Prime+Probe side-channel attack.

As stated before, profiling a portion of the cache becomes more difficult when

the LLC is divided into slices. However, as observed by [42], it is possible to create an *eviction set* without knowing the algorithm implemented. This involves a step prior to the attack where the attacker finds the memory blocks colliding in a specific set/slice. This can be done by creating a large pool of memory blocks, and access them until it is observed that one of them is fetched from the memory. The procedure will be further explained in Section 6.2. A group of memory blocks that fill one set/slice in the LLC will form an *eviction set* for that set/slice as demonstrated in [44, 45, 46].

2.5.2.2 Cache Addressing and Virtual-Physical Memory Mapping

Modern processors use virtual memory to protect processes from accessing the physical memory directly. The OS manages the virtual to physical address translation. The physical memory in most modern systems is divided into *memory pages* of 4KB size. The page size plays a crucial role in the translation stage, since the number of bits from the virtual address that have to be translated directly depends on it. Indeed, if p_o is the page size in bytes, the lower $log_2(p_o)$ bits of the virtual address are not translated by the Memory Management Unit (MMU) and will remain the same in both the physical and virtual address. This portion of the address is called the *page offset*. The rest of the address bits will be refereed to as *virtual page frame number* and the *page frame number* before and after the translation respectively.

In order to efficiently perform this translation, modern processors have several levels of Translation Lookaside Buffers (TLBs). The TLB is a special cache holding the most recently fetched memory pages and their corresponding virtual page frame numbers. This allows the system to first check the TLB for the requested page translation, speeding up the page lookups.

Aimed at more efficient paging, most processors also allow *hugepage* allocations.

Hugepages are substantially larger than regular pages and have a separate TLB. As a result of this, a hugepage holds 2 MB of data while occupying a single TLB entry in contrast to 512 entries would be needed with regular pages which in turn reduces the number of TLB misses.

Memory Addressing in Cache: There are three widely used cache types: direct mapped (each memory block can only go to one fixed location in the cache), fully associative (a memory block can reside in any position in the cache) and set associative (a memory block can reside in a subset of locations in the cache). We will mainly focus on set associative caches since they are the most common choice in modern processors. Set associative caches are defined by 3 main parameters: the cache size s, the cache line length l and the number of ways w for each cache set. Using these parameters, one can calculate the number of sets in the cache as:

$n_s = s/(w * l)$

As Figure 2.5 shows, each of the memory blocks (l size blocks) will reside in a specific set in the cache, mainly defined by its physical address. For the address translation, the physical address ($pfn + p_o$) is divided into three different parts. The lowest $log2(l)$ bits points to a specific location in a cache line. The following $log2(n_s)$ bits indicates the set in which the data resides. The rest of the address bits act as tag and used for matching the corresponding memory block.

2.6 Machine Learning

In this dissertation, we tackle the problems of automation and scalability as well as mitigation of side-channel attacks. To this end, we employ machine learning, specifically Convolutional Neural Networks and Adversarial Learning. Further, we test the viability of our method against Adversarial Learning Defenses. In this chapter,

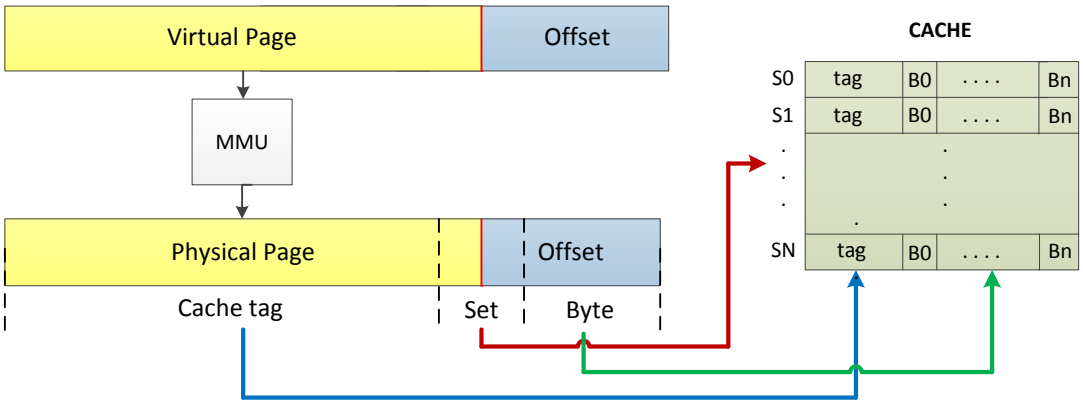


Figure 2.5: Cache accesses when it is physically addressed.

we provide the necessary background material to aid the reader in understanding the aforementioned concepts.

2.6.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are supervised, feed-forward artificial neural networks used in classification tasks. The supervised learning indicating that the data used to train the model is labeled and while the model does not require features to be extracted beforehand, it does require the inputs of different classes to be labeled. CNNs consist of layers of neurons with weights and biases that can learn the important features of the input dataset without human intervention. CNNs do not saturate easily and can reach high accuracy with more training data. Further, they do not require data features to be identified or pre-processed before the training. Instead, relevant features are discovered and learned from the training dataset automatically.

The disadvantage of using CNNs is that the models require a large number of training samples and are computationally expensive to train. Even so, in the past decade, it is shown that CNNs can surpass humans at many tasks that were

previously considered nearly impossible to automate [47, 48, 49, 50, 51, 52]. This breakthrough is fueled by both the increase of GPU powered parallel processing and optimizations in CNNs.

Training a CNN model is done in three phases. First, the labeled dataset is split into three parts; training, validation, and test data. The training data is fed to the CNN with initial hyper-parameters and the classification accuracy is measured using the validation dataset. Guided by the validation accuracy results, the hyper-parameters are updated to increase the accuracy of the model while maintaining its generality. After the model achieves the desired hyper-parameter optimization level, it is tested with the test data and the final accuracy of the model is obtained.

2.6.2 Adversarial Learning

Adversarial Learning (AL) is a subfield of ML that studies the robustness of models against adversarial inputs. It is a very active research area with a plethora of new attacks, defenses and application cases emerging daily [53, 54, 55, 56, 57]. The vulnerability of AL samples stems from the underlying assumption that the training and the test data comes from the same source and have consistent features. Studies have shown, however, by introducing small external noise or what is commonly referred to as `adversarial perturbations`, it is possible to craft adversarial samples and manipulate the output of ML models. In other words, by carefully crafting small changes to an input, one can push it from the boundaries of one class to another. Moreover, due to the high-dimensional space that the classifier operates in, very small perturbations can be enough to push a sample to other classes. While there are many different methods of crafting such perturbations, ideally they are desired to be minimal and not easily detectable.

AL methods on classical ML classifiers (under both white-box and black-box sce-

narios) have been known for quite some time [58, 59, 60, 61, 62, 63]. In 2013 Szegedy et al. [64] introduced the first AL methods on DNNs, showing that very small perturbations that are indistinguishable to the human eye can fool CNN image classifiers such as the ImageNet. The perturbations in the study are calculated using the technique called Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS). This algorithm searches in the variable space to find parameter vectors (perturbations). Later in 2014, Goodfellow et al. [65] improved the attack by using the Fast Gradient Sign Method (FGSM) to efficiently craft minimally different adversarial samples. Unlike the L-BFGS method, the FGSM is computationally conservative and allows much faster perturbation crafting.

In 2016, Papernot et al. [66] further improved upon Goodfellow's FGSM by using Jacobian saliency maps to craft adversarial samples. Unlike the previous attacks, the Jacobian saliency map attack (JSMA) does not modify randomly selected data points or pixels in an image. Instead, it finds the points of importance with regards to the classifier decision and then modifies these specific pixels. These points are found by taking the Jacobian matrix of the loss function given a specific input sample. The JSMA allows an attacker to craft adversarial samples by modifying fewer data points in comparison to FGSM.

In 2016 [67, 68, 69], multiple new AL methods were discovered. Moreover, the research showed that these adversarial samples are transferable i.e. perturbations that can fool a model can also work on other models trained on the same task. In [70], Papernot et al. showed that AL methods can also succeed under the black-box attack scenario where an attacker has access only to the classification labels and not the model parameters like weights, biases or the loss. This prevents the attacker from using the gradient to craft a perturbation. However, it is still possible to use the target model as an oracle that labels the inputs and then use these labeled

images to train a clone classifier. Authors demonstrated the feasibility of the attack on MetaMind and Deep Neural Network classifiers hosted by Amazon and Google. With 84, 96 and 88% misclassification rates respectively, they were able to fool the targeted classifiers.

In [71], researchers have shown that by iteratively morphing a structured input, it is possible to craft adversarial samples against a PDF malware classifier. The attack works by adding and/or removing compilable objects to a PDF and achieves 100% evasion rate. The attack assumes a black-box setting i.e. the attacker does not have access to classification confidence scores and has to rely solely on the output. The study acknowledges that the black-box attack scenario has a cost for obtaining labeled data and uses the number of observations required to quantify this cost.

2.6.3 Defenses Against Adversarial Learning

Adversarial learning (AL) is a problem that plagues machine learning systems. In order to harden machine learning models, some defenses have been proposed. Here, some of the defenses against adversarial learning are listed, specifically, the *adversarial re-training*, *defensive distillation* (DD) and *gradient masking*. These defenses are an integral part of machine learning systems since an attacker capable of overcoming AL would solve one of the fundamental problems of ML. Moreover, it is known that these defenses can be overcome with new types of AL methods or simply by increasing the perturbation size. In short, AL and defenses against it remains a very active research area.

2.6.3.1 Adversarial Re-training

The defense was first proposed by Szegedy et al. in 2013 [64]. Later in 2014, Goodfellow et al. [65] improved the practicality of the method by showing how to

craft adversarial samples efficiently using FGSM, making it easier to craft adversarial samples. In this defense, the model is re-trained using adversarial samples. By doing so, the model is 'vaccinated' against perturbations and can correctly classify them. In other words, the method aims to teach perturbations to the model so that it can generalize better and not be fooled by small perturbations. While this method works successfully against a specific type of attack, it has been shown to fail against attack methods that the model was not trained for.

2.6.3.2 Defensive Distillation

The DD has been proposed by Papernot et al. [72] in 2016 to protect DL models against AL methods. The goal of this technique is to increase the entropy of the prediction vector to protect the model from being easily fooled. The method works by pre-training a model with a custom output layer. Normally, the softmax temperature is set to be as small as possible to train a tightly fitted, highly accurate model. In the custom layer, however, the temperature value is set to a higher value to distill the probability outputs. The first model is trained with the training data using hard labels i.e. the correct class label is set to '1' and all other class labels are set to '0'. After the model is trained, the training samples are fed into it and the probability outputs are recorded as *soft labels*. Then these soft labels are used to train the second, distilled model with the same training data. This process smooths the model surface on directions that an adversary would use to craft perturbations. This smoothing process increases the perturbation size required to fool the model and invalidates some of the previously crafted adversarial samples. This smoothing can be set to different levels by adjusting the temperature value. Note that however, the DD can reduce the classification accuracy significantly if the temperature value is set too high.

2.6.3.3 Gradient Masking

The term gradient masking defense has been introduced in [70] to represent a group of defense methods against adversarial samples. The defense works by hiding the gradient information from the attacker to prevent it from crafting adversarial samples. Papernot et al. [70] however showed that the method fails under an oracle access scenario. An attacker can query the classifier with enough samples to create a cloned classifier. Since the clone and the original classifiers have correlated gradients, the attacker can use the gradient from the clone and craft adversarial samples, bypassing the defense.

Chapter 3

Related Work

This chapter discusses the latest related work in the literature of micro-architectural attacks on cloud and mobile platforms. The listed works are categorized for coherency and discuss work directly related to this dissertation. In the following, we discuss the related work on cache side-channel attacks, co-location detection on cloud, branch prediction attacks, hardware performance counters and micro-architectural attacks on mobile.

3.1 Cache Side-channel Attacks

Many applications being executed in a shared hardware introduces a potential threat to the hardware and software sandboxing techniques. A malicious application can monitor hardware access patterns to recover sensitive information. With the starting points, micro-architectural side-channel attacks have been studied over the last 20 years. More specifically, cache side-channel attacks establish a relationship between the positions accessed in the cache and the data used by the victim. For instance, the first examples of Prime+Probe attacks monitor the L1 cache to deduce information about the victim. However, since the L1 caches are core-private

resources, early attacks were restricted to core co-located processes. This scenario was deemed unrealistic in modern cloud systems. Consequently, micro-architectural attacks did not receive much attention after the first practical realizations. However, with the increasing popularity of shared hardware systems, i.e. cloud and mobile computing, the cache side-channel attack scenario of attacker and victim being able to run processes on the same hardware became realistic and the interest in cache attacks peaked [73, 34]. Follow-up works overcame the issue of only targeting core-private resources: attacks targeting the last-level cache (LLC) have now widely been studied [31, 42, 43]. Since the LLC is shared across cores multi-core processors it provided a suitable side-channel to run cross-core attacks. Further, the timing difference between LLC and memory accesses is higher, making these attacks even more robust.

In general, all the memory-dependent cryptographic algorithms may potentially be exploited by cache attacks, if no countermeasures are provided. Motivated by this observation, a number of researchers have targeted weakly designed software algorithms. The first study considering cache-memory accesses as a covert channel targeting the extraction of sensitive data was done by Hu [74]. However, it was not until 1998 when Kelsey et al. [75] studied the cache hit ratio as a method to deploy the first cache side-channel attacks. Page, in 2003 suggested theoretical methods on cache side-channel attacks [76]. The application of these techniques to recover information from the typical table lookup operations performed in symmetric ciphers was first studied by Tsunoo et al. [77].

As early as in 2003, D. Brumley and Boneh [78] demonstrated timing side-channel vulnerabilities in `OpenSSL 0.9.7` [79], in the context of sandboxed execution in VMMs. The study showed the recovery of RSA private keys from an `OpenSSL`-based web server when victim and attacker ran in the same processor.

Symmetric cryptography is a popular target of side-channel attacks, as demonstrated in [33, 40, 80, 39]). One of the earliest, and most practical cache side-channel attack was that of Bernstein's 2005 cache timing attack against AES [33]. In this work, Bernstein was able to recover an AES key due to micro-architectural timing variations in the cache. In the same line, Bonneau et al. [80] showed how collisions in the last round of AES can affect the overall time execution and give information about the key used by the algorithm. These two attacks can be considered as time-driven cache side-channel attacks since they obtain information by just looking at the overall timing execution. At the same time, several new trace-driven attacks were proposed. Osvik et al. [40] introduced two new side-channel techniques and where able to extract AES keys: `Evict and Time` and Prime+Probe.

Acıiçmez showed that cache attacks not only work when they target the data cache but also when they monitor the instruction cache [41] as well. He used the Prime+Probe technique to monitor whether an RSA operation was calling square or multiply operations, and thereby recovered the private key. Zhang et al. managed to recover an ElGamal encryption key in a cloud scenario running XEN hypervisor when the adversary is co-located in the same core [34]. They used the above-described Prime+Probe technique.

In the more recent years, a new technique emerged for cache analysis: Flush+Reload. The first work to utilize this technique was Gullasch et al. [39], in which they managed to recover an AES encryption key by using the Complete Fair Scheduler to block the encryption execution. In 2013, Yarom et al. [31] used the tool to recover an RSA decryption key running in GnuTLS. Later they used the same technique to recover an ECDSA decryption key from `OpenSSL` [82]. Irazoqui et al. [1] managed to recover an AES encryption key in a real cloud scenario without the necessity of blocking the AES execution (c.f. [39]). Lastly, Zhang et al. [83] demonstrated that

Flush+Reload is also applicable in Product as a Service (PaaS) clouds by recovering sensitive information from a co-located victim.

In addition to the attacks on public key cryptography schemes cache attacks also have been applied to AES [43, 1], ECDSA [91], TLS messages [7], the number of items in a shopping cart [83] or even the key strokes typed in a keyboard [35]. Even further, they recently have been applied to PaaS clouds [83], across processors [92] and in smartphones [36].

3.2 Co-location Detection on Cloud

In 2009, Ristenpart et al. [81] showed that it is possible to solve the co-location problem in cloud environment and therefore use the same hardware resources as a targeted victim. The study targeted EC2 Amazon Web services and demonstrated that an attacker can obtain co-location with 40% chance. This work, for the first time, opened the door for side-channel attacks in the cloud setting. Two years later, Zhang et al. used the cache as a tool to determine whether a user is co-located with someone else or not [73]. In the last few years several methods were proposed to detect co-location on commercial clouds [93]. These works use methods such as deducing co-location from instance and hypervisor IP address, hard disk drive performance degradation, network latency, and L1 cache covert channel. However, in response to these works, most of the proposed techniques have been closed by public cloud administrators.

Bates et al. [93] demonstrated that a malicious VM can inject a watermark in the network flow of a potential victim. In fact, this watermark would then be able to broadcast co-residency information. Again, even though the technique proved to be extremely fast (less than 10 seconds), it was never tested in commercial clouds.

Zhang et al. [94] demonstrated that Platform as a Service (PaaS) clouds are also vulnerable to co-residency attacks. They used the Flush+Reload cache side-channel technique together with a non-deterministic finite automaton method to infer co-location with a particular server. The technique proved to be effective in commercial PaaS clouds like DotCloud or OpenShift, but would never work in IaaS clouds where the memory de-duplication is not implemented, as in most of the commercial IaaS clouds.

In 2015, Varadarajan et al. [95] investigated co-location detection in public clouds by triggering and detecting performance degradation of a web server using the memory bus locking mechanism explored by Wu et al. in 2012 [96] to detect co-location. Simultaneously Xu et al. [97] used the same memory bus locking mechanism to explore co-location threat in Virtual Private Cloud (VPC) enabled cloud systems. Moreover, architectural side-channels can be used to covertly communicate or signal the presence of co-location as demonstrated in [98].

3.3 Branch Prediction Attacks

Time leakage in branch prediction units give rise to another class of side-channel attacks as demonstrated by Acıiçmez et al. where the authors exploited key dependent branches in an RSA computation of OpenSSL [99, 100, 101]. Recently B.B. Brumley and Tuveri [102] demonstrated that the ladder computation in the popular ECDSA implementation of `OpenSSL 0.9.8o` is vulnerable to timing attacks by extracting the private key used in a TLS handshake. In these attacks, authors monitor whether the outcome of a branch is miss-predicted, and use this information to deduce if the branch has been taken or not in a square and multiply operation. These attacks showed that by monitoring the execution time of a branch, an attacker can deduce

if the victim had executed a branch or not and infer security sensitive information.

3.4 Hardware Performance Counters (HPC)

HPCs are used to provide fine-grained information about the execution of a process in order to find bottlenecks and optimize the performance. However, this fine-grained information can be also used as a side-channel leakage source to steal sensitive information, as demonstrated in the works below. In [103] Alam et al. leverages *perf_event* API to detect micro-architectural side-channel attacks. In 2009, Lee et al. [104] showed that HPCs can be used on cloud systems to provide real-time side-channel attack detection. In [105, 106] researchers have shown that HPC can be used to detect cache attacks. Moreover, Allaf et al. [107] used a neural network, decision tree, and kNN to specifically detect Flush+Reload and Prime+Probe attacks on AES.

Moreover, researchers have shown that by using the fine-grain information provided by HPCs, it is possible to violate personal privacy as well. In [108], Gulmezoglu et al. show that HPC traces can be used to reveal the visited websites in a system. The attack relies on ML techniques such as auto-encoder, SVM, kNN and decision trees and works even on privacy conscious Tor browser.

3.5 Micro-architectural Attacks on Mobile Platforms

Micro-architectural components have been widely exploited under non-virtualized and virtualized scenarios. However, little work has been done on exploiting embedded processors such as smartphones and tablets at the hardware level. Although

time driven attacks have proven to be effective in ARM processors [109, 110], these attacks have not been demonstrated on mobile platforms i.e. devices running a mobile OS like Android or iOS. More recently, access-driven side-channel attacks have been exploited to detect ARM Trust-Zone code usage in the L1 cache [111], again not on a mobile platform. Finally in 2016, Lipp et al. [36] managed to run micro-architectural attacks such as Prime+Probe and Flush+Reload on various Android/ARM platforms, proving the practicality of these attacks on mobile devices. In the attack, authors exploited timing differences of accesses from cache and memory to recover sensitive information like keystrokes using a non-suspicious app. Also in 2016, Veen et al. [112] showed that the Rowhammer attack can be performed in Android platforms, without relying on software bugs or user permissions.

Other than cache attacks, there are also other OS based side-channel attacks targeting mobile platforms. These attacks take advantage of hardware related information provided by the OS to extract information. For instance, one can access Linux public directories to monitor the data consumption of each process to build a fingerprinting attack [113]. The network traffic is not the only feature that can be exploited; e.g. per process memory statistics are given by the OS can also be utilized to monitor what a victim application is doing [114] or even recover user's pictures [115].

As for countermeasures, very little work exists to eliminate side-channel attacks implemented in mobile devices. However, there are studies showing that detecting a malicious application is possible. For instance, [116] utilizes static analysis to identify code that executes GUI attacks, whereas [117] focuses on preventing memory-related attacks (including memory bus side-channel attacks) by using ARM-specific features.

Chapter 4

Software Detection Through Shared CPU Cache

Software updates and security patches have become a standard method to fix known and recently discovered security vulnerabilities in deployed software. In public servers, the use of outdated crypto libraries allow adversaries to exploit known weaknesses and launch attacks with significant security impacts. The proposed technique exploits leakages at the hardware level to first, determine if a specific crypto library is running inside a co-located virtual machine (VM) and second to discover the IP of the co-located target. To this end, we use the Flush+Reload cache side-channel technique to measure the time it takes to call (load) a crypto library function. Shorter loading times are indicative of the library already residing in the memory and shared by the VM manager through *memory de-duplication*. We demonstrate the viability of the proposed technique by detecting and distinguishing various crypto libraries, including `MatrixSSL, PolarSSL, GnuTLS, OpenSSL` and `CyaSSL` along with the IP of the VM running these libraries. In addition, we show how to differentiate between various versions of libraries to better select an attack

target as well as the applicable exploit. Our experiments show a complete attack setup scenario with single-trial success rates of up to 90% under light load and up to 50% under heavy load for libraries running in KVM.

4.1 Motivation

Cloud computing has become a major building block in today's computing infrastructure. Many start-up and mid-scale companies such as Dropbox[1] leverage the ability to outsource and scale computational needs to cloud service providers (CSPs) such as Amazon AWS[2] or Google Compute Engine. Other companies may build their computational infrastructure in the form of private clouds, harnessing cost savings from resource sharing and centralized resource management within the company. Nevertheless, one of the main concerns that are slowing the widespread use of such Infrastructure as a Service (IaaS) technologies is potential security vulnerabilities and privacy risks of cloud computing. Usually, CSPs use virtualization to allow multiple tenants to share the same underlying hardware. While the resource sharing maximizes the utilization and drastically reduce cost, ensuring isolation of potentially sensitive data between VMs instantiated by different and untrusted tenants can be a challenge. Indeed, the main security principle in the design and implementation of virtual machine managers (VMMs) has been that of the process and data isolation achieved through *sandboxing*. Although logical isolation ensures security at the software level, a malicious tenant might still extract private information due to leakage coming from *side-channels* such as shared hardware resources. In short, hardware sharing creates an opening for various side-channel attacks developed for non-virtualized environments. These powerful attacks are capable of extracting sen-

[1]Dropbox grew from nil to an astounding 175 million users from 2007 to 2013 [118].
[2]Amazon AWS generated $3.8 billion revenue in 2013 [119].

sitive information, e.g. passwords and private keys, by profiling the victim process.

Until fairly recently, the common belief was that side-channel attacks were not realistic in the cloud setting due to the level of access required to the cloud server, e.g. control of process execution and more specifically the difficulty in co-locating the attack process on the machine executing the victim's process. This belief was overturned in 2009 by Ristenpart et al. [81] who demonstrated that it is possible to solve the co-location detection problem and extract sensitive data across VMs. Using the Amazon Elastic Compute Cloud (EC2) service as a case study, Ristenpart et al. demonstrated that it is possible to identify when an attacker VM is co-located on the same server with a potential victim and therefore using the same hardware as the attacker VM. The work further shows that—once co-located—cache contention between the VMs can be exploited to recover keystrokes across VM boundaries. By solving the co-location problem, this initial result fueled further research in Cross-VM side-channel attacks. Zhang et al. [34] utilized a cache side-channel attack implemented across Xen VMs to recover an ElGamal decryption key from a victim VM. The authors applied a hidden Markov model to process the noisy but high-resolution information across VMs. Shortly thereafter, Yarom and Falkner showed in [31] that RSA is also vulnerable, as well as ECDSA, as shown by Yarom and Benger in [82]. Both attacks use the Flush+Reload technique that exploits the shared L3 cache. It is important to note that since the L3 cache is shared among cores, the attack works even if the victim and the attack processes are located on different cores.

At the system level, one of the most significant consequence of these new high-resolution cache side-channel attacks is that they expose new vulnerabilities in popular crypto libraries such as `OpenSSL`, which were previously considered secure. This forced the crypto library developers to fix their implementations and release patches

to mitigate these new attacks. It should be noted that side-channel attacks are just one more threat to crypto libraries from a long list of vulnerabilities regardless of whether the library is executed in VMs or natively. Therefore, it is crucial to use the most recent version of the crypto libraries where most of the discovered vulnerabilities have been already mitigated via a series of patches. Using an outdated crypto library renders the server vulnerable with potentially devastating consequences.

Good examples of recently outdated crypto libraries are the ones susceptible to Lucky 13 and the infamous Heartbleed attacks. In 2013, AlFardan et al. discovered that most crypto libraries were vulnerable to a padding oracle attack that they named Lucky 13 attack [120]. The attack was able to recover the messages sent in TLS connections by just looking at the time difference caused by invalid padding in digest operations. Although they showed only results in OpenSSL and GnuTLS, most of the crypto libraries were affected by it. Patches were released immediately by library developers. But using a non-patched version of any of these libraries would still leave the door open to this MEE attack.

In 2014, Neel Mehta from the Google security team discovered a dangerous security bug called Heartbleed in the popular `OpenSSL` crypto library [121]. In a nutshell, Heartbleed vulnerability allows a malicious attacker to read more data in the victim's memory than it should, exposing sensitive information like secret keys. The attack quickly became popular in the media and caused grave concern among security researchers for such a simple yet severe vulnerability to go undetected for many years. The cybersecurity columnist Joseph Steinberg argued that the Heartbleed bug is the worst vulnerability found since commercial traffic began to flow on the internet. Indeed, 17% of supposedly secure web servers were found to be vulnerable to the attack. Soon after instances of Heartbleed attack were discovered in the wild. For instance, the Canada Revenue Agency reported the theft of 900

social security numbers [122]. In another instance, the UK parenting site Mumset had several user accounts hijacked [123]. Most of the websites patched the bug within days of its release, but it remains unclear whether the vulnerability has been exploited before its public announcement. It is believed that some attackers might have been exploiting the vulnerability for five months before Mehta's discovery.

The Heartbleed vulnerability presents a striking example of the severe consequences of using an outdated crypto library and a striking example of how people ignore up-to-date software. One month after the Heartbleed bug was discovered, 300k of 600k systems were patched [124]. But the month after that only 9k additional systems was patched leaving the remaining 300k systems still vulnerable. This dramatic drop in the number of patched systems shows that the Heartbleed patching is almost over even though the security risk still persists. In general, even when more critical vulnerabilities are discovered, there is inherent inertia in patching systems gives attackers a window of opportunity to penetrate computing systems. During this narrow window an attacker has to first run a discovery phase where the system is target computing platform through a series of tests to identify the most vulnerable point of entry in the subsequent attack phase [3]. During the first phase, it is critical for the attacker remains stealthy. Therefore, a discovery tool that permits such facility, i.e. covertly detecting a particular installation of an outdated crypto library, will be an indispensable tool in the hands of an attacker.

Contributions

In this chapter, we introduce a method to detect the execution of specific software co-residing in the same host across VM-boundaries. In particular, we show that

[3]Directly running the attack carries varying levels of risks of exposing the attack depending on the nature of the attack.

crypto libraries executed in a co-located server in the cloud can be detected with high accuracy, and that specific versions of those libraries can also be distinguished reliably. After this library detection stage, we show that the IP of the co-located VM running the target library is also recoverable in a short time.

The technique can enable malicious parties, to covertly detect vulnerable crypto libraries/versions prior to performing an attack. The detection method exploits subtle leakages of timing information at the hardware level and runs rather quickly. At the protocol level, the detection technique does not interfere with the target's usual operations. Therefore, the detection method is very hard to detect by the target.

The very same detection technique can be used by cloud service providers to gently detect vulnerable crypto libraries running in their clouds, and notify the clients using these libraries. The detection technique is virtually non-interfering and therefore will not cause any noticeable degradation in the client's system. By this, we show that cache side-channels can also be used in a constructive way to *improve* the security of a cloud architecture. In essence, the technique we introduce brings greater transparency in cloud systems.

In this chapter, we develop a method that detects whether a co-located VM is running a specific crypto library and to furthermore determine the specific version of the library as well as the IP address of the co-located VM. Described attack scenario requires a detection technique that must work across cores to be realistic in a cloud setting. For this purpose, we need to exploit a type of shared resource between cores in multicore systems that leak private information. Several single core resources like branch predictors and TLBs have been exploited in the past. However, information about these shared resources in modern multi-core processors is scant at best, and for most cases is not available to the public. In contrast, most modern processor

caches *can* act as covert and side-channel and can be used for our detection method. Specifically, contributions can be summarized as follows:

- We demonstrate that it is possible to *detect* and *classify* executed code across VM boundaries with high accuracy
- We show *how* the developed method can be applied to determine the crypto library being used by a co-located VM
- Our results show that a co-located VM can discover the use of outdated vulnerable versions of a library; This means that for the first time we show how to distinguish between vulnerable and non-vulnerable crypto-library versions across VMs
- We show that after detecting the cryptographic library, the IP address of the co-located VM can be recovered in seconds to minutes depending on the network size
- We present a test bench with a popular cloud hypervisor KVM that proves the viability of our detection method

We present empirical results for detecting the execution of `MatrixSSL, PolarSSL, GnuTLS, OpenSSL` and `CyaSSL` crypto libraries when running inside a VM in KVM, as well as distinguishing and detecting specific *versions* of such libraries, in particular `OpenSSL` versions 0.9.7a, 0.9.8k, 1.0.0a, 1.0.1c, 1.0.1e, 1.0.1f, and 1.0.1g. Also, we show the time required to recover the IP address of the co-located VM running the aforementioned libraries. Our detection method obtains a success rate of up to 90% under low noise scenarios and a success rate of up to 50% under heavy load scenarios while maintaining a negligible false-positive rate. This means that even when the workload is sufficiently high, our detection method detects almost one out

of two library calls made by the target, and virtually never incorrectly detects a library.

As mentioned before, the motivations for detecting the execution of a specific piece of software being used can be manifold. The knowledge of the crypto library being used can give crucial information to a co-located adversary. For example, each crypto library has unique features and therefore can be used for fingerprinting. Furthermore, common attacks are not addressed in the same way in all the libraries. Some libraries have weaker patches than the others. This tool gives to the attacker the ability to determine whether a library that a co-located tenant is using is vulnerable against a certain type of attack. Here, we focus on the most popular crypto libraries: `GnuTLS, OpenSSL, PolarSSL, MatrixSSL` and `CyaSSL` [125, 126, 127, 128, 79]. These libraries have different cryptographic implementations and from the side-channel point of view some of them can be more secure than others. In this chapter, we give examples of why an attacker could benefit from the knowledge of the crypto library being used.

AES: is the most popular block cipher in use today. Its implementation among the different libraries varies, with `OpenSSL` having cache attack mitigation techniques in their implementation such as bit slicing techniques, while other libraries such as `PolarSSL` do not implement any technique to cope with cache leakage.

MEE attacks: The protection level of common attacks like Beast [129] and Lucky 13 [120] varies among different libraries. For example, we know that `OpenSSL` has a real constant time implementation to avoid padding leakage, whereas `CyaSSL` or `GnuTLS` do not [130, 120].

RSA: RSA is the most widely used public key cryptographic algorithm. Yarom et.al [31] showed that GnuTLS was not protected against cache side-channel attacks. Although a fix was added in the most recent version to address this leakage, an

attacker can still detect calls to a specific non-patched version of the library.

Insecure encryption ciphers: While some crypto libraries such as `CyaSSL` does not support weak encryption algorithms like DES-CBC, others do, e.g. `OpenSSL`. If an attacker can detect the use of vulnerable libraries, our detection technique gives an attacker a good opportunity to perform attacks against it.

The usage of crypto libraries is very typical in many processes performed by a virtual machine. For instance, Mozilla web browser uses NSS (which could be added to the detector) whereas Chrome browser uses `GnuTLS`. However, when a downloading process in the command line is performed, `OpenSSL` is used as the crypto library. Other applications like Steam use `OpenSSL` as well. This means that the observer could easily detect one of those widely used libraries, and in consequence, take advantage of any weakness present on it. Furthermore, This *cross-detection* method applied to crypto libraries allows the observer to profile the usage of certain applications (in case that the application is not using any other shareable code), like the ones mentioned above.

4.2 Detection Method

Here we introduce our library detection method. We distinguish between two scenarios, for which the detection methods slightly differ:

- **Library Detection:** detecting whether a specific library is being used in a co-located virtual machine
- **Version Detection:** detecting whether a particular version of a library is running in a co-located VM.

The Library detection method works by exploiting information leaked through deduplication, detected by using the Flush+Reload technique. It is clear that each

library has unique functions that are called when an SSL/TLS connection is performed. This gives rise to a library identifying process.

4.2.1 Detection Stages

The detector performs *library detection* and *version detection* in several stages. Detailed steps of the detection method are as follows:

Unique Function Identification: The detector identifies functions that are called when an SSL/TLS connection is established. The goal is to pick functions that are unique to the library and therefore have potential in being mapped to a unique hash during the de-duplication process while at the same time marking the event we wish to be able to detect. For the crypto libraries we are targeting, we have pre-selected the identification functions as follows:

- **OpenSSL**: `SSL_CTX_new`. This function creates the context variable to start an SSL/TLS connection and is always called before a SSL/TLS connection.
- **PolarSSL**: `ssl_set_authmode`. This function is called to select the preferred authentication mode in an SSL/TLS connection performed by `PolarSSL` at the beginning of an SSL/TLS connection.
- **GnuTLS**: `gnutls_global_init`. Function for the initialization of `GnuTLS` library variables and error strings that is called before the beginning of each SSL/TLS connection.
- **CyaSSL**: `CyaSSL_new`. Every SSL/TLS connection is associated with a `CyaSSL` object. This object is created by the `CyaSSL_new` function, and therefore has to be called prior to each `CyaSSL` SSL/TLS connection.

- **MatrixSSL**: `matrixSslOpen`. Opening library function performed by `MatrixSSL` prior to any other SSL/TLS functions, making it suitable for our detection method.

Offset Calculation: The detector has to calculate the offset of these functions with respect to the beginning of the library since the ASLR moves the user address space randomly. Functions such as `dlopen` allow to recover the starting virtual address of the monitored library. By further obtaining the virtual address of a specific function, the attacker can calculate the difference between the addresses hence the offset. Another possibility for the attacker is to disable the ASLR in his own OS and use the address corresponding to the targeted function directly.

Flush+Reload as Detection Method: The attacker uses the Flush+Reload attack to detect whether a function from a shared library has been called or not. If any other co-located VM use one of the functions that the attacker is trying to detect, this will be present in the last level of cache and therefore the reload time is going to be smaller. Hence the attacker can conclude that the library which the function belongs to has been called. If it was not accessed, the function will reside in main memory, having a bigger reload time. Figure 4.4 visualizes this last statement via an experiment outcome. The figure shows the reload times of a certain function when monitored by the Flush+Reload technique. The reload time, when a co-located VM is calling it constantly (red bars), differs significantly from the case where it is not called (blue bars). The difference is quite substantial, and if the threshold is set correctly, the noise is very small. Finally, there are special cases where the accesses cannot be detected. These access scenarios are as follows:

- Victim access occurs after the reload stage, and they do not overlap. In this case, the access is not going to be detected by the Flush+Reload technique.

- Victim access and Flush+Reload stage overlap. If the victim access was done slightly before the reload stage, this access will be noticeable. However, if the access was made slightly after the reload stage, it will not be detected.
- Some other process evicts the access prior to the reload stage, and therefore this is not going to be detectable by the Flush+Reload technique.

4.2.2 Preventing Wrong Version Detection

The detection is performed by monitoring unique functions associated with a library/version. These are easy to find across libraries since `OpenSSL` does not use the same function as any other library. However, this is not true for the version detection scenario. The targeted functions in each version can experience three different situations:

Different Versions, Same Page: Figure 4.1 shows the case where two different `OpenSSL` versions are present and the targeted page (in which resides the targeted function) is the same for both of them. Since the memory is divided into pages of 4 KB and KSM works computing hashes at the page level, both pages will create the same hash and they would be merged. Therefore, an attacker would mispredict if the call was made by `OpenSSL` version 1 or `OpenSSL` version 2. In this scenario, the best route to take is to target another function that is not same in both libraries.

Different Versions, Different Functions: If the targeted functions to detect are different even though previous pages are the same, the attacker has no risk on mispredicting the versions, since KSM will never merge two different pages. This is the case we present in Figure 4.2

Different Versions, Same Page, Different Offset: In another situation, the functions are the same in both versions of the library, but since a modification has

been done in some other previous page in the library, the offset will not be the same anymore. Therefore the page address would be different and the result of the hash operation applied to the pages will be different. In this case, the attacker would still be able to recognize different versions even though they have the same function. This is the situation presented in Figure 4.3.

We want to use a function that is called always in an SSL/TLS connections and that establishes a relationship with one single specific library. In the cases that we analyze, we keep using `SSL_CTX_new` because it meets both requirements. We already discussed the first requirement above. To test if we satisfy the second requirement, we analyze the function across all the libraries analyzed by our detector, observing different outcomes. This function changes between some of the versions, e.g. `OpenSSL 0.9.8k` and `OpenSSL 1.0.1a`, and does not change between versions, e.g. `OpenSSL 1.0.1e` and `OpenSSL 1.0.1f`. However, the offset with respect to a page boundary is different among all of them, making the digest from the hash operation different for all of them. Therefore there is no risk that the hypervisor will merge pages from different library versions.

4.3 IP Address Recovery

After determining the library running in the co-located VM, the attacker runs the TLS handshake detector for that specific library and starts the IP address recovery step. To recover the IP address of the co-located VM, the attacker starts sending TLS communication requests to all IP addresses starting with her local subnet and trying a wider range of addresses until a handshake is detected. When she triggers the co-located target VM, the specific library detector that she ran beforehand detects the TLS handshake. Only by observing the detector output and the IP

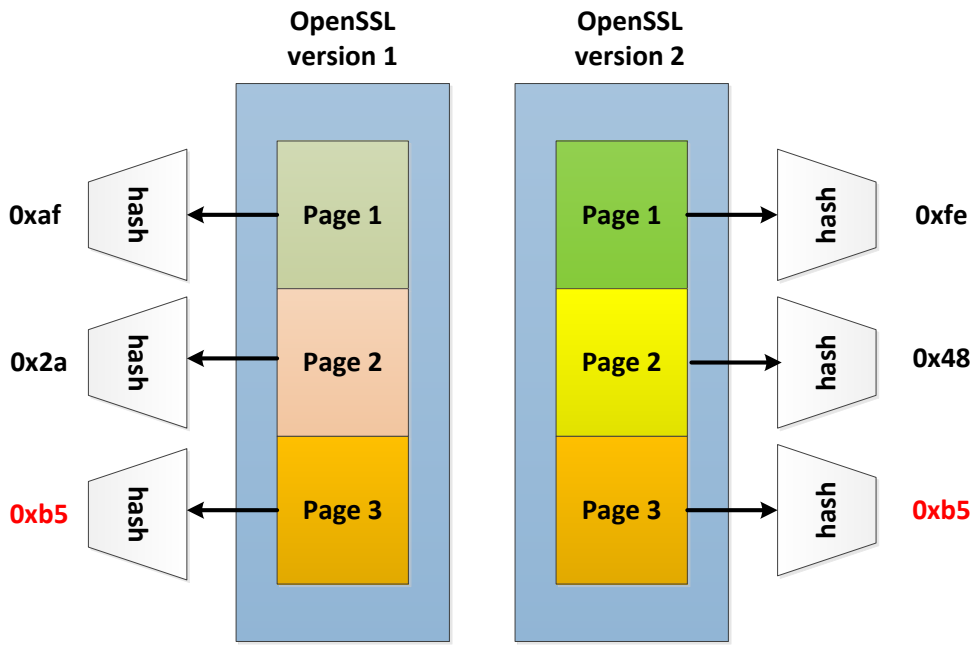


Figure 4.1: KSM when two different versions have different pages, but the targeted page is the same

scanner process, she can then pinpoint and recover the IP address at the time of a detection hit. This step is scripted and requires varying times depending on the detected library TLS setup time. In the worst case with the slowest library, it takes less than 15 seconds to scan 255 different IP addresses for a match.

4.4 Additional Dangers of Version Detection

Version detection goes beyond library detection, as its main goal is to distinguish between different versions of the same software family, such as a crypto library as `OpenSSL`. There are several well-known vulnerabilities in certain `OpenSSL` versions that can enable simple attacks if a malicious party becomes aware of an unpatched implementation. When used on the cloud, an adversary can use the proposed tool

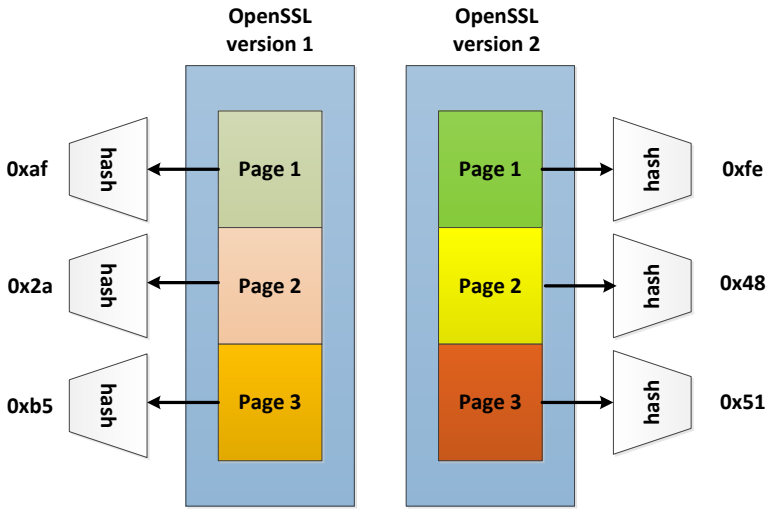


Figure 4.2: KSM when two different versions have different pages, and the targeted page is different

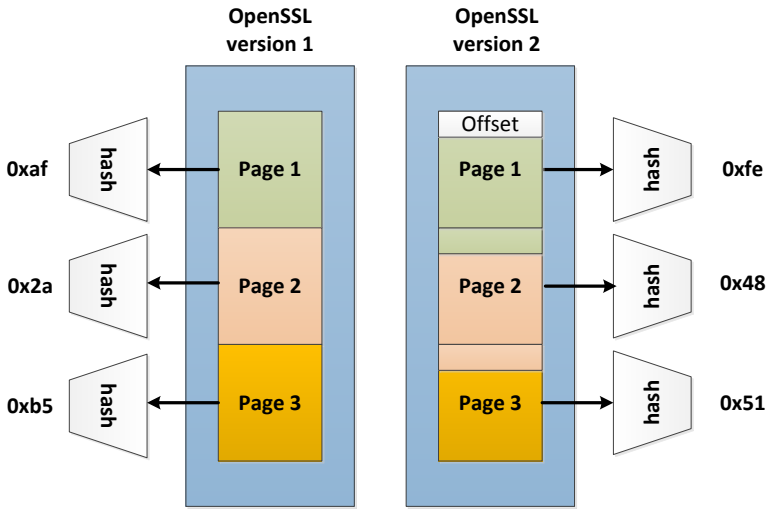


Figure 4.3: KSM when an offset introduced by a modification in the library causes differences on the hash operation.

to detect such outdated versions running in co-located VMs.

Thus, the knowledge of the version enables the adversary to choose the most

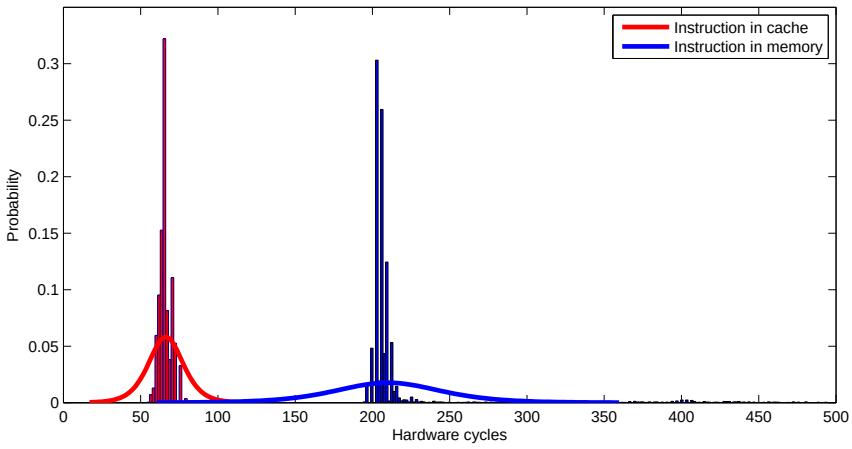


Figure 4.4: Reload time when a co-located VM is using the targeted code (red) and when it is not (blue) on KVM, Intel Xeon 2670

powerful attack against a specific library. Outdated versions of a specific crypto library are common since each OS has its own default installation of a specific version. For example, Ubuntu 12.04 uses `OpenSSL 1.0.1` and Ubuntu 14.04 uses `OpenSSL 1.0.1f`.

We work with seven `OpenSSL` versions; `OpenSSL 0.9.7a`, `OpenSSL 0.9.8k`, `OpenSSL 1.0.0a`, `OpenSSL 1.0.1c`, `OpenSSL 1.0.1e`, `OpenSSL 1.0.1f`, `OpenSSL 1.0.1g`. In the following we list some of the popular vulnerabilities in different libraries and versions [131]. These are flaws that directly affect the mentioned version, but of course, new flaws in more recent versions can be applied to the most outdated ones.

- `OpenSSL 0.9.7`: This is the most outdated version that should not be used under any circumstance. There are many attacks targeting this version, such as AES cache attacks, branch prediction attacks in RSA or attacks on PKCS [132].
- `OpenSSL 0.9.8k`: Vulnerable to Denial of Service attacks, Kerberos crash attacks or flaws in the handling of CMS structures.

- `OpenSSL 1.0.0a`: Vulnerable to Buffer Overrun attacks in TLS, vulnerable to modifications of stored session cache ciphersuites, DOS attacks due to GOST parameters, memory leakage due to failure of byte clearing in SSL3.0, vulnerable to Vaudenay's padding oracle attack [133], weaknesses in the PKCS code exploitable using Bleichenbacher's attack [134].
- `OpenSSL 1.0.1c`: Vulnerable to Lucky 13 attack [120], DOS attack due to failure in OCSP response, DOS attack on AES-NI supporting platform.
- `OpenSSL 1.0.1e`: Vulnerable to Flush+Reload ECDSA attack [31], and to Heartbleed attack [121].
- `OpenSSL 1.0.1f`: Vulnerable to Flush+Reload ECDSA attack [31], and to Heartbleed attack [121].
- `OpenSSL 1.0.1g`: Heartbleed fixed.

Heartbleed attack: The Heartbleed attack is a serious threat that was discovered in `OpenSSL`. The bug allows a malicious attacker to steal sensitive information used by SSL/TLS encryption. It compromises the secret keys used to identify the service providers. The bug was addressed in `OpenSSL 1.0.1g`, which means that previous versions of `OpenSSL` are still vulnerable to the attack. This compromises seriously those users who are still using an outdated version of the popular `OpenSSL` library. Our detection method allows an attacker to detect whether one of the vulnerable versions is being used.

4.5 Experiment Setup and Results

Our measurement setup mimics the cloud setup found in commercial CSPs. All experiments are performed on a machine featuring an eight-core Intel Xeon e5-2670

v2 CPU, running at 2.5 GHz. This CPU is also commonly used by the `m3` instances of Amazon EC2. It features 32 KB of private L1 cache and 262 KB of private L2 cache per core, as well as 20 MB of L3 cache, shared among the cores. KVM is used as hypervisor [135], whereas Ubuntu 12.04 is used for all the guest OSs. KVM's KSM feature is enabled for all experiments, scanning 100 pages each 200 milliseconds (the default values). Virtual machine manager provides a graphic interface for VMs.

In our evaluation setup, we create and use three virtual machines. The first one, VM1, acts as a *user* performing SSL/TLS connections with a library of its choice. The second VM, VM2, acts as *detector*: VM2 aims at detecting the library and specific version used by VM1. The last one, VM3, is used to simulate regular user load and therefore add various levels of noise depending on the **scenario**:

- **Noise-free:** In the first scenario, the victim, VM1, establishes SSL connections, the detector, VM2 executes our script to detect the library/version being called by the victim. No additional noise coming from a different VM is added, i.e. VM3 is idle.

- **Web browsing:** In the second scenario, again the victim VM is performing SSL connections while the detector VM tries to detect which library/version he is using. However, in this case, a third VM is performing web-browsing operations at the same time. It will run a script that opens a webpage every 5 seconds.

- **File sharing:** The third scenario is similar to the second one. In this case, while the victim is running SSL/TLS operations and the detector VM tries to run the detection script, a third Virtual Machine is going to be running a script automatically downloads a 6 KB PDF file every 5 seconds.

- **Media streaming:** In our last case, we want a scenario in which the amount

of CPU load introduced by the noise adder VM is significantly higher than in the previous cases. In this scenario, while the detector VM and victim VM are executing Flush+Reload and SSL/TLS connections respectively, at the same time a third VM continuously streams a movie. To model this situation, we used Netflix as our media streaming software. In particular, we installed the Netflix-desktop version for Unix. Note that the content streamed via Netflix is Digital Rights Management (DRM) protected, therefore encrypted. For encryption, Netflix uses AES-128 in counter mode. The cryptographic operation as well as the media decoding process create significant background noise.

We characterize our noise scenarios in terms of 3 parameters; the additional CPU load that is observed in the hypervisor due to this operations, the network traffic created and the number of cache references in a minute. Table 4.1 summarizes the observed values for each of the characteristics analyzed for the scenarios under consideration. We used the Linux tool `top` to observe the CPU load increase, and `jnettop` to observe the network traffic value `kn` bits per second, and finally we use `perf` to calculate the number of cache references made in each scenario in one minute. We observed that in terms of CPU load, both the web browsing scenario and the file downloading scenario are similar to an increase of 25%. As expected, the media streaming scenario creates more CPU load; almost 4 times more than the previous two scenarios with an increase of around 95%. The situation changes a little in terms of network traffic and cache references, with the file downloading scenario with slightly less noise. We observe an increase in these two parameters in the web-browsing scenario. Again for the media streaming scenario, we observe a substantial increase in both parameters, making us believe that the detector will decrease its efficiency under this scenario.

Table 4.1: Noise parameters in different scenarios

Noise Scenario	**CPU load**	**Network Traffic** (tx-rx)	**Cache usage** [references/min]
Noise free	0%	0	0
Web browsing	20-25%	3.67K-21K	115×10^6
File sharing	20-25%	600-1K	12.4×10^6
Streaming	95%	82K-5.8M	$2,300 \times 10^6$

In the experiments, we followed the approach of single-hit measurements meaning that we calculated the probability of successfully detecting a *single* function call at any time. It is important to note that the scanning process for all the crypto libraries used in the experiments takes only 5 seconds. Therefore, an adversary can easily amplify the detection rate of target library execution, even under heavy noise scenarios by scanning multiple times either continuously or in short intervals.

For our experiment setup, the threshold to distinguish between accessed (function resides in the cache) and not accessed (function resides in the memory) functions is 190 cycles. As it can be seen in Figure 4.4, this threshold is based on experimental measurements and is sufficient to distinguish the two Gaussian distributions. Note that this is a hardware specific threshold and will have to be tuned if the experiments are performed on a different machine. The threshold value is easily obtained by running the required library function on the target platform and measuring the execution time using RDTSC(P) for both memory and cache access. In order to ascertain that the function code resides in the memory and the obtained time belongs to the memory access, CLFLUSH instruction is used to flush the code from all levels of cache.

Table 4.2: Library detection experiment results under different levels of system background noise.

Library	**Detection success rate per scenario**			
	Noise free	**Web browsing**	**File sharing**	**Media streaming**
CyaSSL	90%	85%	85%	50%
OpenSSL	77%	76%	79%	48%
MatrixSSL	71%	72%	76%	41%
PolarSSL	91%	88%	84%	50%
GnuTLS	83%	91%	86%	51%
False positives	0	0.2%	0.4%	0.4%

Table 4.3: Library version detection experiment results under different levels of system background noise.

Library	**Noise free**	**Web browsing**	**File sharing**	**Media streaming**
OpenSSL 0.9.7a	79%	79%	78%	34%
OpenSSL 0.9.8k	85%	84%	87%	50%
OpenSSL 1.0.0a	83%	84%	82%	42%
OpenSSL 1.0.1c	85%	86%	85%	41%
OpenSSL 1.0.1e	80%	86%	84%	44%
OpenSSL 1.0.1f	76%	83%	80%	44%
OpenSSL 1.0.1g	82%	84%	90%	38%
False positives	0.14%	0.29%	0.29%	0.14%

Table 4.4: IP detection rate results.

Library	**Number of Connections per IP**	**Average Scan Time for 255 IPs (sec)**	**Average Detection Hits**
CyaSSL	1 tries	5.57	3.5
OpenSSL	3 tries	6.86	0.77
MatrixSSL	1 tries	3.4	2.66
PolarSSL	3 tries	7.33	1.54
GnuTLS	3 tries	9.7	1.31
False positives	0	0.2%	0.4%

4.5.1 Library Detection

To determine the efficacy of the library detection method we surveyed five crypto libraries, i.e. `OpenSSL 1.0.1 (OS)`, `GnuTLS 26 2.12.14 (OS)`, `MatrixSSL 3.6.1`, `CyaSSL3.0.0 and PolarSSL 1.3.7`. All the libraries are compiled as shared li-

braries. For the purposes of this attack, we assume that one user runs an SSL/TLS connection using one of these five libraries, and a second user has to detect which one was run. We use typical functions that are used in regular SSL/TLS connections (such as library initialization or context creation functions). We run a script that randomly chooses one of the libraries mentioned above and performs SSL/TLS connections using the following tools:

- **OpenSSL**: `s_client` and `s_server` tools provided in the `OpenSSL` library.
- **PolarSSL**: `ssl_client1` and `ssl_server` programs provided by `PolarSSL` to perform the SSL/TLS connection.
- **GnuTLS**: anonymus authentication server and anonymus authentication client examples provided by `GnuTLS` in [136, 137] for our test SSL connection.
- **CyaSSL**: client and server examples provided by `CyaSSL` to perform the SSL/TLS connections.
- **MatrixSSL**: client and server apps provided by `MatrixSSL` that establish a SSL connection between them.

We recorded a total of 100 calls per library. Results are presented in Table 4.2. The success rate represents the percentage of correctly detected libraries. Incorrectly detected libraries, i.e. false positives, are presented in the last row. The first thing to notice here is that we have an exceptionally low false positive rate, ranging from 0 false positives observed in the noise-free scenario to 2 false positives in both the third and forth noisy scenarios out of 500 calls. In the best case (noise-free scenario) the success rate ranges between 70% and 90% depending on the library. Furthermore, in the low noise scenarios these results do not change, meaning that this type of noise would not affect the results. Finally, we observe that when the heavy load scenario

is applied, we do not observe a significant increase in the wrong predictions, but we observe a decrease in the success rate of the library detection. Detection rate goes down from almost 90% to 50% in the best cases, and from 70% to 41% in the worst cases. This means that even when the target is residing in a physical machine with a heavy load, the detector can still detect the target library 50% of the time.

Another thing to note hereis that since we are using the `OpenSSL` version provided by the OS in the download script noise scenario, we are detecting some additional calls of `OpenSSL` from the downloading step. However since these calls are made by another VM and could be considered both correct or incorrect predictions, we did not include them in the results.

4.5.2 OpenSSL Version Detection

The second scenario is where we have a version detection tool. We used 7 different `OpenSSL versions`: `OpenSSL 0.9.7a`, `OpenSSL 0.9.8k`, `OpenSSL 1.0.0a`, `OpenSSL 1-.0.1c`, `OpenSSL 1.0.1e`, `OpenSSL 1.0.1f` (previous to Heartbeat fix) and `OpenSSL 1.0.1g` (heartbeat fixed). We have to be more specific here since we have to look for functions that are different across libraries because otherwise, KSM would merge them even if they are from different libraries. However in all the libraries analyzed, the offset of the function that is being tested (SSL_CTX_new) with respect to the beginning of a page is different, therefore KSM will never merge them, even when the function is the same. SSL_CTX_new is a function that is always called in a new SSL connection. We use the applications `s_server` and `s_client` that `OpenSSL` provides for testing SSL connections. We run a script that randomly chooses one of the above-mentioned versions of `OpenSSL` and performs a TLS connection.

Again we record 100 calls of each one of the version, 700 in total, in each one of the scenarios. Again we are going to run 4 types of experiments: noise-free,

web-browsing, download noise, and heavy Netflix load. Results are presented in Tables 4.3. We see a similar pattern to the one we saw in the library detection. For noise-free and low noise scenarios we observe that the success probability varies from 76% to 90%. We have to remark that in this experiments we are not using the OS provided version of `OpenSSL` (plain 1.0.1) since we did it in the previous tests and therefore we do not detect the file downloads. One could always include this version to the detectors and still, be able to detect it. Another observation is that again, the wrong predictions are fairly low, with 2 wrong predictions in 700 library calls in the worst case. Finally, we observe that when a heavy load is present in the server, the success rate decreases to 50% in the best case and to 34% in the worst case. This shows again that even though heavy load decreases the success rate, one could still detect one library call out of three in the worst case.

4.5.3 IP Detection

For the IP detection experiments, we chose functions that run when a TLS communication handshake is triggered in each particular library. After the initial library detection stage, we ran the IP detector script and the TLS handshake detector to discover the co-located target VM's IP address. To be able to scan a wide range of IP addresses in a short time, we used the timeout command with the TLS client handshake process to eliminate IP addresses with no active TLS servers. Note that this timeout value had to be short enough to allow fast scanning of a large group of IP addresses but also provide enough time to allow the TLS client to run necessary pre-connection processes. To meet these two criteria, we tried different timeout values ranging from 0.001 milliseconds to 1 second for different libraries and determined that 0.01 milliseconds was optimal.

As for TLS Client, we experimented with different TLS clients provided with

libraries and determined that the OpenSSL client was the fastest one amongst the inspected libraries. Therefore for the IP detection stage, we used OpenSSL TLS client to trigger TLS handshake for all libraries. Table 4.4 shows the time it takes to scan 255 IP addresses in the subnet to discover co-located victim VM, as well as the average detection, hits from the co-located attacker VM during the IP detection stage. As seen from the table, for some libraries, namely OpenSSL, PolarSSL and GnuTLS, 3 TLS connection attempts per IP is used while for CyaSSL and MatrixSSL only 1 per IP is used. This is done in order to increase the detection rate while keeping the average scan time within acceptable limits. As the results show, all libraries except the OpenSSL have average detection hits over 1, meaning that they are always detected when the TLS handshake is triggered. As for the OpenSSL, even when 3 connections per IP are established, the detection rate is 0.77 on average and cannot be further increased by repeated measurements in acceptable scan time limits. We believe that this is due to the fact that OpenSSL handles TLS handshake faster than the detector has a chance to detect the handshake. The fact that OpenSSL TLS client was fastest to establish connections in our test and the scan time for 255 IPs was fastest for OpenSSL supports this reasoning.

4.6 Countermeasures against the Flush+Reload

There are many proposed countermeasures against the Flush+Reload attack and of it's variants that rely on memory de-duplication. In this chapter, we list some of these countermeasures and give a brief explanation.

Restricting the clFlush Access

Our detection method is based on the detector's ability to flushing specific memory lines from the cache with the `clflush` command. Prohibiting the usage of the `clflush` command would prevent the attacker from implementing the Flush+Reload attack. Note that disabling the clflush instruction will disrupt memory coherence in devices where memory coherence is not supported. Also note that `clflush`-like instructions can still be substituted by cache priming techniques, as in [138].

Disabling Memory De-duplication

Disabling de-duplication prevents the Flush+Reload based detection of executed code. Even partial disabling, e.g. by marking crypto libraries (or any critical software) to be excluded from de-duplication, can prevent the library detection with minimal effect on performance. The main disadvantage of this countermeasure is the lack of memory usage optimization, especially in multi-tenant systems. Keep in mind that with de-duplication, it is shown to be possible to run over 50 Windows XP guest VMs on a server with 16 GB RAM [15]. Also, note that other spy processes like Prime+Probe may still succeed even when de-duplication is turned off.

Dedicated Hosting/Single Tenant Systems

Public clouds like Amazon EC2 can provide customers with dedicated hosts meaning that only the VMs from a single customer will reside on the underlying hardware platform and no hardware resource will be shared between customers. In this scenario, no cache side-channel attacks can be implemented, since the attacker cannot co-locate with a victim anymore.

Diversifying the Execution Code

One possibility to mitigate cache side-channel attacks is to create different and unique program traces (that perform identical computations) for different executions. This countermeasure, proposed in [139], will prevent the Flush+Reload technique since the specific location of the function that the attacker wants to monitor would be different for different users (and thus, libraries would never be de-duplicated).

Degrading the Granularity of Timers

As cache timing side-channel attacks base their procedure on the accuracy of `rdtscp`-like timers, an easy solution that cloud hypervisors can adopt is to eliminate the access to fine-grain timers from guest VMs. Alternatively, as stated in [140], fine-grain timers could introduce a certain amount of noise so that cache side-channel attacks are no longer applicable. Note that this would only make the Flush+Reload harder under certain attack scenarios. If the attack scenario permits, the attacker can use **increment counters** by continuously incrementing a variable in a separate execution thread and use the counter value as the timer.

Cache Partitioning

As suggested in [141], partitioning the cache is a hardware solution that would mitigate any kind of cache side-channel attack. If the attacker and the victim have associated different portions in the shared level of cache (effectively creating *private* caches), no cache-based attacks are possible. Therefore allocating parts of the cache to a specific VM or a process, even when de-duplication is enabled, would avoid any cache leakage between VMs/processes. The downside of cache partitioning is that

the cache utilization associated with each process decreases significantly, resulting in serious performance penalties.

Randomizing Cache Loads

Another countermeasure is to add a random offset to the location of the sensitive data when the CPU fetches data to the cache, as proposed in [141]. By adding an offset, the physical memory would only have one copy of a shared page, but it would add a random offset when being loaded into the cache. This random offset is private for each processor VM. Therefore, an attacker would have to know the private offset of the victim's process to be able to access the same data and the cache set.

4.7 Conclusion

In this chapter, we presented a detection method to identify the execution of pieces of software on a target's VM across co-located VMs. While the technique is generic and applies to cross-VM settings where de-duplication is enabled, our experiments focused specifically on crypto libraries. We believe that this is a highly relevant use case for the detection method since it enables an attacker to covertly carry out a discovery phase with high precision and great speed.

We demonstrated the viability of the detector by identifying the crypto library and the particular version used by a target. Our work shows that identifying a specific library version being used by a co-located tenant is possible. This enables an attacker to focus on the most viable vulnerability. One clear example is the Heartbleed bug, which was not fixed until `OpenSSL 1.0.1g` and allows an attacker to the extraction of private information.

We presented experiment results on OpenSSL versions under various noise sce-

narios. We observed that in low-noise scenarios the detection rate is up to 90%, whereas in heavy load scenarios the detection rate reaches up to 50%. Nevertheless, we would like to emphasize that even in the worst case scenario with a heavy load, the attacker gains knowledge about the used library after two or three library calls. Finally, we outlined a number of countermeasures which would render the detection technique obsolete.

Chapter 5

Co-location Detection on Cloud

In this chapter, we focus on the problem of co-location as a first step of conducting Cross-VM attacks such as Prime+Probe or Flush+Reload in commercial clouds. We demonstrate and compare three co-location detection methods namely, cooperative Last-Level Cache (LLC) covert channel, software profiling on the LLC and memory bus locking. We conduct our experiments on three commercial clouds, Amazon EC2, Google Compute Engine, and Microsoft Azure. Finally, we show that both cooperative and non-cooperative co-location to specific targets on the cloud is still possible on major cloud services.

5.1 Motivation

As the adoption of cloud computing continues to increase at a dizzying speed, so has the interest in cloud-specific security issues. A new security issue due to cloud computing is the potential impact of shared resources on security and privacy of information. An example is the use of caches to circumvent ASLR [142], one of the most common techniques to prevent control-flow hijacking attacks.

Several other works target the exploitability of cryptography in co-located sys-

tems under increasingly generic assumptions. While early works such as [34] required attacker and victim to run on the same CPU core, latest works [43, 42] work across cores and managed even to drop the memory de-duplication requirement of Flush+Reload attacks [31, 91, 2, 4]. Besides extracting cryptographic keys, there are plenty of other security issues explored in other related studies. Irazoqui et al. [7] study the potential of reviving the partially fixed Lucky 13 attack [120] by exploiting co-location.

All of the above attacks rely on the attacker's ability to co-locate with a potential victim. While co-location is an immediate consequence of the benefits of cloud computing (better utilization of resources, lower cost through shared infrastructure etc.), whether *exploitable* co-location is possible or easy has so far not been studied in detail. In his seminal work, Ristenpart et al. [81] studied the general feasibility of co-location in Amazon EC2, the most popular public cloud service provider (CSP) then and now, in detail. However, the cloud landscape has changed significantly since then: The EC2 has grown exponentially and operates data centers around the globe. A myriad of competitors have popped up, all competing for the rapidly growing customer base [143]. CSPs are also more aware of the potential security vulnerabilities and have since worked on making their systems leak less information across VM boundaries.

Furthermore, in their experiments, both co-located parties were colluding to achieve co-location. That is, both parties were willingly involved in communicating through the CPU cache with each other to verify co-location. While being of high importance to show the feasibility in the first place, trying to co-locate with a specific and most likely unwilling target can be considerably harder. Since that initial work, until very recently only a little work has dealt with a more detailed study on the difficulty of co-location. Therefore, we believe, the problem of co-location on

cloud requires further in-depth analysis examining different detection methods under diverse scenarios and access levels for the attacker. With this motivation, we have discovered three co-location detection methods as; LLC Covert Channel, Software Profiling on LLC, Memory Bus Locking. Below, the details of these methods are provided.

5.2 Threat Models

Here we briefly outline two attacks scenarios for cross-VM attacks on public clouds. The main difference between the two scenarios is whether the target is predetermined or not. As we shall see, this makes a significant difference in terms of the requirements and cost of a successful attack. We provide concrete examples for both scenarios.

5.2.1 Random Victim Scenario

In the random victim scenario, we outline attack in four steps as follows:

1. **Co-location:** The attacker spins instances on the cloud until it is determined that the instance is not *alone*; i.e. is co-located with another VM. Here the goal is to maximize the probability and thereby reduce the cost of co-locating with a viable target. Cheaper instances that use fewer CPU cores tend to share the same hardware in greater numbers. Therefore these instances have a better chance of co-location with other customers. Since we do not discriminate between targets, this step is rather easy to achieve.

2. **Vulnerable Software Identification:** The attacker detects a software package in the co-located VM vulnerable to cross-VM attacks by monitoring corresponding LLC sets of libraries, e.g. an unpatched version of a crypto library.

Cache access/performance and more broadly fingerprinting based techniques do exist in the literature to make successful attacks in the cloud environment [19, 94, 3]. Here, instances with a lower number of tenants are less noisy therefore have a higher success rate of library detection and the actual attack.

3. **Cross-VM Secret Extraction:** Here the attacker runs one of the cross-VM attacks [5, 43] on the identified target. By exploiting cross-VM leakage the attacker would be able to recover sensitive information ranging from specialized pieces of information such as cryptographic keys to higher level information such as browsing patterns, shopping cart, system load or any sensitive information of value. Noise plays a significant role in the reliability of the extraction technique. Since co-location (first step) is easy to achieve, it is (almost) always advisable to opt for a less populated low noise instance to improve the chance of a successful attack in the later steps.

4. **Value Extraction:** The result is some sensitive information that can be turned into value with additional mild effort. For example, some information is valuable in its own right and can be converted into money with little or no effort, e.g, bitcoins, credit card information, credentials for online banking. Some others require further effort such as TLS session encryption key (secret key), e.g. for a Netflix streaming session. If the recovered secret is a private key of a public key encryption scheme (e.g. RSA secret key used a TLS handshake) the attacker needs the identity of the owner (website/company) to have further use for the secret key. In this case, he may check the private key against public key repositories for noise correction and target identification.

5.2.2 Targeted Victim Scenario

This is the complementary scenario where we are given some identification information about the target.

1. **IP Extraction:** The attacker wants to focus its cycles on a server or group servers that belong to an individual, cloud-backed business, e.g. Dropbox or Netflix, or group/entity, e.g. dissidents of a political party. Here we assume that the attacker is capable of resolving the identification information to an IP or group of IPs of the target. In practice, this can be achieved rather easily by using public information and by using simple commonly available network tools such as `traceroute`/`tracepath`, `nmap` etc.

2. **Targeted Co-location:** The attacker creates instances on the cloud until one is co-located with the target instance on the same physical machine. The identification information of the victim, e.g. IP address, is used for co-location detection. For instance, using the IP the attacker can query the server creating CPU load and then run co-location tests. While co-location detection will be easier in this scenario due to the trigger; we will need many more trials to land on the same physical machine as the victim[1]. Nevertheless, we can accelerate targeted co-location by *searching*, for instance, only in the same cloud region as the victim instance using the publicly available AWS IP lists [144]. Note that while we applied this method to AWS, it also holds true for other public cloud services. Further, we can obtain finer grain information about the target's location simply by running `traceroute` or `tracepath` on the victim IP.

[1]Note that if the physical machine is already filled with the maximum number of allowed instances, then co-location may not be possible at all. In this case, a clever albeit costly strategy would be to first mount a denial of service attack causing the target instance to be replicated and then try co-locating with the replicas.

3. **Vulnerable Software Identification:** Since we know the identity of our target, it is safe to assume that we have some rudimentary understanding of the victim's setup including OS, communication and security protocols used etc. Even if this is not the case, it would be possible to run a discovery stage to survey the victim machine using its IP and by detecting process fingerprints through cross-VM leakage.

4. **Value Extraction:** The attacker exploits cross-VM leakage to recover sensitive information. Further processing may allow enhancing the quality of the recovered data using publicly available information. For instance, a noisy private key can be processed with the aid of the public key contained in the certificate belonging to the target to remove any imperfections.

5.3 LLC Covert Channel

The LLC is shared across all cores in most modern CPUs and is semi-transparent to all VMs running on the same machine. By semi-transparent, we mean that all VMs can utilize the entire LLC but cannot read each other's data. We exploit this behavior to establish a covert channel between VMs in the cloud. The covert channel works by two VMs writing to a specific set-slice pair in the LLC and detecting each other's accesses. LLC set address can easily be deduced from the virtual addresses available to VMs using hugepages as done in [43, 42, 5]. The cache slice, on the other hand, cannot be determined with certainty unless the slice selection algorithm of the CPU is known. However, the covert channel can still work by priming more sets and accessing lines that go to the targeted set, regardless of its slice.

5.3.1 Prime+Probe in LLC

In the LLC, the number of lines required to fill a set is equal to the LLC associativity. However, when multiple users access the same set, one will notice that fewer than 20 lines are needed to trigger evictions. By running the following test concurrently on multiple instances, we can verify co-location. The test works as follows:

- Calculate the set number by using the address bits that are not affected by the virtual to physical address translation. Prime a memory block M_0 in the set.

- Access more memory blocks $M_1, M_2, \ldots, M_n$ that go to the same set. Note that since the slice selection algorithm for the specific CPU is necessary to address a set/slice pair with certainty, the number of memory blocks n needs to be larger than the set associativity times the number of slices.

- Access the memory block M_0 and check for eviction from the LLC. If evicted, we know that the required b memory blocks that fill the set are among the accessed memory blocks $M_1, M_2, \ldots, M_n$.

- Starting from the last memory block accessed, remove one block and repeat the above protocol. If M_0 still has high access time, M_i does not reside in the same slice. If b_0 is now located in the cache, we know that b_i resides in the same cache slice as b_0 and therefore go to the same set.

- Once the b memory blocks that fill a slice are identified, we just access additional memory blocks and check whether one of the primed b memory blocks has been evicted, indicating that they collide in the same slice.

The covert channel works by continuously accessing data that goes to a specific cache set and measuring the access time to determine if a newly accessed data has

evicted an older entry from the set. Due to this continuous cache line creation, when the second party makes accesses to the monitored set, they are detected. In general, if there is no noise present, the number of lines that can go to a set without triggering an eviction is equal to the associativity of the cache, assuming a first-in-first-out (FIFO) cache replacement policy is employed.

When two VMs try to fill the same set, they have to access less number of data blocks to fill the specified cache hence detecting the co-location. Using the number of blocks necessary to fill a specific set with and without another instance interfering, we calculate a co-location confidence ratio.

5.4 Software Profiling on LLC

The software profiling method works in a realistic setting with minimal assumptions. The method works in a non-cooperative scenario where the target does not participate in covert communication and continues its regular operation. The method does not require memory de-duplication or any form of shared libraries. It employs the Prime+Probe to monitor and profile a portion of the LLC while a targeted software is running. As for the memory addressing, we profile the targeted code address as a relative address to the page boundary. Since the targeted library will be page aligned, the target code's relative address (the page offset) will remain the same between runs. Using this information, we can reduce our search space in the detection stage. Therefore, we need to monitor only 320 different set-slice pairs such as $X \mod 64 = Y$ where X is 320 different set numbers (since we have 10 cores and 32 different set numbers satisfying the equation) and Y is the first 6 bits (the first 6 bits of the LLC set number is directly converted to physical address) of the set number for the desired function.

For RSA detection, the slice selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC at a reasonable time. Without the algorithm, it would take too much time to monitor potential cache sets. For our experiments, we have used the algorithm that was reverse engineered in [5].

In summary, there are two stages to the software profiling on LLC;

- **Profiling Stage:** The first step of the profiling is to monitor the targeted LLC sets while the profiled code, the software is not running. The purpose of this stage is to measure the idle access time of 20 lines for each set to have a threshold to detect whether there is a cache miss or not in the next stage.

- **Detection Stage:** We send RSA decryption requests to candidate IPs in order to discover the IP address of the victim. After triggering the decryption we begin to monitor the portion of LLC to detect accesses triggered by the decryption. If we detect accesses in targeted set-slice pairs then we know that the correct IP address is found. As a double check, in addition to the RSA detection, we also detect AES encryption. In order to so, we monitor another portion of the LLC where the AES T-tables potentially reside. And if the victim is co-located with the attacker, we can detect and monitor these T-table accesses.

5.5 Memory Bus Locking

This detection method uses the overall performance degradation of a system caused by the memory bus locking to detect potential co-location. The memory bus locking method exploits a cache coherency subroutine used to ensure atomicity of certain operations. In the following, we explain these special instructions.

5.5.1 Atomic Operations

Atomic operations are an integral part of modern computing systems. They are used to implement mutex and semaphores, sustain cache coherency, avoid race conditions and even serve interrupts. These instructions cannot be uninterrupted during their execution and appear to the rest of the system as instant. As explained in more detail in Section 2.4, the x86 architecture has many atomic instructions that can be executed atomically with a lock prefix. Moreover, the XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use.

Exotic Atomic Operations: are special atomic operations that work on uncacheable memory and trigger system-wide memory bus locking. The fact that some addresses are uncacheable can be due to data in the operand spanning multiple cache or memory lines as in the case of a word-tearing or it can be due to the operand address corresponds to a reserved space on the physical memory. In any case, an Exotic Atomic Operation triggers memory bus locking and flushing of the ongoing memory operations in all of the CPUs of the system to ensure atomicity and data coherence. As expected, this results in a heavy performance penalty to the overall system, especially the memory transactions. Moreover, since instructions might take different clock cycles to execute, in order to maximize the flushing penalty, all atomic instructions available to the platform should be tested to see how long each instruction takes to complete. Since the flushing is succeeded with the atomic operation itself, the longer the instruction executes, the worse the performance hit to the system becomes. In order to maximize the performance degradation, we tested all atomic instructions available to the platforms and measured how long each instruction takes to execute. In our experiments, we have used the XADDL instruction since it resulted in the strongest penalty to our test platforms. By triggering the memory bus locking via exotic atomic operations, we slowdown a server process

running in the cloud and detect co-location **without cooperation** from the victim side.

5.5.2 Cache Line Profiling Stage

Our attack is CPU-agnostic and employs a short, preliminary cache profiling stage. This stage eliminates the need for information like the cache line size and the cache access time. Our purpose here is to obtain data addresses that span multiple cache lines hence triggers a bus lock. First, we allocate a block of small, page-aligned memory using *malloc*. After the allocation, we start performing atomic operations on this block in a loop of 256 since no modern cache line is expected to be larger than 256 bytes. In each loop, we move our access pointer by one and record atomic operation execution times. When we observe a time larger than the pre-calculated average, we record the address. After all 256 addresses are tested, we obtain a list of addresses that span across multiple cache lines. Later during the locking stage, we operate only on these addresses rather than a continuous array, making the attack more efficient.

5.5.3 Dual Socket Results

Memory bus locking works on systems with multiple CPU sockets. Even further, our tests reveal that the bus locking penalty clearly reveals whether the target and the attacker run in the same socket or not. As seen in Figure 5.1, the memory access time is clearly distinguishable between the same socket and different socket locks on a dual socket system with two Intel Xeon E5-2609 v2 CPUs. Note that this information is important for the attacker since an architectural attack using the LLC requires the attacker and the target to be running in the same socket.

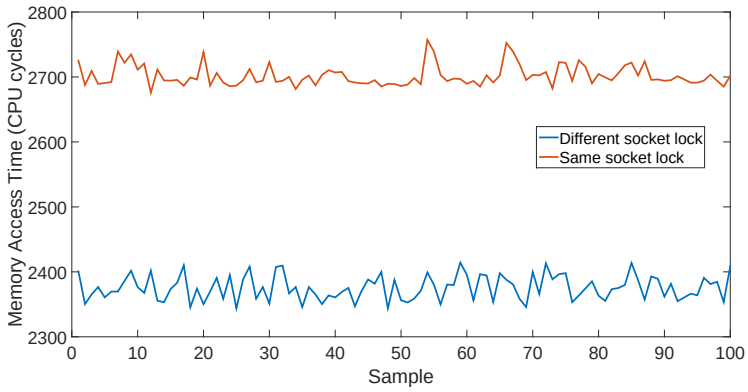


Figure 5.1: The memory access times during a bus lock triggered with the XADDL instruction showing when the attacker resides in the same socket (red) and different sockets (blue).

5.6 Commercial Clouds Experiment Results

In all three aforementioned commercial clouds, we have launched 4 accounts with 20 instances per account, achieving co-location in each cloud. Also, note that we only classify the instances running in the same CPU socket as co-located and ignore the ones running on different sockets.

Amazon EC2: In Amazon EC2 we used **m3.medium** instance types that have balanced CPU, memory and network performance. This instance type has 1 vCPU, 3.75 GB of RAM and 4 GB of SSD storage. According to Amazon EC2 Instance Types web page [145], these instances use 10 core Intel Xeon E5-2670 v2 (Ivy Bridge) processors.

Out of 80 instances launched, we have obtained 7 co-located pairs and one triplet verified by the tests. Moreover, we have tried to co-locate with instances that have launched previously. Surprisingly, we have been able to co-locate with instances that have launched 6 months prior.

Google Compute Engine: In GCE, we used **n1-standard-1** type instances running on 2.6 GHz Intel Xeon E5 (Sandy Bridge), 2.5 GHz Intel Xeon E5 v2 (Ivy

Bridge), or 2.3 GHz Intel Xeon E5 v3 (Haswell) processors according to [146]. Out of 80 instances launched, we have obtained only 4 co-located pairs.

Microsoft Azure: In Azure, we used **extra small A0** instance types with 1 virtual core, 750 MB RAM, maximum 500 IOPS, and 20 GB disk storage that is specified as neither SSD nor HDD [147]. Out of 80 instances launched, we have obtained only 4 instances that were co-located. However, this was partly due to the highly heterogeneous CPU pool that Azure employs. Our first account had instances with AMD Opteron CPUs while the second had Intel E5-2660 v1 and the last two had Intel E5-2673 v3. Naturally, we could only achieve co-location among instances that have the same CPU model. Out of 40 Intel E5-2673 v3 instances, we detected 4 co-located instances.

5.6.1 LLC Covert Channel

In the following, we present the results of the LLC covert channel experiments. The confidence ratio is highest at 1 as seen in Figure 5.2. There are 8 instances (meaning 4 pairs) that have higher than 50% confidence ratio among 80 and the co-located pairs are found by binary search at the end. Hence, it is confirmed that they are indeed co-located with each other.

5.6.2 LLC Software Profiling

We conducted the LLC Software Profiling experiments on the co-located Amazon EC2 instances with 10 core E5-2670 v2 processors. As for the software target, in order to demonstrate the versatility of the attack, we chose the RSA (Libgcrypt version 1.6.2) that uses sliding window exponentiation and the AES (OpenSSL version 1.0.1g, C implementation) that uses T-tables. Note that the detection method is not limited to these targets since the attacker can run and profile any software

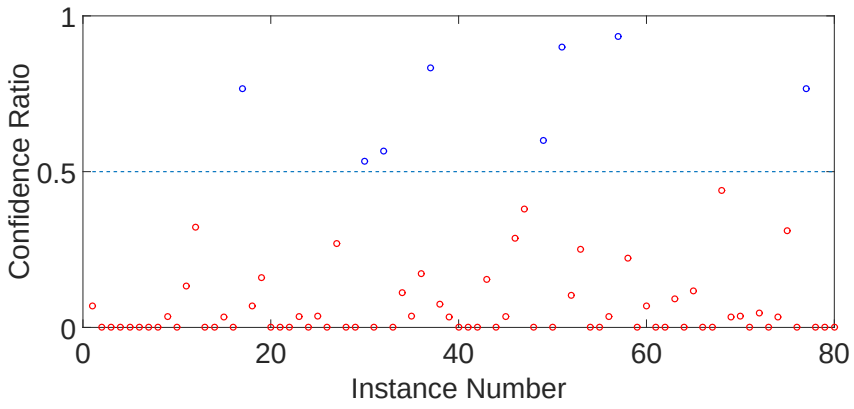
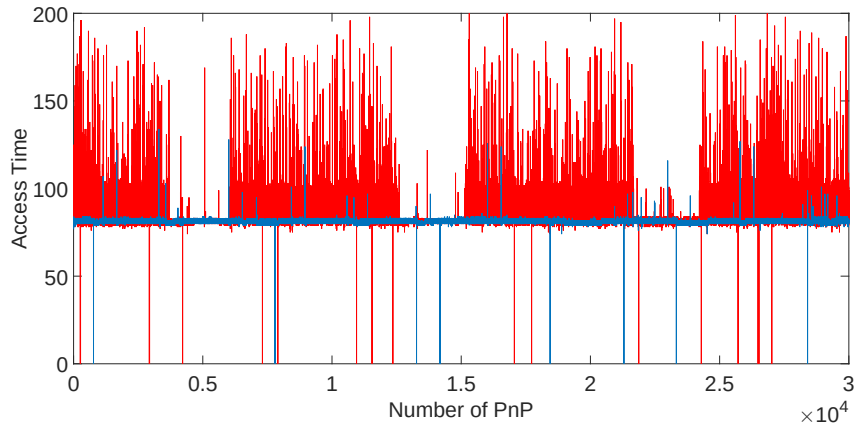


Figure 5.2: GCE LLC Test Confidence Ratio Comparison

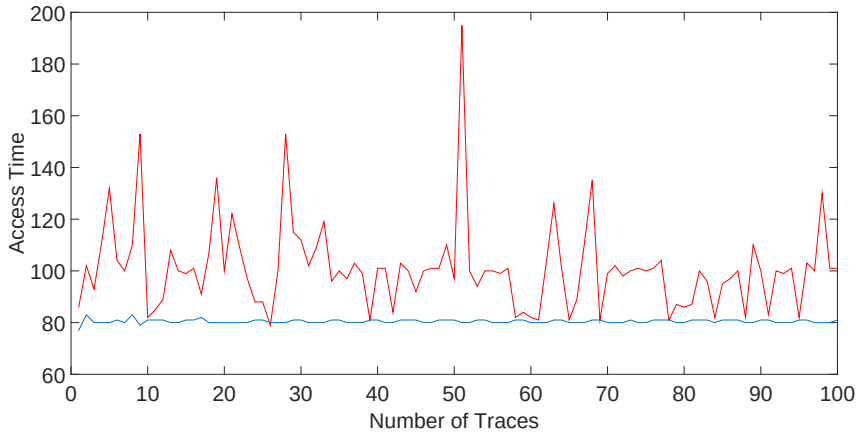
which uses shared library in his instance and perform the attack.

For RSA detection, the slice selection algorithm of the CPU is required to locate the targeted multiplication code in the LLC within a reasonable time. In our experiments, we have used the algorithm that was reverse engineered by Inci et. al in [5].

The first step of the profiling is to monitor the targeted LLC sets while the profiled code, RSA is not running. After the regular operation of sets is observed, the RSA request is sent to several IP addresses, starting from the attacker's own subnet. As soon as the request is sent, the profiling starts and traces are recorded by the Prime+Probe. If the RSA decryption is running on the other VM, the pattern of multiplication can be observed as in Figure 5.3. In general, the multiplication is performed between 2000-8000 traces. In these traces, we look for the delta of two profiles for each set-slice pair. In Figure 5.4, the difference between the two profiles is illustrated for two co-located instances. Both figures show that there are two set-slice pairs with significantly higher access times (4-8 cycles) on average of 10 experiments. Hence, it can be concluded that these two sets are used by RSA decryption and this candidate instance is probably co-located with the attacker.



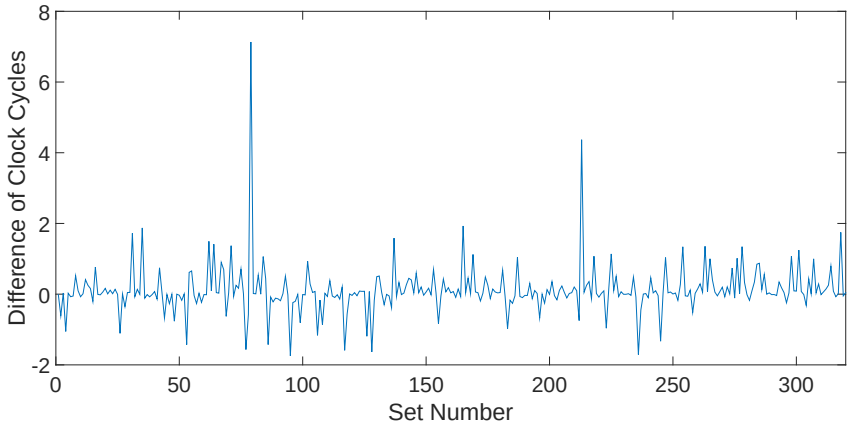
(a) RSA Pattern



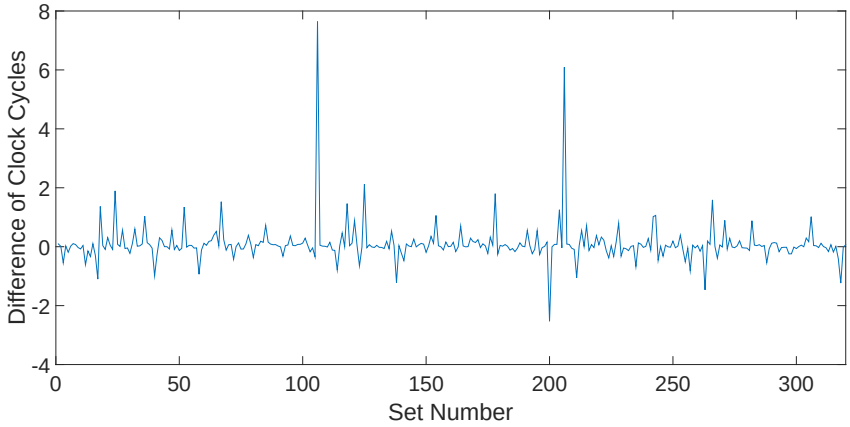
(b) AES Pattern

Figure 5.3: Red and blue lines represent idle and RSA decryption/AES encryption access times respectively

After we obtain IP addresses of several potential co-located instances, we trigger AES encryption by sending random ciphertexts and at the same time monitoring the LLC. For this part of the detection stage, since AES encryption is much faster than RSA decryption we can only catch one access to monitored T-table position. Hence, we send 100 AES encryption requests to each instance in the IP list. If we observe 90% cache miss for one of the set-slice pairs, it can be concluded that the AES encryption is performed by the co-located instance, as seen in Figure 5.3(b).



(a) RSA Analysis for the first co-located instance



(b) RSA Analysis for the second co-located instance

Figure 5.4: The difference of clock cycles between base and RSA decryption profiling for each set-slice pairs over 10 experiments

5.6.3 Memory Bus Locking

The performance degradation due to the memory bus locking is application specific. Therefore we tested various applications as seen in Table 5.1 to see how each one is affected. As expected, applications with frequent memory accesses are more affected by locking. For example, the GnuPG which mostly uses the ALU and does seldom memory access slowed down by 29%. An Apache web server that frequently loads content from memory, on the other hand, has a slowdown by the factor of 4.28.

In addition to specific software performance degradation, we also measured the

Table 5.1: Application slowdown on an Intel Xeon 2640 v3 due to Memory Bus locking triggered on a single core.

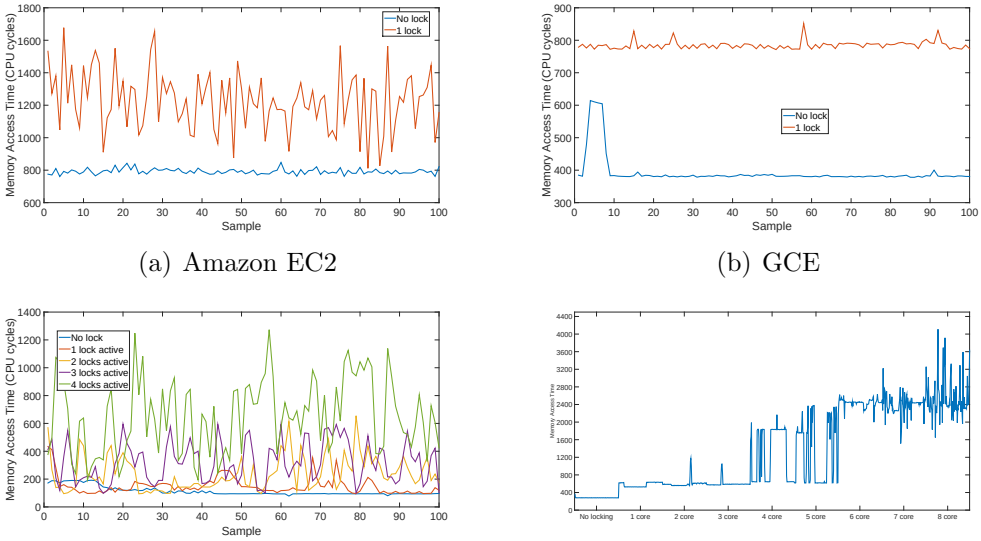
Process	Normalized Execution Time
Apache	4.28x
PHP	0.1x
GnuPG	0.29x
HTTPerf	0.29x
Memory Access	5.38x
RAMSpeed int	5.01x
RAMSpeed fp.	4.88x
Media Stream	2.36x

effect of multiple locks executed in parallel. To do so, we have used the `openmp` parallel programming API [148] and ran the lock in multiple threads. Figure 5.5(d) shows the memory access times when 0 to 8 locks run in parallel. As the figure shows, the first lock does slow down the memory accesses by 100% while the second and third locks do not further degrade the memory performance. However, after the fourth and fifth locking threads, we observe even stronger degradation.

5.6.4 Comparison of Detection Methods

As explained in Section 5.2, co-location can be exploited in both random and targeted victim scenarios. An attacker can directly look for attack vectors to steal information from her neighbors or go after a specific target and spin up instances until co-located. However, if the detection method does not provide reliable results, the attacker can discard the co-located instances or even have false positives due to noise. Therefore a useful and efficient co-location detection method is essential.

The Table 5.2 shows that all three methods inspected in this study work with high accuracy in a real commercial cloud setting. All methods work with minimal requirements, no hypervisor access or specific hardware. In comparison, while the



(a) Amazon EC2

(b) GCE

(c) MicrosoftAzure

(d) Lab setup using Intel Xeon E5-2640 v3

Figure 5.5: Memory access times with and without an active memory bus lock of a) Amazon EC2 m3.medium instance b) GCE n1-standard1 instance c) Microsoft Azure A0 instance d) Lab setup (Intel E5-2640 v3)

Table 5.2: Comparison of co-location detection methods. *OPD: Observed Performance Degradation

Detection Method	Worst Case	Average	Best Case
Memory Bus Locking OPD*	0.1x	3.28x	6.1x
LLC Covert Channel	53%	73.5%	93%
LLC Software Profiling	50%	70%	90%

memory bus locking has the least clear co-location signal in the worst case, the other two methods are more prone to the LLC noise. Also, as seen in Table 5.1 the memory bus locking gives more reliable results with applications with frequent memory accesses. For the uncooperative co-location scenario, depending on the workload of the target instance, one can use either the memory bus locking or the software profiling to detect co-location with high accuracy.

5.7 Conclusion

In conclusion, we represent three co-location detection methods working in the three most popular commercial clouds (Amazon EC2, Google Compute Engine, Microsoft Azure) and compare their efficiencies. In addition, for the first time, we have achieved targeted co-locations in Amazon EC2 Cloud by applying the LLC software profiling for AES and RSA processes. For the memory bus locking method, we have observed that frequent memory accesses lead to more significant degradation. As for the cache covert channel, we show that the method works in a cooperative scenario with high accuracy. And finally, we presented the LLC software profiling technique that can be used for a variety of purposes including co-location detection without the help of memory de-duplication or cooperation from the victim side.

Chapter 6

RSA Key Recovery on Cloud

Cloud services keep gaining popularity despite security concerns. While non-sensitive data is easily trusted to the cloud, security-critical data and applications are not. The main concern with the cloud is the shared resources like the CPU, memory and even the network adapter that provide subtle side-channels to malicious parties. We argue that these side-channels indeed leak fine-grained, sensitive information and enable key recovery attacks on the cloud. Even further, as a quick scan in one of the Amazon EC2 regions shows, a high percentage -55%- of users run outdated, leakage prone libraries leaving them vulnerable to mass surveillance.

The most commonly exploited leakage in the shared resource systems stems from the cache and the memory. High resolution and the stability of these channels allow the attacker to extract fine-grained information. In this chapter, we develop a novel co-location detection method and employ the Prime+Probe attack to retrieve an RSA secret key from a co-located instance in Amazon EC2. Finally, we employ noise reduction to deduce the RSA private key from the monitored traces. By processing the noisy data we obtain the complete 2048-bit RSA key used during the decryption.

6.1 Motivation

Cloud computing services are more popular than ever with their ease of access, low cost, and real-time scalability. With increasing adoption of cloud, concerns over cloud-specific attacks have been rising and so has the number of research studies exploring potential security risks in the cloud domain. The main enabler for cloud security is the seminal work of Ristenpart et al. [81]. The work demonstrated the possibility of co-location as well as the security risks that come with it. The co-location is the result of resource sharing between tenant Virtual Machines (VMs). Under certain conditions, the same mechanism can also be exploited to extract sensitive information from a co-located victim VM, resulting in security and privacy breaches. Methods to extract information from VMs have been intensely studied in the last few years however remain infeasible within public cloud environments, e.g. [83, 149, 3, 150]. The potential impact of attacks on crypto processes can be even more severe since cryptography is at the core of any security solution. Consequently, extracting cryptographic keys across VM boundaries has also received considerable attention lately. Initial studies explored the Prime+Probe technique on L1 cache [34, 151]. Though requiring the attacker and the victim to run on the same physical CPU core simultaneously, the small number of cache sets and the simple addressing scheme made the L1 cache a popular target. Follow up works have step by step removed restrictions and increased the viability of the attacks. The shared Last Level Cache (LLC) now enables true cross-core attacks [31, 91, 1] where the attacker and the victim share the CPU, but not necessarily the CPU core. Most recent LLC Prime+Probe attacks no longer rely on de-duplication [42, 43] or core sharing, making them more widely applicable.

With the increasing sophistication of attacks, participants of the cloud indus-

try ranging from Cloud Service Providers (CSPs) to hypervisor vendors, up all the way to providers of crypto libraries have fixed many of the newly exploitable security holes through patches [152, 153, 154]—many in response to published attacks. However, many of the outdated cryptographic libraries are still in use, opening the door for exploits. A scan over the entire range of IPs in the South America East region yields that 55% of TLS hosts installed on Amazon EC2 servers have not been updated since 2015 and are vulnerable to an array of more recently discovered attacks. Consequently, a potential attacker such as a nation state, hacker group or a government organization can exploit these vulnerabilities for bulk recovery of private keys. Besides the usual standard attacks that target individuals, this enables mass surveillance on a population thereby stripping the network from any level of privacy. Note that the attack is enabled by our trust in the cloud. The cloud infrastructure already stores the bulk of our sensitive data. Specifically, when an attacker instantiates multiple instances in a targeted availability zone of a cloud, she co-locates with many vulnerable servers. In particular, an attacker trying to recover RSA keys can monitor the LLC in each of these instances until the pattern expected by the exploited hardware level leakage is observed. Then the attacker can easily scan the cloud network to build a public key database and deduce whom the recovered private key belongs to. In a similar approach, Heninger et al. [155] scan the network for public keys with shared or similar RSA modulus factors due to poor randomization. Similarly, Bernstein et al. [156] compiled a public key database and scanned for shared factors in RSA modulus commonly caused by broken random number generators.

In this chapter, we explain all the necessary steps to recover RSA decryption keys in the Amazon EC2 cloud and present our results. More precisely, we utilize the LLC as a covert channel both to co-locate and perform a cross-core side-channel

attack against a recent cryptographic implementation. Our results demonstrate that even with complex and resilient infrastructures, and with properly configured random number generators, cache attacks are a big threat in commercial clouds.

Contributions

This chapter presents a full key recovery attack on a modern implementation of RSA in a commercial cloud and explores all steps necessary to successfully recover both the key and the identity of the victim.

This attack can be applied under two different scenarios:

1. **Targeted Co-location:** In this scenario, we launch instances until we co-locate with the victim as described in [95, 6]. Upon co-location the secret is recovered by a cache enabled cross-VM attack.

2. **Bulk Key Recovery:** We randomly create instances and using cross-VM cache attacks recover imperfect private keys. These keys are subsequently checked and against public keys in the public key database. The second step allows us to eliminate noise in the private keys and determine the identity of the owner of the recovered key.

Unlike in earlier bulk key recovery attacks [155, 156] we do not rely on faulty random number generators but instead exploit hardware level leakages.

Our specific technical contributions are as follows:

- We first demonstrate that the LLC contention based co-location detection tools are plausible in public clouds

- We describe how to apply the Prime+Probe attack to the LLC and obtain RSA leakage information from co-located VMs

- Last, we present a detailed analysis of the necessary post-processing steps to cope with the noise observed in a real public cloud setup, along with a detailed analysis on the CPU time (at most 30 core-hours) to recover both the noise-free key and the owner's identity (IP).

6.2 Co-locating on Amazon EC2

In order to perform our experiments across co-located VMs, we first need to make sure that they are running in the same server. We present the LLC as an exploitable covert channel with the purpose of detecting co-location between two instances. For the experiments, we launched 4 accounts (named A, B, C, and D) on the South American Amazon EC2 region and launched 20 `m3.medium` instances in each of these accounts, 80 instances in total.

On these instances, we performed our LLC co-location detection test and obtained co-located instance pairs. In total, out of 80 instances launched from different accounts, we were able to obtain 7 co-located pairs and one triplet. Account A had 5 co-located instances out of 20 while B and C had 4 and 7 respectively. As for account D, we had no co-location with instances from other accounts. Overall, assuming that the account A is the target, next 60 instances launched in accounts B, C, D have 8.3% probability of co-location with the target. Note that all co-locations were between machines from different accounts. The experiments did not aim at obtaining co-location with a single instance, for which the probability of obtaining co-location would be lower.

6.2.1 Revisiting Known Detection Methods

To achieve co-location, we first tried to reproduce Ristenpart et al.'s [81] work where they achieved co-location on Amazon EC2 in 2009. In [81], the authors use Hypervisor IP, Instance IP, Ping Test and Disk Drive Test as tools for co-location detection. We have tried these methods as well and found that Amazon, in fact, did a good job of fixing these attack vectors and they are no longer useful. Even though these methods did not provide fruitful results in Amazon EC2, we believe that they may still prove viable on other clouds. Therefore we explain our experience with each of the aforementioned tests in the following.

Hypervisor IP: Using the `traceroute` tool, we collected first hop IP addresses from our instances. The idea behind this collection is that instances located in the same physical machine should have the same first hop address, presumably the hypervisor IP. However, experiments show that there are only a few different first hop IP addresses used for a large number of instances, only four for our 80 instances. Also, with repeated measurements, we noticed that these addresses were actually dynamic, rather than assigned IPs. Even further, we later confirmed by LLC test results that co-located instances do not share the same first hop address, making this detection method useless.

Instance IP: Like [81], we also checked for any possible algebraic relation or proximity between our instance IP addresses. After detecting co-located instances with the LLC test, we checked both internal and external IP addresses of co-located instances and concluded that IP address assignment is random and does not leak any information about co-location in Amazon EC2 anymore.

Ping Test: In a network, ping delay between two nodes depends on various factors such as network adapters of nodes, network traffic and most importantly the

number of hops between the nodes. In [81], authors used this information to detect co-location on the assumption that co-located instances have shorter ping delays. By sending pings from each instance to all other 80, we obtained round trip network delays for all instances. From each account, we sent 20 repeated pings and obtained maximum, average and minimum ping times. We discarded the maximum ping values since they are highly affected by network traffic and do not provide reliable information. Average and minimum ping values, on the other hand, are more directly related to the number of hops between two instances. While co-location correlates with lower ping times, it fails to provide conclusive evidence for co-location.

Figure 6.1 shows the heat map of our ping timings, dark blue indicating lower and red representing high round trip times. Also, x and y-axes represent ping source and target instances respectively. Visual inspection of the figure reveals: (i) Diagonal representing the self-ping (through external IP) time is clearly distinguishable and low compared to the rest of the targets; (ii) Network delay of the source instance affects the round trip time significantly, requiring an in-depth analysis to find relatively close instances; (iii) Last 20 instances that belong to account D have significantly lower and uniform overall ping delays than the rest of the accounts.

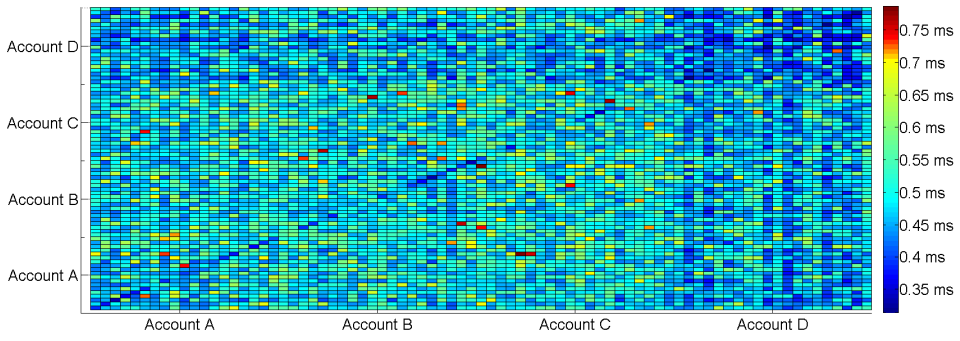


Figure 6.1: Ping time heat map for all 80 instances created using minimum ping times for each source instance

In order to eliminate the delay stemming from the source instance, we decided

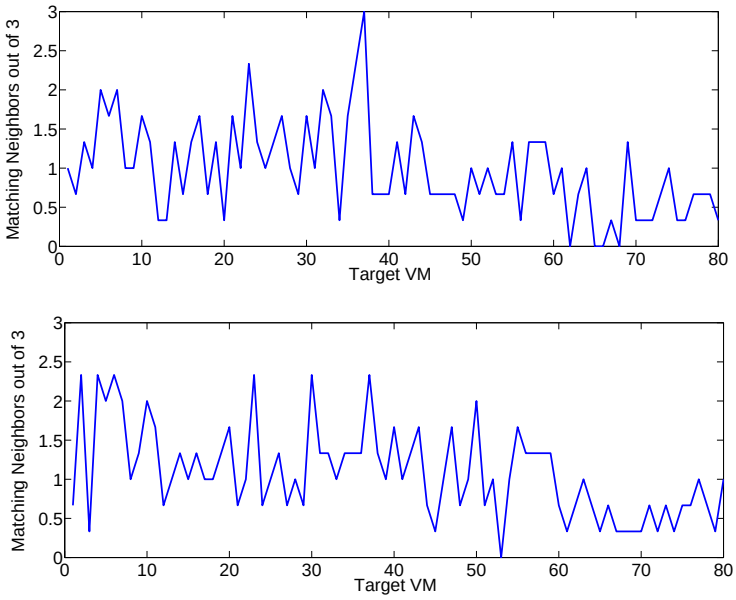


Figure 6.2: Consistent number of neighbors according to their average and minimum ping response times respectively

to find the 3 closest neighbors of each instance rather than applying a constant threshold. After filtering strong outliers, we used multiple datasets with average and minimum ping times to create a more reliable metric. For different datasets, consistent closest neighbors to an instance indicate either co-location or sharing the same subnet. Using three datasets taken at different times of day, we created Figures 6.2(a) and 6.2(b) that show consistent close neighbors according to average and minimum ping times respectively, for all 80 instances. As seen from figures, network is highly susceptible to noise which consistency of ping times significantly. As seen in Figure 6.2(a), apart from instance B17, no instance had consistent low ping neighbors nearly enough to suspect a co-location. In conclusion, even though the ping test reveals some information about the proximity of instance networks, as seen from self-ping times, it is not fine-grain enough to be used for co-location detection.

Disk Drive Benchmark: To replicate [81]'s disk drive test, we used the `dd` tool

due to its file size and repetition flexibility. `dd` is a Unix command line utility used to copy files, backup disks and perform disk benchmarks. In our performance degradation tests, we used `dd` as a tool to measure disk performance under simultaneous heavy load from multiple instances and use the results to detect possible co-locations. Using `dd`, we repeatedly wrote various size blocks of zeros to the disk drive and measured the write speed. Also, in order to maximize the performance degradation, we tried various file sizes ranging from 1 KB to 2 GB to find an optimal detection point. Instances from all four accounts were scheduled to perform the test at the same time. In order to achieve synchrony, we updated instance times to `time.nist.gov` prior to each test. The aim of this test was to observe any potential performance degradations in disk write speeds due to two or more co-located instances.

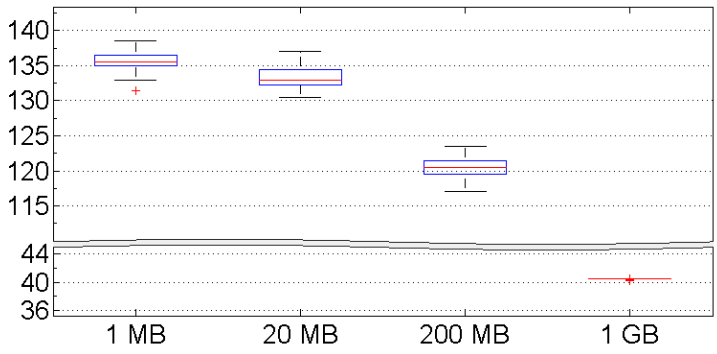


Figure 6.3: `dd` performance results for 2 GB, 200 MB, 20 MB and 1 MB blocks.

Our `dd` test results are presented in Figure 6.3 where the horizontal and vertical axis represents the instance number and the disk write speed in MBps, respectively. Note that, we did not include any results for files smaller than 1 MB since `dd` does not provide stable results for such files. Figure 6.3 clearly shows that disk performance is almost constant for all instances, including the co-located ones. This means that either all 80 instances are using separate disk drives or more likely that

every instance has its disk access bandwidth limited to a certain value to prevent bottlenecks. In conclusion, the performance degradation in co-located instances is unstable, negligibly low at about $2 \sim 3$ percent, and thus inconclusive.

6.2.2 The LLC Co-location Method

The LLC is shared across all cores of most modern Intel CPUs, including the Intel Xeon E5-2670 v2 used (among others) in Amazon EC2. Accesses to LLC are thus transparent to all VMs co-located on the same machine, making it the perfect domain for covert communication and co-location detection.

Our LLC test is designed to detect cache lines that are needed to fill a specific set in the cache. In order to control the location that our data will occupy in the cache, the test allocates and works with hugepages.[1]In normal operation with moderate noise, the number of lines to fill one set is equal to LLC associativity, which is 20 in Intel Xeon E5-2670 v2 used in our Amazon EC2 instances. However, with more than one user trying to fill the same set at the same time, one will observe that fewer than 20 lines are needed to fill one set. By running this test concurrently on a co-located VM pair, both controlled by the same user, it is possible to verify co-location with high certainty. The test performs the following steps:

- Prime one memory block b_0 in a set in the LLC
- Access additional memory blocks $b_1, b_2, \ldots, b_n$ that occupy the same set, but can reside in a different slice.
- Reload the memory block b_0 to check whether it has been evicted from the LLC. A high reload time indicates that the memory block b_0 resides in the

[1]The co-location test has to be implemented carefully, since the heavy usage of hugepages may yield into performance degradation. In fact, while trying to achieve a quadruple co-location, Amazon EC2 stopped our VMs due to performance issues.

RAM. Therefore we know that the required m memory blocks that fill a slice are part of the accessed additional memory blocks $b_1, b_2, \ldots, b_n$.

- Subtract one of the accessed additional memory blocks b_i and repeat the above protocol. If b_0 is still loaded from the memory, b_i does not reside in the same slice. If b_0 is now located in the cache, it can be concluded that b_i resides in the same cache slice as b_0 and therefore fill the set.

- Count the number of memory blocks needed to fill a set/slice pair. If the number is significantly different from the associativity, it can be concluded that we have cache contention across co-located VMs.

The LLC is not the only method that we have tried in order to verify co-location. As detailed in Section 6.2.1, we have also tried the methods used in [81]. However, the experiments show that the LLC test is the only decisive and reliable test that can detect whether two of our instances are running in the same CPU in Amazon EC2. We performed the LLC test in two steps as follows:

1. **Single Instance Elimination:** The first step of the LLC test is the elimination of single instances i.e. the ones that are not co-located with any other in the instance pool. To do so, we schedule the LLC test to run at all instances at the same time. Instances not detecting co-location is retired. For the remaining ones, the pairs need to be further processed as explained in the next step. Note that without this preliminary step, one would have to perform $n(n-1)/2$ pair detection tests to find co-located pairs, i.e. 3160 tests for 80 instances. This step yielded 22 co-located instances out of 80.

2. **Pair Detection:** Next we identify pairs for the possibly co-located instances. The test is performed as a binary search tree where each instance is tested against all the others for co-location.

6.2.3 Unsuccessful Co-location Detection Methods

CPU Benchmarking: To create a bottleneck at the CPU level, we used the `Hardinfo` CPU benchmark suite. The suite provides a wide range of benchmarks, namely CryptoHash, Fibonacci number calculation, N-Queens test, FFT calculation, and Raytracing. However, our experiments show that the instance isolation and the resources management in Amazon EC2 prevents the test from creating noticeable performance degradation hence indicating co-location.

AES-NI Benchmarking: AES-NI is the high-performance AES hardware module found in most modern Intel processors including the ones used in Amazon EC2. The Intel Xeon E5-2670 v2 datasheet [157] does not specify whether each core has its own module or all cores use a single AES-NI module. We suspected that by creating bottlenecks in the shared AES-NI module we could detect co-location. However, our experiments revealed that the AES-NI modules are not shared between cores and each CPU core uses its own module, making this method uses only for same-core detection. This prevents the AES-NI benchmark to be used for co-location detection, unless two instances are using the same physical core, as separate threads.

6.3 Tricks and Challenges of Co-location Detection

During our experiments on Amazon EC2, we have observed various problems and interesting events related to the underlying hardware and software. Here we go over these observations and discuss what to expect when experimenting on Amazon EC2.

Hardware Complexity

Modern Amazon EC2 instances have much more advanced and complex hardware components like 10 core, 20 thread CPUs and SSDs. Thus, our cache profiling techniques have to be adapted to handle servers with multiple slices that feature non-linear slice selection algorithms.

Co-located VM Noise

Compute cloud services including Amazon EC2 maintain a variety of services and servers. Most user-based services, however, quiet down when users quiet down, i.e. after midnight. Especially between 2 a.m. and 4 a.m. Internet traffic, as well as computer usage, is significantly lower than the rest of the day. We confirmed this assumption by measuring LLC noise in our instances and collected data from 6 instances over the course of 4 weekdays. Results are shown in Figure 6.4. LLC noise and thus server load are at its peak around 8 p.m. and lowest at 4 a.m. We also

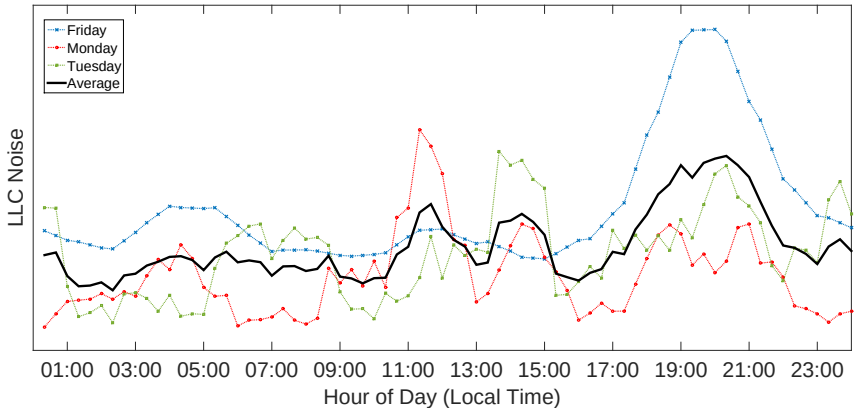


Figure 6.4: LLC Noise over time of day, by day (dotted lines) and on average (bold line).

measured the noise observed in the first 200 sets of the LLC for one day in Figure 6.5. The y-axis shows the probability of observing a cache access by a co-located user

other than victim during a Prime+Probe interval of the spy process (i.e. the attacker cannot detect the cache access of the victim process). The measurements were taken every 15 minutes. A constant noise floor at approx. 4.5% is present in all sets. Sets 0 and 3 feature the highest noise, but high noise (11%) is observed at the starting points of other pages as well. In fact, certain set numbers (0,3,26,39,58) mod 64 seem to be predictably more noisy and not well suited for the attack.

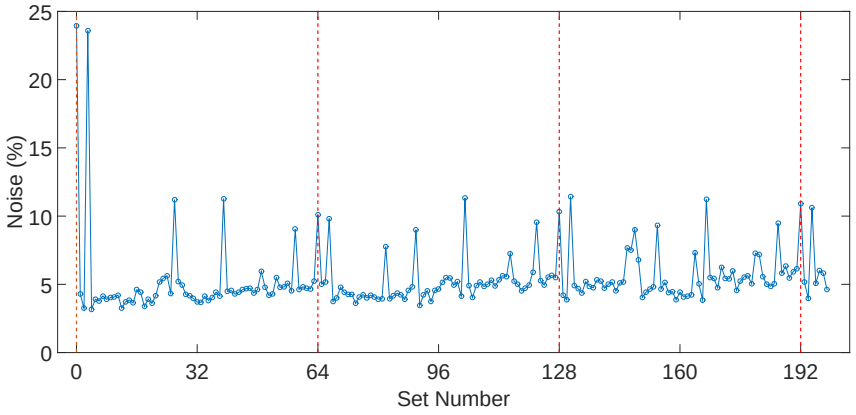


Figure 6.5: Average noise for the first 200 sets in a day. Red lines are the starting points of pages.

Dual Socket machines

We did not find evidence of dual socket machines among the medium instances that we used in both co-location and attack steps. Indeed once co-located, our LLC co-location test always succeeded over time, even after a year. If the instances were to reside in dual socket machines and the VM processes moved between CPUs, the co-location test would have failed. However, even in that case, repeated experiments would still reveal co-location just by repeating the test after a time period enough to allow a socket migration.

Placement Policy

During our experiments, we have observed that the instances launched within short time intervals of each other were more likely to be co-located. To exploit the placement policy and increase chances of co-location, one should launch multiple instances in a close time interval with the target. Note that two instances from the same account are never placed on the same physical machine. While this increases the probability of co-locating with a victim in a practical attack scenario, it also makes it harder to achieve co-location for controlled experiments.

Hypervisor Hardware Obfuscation

In reaction to [81], Amazon has fixed information leakages about the underlying hardware by modifying their Xen Hypervisor. Currently, no sensor data such as fan speed, CPU and system temperature or hardware MAC address is revealed to instances. Serial numbers and all other hardware identifiers are either emulated or censored, mitigating any co-location detection using this information.

Instance Clock Decay

In our experiments using Amazon EC2, we have noticed that over time, system clocks of these instances fallback. More interestingly, after detecting co-location using the LLC test, we discovered that co-located instances have the same clock degradation with 50 nanoseconds resolution. We believe that this information can be used for co-location detection as well.

Clock Differences Between Instances

In order to run performance degradation tests simultaneously, OS clocks in instances must be synchronized. In order to assure synchronization during tests, we have updated the system times using `ntpdatetime.nist.gov` command multiple times during experiments to keep the synchronization intact. Note that without time synchronization, we have observed differences in system times between instances of up to a minute.

Instance Retirement

A very interesting feature of Amazon EC2 is instance retirement in case of perceived hardware failure or overuse. Through constant resource monitoring, EC2 detects significant performance degradation on one of the hardware components such as disk drive, network adapter or a GPU card in a physical system, and marks the instance for retirement. If there is no malfunction or hazardous physical damage to the underlying hardware, e-mail notifications are sent to all users who have instances running on the physical machine. If there is such an immediate hardware problem, instances are retired abruptly and a notification e-mail is sent to users afterward. We observed this behavior on our triple co-located instances (across three accounts). While running our performance tests to create a bottleneck and determine co-location, we received three separate e-mails from Amazon to the three involved accounts notifying us that our instances A5, B7 and C7 had a hardware failure and are scheduled for instance retirement. The important thing to note here is that, via our tests, we have previously determined that these instances A5, B7, and C7 are co-located on the same physical machine. We assume that our performance-based co-location tests were the cause of the detected performance degradation in the system that

raised flags with Amazon EC2 health monitoring system, resulting in the instance retirement.

Network Scanning the Amazon EC2

We utilize the `nmap` tool to scan the entire Amazon EC2 network in South America, which consists of 10 basic IP ranges, each one with different sizes. In particular, we are looking for information on port 443. This will serve us to construct our public key database in case the identity of the target is not known. The results are presented in Figure 6.6. It can be observed that the distribution over the regions is not uniform.

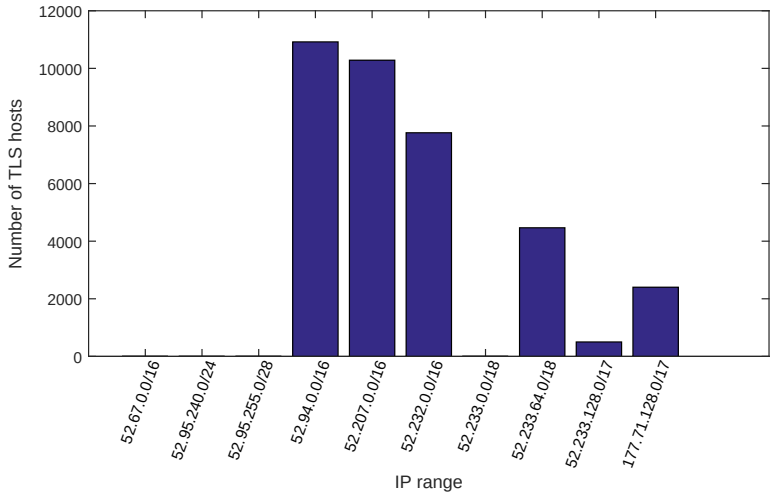


Figure 6.6: Number of TLS hosts in the South American region of Amazon EC2 by IP range

6.4 Cross-VM RSA Key Recovery

To prove the viability of the Prime+Probe attack in Amazon EC2 across co-located VMs, we present an expanded version of the attack implemented in [42] by showing

its application to RSA. It is important to remark that the attack *is not* processor specific, and can be implemented in any processor with inclusive last level caches. In order to perform the attack:

- We make use of the fact that the offset of the address of each table position entry does not change when a new decryption process is executed. Therefore, we only need to monitor a subsection of all possible sets, yielding a lower number of traces.

- Instead of the monitoring both the multiplication and the table entry set (as in [42] for El-Gamal), we *only monitor a table entry set in one slice*. This avoids the step where the attacker has to locate the multiplication set and avoids an additional source of the noise.

The attack targets a sliding window implementation of RSA-2048 where each position of the pre-computed table will be recovered. We will use Libgcrypt 1.6.2 as our target library, which not only uses a sliding window implementation but also uses CRT and message blinding techniques [158]. The message blinding process is performed as a side-channel countermeasure for `chosenciphertext` attacks, in response to studies such as [89, 88].

We use the Prime+Probe side-channel technique to recover the positions of the table T that holds the values $c^3, c^5, c^7, \ldots, c^{2^W-1}$ where W are the window size. For CRT-RSA with 2048 bit keys, $W = 5$ for both exponentiations d_p, d_q. Observe that, if all the positions are recovered correctly, reconstructing the key is a straightforward step.

Recall that we do not control the victim's user address space. This means that we do not know the location of each of the table entries, which indeed changes from execution to execution. Therefore we will monitor a set hoping that it will be

accessed by the algorithm. However, our analysis shows a special behavior: each time a new decryption process is started, even if the location changes, the offset field does not change from decryption to decryption. Thus, we can *directly* relate a monitored set with a specific entry in the multiplication table.

The knowledge of the processor in which the attack is going to be carried out gives an estimation of the probability that the set/slice we monitor collides with the set/slice the victim is using. For each table entry, we fix a specific set/slice where not much noise is observed. In the Intel Xeon E5-2670 v2 processors, the LLC is divided into 2048 sets and 10 slices. Therefore, knowing the lowest 12 bits of the table locations, we will need to monitor *one* set/slice that solves $s \mod 64 = o$, where s is the set number and o is the offset for a table location. This increases the probability of probing the correct set from $1/(2048 \cdot 10) = 1/20480$ to $1/((2048 \cdot 10)/64) = 1/320$, reducing the number of traces to recover the key by a factor of 64. Thus our spy process will monitor accesses to *one* of the 320 set/slices related to a table entry, hoping that the RSA encryption accesses it when we run repeated decryptions. Thanks to the knowledge of the non-linear slice selection algorithm, we can easily change our monitored set/slice if we see a high amount of noise in one particular set/slice. Since we also have to monitor a different set per table entry, it also helps us to change our eviction set accordingly.

The threshold is different for each of the sets since the time to access different slices usually varies. Thus, the threshold for each of the sets has to be calculated before the monitoring phase. In order to improve the applicability of the attack, the LLC can be monitored to detect whether there are RSA decryptions or not in the co-located VMs as proposed in [6]. After it is proven that there are RSA decryptions the attack can be performed.

In order to obtain high-quality timing leakage, we synchronize the spy process

and the RSA decryption by initiating a communication between the victim and attacker, e.g. by sending a TLS request. Note that we are looking for a particular pattern observed for the RSA table entry multiplications, and therefore processes scheduled before the RSA decryption will not be counted as valid traces. In short, the attacker will communicate with the victim before the decryption. After this initial communication, the victim will start the decryption while the attacker starts monitoring the cache usage. In this way, we monitor 4,000 RSA decryptions with the same key and same ciphertext for each of the 16 different sets related to the 16 table entries.

We investigate a hypothetical case where a system with dual CPU sockets is used. In such a system, depending on the hypervisor CPU management, two scenarios can play out; processes moving between sockets and processes assigned to specific CPUs. In the former scenario, we can observe the necessary number of decryption samples simply by waiting over a longer period of time. In this scenario, the attacker would collect traces and only use the information obtained during the times the attacker and the victim share sockets and discard the rest as missed traces. In the latter scenario, once the attacker achieves co-location, as we have in Amazon EC2, the attacker will always run on the same CPU as the target hence the attack will succeed in a shorter span of time.

6.5 Leakage Analysis Method

Once the online phase of the attack has been performed, we proceed to analyze the leakage observed. There are three main steps to process the obtained data. The first step is to identify the traces that contain information about the key. Then we need to synchronize and correct the misalignment observed in the chosen traces. The last

step is to eliminate the noise and combine different graphs to recover the usage of the multiplication entries. Among the 4,000 observations for each monitored set,

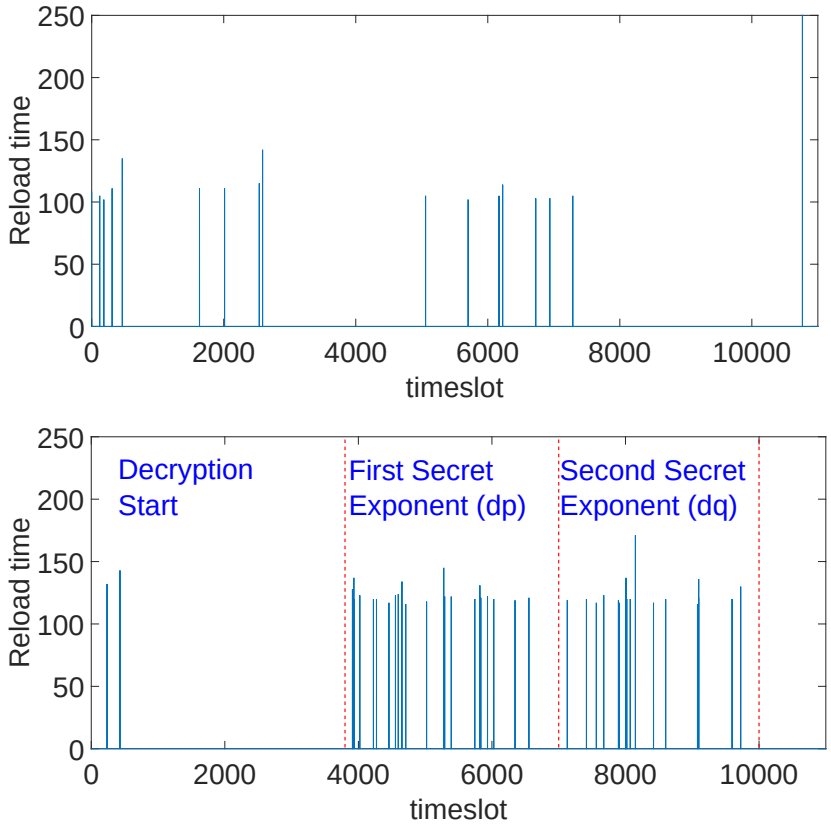


Figure 6.7: Different sets of data where we find a) trace that does not contain information b) trace that contains information about the key

only a small portion contains information about the multiplication operations with the corresponding table entry. These are recognized because their exponentiation trace pattern differs from that of unrelated sets. In order to identify where each exponentiation occurs, we inspected 100 traces and created the timeline shown in Figure 6.7(b). It can be observed that the first exponentiation starts after 37% of the overall decryption time. Note that among all the traces recovered, only those that have more than 20 and less than 100 peaks are considered. The remaining ones are discarded as noise. Figure 6.7 shows measurements where no correct pattern

was detected (Fig. 6.7(a)), and where a correct pattern was measured (Fig. 6.7(b)).

In general, after the elimination step, there are $8-12$ correct traces left for each set. We observe that data obtained from each of these sets corresponds to 2 consecutive table positions. This is a direct result of CPU cache prefetching. When a cache line that holds a table position is loaded into the cache, the neighboring table position is also loaded due to cache locality principle.

For each graph to be processed, we first need to align the creation of the look-up table with the traces. Identifying the table creation step is trivial since each table position is used twice, taking two or more time slots. Figure 6.8(a) shows the table access position indexes aligned with the table creation. In the figure, the top graph shows the correct table accesses while the rest of the graphs show the measured data. It can be observed that the measured traces suffer from misalignment due to noise from various sources e.g. RSA or co-located neighbors.

To fix the misalignment, we take the most common peaks as a reference and apply a correlation step. To increase efficiency, the graphs are divided into blocks and processed separately as seen in Figure 6.8(a). At the same time, Gaussian filtering is applied to peaks. In our filter, the variance of the distribution is 1 and the mean is aligned to the peak position. Then for each block, the cross-correlation is calculated with respect to the most common hit graph i.e. the intersection set of all graphs. After that, all graphs are shifted to the position where they have the highest correlation and aligned with each other. After the cross-correlation calculation and the alignment, the common patterns are observable as in Figure 6.8(b). Observe that the alignment step successfully aligns measured graphs with the true access graph at the top, leaving only the combining and the noise removal steps. We combine the graphs by simple averaging and obtain a single combined graph.

In order to get rid of the noise in the combined graph, we applied a threshold

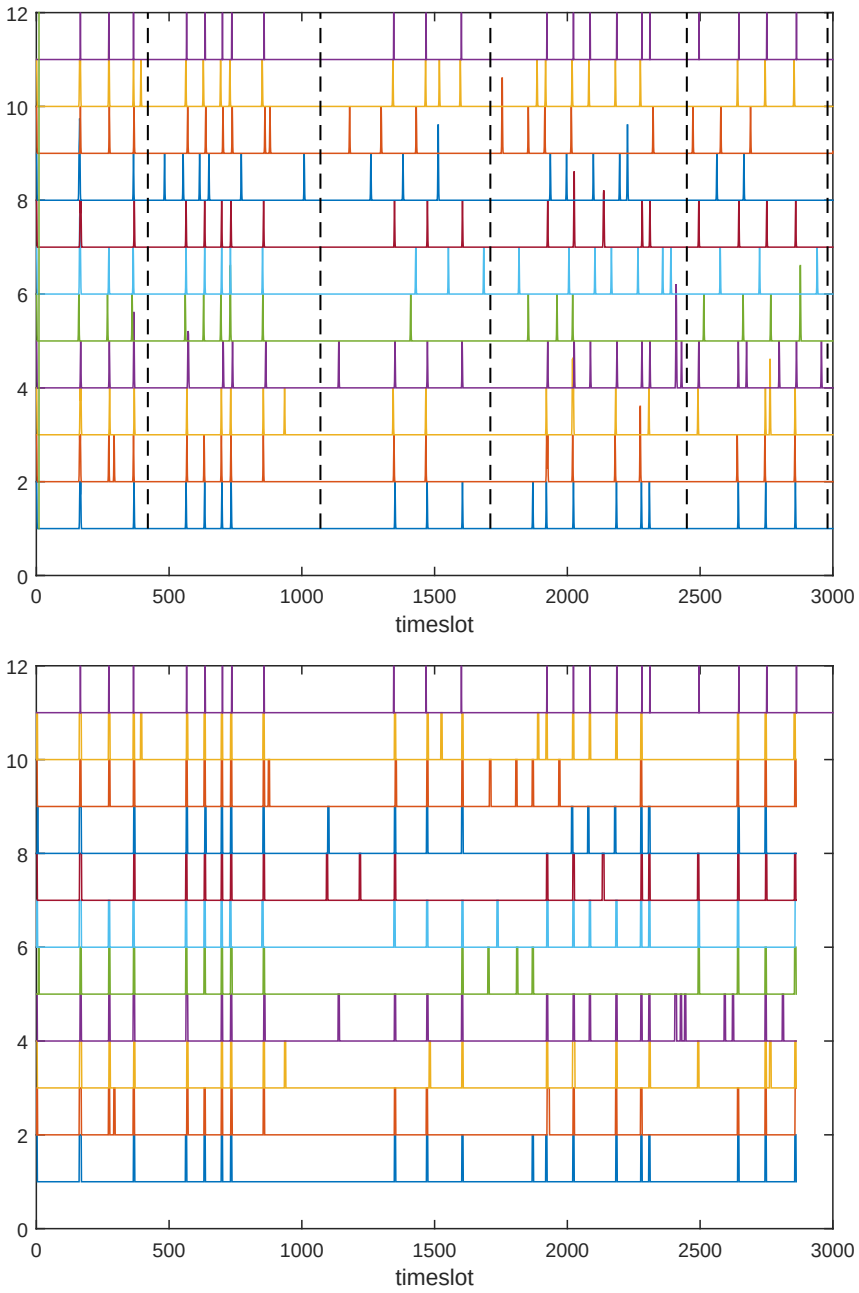


Figure 6.8: 10 traces from the same set where a) they are divided into blocks for a correlation alignment process b) they have been aligned and the peaks can be extracted

filter as can be seen in Figure 6.9(a). We used 35% of the maximum peak value observed in graphs as the threshold value. Note that a simple threshold was sufficient to remove noise terms since they are not common between graphs.

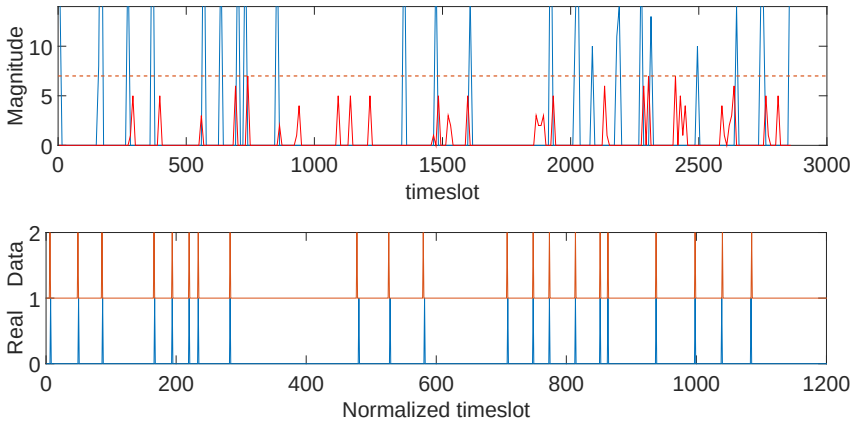


Figure 6.9: Eliminating false detections using a threshold (red dashed line) on the combined detection graph (top). Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution.

Now we convert scaled time slots of the filtered graph to real-time slot indexes. We do so by dividing them with the spy process resolution ratio, obtaining the Figure 6.9(b). In the figure, the top and the bottom graphs represent the true access indexes and the measured graph, respectively.

Also, note that even if additional noise peaks are observed in the obtained graph, it is very unlikely that two graphs monitoring consecutive table positions have noise peaks at the same time slot. Therefore, we can filter out the noise stemming from the prefetching by combining two graphs that belong to consecutive table positions. Thus, the resulting indexes are the corresponding timing slots for look-up table positions.

The very last step of the leakage analysis is finding the intersections of two graphs that monitor consecutive sets. By doing so, we obtain accesses to a single table position as seen in Figure 6.10 with high accuracy. At the same time, we have a total of three positions in two graphs. Therefore, we also get the positions of the neighbors. A summary of the result of the leakage analysis is presented in Table 6.1. We observe that more than 92% of the recovered peaks are in the correct position.

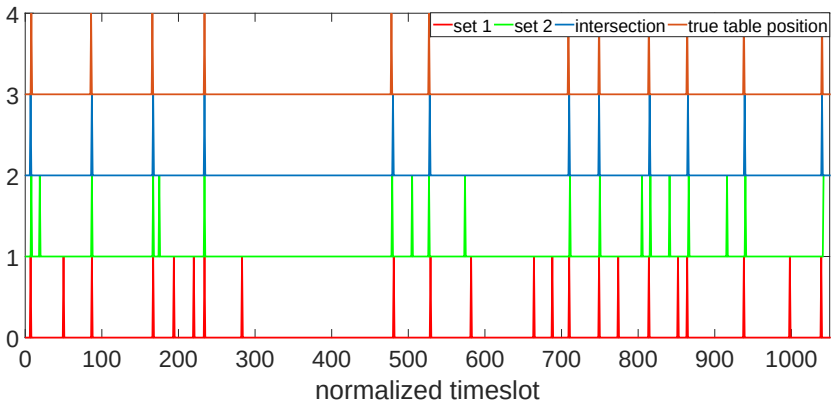


Figure 6.10: Combination of P+P results of two LLC sets.

However, note that by combining two different sets, the wrong peaks will disappear with high probability, since the chance of having wrong peaks in the same time slot in two different sets is very low.

Table 6.1: Successfully recovered peaks on average in an exponentiation

Average Number of traces/set	4000
Average number of correct graphs/set	10
Wrong detected peaks	7.19%
Missdetected peaks	0.65%
Correctly detected peaks	92.15%

Chapter 7

A Micro-architectural QoS Attack on Smartphones

Smartphone apps need optimal performance and responsiveness to rise among numerous rivals on the market. Further, some apps like media streaming or gaming apps cannot even function properly with a performance below a certain threshold. In this chapter, we present the first performance degradation attack on Android OS that can target rival apps using a combination of logical channel leakages and low-level architectural bottlenecks in the underlying hardware.

To show the viability of the attack, we design a proof-of-concept app and test it on various mobile platforms. The attack runs covertly and brings the target to the level of unresponsiveness. With less than 10% CPU time in the worst case, it requires minimal computational effort to run as a background service and requires only the UsageStats permission from the user. We quantify the impact of our attack using 11 popular benchmark apps, running 44 different tests. The measured QoS degradation varies across platforms and applications, reaching a maximum of 90% in some cases. The attack combines the leakage from logical channels with low-

level architectural bottlenecks to design a malicious app that can covertly degrade Quality of Service (QoS) of any targeted app. Furthermore, our attack code has a small footprint and is not detected by the Android system as malicious. Finally, our app can pass the Google Play Store malware scanner, Google Bouncer, as well as the top malware scanners in the Play Store.

7.1 Motivation

Smartphones are now integrated into all facets of our lives—facilitating our daily activities from banking to shopping and from social interactions and to monitoring our vital health signs. These services are supported by numerous apps built by an army of developers. The mobile ecosystem is growing at an astounding rate with more than 2.4 million apps as of September 2016 [159], running on billions of devices. According to [160], more than a million new Android devices are activated worldwide, downloading billions of apps and games each month. Moreover, app revenue totaled to 45 billion dollars in 2015, proving a lucrative business. App developers big and small are under enormous competition trying to get a foothold in this growing market and a share of the huge revenue. As expected, there is fierce competition amongst competing apps with similar functionality trying to earn a top ranking in the app store.

In such a cutthroat market, competing app vendors have a strong incentive to cheat to get ahead in the competition. In the mobile app market, a strong delivery channel is the app store where apps which have received high user ratings are featured on the main page, and ones with low ratings are essentially buried in the listings and thus have become invisible to the users. Therefore, if an app developer can force a negative user experience during the use of a competitor's app, that could

essentially render the competing app invisible.

To prevent malicious interference mobile platform vendors commonly implement app level sandboxing. For instance, on the Application Fundamentals web page, Google states that: "Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps." [161]. While the first part of this statement is correct, the second part is not. Android apps running on a system share the same underlying physical hardware and therefore are not truly isolated from each other. This lack of physical isolation can be exploited, giving an app advantage over a competitor's app. Victim app's operations can be manipulated, degraded or even brought to a halt. This would be particularly destructive in real-time applications such as trading, online gaming, and live streaming.

Malware in Android devices is generally separated into five categories as follows; Information Leakage, Privilege Escalation, Financial Charge, Ransomware, and Adware. Here, we introduce a new category, Quality of Service (QoS) attacks. QoS attacks aim to degrade and or disrupt the functionality of legitimate services. In this case, we aim to degrade the performance of other apps installed on the same mobile device using similar techniques as micro-architectural attacks. Here, we show that such an exploit is indeed practical. We introduce a technique that exploits architectural bottlenecks, in combination with logical channel leakages to degrade the performance of victim apps. The attack app does not require any root or peripheral access privileges as needed in sensor-enabled attacks. Also, the performance footprint of the attack code is very low, the attack is not detected by any malicious code monitor. We quantify the performance degradation with a wide variety of benchmarks on various platforms. Since our attack vector only employs a widely used feature of modern microprocessors, i.e. memory bus locking, the degradation attack is hard to detect and mitigate. Furthermore, our app passed the Google Play Store

malware scan and an additional 23 of the most popular malware scanners listed in the Play Store. This shows evidence that new threats do not necessarily fit into traditional malware definitions and require more in-depth analysis with a broader perspective.

Contributions

This chapter presents and explores all the necessary steps to successfully implement a Quality of Service (QoS) attack on mobile devices running Android OS. Specifically, in this chapter we,

- present the first QoS attack on Android OS that combines architectural bottlenecks and logical channel leakages to significantly degrade the performance of a victim app.
- show that our attack is stealthy and hard to mitigate by showing that it cannot be detected by the Android OS, Google Play Store malware scan or malware scanner apps. Further, the attack exploits the memory bus locking, a widely used feature of modern microprocessors hence is hard to mitigate.
- test and quantify the QoS degradation caused by our attack using the most popular benchmarks in the Play Store.

7.2 The QoS Attack Methodology

In order to perform the QoS attack on Android, we need to overcome two separate problems: 1) Detecting when the victim app is in active use i.e. in the foreground. 2) Performing an Exotic Atomic Operation and triggering a memory bus lock.

The first part of the attack is crucial to ensure that the user does not suspect the attacker app as the culprit behind the slowdown but rather blames it on the victim app. If the attacker was to trigger memory bus locking continuously, even from the background, it would appear as the system altogether has a performance problem. Or even worse, the user could trace the slowdown back to the attacker app and uninstall it, rather than the victim app and defeat the purpose of the attack. Therefore, the first problem the attacker has to overcome is detecting the victim app launch.

The second part of the attack is degrading the victim app's performance whenever it is active. The attacker can achieve this by performing Exotic Atomic Operations that trigger memory bus locking, resulting in significant performance overhead to the system. By triggering the lock while the victim app is running, the attacker can flush the ongoing memory operations in the CPU, disrupting the victim app's operation for over 10K cycles. By continuously doing this while the victim app is in the foreground, the attacker can lead the user to think that the victim app has sub-optimal performance and consequently uninstall it.

Our attack consists of the following steps;

1. Launch the attacker app, create a `Sticky` background service meaning that the service will stay active even if the attacker app is closed or even shut down by the user.

2. Run cache profiling tool to obtain spanning addresses to perform Exotic Atomic Operations.

3. Check for victim app launch from the background service. Wait until the user puts the victim app in the foreground.

4. When the victim launch is detected, start the `Exotic Atomic Operation` loop, degrading the QoS of the targeted app.

5. Keep the loop running until the target app is no longer in the foreground. Stop the QoS degradation attack and release the system bottleneck as soon as the user quits the victim app.

6. Repeat until the user removes the victim app.

In the following, we describe the details of our attack as well as the design and implementation of our attacker app.

Detecting Victim Launch

In order to know when the targeted app is running, we use logical channels that are available to apps in Android OS. However, as Android OS evolved, some of these channels have been closed by the deprecated APIs. In the following, we show how to deduce the foreground app in different versions of Android OS, through various channels.

pre-Android 5.0 (API Level 21)

In Android 5.0 (API 21), it is possible to get the list of running apps on the device as well as the foreground app. By using the runningAppProcess method from the `ActivityManager` class, an app can get the list as well as a binary value `LRU` that holds whether or not the app is the least recently used app. By continuously monitoring the LRU value of a process, an attacker detects when the victim app is in the foreground.

Android 5.0+ (API Level 21+)

With Android 5.0 and forward, Android OS limited access to other apps due to privacy and malware concerns. After the deprecation of the APIs to retrieve running apps, the background apps are now hidden to user level apps. Evidently,

Play Store still has many applications like Task Managers, Memory Optimizers etc. that can detect apps running in the system. The question is if the running apps are hidden, how do these Task Managers still retrieve the list of running. The answer lies in logical channel leakages. Here we discuss various methods that can be used to retrieve running apps and subsequently the foreground app.

Package Usage Stats Permission: With UsageStats class added in Android 5.0, apps can obtain various information such as the Last Time Used (LTU), package name and total time spent in the foreground about all the apps running on the device. Although there is no information provided about whether or not an app is in the foreground, it is still possible to infer this information using other data as follows. Using the `getLastTimeUsed` function, it is possible to check when an app, the victim, in this case, was put in the foreground by the user. Note that this value is updated as soon as the user puts the app in the foreground. However, during the use of the app in the foreground, the LTU value remains constant. The value is again updated when the user changes activities and puts the victim app to the background. Using this information, one can monitor the LTU value of an app as shown in Algorithm 1 and deduce the foreground activity of the victim app.

Algorithm 1: Victim app detection algorithm

```
function the foreground_Check();
while SwitchON// The start service switch do
    if inUse && !inUse_old then
        isActive = TRUE; // Victim app put in the foreground
    if isActive then
        lock(); // Bus locking function
        inUse_old = inUse;
        inUse = Check_Usage();
        if inUse && !inUse_old then
            isActive = FALSE; // Victim app put off foreground
```

Timing the killBackgroundProcesses Function: After Android 5.0 (API Level 21), some of the Task Managers in the Play Store changed the way they worked to keep their functionality. Instead of getting the list of running apps, they now retrieve the list of all installed apps using the system provided `getInstalledApplication` function. Then using the `killBackgroundProcesses`, go over the whole list and try to kill all listed apps using app package names. Note that this function requires the `KILL_BACKGROUND_PROCESSES` permission and kills only the apps that were running.

In our attack scenario, we can improve this method to detect whether the victim app is in the foreground. For the apps in the list that were not running, the function simply moves the next item. An attacker can periodically call this function, time each call, and kill the victim app. If the call of the function is above a certain threshold, the attacker can then deduce that the victim app was indeed running. Furthermore, if the victim app is in the foreground, the function cannot kill it at all. By detecting such unsuccessful kill requests, the attacker can deduce when the victim is in the foreground. Moreover, this method requires an only the `KILL_BACKGROUND_PROCESSES` function that is Normal Permission and does not require explicit user consent.

Monitoring Hardware Resources: Hardware resources in Android devices are shared hence accessible to all apps (given the permission). However, even with legitimate access rights, two apps cannot use certain resources simultaneously. For instance, the camera can only be accessed by a single app at any given time. So, when two apps try to access the camera API at the same time, only the first one is served while the second one receives a busy respond. This shared resource allows an attacker to monitor access to a hardware API and detect when a competitor app is in use.

Getting the List of Running Services: After Android 5.0, the list of running

apps became hidden while the list of running services did not. Using this information, it is possible to check whether an installed application has any active service indicating recent use. While a low-grain estimation, this information can still be used to detect whether an app has run recently. This method is especially useful in cases where victim app services launch after the app comes to the foreground.

Reading the System Logs: In Android, many events including warnings, errors, crashes, and system-wide broadcasts are written to the system log. The operating system along with apps write to this log. Using this information, an attacker determines which apps have written to the log recently and monitor the victim app usage. Note that, since Android 4.1 (API level 16), system logs are accessible by only system/signature level apps meaning that third-party apps cannot read system logs.

Niceness of Apps: Another way of detecting foreground app, is using `getRunningAppProcess` and to check niceness of apps. When the app is in the foreground, niceness value decreases to give the user a smoother experience. Therefore by constantly monitoring the niceness value of an app, one can defer the foreground activity of a target app.

7.2.1 Cache Line Profiling Stage

Our attack is CPU-agnostic and employs memory bus locking regardless of the total cache size, cache line size or the number of cache sets. We achieve this by detecting uncacheable memory blocks with a quick, preliminary cache profiling stage. The profiling eliminates the need to know the CPU specifications e.g. the cache line size. Moreover, the Java code in Android apps is compiled to run on the JVM, resulting in changes to the cache addresses. By employing this profiling stage, however, we can ignore address changes at the runtime.

In order to obtain a data block that spans multiple cache lines, we first allocate a block of page-aligned memory using `AtomicIntegerArray` object from the `java.util.concurrent.atomic` class. Note that the size of this array should be large enough to contain multiple uncacheable addresses but not so large that it would trigger an Out-of-Memory (OOM) error and crash the app. In our experiments we have used *array_length* of 1024K to satisfy both conditions.

After the allocation, choose the ideal atomic operation to be used with the memory bus locking. While there are many atomic operations to choose from, it is most beneficial for the attacker to choose the operation that takes the longest time to perform. Since the memory bus lock remains active until the atomic operation is fully completed, longer operations result in stronger degradation to the system. In order to maximize the performance penalty, we have tested various operations such as `compareAndSet, decrementAndGet, addAndGet, AndDecrement` and `getAndIncrement`. Our results showed that the `getAndIncrement` operation was taking the longest time (10K-12K nanoseconds) hence was selected to be used in the attack. After choosing the atomic operation, we first operate continuously on a single address to get a baseline execution time. After the baseline execution time is established -without the bus locking- we start performing on the array as described in Algorithm 2. Starting from the beginning of the allocated array, we increment the array index by one in each loop and record the execution time. When a significantly longer execution time is detected, it is evident that the address is uncacheable or spanning multiple cache lines, therefore triggering the memory bus lock. After all the addresses are tested and the spanning ones recorded, we obtain a list of addresses that satisfy the `Exotic Atomic Operation` condition. We later use these addresses to lock the memory bus in our QoS degradation attack. Note that it is not necessary to obtain a long list since the attacker can de-allocate the

array and reuse the same addresses.

Algorithm 2: Cache line detection algorithm

```
arr = Atomic Integer Array of length array_length Output: List of cache
        line spanning addresses
for each i smaller than array_length do
    startTime = System.nanoTime();
    value = arr.getAndIncrement(i);
    operation_time = System.nanoTime() - startTime;
    if operation_time¿Pre_calculated_average then
        exotic_address[index++] = i;
```

Timing the Operations: To time the performed atomic operations, we use the system provided `nanotime()` function. In theory, this function returns the JVM's high-resolution timer in nanoseconds. However in practice, due to numerous delays stemming from both hardware and software, we have observed varying timer resolutions. With the test devices that are used in experiments, best timer resolutions that we have observed were 958, 468, 104 and 104 nanoseconds on Galaxy S2, Nexus 5, Nexus 5X and Galaxy S7 Edge respectively. Considering that the aforementioned devices have CPUs running in the range of 1.2 to 2.26 GHz and assuming that the devices were running at the highest possible CPU speeds, we can estimate the timer resolution in CPU cycles. By multiplying each CPU clock with the minimum timer resolution in nanoseconds, we get 223, 187, 1057 and 1149 CPU cycles of timer resolution for each device. While the low-resolution timer would present a problem for cache attacks, it is sufficient to distinguish between regular and Exotic Atomic Operations. Remember that we are measuring the execution time of atomic operations on different memory addresses to detect uncacheable addresses where this operation will incur a heavy timing penalty. In average, the regular atomic operations take around 1686, 1610, 844 and 369 nanoseconds in our test platforms. While Exotic Atomic Operations take around 3000-20000 nanoseconds as shown in Figure 7.1.

Since the gap between the two is large enough, we can distinguish between the two using the `nanotime()` timer.

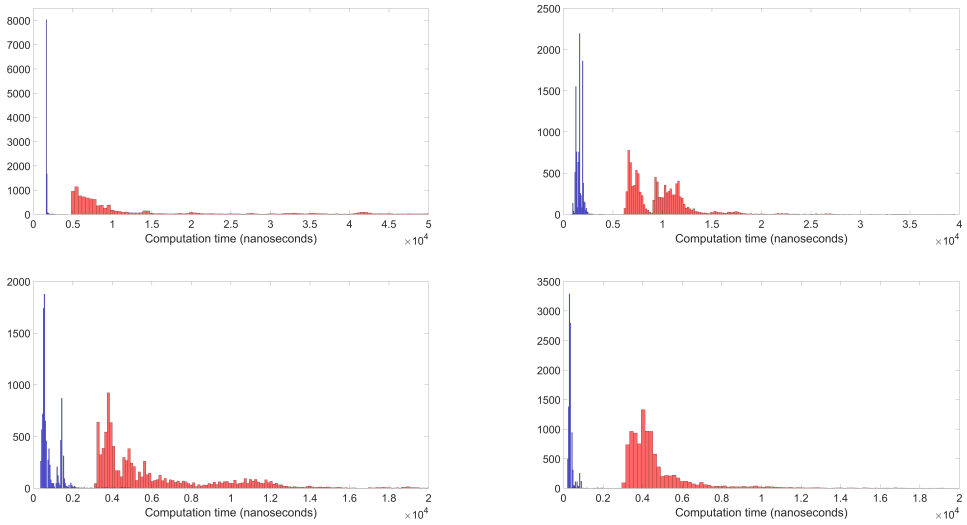


Figure 7.1: Histograms of regular (blue) and exotic (red) atomic operation of the test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively.

7.2.2 Attacker App Design and Implementation

We have designed a simple, lightweight proof-of-concept app to turn the performance penalty into an attack and tested it on various platforms and target apps. The app is designed to work on all devices that have Android 5.0 (API level 21) or a newer version of Android. According to [162] this covers 58.4% percent of all Android devices as of November 2016. Since the app a proof-of-concept, it uses the Package Usage Stats permission. This permission allows the app to get the list of running apps their last active times. Note that this permission opens a prompt and requires the user to explicitly give permission to the app. Our app has a simple interface that includes two activities. The first activity prompts the user to give the necessary permission after which the user can open the second activity. As seen in Figure 7.2, the app opens with an activity that shows disclaimer. On this

activity screen Figure 7.2.2, the user has to give the necessary Package Usage Stats permission to the app or otherwise the app will not enter the app selection activity. To give the permission, the user only has to click on the "Give Ordinary Permissions" button and will automatically be forwarded to the necessary system settings page. After the permission is obtained, the user can return to the opening activity and click on the Go To App Selection Screen. On the app selection activity, as shown in Figure 7.2(b), the user can select any of the apps that were used in the last 24 hours as the target. After that, the user clicks on the "Start Slowdown Service" switch and the selected app name is passed to the background service. Now, the background service of the attacker app continuously monitors the list of used apps. When the selected app is detected to be put in the foreground, the service starts the attack and degrades the target's performance. The QoS attack continues until the user exits the selected app.

7.3 Experiment Setup and Results

In this chapter, we give the details of our experiment setup, the devices that we have tested our attack on and finally present the performance degradation observed by the selected benchmarks apps.

7.3.1 Experiment Setup

In order to test the level of degradation in the quality of service that the attack can cause, we performed experiments on various smartphones. Also, since regular Android apps generally do not provide performance statistics, we have used benchmarks to quantify the QoS degradation. In our experiments, we have collected performance measurements with and without the attacker app running in the background.

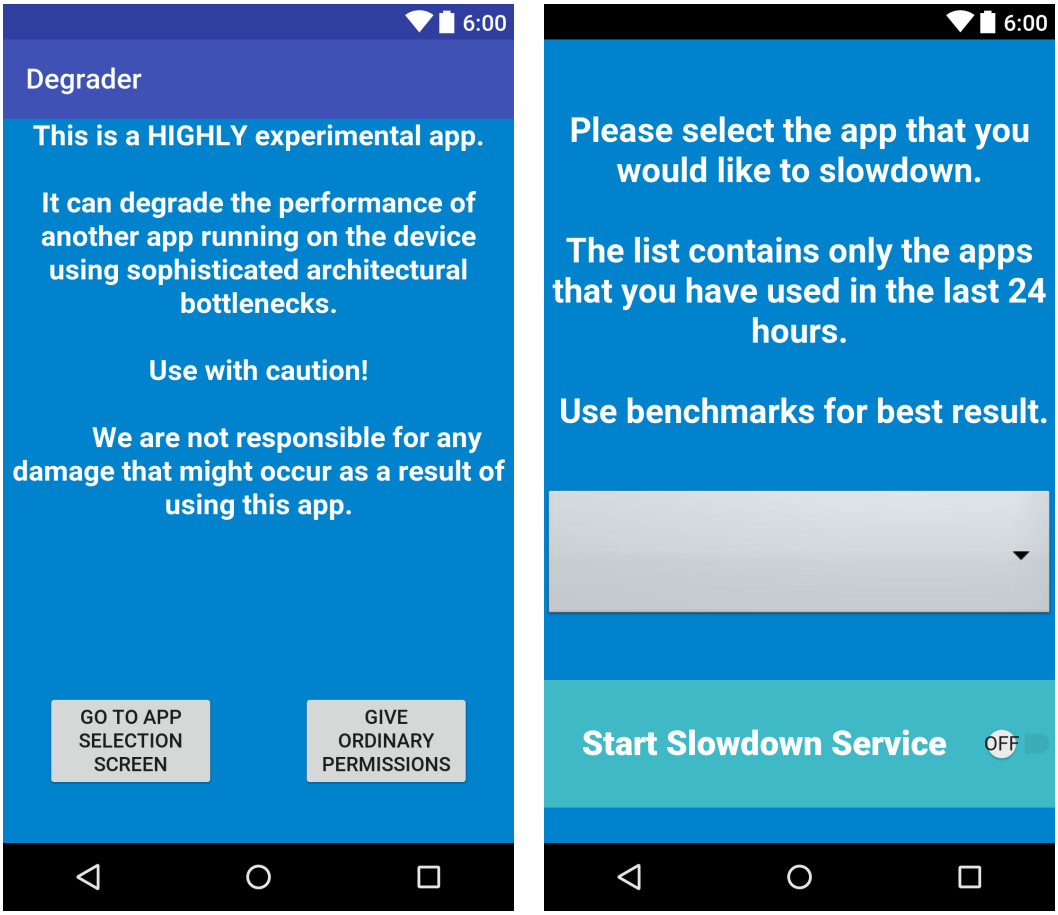


Figure 7.2: Attacker app interface.

As test platforms, we have used four different mobile devices namely Galaxy S2, Nexus 5, Nexus 5X and Galaxy S7 Edge. We have selected these devices to show the viability of the attack on different mobile CPUs. Also to add variety, we have updated these devices to different versions of Android. We have updated Galaxy S2, Nexus 5 and Nexus 5X and Galaxy S7 Edge to Android 4.1.2 (API Level 16), 5.1.1 (API Level 22), 6.0.1 (API Level 23), 6.0.1 (API Level 23) respectively. As expected, we were able to perform our attack on all test devices, regardless of what version of Android they had running. Note that we have not used any unofficial Android distribution, ROM or side-loaded any patches. All the test devices use their stock Android ROMs without any modification. We present the detailed specifications of

the devices in Table 7.1.

Since all Android devices employ a type of battery optimization, we wanted to make sure that our experiments would not be affected by this in any way. Therefore, to ensure optimal performance, all the experiments were performed while the test devices were fully charged and connected to a power outlet. Furthermore, Android monitors the data from the temperature sensors and adjusts the system performance to prevent overheating. To prevent any slowdown due to high temperature, we have placed the test devices apart from each other made sure that they are properly ventilated.

Table 7.1: Specifications of the test devices used in experiments

	Galaxy S2	Nexus 5	Nexus 5X	Galaxy S7 Edge
Android Version	4.1.2	5.1.1	6.0.1	6.0.1
API Level	16	22	23	23
SoC	Exynos 4	Snapdragon 800	Snapdragon 808	Snapdragon 820
ARM Core	Cortex-A9	Krait 400	4xCortex-A53 + 2xCortex-A57	4x Kryo
Number of Cores	2	4	4+2	2+2
CPU Clock (GHz)	1.2	2.26	1.4 + 1.8	1.6 + 2.15
Big-Little	no	no	yes	yes
CPU Architecture	32-bit	32-bit	64-bit	64-bit
ARM version	v7-A	v7	v8-A	v8-A

7.3.2 Test Targets: Performance Benchmarks

In order to quantify the level of degradation caused by the attack, we have used benchmarks apps, performing various tests. With these tests, we have measured the performance of the devices performing high-level operations such as 3D processing, 2D image processing, streaming as well as low-level tests like ALU computations, memory read and write access time, bandwidth and latency.

To prevent any bias and provide a fair comparison, we have used the top down-

loaded benchmarks in the Google Play store. In total, we have used 11 benchmark apps. Other than the CPU Prime Benchmark, all of these benchmarks has numerous available tests to score different aspects of the device. For instance, the AnTuTu Benchmark performs 3D computation, CPU, RAM, and UX (User Experience) tests to measure different aspects of the device performance, providing a separate score for each category. Including these tests, we have monitored the QoS strength of our app with 45 different tasks. Note that while some of the benchmark apps like 3DMark, GeekBench, GFXBench, and PCMark were not available for the older Galaxy S2 running Android 4.1.2, rest was available to all of the tested devices.

Complete list of benchmarks that we have used is as follows; 3DMark (Ice Storm Unlimited, Slingshot Unlimited), AnTuTu Benchmark (3D, CPU, RAM, UX), Benchmark&Tuning, CF-Bench (Java, Native, Overall), CPU Prime Benchmark, Geekbench 4, GFXBench GL (ALU 2, Driver 2, Manhattan 3.0, Texturing, T-Rex), PassMark Performance Test Mobile (2D Graphics, 3D Graphics, CPU, Disk, RAM, System), PCMark (Storage, Word, Work 2.0), Quadrant Standard (CPU, I/O, Memory, Overall), Vellamo (Browser - Chrome, Metal, Multi-Core).

7.3.3 Degradation Results

As mentioned earlier, regular Android apps generally do not output performance statistics. While the slowdown result of the attack and other effects are visible to the human eye, the visual slowdown is not quantifiable. To overcome this problem, we have decided to use benchmarks that can output system performance at any given time. To demonstrate and quantify the performance degradation caused by our QoS attack on different functions of the system and apps, we have used numerous benchmarks. This allowed us to measure system performance both with (degraded performance) and without (baseline performance) the attack running in

the background, giving us a clear contrast for each test device. Here, we present these results.

Our results show that we can significantly degrade the QoS of various apps. As shown in Figure 7.3, test benchmarks show varying levels of performance degradation, up to 90.98% compared to the baseline results. Also, it is evident that different test devices show diverse levels of degradation due to the difference in the underlying hardware. For instance, Nexus 5 has an average degradation of about 20% while Nexus 5X has about 45%.

Figure 7.4 shows how much performance degradation was observed by each benchmark when targeted by the attack. It is evident that almost all benchmarks show strong degradation in performance. Also, as mentioned before, one of our test devices, the Galaxy S2 with Android 4.1.2 did not support all of the benchmarks. These unavailable tests are shown with 0% degradation in the figure. In Figure 7.5, we represent normalized benchmark results i.e. ratio of degraded performance to baseline results. Note that like Figure 7.4, benchmarks are numbered as in Table 7.2.

Names of the benchmark suites and the specific tests are given in Table 7.2. Note that, while many of these benchmarks have subtest, they are not represented for clarity. Instead of giving results for each subtest, we only represent the average degradation. Finally, benchmark scores vary greatly depending on the computation power of the device as expected. Therefore, we represent degradation percentages rather than actual scores.

7.3.4 Stealthiness of the Attacker App

Our attack is hard to detect and runs with a minimal footprint on the system. During our experiments, we have continuously monitored the CPU usage of our app through Android Monitor provided by the `Android Studio`. We have observed

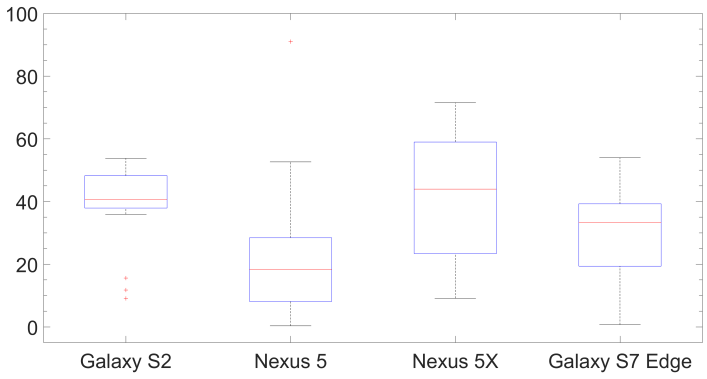


Figure 7.3: Quartile representation of benchmark performance degradations. Vertical axis represents performance degradation percentage compared to the baseline and red lines mark the average degradation.

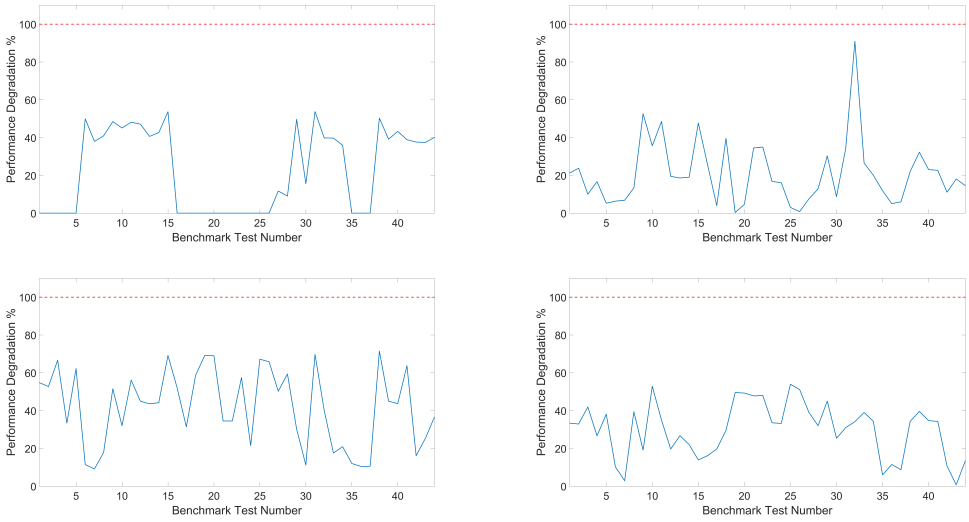


Figure 7.4: Performance degradation percentages of benchmarks on test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively. Red dashed lines represent the maximum possible degradation, i.e. 100%. Benchmarks are numbered as in Table 7.2.

that our app has low CPU usage, even at the times of performing Exotic Atomic Operations continuously. For all the tested devices, CPU usage of the attacker app never exceeded 10% mark, showing light CPU usage.

To show that our app is stealthy and can pass modern malware scanners, we

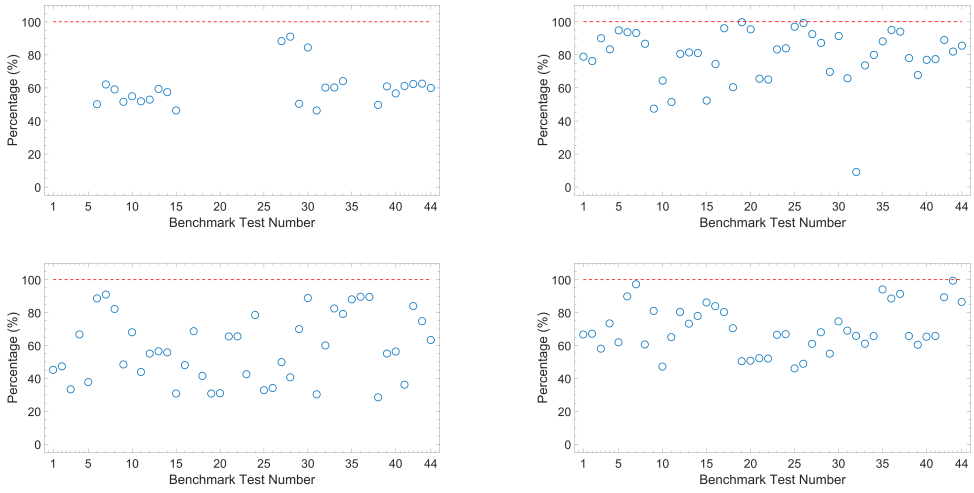


Figure 7.5: Normalized benchmark results of test devices, Galaxy S2, Nexus 5, Nexus 5x and Galaxy S7 Edge respectively. Red dashed lines represent baseline performance for each benchmark while blue lines represent results under attack. Benchmarks are numbered as in Table 7.2.

have used 23 of the most popular malware scanners on the Google Play Store. At the time of testing, none of the malware scanners (see Appendix 7.3.4) were able to detect our attacker app as malware even though it was causing significant distress to the operation of the device. We believe that this is due to the following reasons:

1. Unlike other micro-architectural attacks e.g. Rowhammer and cache attacks (Flush+Reload Evict&Reload, Prime+Probe etc.), our QoS attack does not require evicting memory blocks from the cache. Since it does not require eviction or continuous monitoring of a specific memory block, there is no continuous data access.

2. The memory bus trigger does not have to run at a high frequency to achieve performance degradation. In fact, a single bus lock results in a performance bottleneck for over 12395, 9766, 6128 and 4693 nanoseconds on average on our test devices, as shown in Figure 7.1. So, as long as the attacker can trigger the bus lock about every 10K+ cycles, the system will stay in a continuous

Table 7.2: QoS attack performance degradation quantified by various benchmarks.

Benchmark	Galaxy S2 %	Nexus 5 %	Nexus 5X %	Galaxy S7 Edge %
1. 3DMark (Ice Storm Unlimited Graphics Average)	NA	21.15	54.78	33.36
2. 3DMark (Ice Storm Unlimited Physics Average)	NA	23.82	52.73	32.87
3. 3DMark (Slingshot Unlimited Graphics Average)	NA	10.00	66.67	41.91
4. 3DMark (Slingshot Unlimited Physics Average)	NA	16.67	33.33	26.57
5. AnTuTu Benchmark (3D)	NA	5.25	62.25	38.12
6. AnTuTu Benchmark (CPU)	49.90	6.36	11.44	10.18
7. AnTuTu Benchmark (RAM)	37.97	6.83	9.08	2.84
8. AnTuTu Benchmark (UX)	40.93	13.34	17.81	39.40
9. Benchmark & Tuning (CPU)	48.43	52.61	51.53	19.06
10. Benchmark & Tuning (I/O)	45.08	35.65	31.97	52.87
11. Benchmark & Tuning (Memory)	48.13	48.56	56.14	34.96
12. CF-Bench (Java)	47.11	19.48	44.94	19.63
13. CF-Bench (Native)	40.61	18.61	43.55	26.73
14. CF-Bench (Overall)	42.59	19.01	44.16	22.00
15. CPU Prime Benchmark	53.66	47.73	69.19	13.85
16. Geekbench 4 (Compute Overall)	NA	25.61	51.96	16.12
17. Geekbench 4 (Multi-C Overall)	NA	3.96	31.36	19.69
18. Geekbench 4 (Single-C Overall)	NA	39.62	58.52	29.50
19. GFXBench GL (ALU 2 (Frames))	NA	0.31	69.20	49.54
20. GFXBench GL (ALU 2 Offscreen (Frames))	NA	4.63	68.99	49.26
21. GFXBench GL (Driver 2 Overhead (Frames))	NA	34.50	34.50	47.74
22. GFXBench GL (Driver 2 Overhead Offscreen (Frames))	NA	34.98	34.47	47.98
23. GFXBench GL (Manhattan 3.0 (Frames))	NA	16.78	57.48	33.53
24. GFXBench GL (Manhattan 3.0 Offscreen (Frames))	NA	16.11	21.43	33.17
25. GFXBench GL (Texturing (MTexels/s))	NA	3.01	67.18	53.94
26. GFXBench GL (Texturing Offscreen (MTexels/s))	NA	0.81	65.84	51.11
27. GFXBench GL (T-Rex Offscreen)	11.71	7.50	50.17	39.06
28. GFXBench GL (T-Rex)	9.02	12.82	59.37	31.98
29. PassMark Performance Test Mobile (2D Graphics)	49.66	30.33	30.12	45.03
30. PassMark Performance Test Mobile (3D Graphics)	15.55	8.69	11.18	25.34
31. PassMark Performance Test Mobile (CPU)	53.75	34.23	69.71	30.95
32. PassMark Performance Test Mobile (Disk)	39.80	90.98	39.98	34.14
33. PassMark Performance Test Mobile (RAM)	39.74	26.45	17.50	38.98
34. PassMark Performance Test Mobile (System)	35.89	20.09	20.89	34.28
35. PCMark (Storage)	NA	11.89	12.00	5.96
36. PCMark (Work 2.0)	NA	5.03	10.40	11.47
37. PCMark (Work)	NA	6.00	10.54	8.66
38. Quadrant Standard (CPU)	50.31	22.03	71.50	34.26
39. Quadrant Standard (I/O)	39.12	32.28	44.94	39.56
40. Quadrant Standard (Memory)	43.30	23.06	43.69	34.68
41. Quadrant Standard (Overall)	38.88	22.65	63.78	34.23
42. Vellamo (Browser - Chrome)	37.62	11.06	16.08	10.70
43. Vellamo (Metal)	37.42	18.07	25.24	0.68
44. Vellamo (Multi-Core)	40.08	14.50	36.66	13.54

state of a bottleneck. The fact that the attacker issues only 1 CPU instruction every 10K+ cycles, keeps the CPU load minimal. If a user was to use a task manager to inspect CPU usages of different apps or check the system logs to do the same, he/she would only see CPU use of unsuspecting 10% by the attacker app. In addition to that, remember that the memory bus locking

part of our attack is active only when the target app is in the foreground. So, unless the target device supports split screen and multitasking, and both the task manager and the victim app support split screen, the user cannot even observe this 10% percent load. While the bus locking is not active, the load of our background service is nominal, between 0.01% and 1%.

3. The current malware and anti-virus scanners are not adequately suited to detect micro-architectural attacks or bottlenecks. In these types of attacks, the attack surface and threat models are completely different from those of classical malware.

4. The attack is hard to detect with dynamic or static analysis techniques because the attacker app performs legitimate data operations and does not attempt to access any unauthorized APIs or data. Any app can perform atomic operations and trigger the bus lock without an ill-intent. Therefore it is not practical to simply detect these instructions. Further, the attacker app does not contain any information like the package name of the victim app which could give away the attack during static analysis. As for dynamic analysis, the scanner would have to trigger a bus lock which would require running the victim app which may not be possible if the victim is later provided to the app via the web connection. Finally, since no compute intensive operation like increment counters are used in the attack, it would be so very hard to observe the degradation during the scan and tag it as malware.

Complete List of Malware Scanners 360 Security - Antivirus Boost, Anti Spy (SpyWare Removal), AntiVirus FREE 2016 - Android, Avira Antivirus Security, Bitdefender Antivirus Free, CCleaner, Clean Master (Boost & AppLock), CM Security AppLock AntiVirus, Comodo Mobile Security, Dr. Safety - BEST FREE

ANTIVIRUS, Droidkeeper - Free Antivirus, FireAMP Mobile, FREE Spyware & Malware Remover, GO Security, Antivirus AppLock, Hidden Device Admin Detector, Kaspersky Antivirus & Security, Lookout Security & Antivirus, Malware Sleuth, Malwarebytes Anti-Malware, McAfee SpyLocker Remover, Mobile Security & Antivirus, Norton Security and Antivirus, Open Vaccine, Security & Power Booster -Free, Stubborn Trojan Killer.

7.4 Detection and Mitigation

In order to detect and mitigate this attack, we propose the following countermeasures. **Atomic Operation Alignment Check:** The operating system can inspect atomic operation operands for uncacheable memory addresses. When such an address is detected, the OS that can increment the count of recent atomic operations on uncacheable addresses. If the count reaches a preselected threshold, then the OS can give the app issuing these atomic operations a timeout or shut it down altogether. Though this countermeasure would incur a performance overhead, the penalty would be smaller than bus locking. Alternatively, the OS can move the operating data to a cacheable address and prevent bus locks.

Memory Bandwidth Monitoring: The operating system can periodically monitor memory bandwidth and stop or slow down any process that triggers bus locking frequently. In the case of an active attack, this method would allow the OS to detect the culprit and stop the process/app while allowing non-attack locks to be performed as usual.

Closing Inter-app Logical Channel Leaks: As discussed in Section 7.2 in detail, there are numerous ways to know which apps are installed or running on an Android device. When this information is obtained by the attacker, it is a matter of resource

monitoring (through logical or side-channels) to detect victim launch. So, if an attacker cannot obtain any information about installed or running apps on the device, then the detection stage would rely only on more noisy side-channels like shared hardware monitoring. While this countermeasure would strongly impact existing apps using information like installed apps, it would also make the detection part of the attack more difficult.

Restricting Access to UsageStats: UsageStats permission is used by numerous apps that need to monitor user's app usage. For instance, a mobile data operator might want to check which apps are currently active so that they can disable the data usage counter for a specific app e.g. Youtube or Netflix. However, it is also dangerous permission in the sense that it looks innocent to a naive user but has crucial consequences as demonstrated. So, instead of removing this permission altogether, making it a signature/system level permission would protect this critical information while allowing trusted parties e.g. data operators to keep the functionality.

7.5 Conclusion

In conclusion, we show that low-level architectural attacks are a real threat in mobile devices. Battery, compute power and storage restrictions of mobile devices require strong optimizations and create very suitable candidates for these types of attacks. Further, combining architectural attacks with logical channel leakages open a wide range of exploits for malicious parties. In this chapter, we have shown that malicious parties can exploit the underlying shared hardware to degrade or even halt operations of other apps using this combination and inter-app interactions have to be controlled carefully.

7.6 Ethical Concerns and Responsible Disclosure

The attacker app that we have designed is uploaded to the Google Play Store and since it is not detected as malware by the Play Store, it is currently available for download. Since the app causes a performance degradation to the overall system, we have clearly warned users in the install page that this is an experimental app aiming to degrade the performance of the overall system and should be used with caution. We hope that this warning will be sufficient to prevent any accidental installations that might result in unwanted performance degradation. Further, to prevent hogging of the system resources, we have significantly limited the performance degradation power of the app by decreasing the frequency of memory bus lock triggers. Finally, we have informed the Android security team of our findings in advance to this publication and made our bug submission through the AOSP bug report portal.

Chapter 8

Machine Learning for and against Side-channel Attacks

8.1 Motivation

Side-channel attacks (SCA) require analysis of large amounts of noisy data with human interpretation and intuition. This requirement presents a challenge for automation and scalability of SCAs. In a real-world scenario, it is not practical for an attacker to spend long periods of time to find a side-channel vulnerable target, obtain leakage traces and then analyze these traces to recover sensitive information that may or may not be there. Such a task however is perfectly suited for modern machine learning (ML). Moreover, Adversarial Learning, a subfield of ML, can be utilized to create smart noise with minimal changes to the input but change the output of ML systems drastically. Taking advantage of this, we propose to retrofit Adversarial Learning as a countermeasure to side-channel attacks. In this chapter, we show in detail how ML can be utilized both to perform side-channel attacks and to prevent them.

In the last decade, ML solutions have been applied to variety of tasks such as image classification [52, 51, 163], speech recognition [49], lip reading [50], verbal reasoning, self-driving cars, playing competitive video games [164, 165] and even writing novels [166]. As expected with any booming technology, it is also utilized by malicious actors. For instance, there have been cases of AI-generated content on the Internet boards to create or shift public opinion by social engineering. The latest known example being the AI-generated fake comments on the FCC net-neutrality boards where millions of AI-generated comments were posted [167, 168]. These messages were grammatically and semantically sound and not easily detectable as crafted. Another malicious use of AI is for spam and phishing attacks. Using the sentiment analysis, hackers are now crafting tailored phishing e-mails that have higher 'yield' rates than ones written by humans [169, 170]. Beyond crafting semantically sound text, modern AI system can even participate in hacking competitions where human creativity and intuition was thought to be irreplaceable.

According to a survey among cybersecurity experts [171], the use of AI for cyber-attacks will inevitably become more common over time and the academic literature already shows the use of ML for SCA. In 2011, Hospodar et al. [172] demonstrated the first use of ML, LS-SVM specifically, on a power SCA on AES, showing that the ML approach yields better results than traditional template attacks. Later, Heuser et al. [173] showed the superiority of multi-class SVM for noisy data in comparison to the template attacks. In 2012, Zhang et al. [34] demonstrated the use of multi-class SVM to extract RSA decryption keys from noisy side-channel leakage trace. In addition to SVMs, Neural Networks (NN) is also a popular tool among side-channel researchers. Martinasek et al. [174, 175] showed that NNs could be used to classify AES keys from power measurements with a success rate of 96%. In 2015, Beltramelli [176] used LSTM NN to collect meaningful keystroke data via

motions of the smart-watch user. In 2016, Maghrebi et al. [177] compared four DL based techniques with template attacks to attack an unprotected AES implementation using power consumption and showed that CNN outperforms template attacks. Finally in 2017, Gulmezoglu et al. [108] showed that ML can be used to extract meaningful information from the cache side-channel to recover web traffic.

The most straightforward countermeasure against side-channel attacks is to drown the sensitive computation with noise to cloak it from the attacker. However, this defense method is computationally expensive. In this chapter, we show that we can do much better than adding random noise. By using adversarial learning (AL), we can craft significantly smaller, intelligent noise to accompany processes. By using AL, we achieve a better cloaking effect with much smaller changes to the trace and minimal overhead to the system. Also, the proposed defense does not require the redesign of the software or the hardware stacks, making it practically deployable. In addition to that, it can be deployed as an opt-in service that users can enable or disable at wish, depending on the sensitivity of specific computations.

In summary, as stated in [178], attacking an ML system is easier than defending it. We exploit this advantage and flip the position of the attacker and defender to use AL as a defensive tool. In this chapter, we propose the use of AL to cloak side-channel leakage of processes and to protect them from ML-capable adversaries.

Contributions

In this dissertation, we list micro-architectural threats to modern computing systems by presenting attacks, providing experiment results and proposing countermeasures.

This work shows the beneficial use of adversarial learning against ML equipped side-channel attackers and proposes a framework to efficiently cloak side-channel leakage. In addition, potential methods to bypass this defense have been tested and

proven ineffective. Specifically in this chapter we;

- show that processes running on a shared system can be accurately identified by a deep learning (DL) or a classical ML classifier by their side-channel leakage. To demonstrate, we classify 20 types of processes using readily available, high-resolution HPCs.
- investigate the effects of various leakage trace parameters like the number of features, samples, and data collection intervals on the classifier accuracy.
- use various AL methods to craft perturbations to test and quantify the efficiency of each method against the side-channel classifier.
- show how to execute AL perturbations side-by-side with the original process to cloak the side-channel leakage. We show that this is a strong defense against an AI-equipped attacker.
- show that even when an attacker hardens her classifier against AL with adversarial re-training and defensive distillation, AL methods can overcome these defenses and still fool the side-channel classifier.

8.2 Training Classifiers to Process Side-channel Leakage

There are many types of side-channel leakage such as EM, power, timing etc. Even for micro-architectural leakage, it may be stemming from the CPU cache, DRAM or Hardware Performance Counters (HPCs). This leakage can be extracted using various attack methods like Flush+Reload, Prime+Probe or by requesting data

directly from the OS. In this study, we chose to use HPCs as our leakage source to be processed by a machine learning classifier.

8.2.1 Profiling Software using HPC Traces

HPCs are special purpose registers that provide detailed information on low-level hardware events in computer systems. These counters periodically count specified event like cache accesses, branches, TLB misses and many others. This information is intended to be used by developers and system administrators to monitor and fine-tune the performance of applications. The availability of a specific counter depends on the architecture and model of the CPU. Among many available HPCs, we have selected the following 5 for the classification task;

1. **Total Instructions:** the total number of retired i.e. executed and completed CPU instructions.
2. **Branch Instructions:** the number of branch instructions (both taken and not taken).
3. **Total Cache References:** the total number of L1, L2, and L3 cache hits and misses.
4. **L1 Instruction Cache Miss:** the occurrence of L1 cache instruction cache misses.
5. **L1 Data Cache Miss:** the occurrence of L1 cache data cache misses.

We have selected these HPCs to cover a wide selection of hardware events with both coarse and fine-grain information. For instance, the `Total Instructions` does not directly provide any information about the type of instructions being executed. However, the execution time is dependent on the type of instructions even if the data is loaded from the same cache level. The time difference translates indirectly

into the total instructions executed in the given time period and leaks information.

The `Branch Instructions` HPC provides valuable information about the execution flow of a program. Whether the branches are taken or not taken, the total number of branches in the executed program remains constant for a given input and execution path. This constant in the leakage trace helps eliminate noise elements and increases classification accuracy.

The `Total Cache References` HPC provides similar information to the Branch Instructions HPC in the sense that it does not leak information about the finer details like the specific cache set or even the cache level. However, it carries information regarding the total memory access trace of the program. Regardless of the data being loaded from the CPU cache or the memory, the total number of cache references will remain the same for a given process.

The `L1 Instruction Cache Miss` and the `L1 Data Cache Miss` HPCs provide fine-grain information about the *Cold Start* misses on the L1 cache. Since the L1 cache is small, the data in this cache level is constantly replaced with new data, incrementing these counters. Moreover, separate counters for the instruction and the data misses allows the profiler to distinguish between arithmetic and memory intensive operations and increases the profile accuracy. Finally, all five of the HPCs are interval counters meaning that they count specific hardware events within selected time periods.

8.2.2 Experiment Setup

Classifier models are trained and tested on a workstation with 10-core Intel i7-7900X CPU, two Nvidia 1080Ti GPUs (Pascal architecture), and 64 GB of RAM. Training is performed using the GPUs to utilize parallelism and decrease training time. On the software side, the classifier model is coded using *Keras v2.1.3* with *Tensorflow-*

GPU v1.4.1 back-end and other Python3 packages such as *Numpy v1.14.0, Pandas, Sci-kit, H5py etc.*

Side-channel (the HPC) traces are collected from a server with Intel Xeon E5-2670 v2 CPU, running Ubuntu 16 LTS. This specific CPU model has 85 possible hardware events of which 50 are available to user-space. To access these HPCs, the *QuickHPC* [179] tool is used. QuickHPC is developed by Marco Chiappetta to collect high-resolution HPC data using the PAPI back-end. It provides a fast and easy-to-use interface to HPCs.

8.2.3 Classifier Design and Implementation

Here, we give the details of the design and implementation of classifiers that can identify processes using the system HPC trace. To show the viability of such classifier, we chose 20 different ciphers from the OpenSSL 1.1.0 library as the classification target. Note that these classes include ciphers with both very similar and extremely different performance traces e.g. AES-128, ECDSAB571, ECDSAP521, RC2, and RC2-CBC. Moreover, we also trained models to detect the version of the OpenSSL library for a given cipher. For this task, we used OpenSSL versions 0.9.8, 1.0.0, 1.0.1, 1.0.2 and 1.1.0.

8.2.3.1 Classical ML Classifiers

Here, we refer to non-neural network classification methods as classical ML classifiers. In order to contrast classical ML methods with CNNs, we trained a number of different models using the Matlab Classification Learning Toolbox. The trained classifiers include SVMs, decision trees, kNNs and variety of ensemble methods.

8.2.3.2 CNN Classifier

We designed and implemented the CNN classifier using Keras with Tensorflow-GPU back-end. The model has a total of 12 layers including the normalization and the dropout layers. In the input layer, the first convolution layer, there are a total of 5000 neurons to accommodate the 10 milliseconds of leakage data with 5000 HPC data points. Since the network is moderately deep but extremely wide, we used 2 convolution and 2 MaxPool layers to reduce the number dimensions and extract meaningful feature representations from the raw trace.

In addition to convolution and MaxPool layers, we used batch normalization layers to normalize the data from different HPC traces. This is a crucial step since the hardware leakage trace is heavily dependent on the system load and scales with overall performance. Due to this dependency, the average execution time of a process, or parts of a process can vary from one execution to another. Moreover, in the system-wide leakage collection scenario, the model would train over this system load when it should be treated as noise. If not handled properly, the noise and the shifts in the time domain results in over-fitting the training data with the dominant average execution time, decreasing the classification rate. By using the batch normalization layer, the model learns the features within short time intervals and the relation between different HPC traces. On the output layer, we use 20 neurons with softmax activation, representing 20 classes of processes. Finally, we use *Categorical Cross-entropy* loss function with the *Adam Optimizer* to train the model.

The CNN classifier is constructed using the layers given below;

1. Convolution layer (50, (10,1))
2. MaxPool Layer (10,1)
3. Batch Normalization Layer
4. Dropout Layer (0.25)

5. Convolution Layer (100, (10,1))
6. MaxPool Layer (10,1)
7. Batch Normalization Layer
8. Dropout Layer (0.25)
9. Flatten Layer
10. Dense Layer (400)
11. Dropout Layer (0.25)
12. Dense Layer (20)

The classifiers are trained to identify 20 classes of crypto ciphers as well as five different versions of OpenSSL, as detailed in Section 8.2.3. Unless otherwise stated, the classifier is trained with data from the cipher implementations of OpenSSL 1.1.0, the same version we use to craft adversarial samples against in later experiments. Moreover, to show the versatility of the classifier i.e. that it is not limited to a certain process type or a library, we trained models to classify 20 different ciphers from OpenSSL versions 0.9.8, 1.0.0., 1.0.1, 1.0.2 and 1.1.0. Further, we trained models to distinguish between different versions of the same process e.g. OpenSSL 1.0.0 implementation of AES-128-CBC vs any other version of the OpenSSL.

8.2.4 Classification Results

8.2.4.1 Classical ML Classifiers

In our training of ML classifiers, the first challenge was the fact that the side-channel leakage data is extremely wide. We have chosen to train our models with samples consisting of 1000 data points per HPC with 5 HPCs total. This parameter selection is done empirically to provide good cloaking coverage and train highly accurate models as explained in Section 8.2.4.2. Using 1000 data points with 10

micro-second intervals per HPC allows us to obtain useful data in a short observation window. Nevertheless, 5000 dimensions are unusually high for classifiers, especially for a multi-class (20) classifier. To find the optimal setting for the hardware leakage trace, we tried different parameters with each classifier. For instance, in the case of decision trees, we have trained the 'Fine Tree', 'Medium Tree' and 'Coarse Tree' classifiers, allowing 5, 20, and 100 splits (leaves in the decision tree) respectively. For the case of Gaussian SVM, fine, medium and coarse refers to the kernel scale set to sqrt(P)/4, sqrt(P) and sqrt(P)*4 respectively. As for the kNN, the parameters refer to the number of neighbors and the different distance metrics. Results for the classical ML classifiers are presented in Table 8.1 and show that these models can, in fact, classify processes using their HPC traces. Note that the Quadratic Discriminant did not converge to a solution before PCA hence no score is given in the table.

8.2.4.2 CNN Classifier

For the CNN classifier, we firstly investigated the effect of the number of HPCs collected and trained our models for 100 epochs with data from a varying number of HPCs. Not surprisingly, even with only 1 HPC, our CNN classifier achieved 81% validation accuracy, although after a high number of epochs. Moreover, after the 30th epoch, the model over-fitted the training data i.e. the validation accuracy started to drop while the training accuracy kept increasing. When we increased the number of HPCs collected, our models became much more accurate and achieved over 99% validation accuracy as seen in Figure 8.2. Moreover, when we use the data from all 5 HPCs, our model achieved 99.8% validation accuracy in less than 20 epochs. While our validation accuracy saturates even with only 2 HPCs, *Total Instructions* and *Branch Instructions* we have decided to use all 5 of them. We

Table 8.1: Application classification results for the classical ML classifiers with and without PCA feature reduction.

Classification Method	**Without PCA**	**With PCA (99.5% variance)**
Fine Tree	98.7	99.9
Medium Tree	85.4	94.8
Coarse Tree	24.9	25
Linear Discriminant	99.6	99.7
Quadratic Discriminant	N/A	99.4
Linear SVM	99.9	98.2
Quadratic SVM	99.9	96.9
Cubic SVM	99.9	94.3
Fine Gaussian SVM	40	88.2
Medium Gaussian SVM	98.3	92.1
Coarse Gaussian SVM	99.7	13.4
Fine kNN	96.8	11.1
Medium kNN	94.9	7.8
Coarse kNN	85.5	5.2
Cosine kNN	92.5	19.6
Cubic kNN	85.2	7.7
Weighted kNN	95.9	8.3
Boosted Trees	99.2	99.8
Bagged Trees	99.9	94.8
Subspace Discriminant	99.8	99.7
Subspace kNN	84.8	88.1
RUSBoosted Trees	76	92.8
Best	**99.9**	**99.9**

made this decision because, in a real-world attack scenario, an attacker might be using any one or more of the HPCs. Since it would not be known which specific hardware event(s) an attacker would monitor, we decided to use all 5, monitoring different low-level hardware events to provide a comprehensive cloaking coverage.

To determine the optimum number of features per HPC, we have trained multiple models with varying input sizes. As shown in Figure 8.1, the validation accuracy saturates at 1000 and 2000 features and the validation loss increases after 1000 features. For this reason, we chose to use 1000 features for our experiments.

After deciding to use data from 5 HPCs, we investigated how the number of

training samples affects the CNN classifier validation accuracy. For that, we have trained 6 models with a varying number of training samples. For the first model, we have used only 100 samples per class (2000 samples in total) and later on trained models with 300, 1000, 3000, 10000 and 30000 samples per class. In the first model, we achieved 99.8% validation accuracy after 40 epochs of training. When we trained models with more data, we have reached similar accuracy levels in much fewer epochs. To make a good trade-off between the dataset size and training time, we have opted to use 1000 samples per class. This model reaches 100% accuracy with 20 epochs of training as shown in Figure 8.3. Finally, our last model achieved 100% accuracy just after 4 epochs when trained with 30000 samples per class.

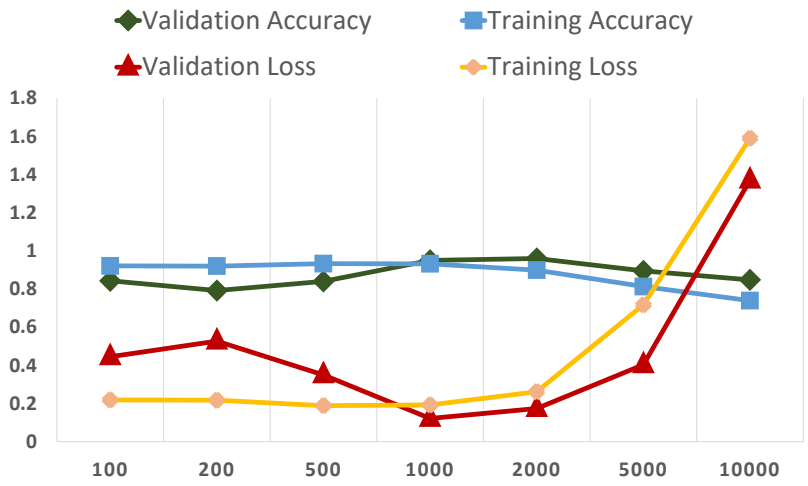


Figure 8.1: Results for the CNN classifier trained using varying number of features. Models reach highest validation accuracy with 1000 and 2000 features.

We also verified that different versions of OpenSSL can be distinguished for each cipher. For each of the 20 analyzed ciphers, we built classifiers to identify to which of the five analyzed versions they belong. Figure 8.4 presents the classification results of the two models trained using 1-5 HPC traces respectively. As cipher updates between versions can be very small, the added information from sampling several

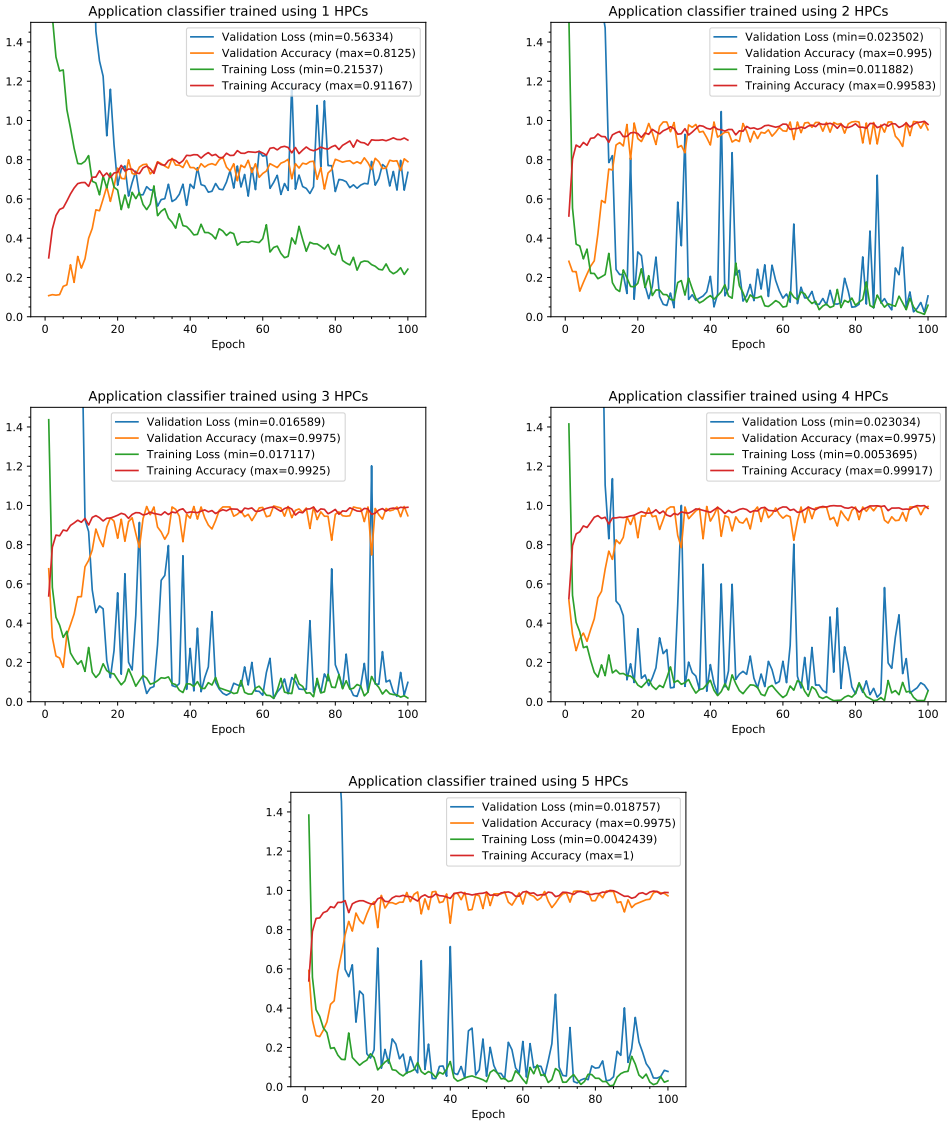


Figure 8.2: Results of CNN classifiers trained with varying number of HPCs. Even using data from a single HPC trace is enough to achieve high accuracy top-1 classification rate, albeit taking longer to train.

HPCs is useful to obtain high classification rates, as can be seen from the results.

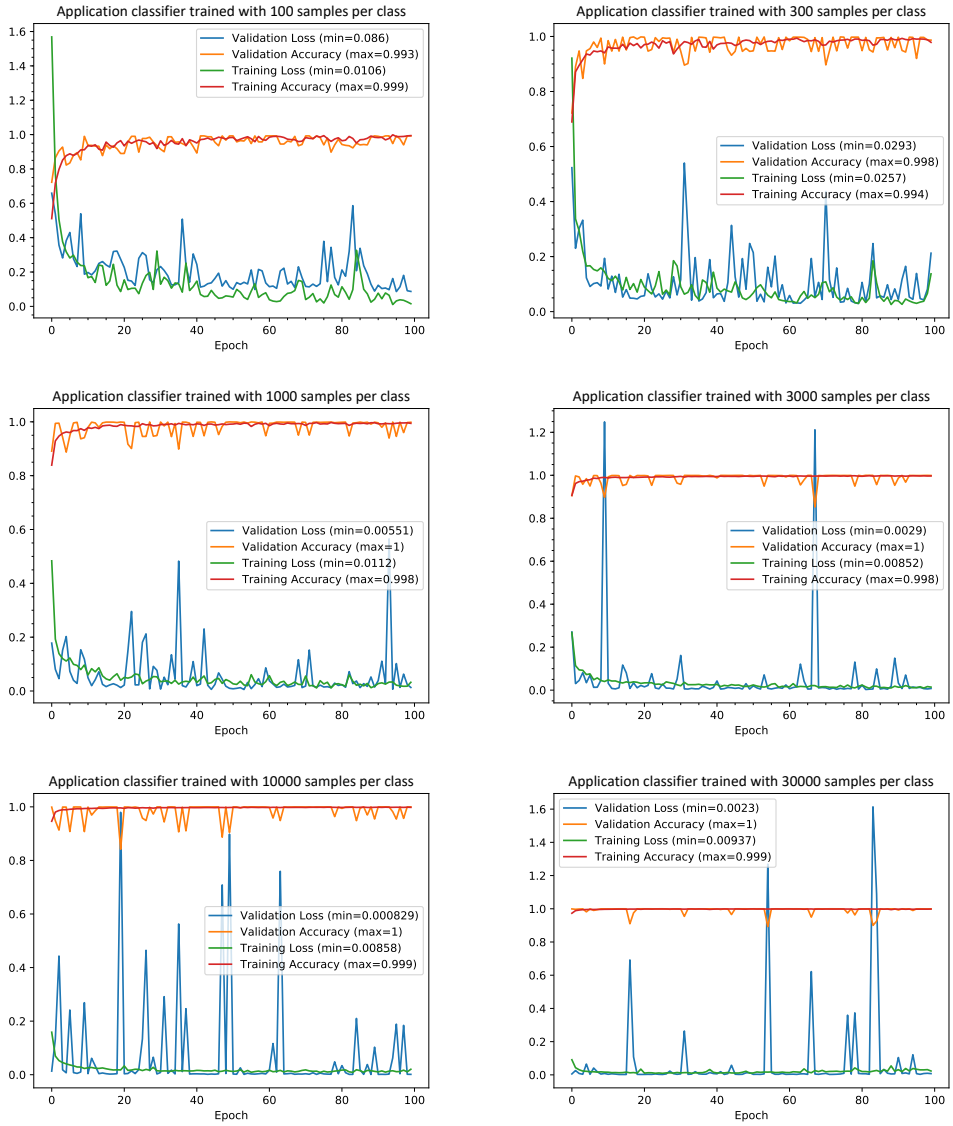


Figure 8.3: Results of CNN classifiers trained with 100 and 1000 samples per class. The first model reaches 99% accuracy in 40 epochs. When the number of samples per class is increased to 1000, we achieve same accuracy in a few epochs of training.

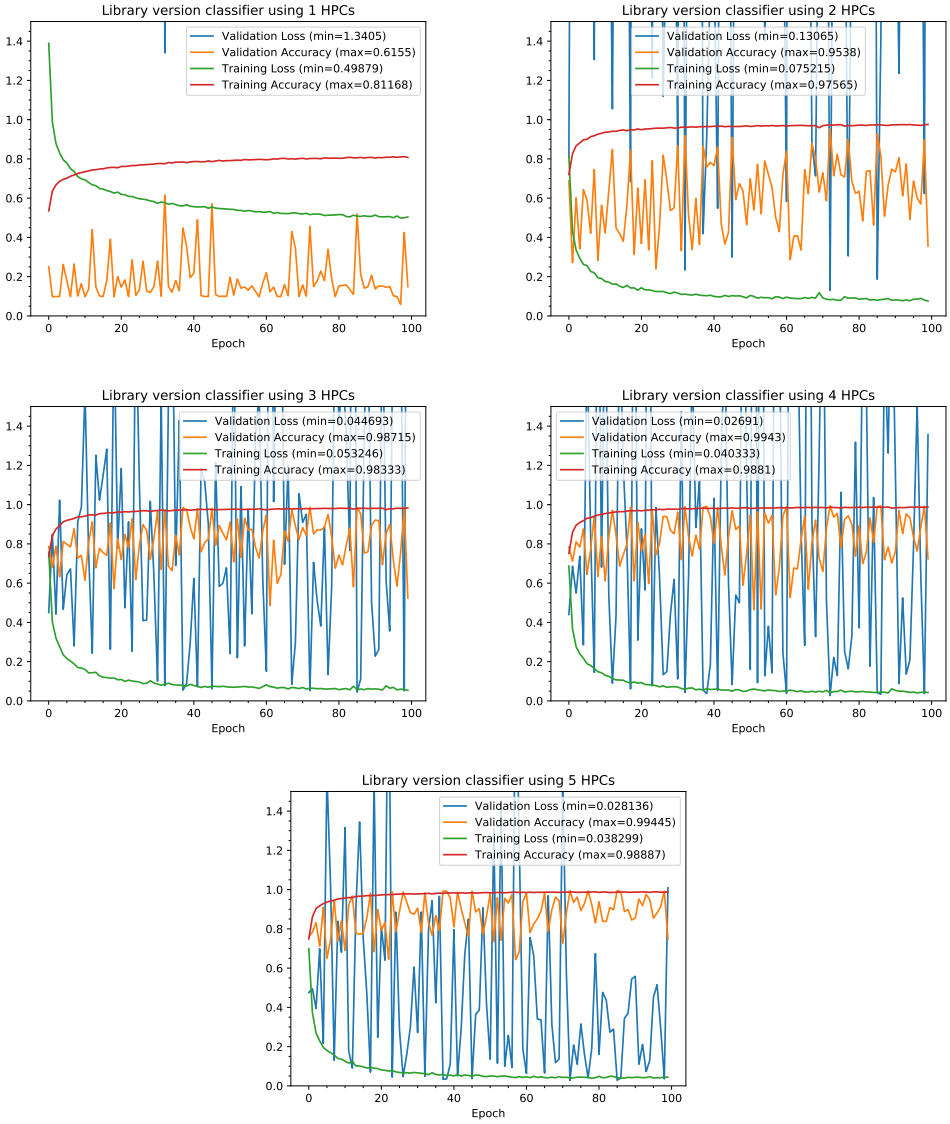


Figure 8.4: Results of the CNN trained for OpenSSL version detection, using varying number of HPCs. When trained with only a single HPC, the validation accuracy saturates at 61%. When all 5 HPCs are used, the validation accuracy reaches 99%.

8.3 DeepCloak, A Framework to Cloak Side-channel Leakage

In previous chapters, we have demonstrated that side-channel leakage can be processed by both classical machine learning models and convolutional neural networks. Here, we propose Deepcloak as a countermeasure against such processing. We argue that, due to the heavy manual processing overhead of side-channel attacks, it is very likely that malicious parties will resort to using machine learning. With the use of machine learning, attackers can both automate and scale side-channel attacks, especially in cloud environment. In this tangent, we propose to exploit an inherent machine learning weakness, vulnerability to adversarial learning, as a way to defend against such attackers.

DeepCloak provides a framework to profile security sensitive code and crafts adversarial noise that will cloak the code's side-channel leakage. DeepCloak achieves this creating executable adversarial perturbations on the system. Our goal is to show that side-channel classifiers, specifically the complex and powerful DL-based classifiers, can be successfully fooled by utilizing AL as a defensive tool. To validate this hypothesis, we first train DL-based classifiers using real side-channel leakage. Then, we show the classification accuracy degradation as the result of AL methods. And finally, we show that even if the DL-based classifier is aware of the DeepCloak and applies adversarial re-training or defensive distillation, the outcome does not change and the defense holds. In our experiments, we take the following steps:

1. Train the process classifier C using side-channel leakage.

2. Craft AL samples δ to cloak the user processes and force C to misclassify.

3. Train a new classifier C' with defenses against AL; Defensive Distillation and

Adversarial Re-training.

4. Test previously crafted AL samples δ against the hardened classifier C'. Then craft and test new AL samples δ' against the hardened classifier C' and show that the AL method still succeeds.

5. Show the use of code gadgets to execute the necessary perturbations on an x86 system.

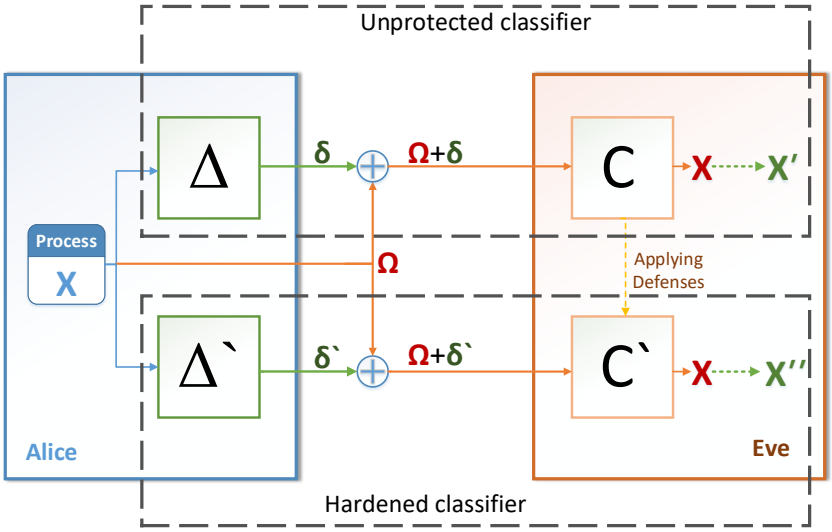


Figure 8.5: The outline of the cloaking methodology.

We outline this methodology in Figure 8.5. In the first stage, Alice, the defender runs the process $\mathbf{X}$, creating the leakage $\mathbf{\Omega}$. Eve, the attacker monitors the leakage and identifies the process $\mathbf{X}$ via the classifier, C. Then, Alice crafts and executes the perturbation δ in a separate thread alongside $\mathbf{X}$, forcing C to misclassify $\mathbf{X}$ as $\mathbf{X}'$. Eve then hardens $\mathbf{C}$ with AL defenses. Now, Eve can classify $\delta' + \mathbf{\Omega}$ partially correct. In the final stage, Alice first tests the previously crafted adversarial samples against Eve's hardened classifier C'. Then, Alice updates her adversarial sample crafting target to fool C' rather than the original classifier C. Alice crafts δ' against $\mathbf{C}'$, and $\mathbf{X}$ is again misclassified.

We apply this methodology to a scenario where an attacker trains a CNN to classify running processes using the system HPC leakage trace as the input. This information is extremely useful to the attacker since it helps to choose a specific attack or even pick a vulnerable target from many. Once a vulnerable target is found, then the attacker can perform micro-architectural or application-specific attacks. To cloak this information leakage, the defender attempts to mask the process signature. The masking should not interfere with the running process or create too much overhead to the overall system. This is why crafting minimal perturbations is crucial to the practicality of our proposed defense.

The attacker periodically collects five HPC values for 10 milliseconds total with 10-microsecond intervals. This results in a total of 5000 data points per trace. Later, the trace is fed into classical machine learning and DL classifiers. In this chapter, we explain our choice of the specific HPCs, the application-classifier design and implementation details, the AL methods applied to these classifiers. We finally test the efficiency of adversarial re-training and defensive distillation against our cloaking method.

8.3.1 Adversarial Learning Attacks

AL remains an important open research problem in AI. Traditionally, AL is used to trick AI classifiers and test model robustness against malicious inputs. Here, however, we propose to use AL as a defensive tool to mask the side-channel trace of applications and protect against micro-architectural attacks and privacy violations. In the following, we explain the specific AL methods that we have used. We consider the following attacks:

- **Additive Gaussian Noise Attack (AGNA):** Adds Gaussian Noise to the input trace to cause misclassification. The standard deviation of the noise is

increased until the misclassification criteria are met. This AL method is ideal to be used in the cloaking defense due to the ease of implementation of the all-additive perturbations. A sister-process can actuate such additional changes in the side-channel trace by simply performing operations that increment specific counters like cache accesses or branch instructions.

- **Additive Uniform Noise Attack (AUNA):** Adds uniform noise to the input trace. The standard deviation of the noise is increased until the misclassification criteria are met. Like the AGNA, this attack is also easy to implement as a sister-process due to its additive property.

- **Blended Uniform Noise Attack (BUNA):** Blends the input trace with Uniform Noise until the misclassification criteria are met.

- **Contrast Reduction Attack (CRA):** Calculates perturbations by reducing the 'contrast' of the input image until a misclassification occurs. In the case of the side-channel leakage trace, the method smooths the trace by reducing the distance between the minimum and the maximum HPC counts within the trace.

- **Gradient Attack (GA):** Uses the loss gradient with regards to the input trace. The magnitude of the added gradient is increased until the misclassification criteria are met. The attack only works when the model gradient is available.

- **Gaussian Blur Attack (GBA):** Adds Gaussian Blur to the input trace until a misclassification occurs. Gaussian blur smooths the input trace and reduces the amplitude of outliers. Moreover, this method reduces the resolution of the trace and cloaks the fine-grain leakage.

- **Fast Gradient Sign Method (FGSM) [65]:** The method has been proposed by Goodfellow et al. in 2014. FGSM works by adding the sign of the elements of

the gradient of the cost function with regards to the input trace. The gradient sign is then multiplied with a small constant that is increased until a misclassification occurs.

- **L-BFGS-B Attack (LBFGSA) [180]:** The attack utilizes the modified Broyden-Fletcher-Goldfarb-Shanno algorithm, an iterative method for solving unconstrained nonlinear optimization problems, to craft perturbations that have minimal distance to the original trace. The attack morphs the input to a specific class. However, in our experiments, we did not target a specific class and chose random classes as the target.

- **Saliency Map Attack (SMA) [181]:** Works by calculating the forward derivative of the model to build an adversarial saliency map. This map reveals which input features e.g. pixels in an image, have a stronger effect on the targeted misclassification. Using this information, an adversary can modify only the features with high impact on the output and produce minimal perturbations.

- **Salt and Pepper Noise Attack (SPNA):** Works by adding Salt and Pepper noise (also called impulse noise) to the input trace until a misclassification occurs. For images, salt and pepper values correspond to white and black pixels respectively. For the side-channel leakage trace, however, these values correspond to the upper and the lower bounds in the trace.

8.3.2 AL Results on the Unprotected Model

After training and testing our side-channel classifier model, we proceeded to craft adversarial samples against this unprotected classifier. There are numerous publicly available libraries like Cleverhans [182], Foolbox [183] and IBM's ART [184], that can create adversarial samples against a given ML model. As long as the same

adversarial attack method is used, there is no difference between the produced adversarial perturbation. In our study, we have used the Foolbox library. Note that, it is also possible to create these perturbations with other libraries or manually, as the adversarial attacks are independent of the specific library used. The Foolbox library provides numerous AL attack methods and provides an easy to use API. For a given adversarial sample crafting method, the Foolbox calculates the necessary perturbations on a given input sample-classifier model pair to 'fool' the given model. Detailed information about these attacks can be found in Section 8.3.1.

Table 8.3 presents the classification accuracy of perturbed samples. As the results show, almost all test samples are misclassified by the classifier model with very high accuracy at over 86%. Another important metric for the AL is the L1-norm distance (L1D) and the L2-norm distance (L2D) of the perturbed input from the original. These metrics quantify the size of the changes i.e. perturbations made to the original trace by various AL methods. The difference between the L1D and the L2D is that the latter is more sensitive to the larger changes due to the square operation. For instance, if a perturbation requires a significant change in 1 sample point among the 5000 features, it will have a stronger impact in the final L2D value than average change distributed over a few points. L1D, however, is more dependent on the overall change in the trace, i.e. all 5000 sample points have the same impact on the final distance. Our results show that with most AL methods, perturbation L1D is around or well below 1% and within the ideal range. Remember, the smaller the perturbation, easier it is to actuate it as a cloaking process.

8.3.3 AL Results on the Hardened Model

Here we present the results of the AL methods on the hardened classifier models to show the robustness of our cloaking mechanism against classifiers hardened with

Table 8.2: Classification confidence and L1D results of the unprotected CNN classifier.

Adversarial Attack	Original Sample Classification Confidence.	Perturbed Sample MisClassification Confidence.	L1D
AGNA	99	96	0.00294
AUNA	99	97	0.00292
BUNA	99	99	0.05000
CRA	99	99	0.05254
GBA	99	97	0.00080
GSA	99	99	0.00499
LBFGSA	99	86	0.00025
SMA	99	92	0.00001
SPNA	99	96	0.01528

Adversarial Re-training and DD. To recap the scenario, Alice the defender wants to cloak her process by adding perturbations to her execution trace so that eavesdropper Eve cannot correctly classify what Alice is running. Then Eve notices or predicts the use of perturbations on the data and hardens her classifier model against AL methods using adversarial re-training and DD.

To test the attack scenario on hardened models, we first craft 100,000 adversarial samples per AL method against the unprotected classifier. Then we harden the classifier with the aforementioned defense methods and feed the adversarial samples. Here, we aim to measure the level of protection provided by the adversarial re-training and the DD methods.

As presented in Table 8.6, the application of both adversarial re-training and DD invalidates some portion of the previously crafted adversarial samples. For the adversarial re-training, the success rate varies between 99% (FGSM) and 4% (SPNA). In other words, 99% of the adversarial samples crafted using FGSM against the unprotected model are invalid on the hardened model. As for the DD, the rate ranges from 61% up to 100%. Impressively, 100% of the adversarial samples crafted using the BUNA are ineffective against the DD hardened model at distillation

Table 8.3: Classification confidence of unprotected and hardened (with adversarial re-training) classifiers under AL.

	Classification Conf.		**Misclassification Conf.**	
	Unprotected Classifier	Hardened Classifier	Unprotected Classifier	Hardened Classifier
AGNA	100	92	99.4	82
AUNA	100	91	99.25	82
BUNA	100	96	99.66	93
CRA	100	97	98.83	98
FGSM	99.24	88	99.65	97
GBA	100	93	99.32	74
LBFGSA	100	89	96.04	97
SLSQPA	100	89	100	72
SMA	100	88	95.66	63
SPNA	100	92	100	74

Table 8.4: L1-norm and L2-norm distances of adversarial samples crafted against unprotected and hardened (Adversarial Re-training) classifiers.

	Unprotected Classifier		**Hardened Classifier (Adv. Re-training)**	
	L1D	**L2D**	**L1D**	**L2D**
AGNA	0.00055	1.33E-06	0.00294	1.7E-05
AUNA	0.00063	1.26E-06	0.00332	1.7E-05
BUNA	0.00073	1.05E-06	0.05	3.7E-05
CRA	0.00142	4.29E-06	0.04999	3E-05
FGSM	4.93E-05	2.46E-09	0.00398	8.9E-07
GBA	0.00022	2.34E-07	0.00071	9.1E-06
LBFGSA	0.00178	2.54E-05	0.00596	4.1E-08
SLSQPA	0.49939	0.332691	0.00031	32.9833
SMA	5.53E-05	8.87E-08	0.00008	1.1E-07
SPNA	0.0002	0.000199	0.08268	0.1139

Table 8.5: Average L1Ds of perturbations crafted against hardened classifiers.

Adversarial Method	**Unprotected Model**	**Adversarial Re-trained**	**DD with T=1**	**DD with T=5**	**DD with T=10**	**DD with T=30**	**DD with T=50**	**DD with T=100**
AGNA	0.0029	0.0029	0.0004	0.0006	0.0106	0.0045	0.018	0.0043
AUNA	0.0029	0.0033	0.0003	0.0007	0.0111	0.0051	0.0179	0.0045
BUNA	0.05	0.05	0.0499	0.082	0.1029	0.0501	0.1638	0.0775
CRA	0.0525	0.05	0.05	0.0855	0.1078	0.05	0.2799	0.0745
FGSM	0.005	0.006	0.005	0.1367	0.0248	0.0106	0.0167	0.0055
GBA	0.0008	0.0007	0.0006	0.0006	0.0499	0.0017	0.0006	0.0008
LBFGSA	0.0003	0.0003	0.0053	0.0002	0.0021	0.0009	0.0052	0.0092
SMA	7E-06	8E-05	3E-05	7E-05	4E-05	1E-05	1E-05	2E-05
SPNA	0.0153	0.0827	0.0106	0.0036	0.018	0.0026	0.02	0.0198

Table 8.6: Effectiveness of adversarial re-training and DD on 100,000 previously crafted adversarial samples. The results show what percentage of previously successful adversarial samples are ineffective on the hardened models.

Adversarial Attack	Adversarial Re-training	DD with T=1	DD with T=2	DD with T=5	DD with T=10	DD with T=20	DD with T=30	DD with T=40	DD with T=50	DD with T=100
AGNA	42	77	60	70	83	83	63	64	62	75
AUNA	43	77	60	70	83	82	63	65	61	75
BUNA	94	92	92	91	94	94	94	96	94	100
CRA	94	95	99	94	94	94	94	99	88	95
FGSM	99	91	90	91	99	99	99	95	99	98
GA	97	99	72	83	99	99	96	80	90	90
GBA	84	83	84	88	82	94	93	91	93	88
LBFGSA	51	76	63	63	78	87	65	65	63	71
SMA	26	71	50	52	62	82	49	47	32	48
SPNA	4	84	76	78	94	93	76	79	73	80

temperature T=100.

Also, adversarial samples have up to 29% lower misclassification confidence compared to the unprotected model. The adversarial samples are still misclassified with high confidence, in the range of 63-98%.

In short, by using the adversarial re-training or the DD, Eve can indeed harden her classifier against AL. However, keep in mind that Alice can observe or predict this behavior and introduce new adversarial samples targeting the hardened models. Below, we discuss the results of our experiments against such hardened models.

8.3.3.1 Adversarial Re-training

After training the DL classifier model and crafting adversarial samples, we use these perturbations as training data and re-train the classifier. The motivation here is to teach the classifier model to detect these perturbed samples and correctly classify them. With this re-training stage, we expect to see whether we can 'immunize' the classifier model against given AL methods. However, as the results in Table 8.3 show, all of the AL methods still succeed albeit requiring marginally larger perturbations. Moreover, while we observe a drop in the misclassification confidence, it is still quite high at over 63% i.e. Eve's classifier can be fooled by adversarial samples.

8.3.3.2 Defensive Distillation

To implement defensive distillation, we have used the technique proposed in [72] and trained hardened models with DD at various temperatures ranging from 1 to 100. Note that while the DD has already been proven broken [185], we include it in our study to show the change of the adversarial perturbation size and do not suggest it as a proper defense against adversarial samples. Our results show that, even if the eavesdropper Eve hardens her classifier with DD, it is still prone to AL methods albeit requiring larger perturbations in some cases. In Table 8.5, we present the perturbation size as L1D i.e. the L1-norm distance, of various attack methods on both unprotected and hardened models. Our results show that the application of DD indeed provides some level of robustness to the model and increases the perturbation sizes. However, this behavior is erratic compared to the adversarial re-training defense i.e. the L1D is significantly higher at some temperatures while much smaller for others. For instance, the L1D for the AUNA perturbations against the unprotected model is 0.00292 in average for 100,000 adversarial samples. L1D for the same attack drops to 0.00033 when DD is applied with temperature T=1. This in turn practically makes Eve's classifier model easier to fool. The same behavior is observed with the adversarial samples crafted using AGNA and GBA as well. For the cases that DD actually hardens Eve's model against adversarial samples, the L1D is still minimal. Hence, Alice can still successfully craft adversarial samples with minimal perturbations and fool Eve's model.

8.3.4 Perturbation Execution

After the user process is profiled and an appropriate perturbation has been calculated, the defender needs to execute this perturbation. In order to do so, one can

either modify and fake the system HPC output through the OS/VMM or execute carefully crafted code snippets. Assuming an unprivileged user without the ability to modify the system HPC directly, we show the following methods to modify HPCs by execution. By using the methods below, one can execute AL perturbations and manipulate the overall system HPC trace. Since the attacker profiles the overall system execution trace, she cannot filter these modifications caused by our cloaking process.

The perturbation execution gadgets are assumed to run concurrently to the original process either as separate processes or threads. Each of the HPCs used in the profiling can be manipulated as explained below. Moreover, if the defender wants to implement a non-additive AL perturbation and decrease a specific HPC, the *NOP* instruction can be inserted into the program code. Whenever the NOP instruction is executed, it would halt the original process hence decrease HPC over a certain time period.

- **Total Instructions:** Execution and retirement of any instruction increments this HPC. If no instructions are executed and the CPU is stalled, this counter does not increase.

- **Branch Instructions:** Execution of any of the following x86 branch instructions can be used to increment this HPC; JNE (Jump if not equal), JE (Jump if equal), JG (Jump if greater), JLE (Jump if less than or equal), JL (Jump if less than), JGE (Jump if greater or equal).

- **Total Cache References:** A simple looped access to a cached variable increases this counter. Note that this HPC counts the access attempts to all cache levels. Moreover, even if the operation results in a cache miss, the attempt still counts as a cache reference and the count is incremented.

- **L1 Instruction Cache Miss:** The *ClFlush* instruction can be used to flush a previously executed instruction code e.g. a function, from the cache. Later, when this piece of code is executed again, it will cause a cache miss and increases the counter.

- **L1 Data Cache Miss:** This HPC can be incremented via multiple different methods. For instance, pointer-chasing buffers can be used to ensure cache misses. Alternatively, one can use the *ClFlush* instruction to flush previously accessed data from the cache and force a cache miss.

Chapter 9

Conclusion

Side-channel leakage on shared hardware systems poses a real and present danger to the security and the privacy of users. Even when the software is perfectly isolated, co-located tenants still share the underlying hardware. As we demonstrated in this thesis, the software isolation is not sufficient and shared resources leave cloud and mobile systems vulnerable to side-channel and QoS attacks. To demonstrate that side-channel attacks are now practical threats, we set out to implement such an attack on an actual cloud, outside of the controlled lab environment. In order to perform a side-channel attack on cloud, we tackled the co-location detection problem and devised 3 new methods to reliably detect co-location. Then we applied the Prime+Probeattack to recover an RSA secret key from a co-located virtual machine that was running on the same system. In addition to side-channels, we introduced and demonstrated a practical QoS attack. We implemented the attack within an Android app and showed that it is practical to successfully develop and deploy such a malware. The developed app was scanned by state-of-the-art malware scanners but none of the scanners were able to detect it as malicious.

Next, we showed that it is possible for malicious actors to utilize ML to over-

come automation and scalability challenges of side-channel attacks. By using ML, we showed how to efficiently recover information from multiple victims in much shorter time. Considering the wide adoption of AI across many disciplines, it would be naive to think that malicious actors do not already use ML. To this end, we argued that we can be proactive and proposed the DeepCloak framework. As the side-channel literature shows, there is a clear need for users to cloak their execution fingerprints from the underlying shared system. With the DeepCloak framework, we took the first step in this direction. Specifically, by making novel, defensive use of adversarial crafting we introduced a new cloaking defense against the side-channel leakage. We demonstrated the threat that side-channel leakage poses by using leakage profiles of different processes to train highly accurate ML and DL classifiers. We further investigated the effects of different parameters on the learning rate and testing accuracy. While side-channel leakage is a strong threat to shared hardware systems, we showed that the leakage can be efficiently cloaked using carefully crafted adversarial perturbations. Moreover, we investigated classifier hardening methods that can potentially help an attack to bypass our defense. We showed that even if DD or adversarial re-training is used, we can still cloak the side-channel leakage. We showed that AL perturbations can be executed as a sister-process, running side-by-side with the original and force misclassification to the attacker's model without significant overhead. The adversarial crafting-based cloaking mechanism that we have outlined in the DeepCloak can enable cloud users to have higher levels of security on demand for sensitive operations. Moreover, the efficiency of this defense can be improved for shared hardware systems like the cloud. Finally, to the best of our knowledge, this work is the first use of adversarial crafting for defensive purposes against side-channel attacks. We envision the same approach to be useful in other application scenarios.

Bibliography

[1] Gorka Irazoqui, Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer International Publishing, 2014.

[2] Gorka Irazoqui, Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Fine Grain Cross-VM Attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing, BDCloud 2014, Sydney, Australia, December 3-5, 2014*, pages 737–744.

[3] Gorka Irazoqui, Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies*, 1(1):25–40.

[4] Berk Gülmezoğlu, Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on AES. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[5] Mehmet Sinan İnci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[6] Thomas Eisenbarth Berk Sunar Mehmet Sinan İnci, Berk Gülmezoğlu. Co-location detection on the cloud. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, volume 7. Springer, 2016.

[7] Gorka Irazoqui, Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96. ACM, 2015.

[8] Berk Gülmezoğlu, Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross-VM Cache Attacks on AES. *IEEE Transactions on Multi-Scale Computing Systems*, 2(3):211–222, 2016.

[9] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813, page 368. Springer, 2016.

[10] Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Efficient, adversarial neighbor discovery using logical channels on Microsoft Azure. In *ACSAC*, pages 436–447, 2016.

[11] Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Hit by the Bus: QoS Degradation Attack on Android. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 716–727, New York, NY, USA, 2017. ACM.

[12] Mehmet Sinan İnci, Thomas Eisenbarth, and Berk Sunar. Deepcloak: Adversarial crafting as a defensive measure to cloak processes. *arXiv preprint arXiv:1808.01352*, 2018.

[13] Cong Chen, Mehmet Sinan İnci, Mostafa Taha, and Thomas Eisenbarth. SpecTre: A Tiny Side-Channel Resistant Speck Core for FPGAs. *CARDIS 2016*, 2016.

[14] M. T. Jones. Anatomy of Linux kernel shared memory. `http://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/l-kernel-shared-memory-pdf.pdf/`, April 2010.

[15] Kernel Samepage Merging. `http://kernelnewbies.org/Linux_2_6_32\#head-d3f32e41df508090810388a57efce73f52660ccb/`.

[16] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux symposium*, pages 19–28, 2009.

[17] Analyzing shared memory opportunities in different workloads. `http://os.itec.kit.edu/downloads/sa_2011_groeninger-thorsten_shared-memory-opportunities.pdf`.

[18] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Effects of memory randomization, sanitization and page cache on memory deduplication. In *European Workshop on System Security (EuroSec 2012)*, 2012.

[19] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security*, page 1. ACM, 2011.

[20] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Software side channel attack on memory deduplication. In *ACM Symposium on Operating Systems Principles (SOSP 2011), Poster session*, 2011.

[21] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[22] Smartphone OS Market Share, 2015 Q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[23] Threat Report: H1 2014. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf.

[24] Threat Report: H2 2014. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H2_2014.

[25] Android Security Documentation. https://source.android.com/devices/tech/security/index.html.

[26] Android system permissions. https://developer.android.com/guide/topics/security/permissions.html.

[27] David A. Rusling. ARMv7a Architecture Overview. presentation, 2010. https://wiki.ubuntu.com/Specs/M/ARMGeneralArchitectureOverview?action=AttachFile&do=get&target=ARMv7+Overview+a02.pdf.

[28] ARM Synchronization Primitives. http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf.

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

[30] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.

[31] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[32] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011.

[33] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[34] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS 2012*, pages 305–316, 2012.

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

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 235–252. USENIX Association, 2016.

[37] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[38] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[39] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. *IEEE Symposium on Security and Privacy*, 0:490–505, 2011.

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, pages 11–18, New York, NY, USA, 2007. ACM.

[42] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[43] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Euromicro DSD*, 2015.

[45] Clementine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aureien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters . In *RAID 2015*, 2015.

[46] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel Last-Level Cache. Cryptology ePrint Archive, Report 2015/905, 2015. `http://eprint.iacr.org/`.

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

[48] Google Brain Chief: AI tops humans in computer vision, and healthcare will never be the same. `https://siliconangle.com/blog/2017/09/27/google-brain-chief-jeff-dean-ai-beats-humans-computer-vision\-healthcare-will-never/`, Sep 2017.

[49] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256*, 2016.

[50] Yannis M Assael, Brendan Shillingford, Shimon Whiteson, and Nando de Freitas. Lipnet: Sentence-level lipreading. *arXiv preprint arXiv:1611.01599*, 2016.

[51] Kun-Hsing Yu, Ce Zhang, Gerald J Berry, Russ B Altman, Christopher Ré, Daniel L Rubin, and Michael Snyder. Predicting non-small cell lung cancer prognosis by fully automated microscopic pathology image features. *Nature communications*, 7, 2016.

[52] Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, 316(22), 2016.

[53] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[54] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Dawn Song, Tadayoshi Kohno, Amir Rahmati, Atul Prakash, and Florian Tramer. Note on attacking object detectors with adversarial stickers. *arXiv preprint arXiv:1712.08062*, 2017.

[55] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.

[56] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[57] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv:1710.08864*, 2017.

[58] Daniel Lowd and Christopher Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[59] Pavel Laskov and Richard Lippmann. Machine learning in adversarial environments, 2010.

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

[61] Yan Zhou, Murat Kantarcioglu, Bhavani Thuraisingham, and Bowei Xi. Adversarial support vector machine learning. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1059–1067. ACM, 2012.

[62] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402, 2013.

[63] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4):984–996, 2014.

[64] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

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

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

[67] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*, 2016.

[68] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

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

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

[71] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

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

[73] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[74] Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, SP '92, pages 52–, Washington, DC, USA, 1992. IEEE Computer Society.

[75] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side Channel Cryptanalysis of Product Ciphers. *J. Comput. Secur.*, 8(2,3):141–158, August 2000.

[76] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002(169), 2002.

[77] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, and Maki Shigeri. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS*, pages 62–76. Springer-Verlag, 2003.

[78] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[79] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.

[80] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems—CHES 2006*, volume 4249 of *Springer LNCS*, pages 201–215. Springer, 2006.

[81] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[82] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `https://eprint.iacr.org/2014/140.pdf`.

[83] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 990–1003, New York, NY, USA, 2014. ACM.

[84] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal I. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Verlag, 1996.

[85] David Brumley and Dan Boneh. Remote timing attacks are practical. volume 48, pages 701–716. Elsevier, 2005.

[86] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology CRYPTO 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397.

[87] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261.

[88] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *CHES*, Lecture Notes in Computer Science, pages 207–228. Springer, 2015.

[89] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pages 444–461.

[90] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. In *CHES 2015*.

[91] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In *CHES*, pages 75–92, 2014. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[92] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '16. ACM, 2016.

[93] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. On detecting co-resident cloud instances using network flow watermarking techniques. *Int. J. Inf. Secur.*, 13(2):171–189, 2014.

[94] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *CCS*, pages 990–1003, 2014.

[95] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, Washington, D.C. USENIX Association.

[96] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[97] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security*, pages 929–944, 2015.

[98] Maurice, C. and Weber, M. and Schwarz, M. and Giner, L. and Gruss, D. and Carlo, A. B. and Mangard, S. and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS 2017*.

[99] Onur AcıİÇmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. Cryptology ePrint Archive, Report 2007/039, 2007. `http://eprint.iacr.org/2006/351.pdf`.

[100] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. pages 312–320, 2007.

[101] Onur Acıiçmez, Çetin K. Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology CT-RSA 2007*, volume 4377, pages 225–242.

[102] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*, pages 355–371. Springer, 2011.

[103] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks. Cryptology ePrint Archive, Report 2017/564, 2017. `https://eprint.iacr.org/2017/564`.

[104] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[105] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Protecting legacy processes against cache attacks. *arXiv preprint arXiv:1709.01795*, 2017.

[106] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 224–235, New York, NY, USA, 2018. ACM.

[107] Zirak Allaf, Mo Adda, and Alexander Gegov. A comparison study on flush+reload and prime and probe attacks on aes using machine learning approaches. In Fei Chao, Steven Schockaert, and Qingfu Zhang, editors, *Advances in Computational Intelligence Systems*, pages 203–213, Cham, 2018. Springer International Publishing.

[108] Berk Gülmezoğlu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, pages 80–97, 2017.

[109] Raphael Spreitzer and Thomas Plos. On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In *Network and System Security - NSS 2013, 7th International Conference, Madrid, Spain, June 3-4, 2013, Proceedings*, volume 7873 of *Lecture Notes in COMPUTER Science*, pages 656 – 662.

[110] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke. Differential cache-collision timing attacks on AES with applications to embedded CPUs. In *CT-RSA 2010*, pages 235–251. Springer, 2010.

[111] Billy Bob Brumley. Cache storage attacks. In *Cryptographers' Track at the RSA Conference*, pages 22–34. Springer, 2015.

[112] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS 2016*.

[113] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *CCS 2013*.

[114] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *2012 IEEE Symposium on Security and Privacy*, pages 143–157. IEEE, 2012.

[115] Chen, Q. A. and Qian, Z. and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security 2014*.

[116] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *IEEE S&P 2015*.

[117] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. ASPLOS 2015.

[118] The dropbox blog. `https://blog.dropbox.com/2013/07/dbx/`.

[119] Amazon AWS: 3.8 billion revenue in 2013. `https://readwrite.com/2013/01/14/amazon-web-services-can-it-win-the-enterprise`.

[120] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540, May 2013.

[121] Heartbleed bug. `http://heartbleed.com/`.

[122] CBC news. Heartbleed bug: 900 SINs stolen from Revenue Canada. http://www.cbc.ca/news/business/heartbleed-bug-rcmp-asked-revenue-canada-to-delay-news-of-sin-thefts-1.2609192l, April 2014.

[123] BBC news. Heartbleed hacks hit Mumsnet and Canada's tax agency. `http://www.bbc.com/news/technology-27028101`, April 2014.

[124] The Guardian. More than 300k systems 'still vulnerable' to Heartbleed attacks. `http://www.theguardian.com/technology/2014/jun/23/heartbleed-attacks-vulnerable-openssl`, July 2014.

[125] CyaSSL: Embedded SSL library wolfSSL. `http://www.wolfssl.com/yaSSL/Home.html`, May 2014.

[126] MatrixSSL: Open source embedded SSL. `http://www.matrixssl.org/`, May 2014.

[127] Nikos Mavrogiannopoulos and Simon Josefsson. GnuTLS: The GnuTLS transport layer security library. May 2014.

[128] PolarSSL. PolarSSL: Straightforward,secure communication. www.polarssl.org.

[129] Dan Goodin. Hackers break SSL encryption used by millions of sites. http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/, 2011.

[130] Thai Duong and Juliano Rizzo. Here Come The XOR Ninjas. 2011.

[131] OpenSSL vulnerabilities. https://www.openssl.org/news/vulnerabilities.html.

[132] Vlastimil Klíma, Ondrej Pokornỳ, and Tomáš Rosa. Attacking rsa-based sessions in ssl/tls. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–440. Springer, 2003.

[133] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In *Proceedings of In Advances in Cryptology - EUROCRYPT'02*, pages 534–546. Springer-Verlag, 2002.

[134] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS1. pages 1–12. Springer-Verlag, 1998.

[135] Kernel based virtual machine. http://www.linux-kvm.org/page/Main_Page, April 2014.

[136] GnuTLS server examples. http://www.gnutls.org/manual/html_node/Server-examples.html, April 2014.

[137] GnuTLS client examples. http://www.gnutls.org/manual/html_node/Client-examples.html, April 2014.

[138] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Jackpot stealing information from large caches via huge pages. Cryptology ePrint Archive, Report 2014/970, 2014. http://eprint.iacr.org/.

[139] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity., 2015.

[140] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46. ACM, 2011.

[141] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM.

[142] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 191–205.

[143] Sharon Gaudin. Public cloud market ready for 'hypergrowth' period. Computerworld Article, April 2014. `http://www.computerworld.com/article/2488572/cloud-computing/public-cloud-market-ready-for--hypergrowth--period.html`.

[144] AWS IP Address Ranges. `https://ip-ranges.amazonaws.com/ip-ranges.json`.

[145] Amazon EC2 Instances. `http://aws.amazon.com/ec2/instance-types/`, 2016.

[146] Google Compute Engine Instance Types. `https://cloud.google.com/compute/docs/machine-types`, 2016.

[147] Microsoft Azure Sizes for virtual machines. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-size-specs/`.

[148] The OpenMP® API specification for parallel programming.

[149] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1406–1418, New York, NY, USA, 2015. ACM.

[150] Kuniyasu Suzaki, Kengo Iijima, YAGI Toshiki, and Cyrille Artho. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 96(1):215–224, 2013.

[151] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 490–505, Washington, DC, USA, 2011. IEEE Computer Society.

[152] Fix Flush and Reload in RSA. `https://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000329.html`.

[153] Transparent Page Sharing: additional management capabilities and new default settings. `http://blogs.vmware.com/security/vmware-security-response-center/page/2`.

[154] OpenSSL fix flush and reload ECDSA nonces. https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=2198be3483259de374f91e57d247d0fc667aef29.

[155] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, 2012. USENIX.

[156] Daniel J Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko Van Someren. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology-ASIACRYPT 2013*, pages 341–360. Springer, 2013.

[157] Intel Xeon 2670-v2. http://ark.intel.com/es/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz.

[158] Libgcrypt. The libgcrypt reference manual. http://www.gnupg.org/documentation/manuals/gcrypt/.

[159] Number of available applications in the Google Play Store from December 2009 to September 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[160] Android, the world's most popular mobile platform. Online, Nov 2016. https://developer.android.com/about/android.html.

[161] Application Fundamentals. Online, Nov 8 2016. https://developer.android.com/guide/components/fundamentals.html.

[162] Android Dashboards. Online, Nov 2016. https://developer.android.com/about/dashboards/index.html.

[163] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, George E Dahl, Timo Kohlberger, Aleksey Boyko, Subhashini Venugopalan, Aleksei Timofeev, Philip Q Nelson, Greg S Corrado, et al. Detecting cancer metastases on gigapixel pathology images. *arXiv preprint arXiv:1703.02442*, 2017.

[164] AI beats professional players at Super Smash Bros. video game. https://www.newscientist.com/article/2122452-ai-beats-professional-players-at-super-smash-bros-video-game/. Accessed: 2017-10-27.

[165] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games. *arXiv preprint arXiv:1703.10069*, 2017.

[166] Elle O'Brien. Romance novels, generated by artificial intelligence. https://medium.com/towards-data-science/romance-novels-generated-by-artificial-intelligence-1b31d9c872b2, Aug 2017.

[167] Jon Brodkin. 2 million people-and some dead ones-were impersonated in net neutrality comments. https://arstechnica.com/tech-policy/2017/12/dead-people-among-millions-impersonated-in-fake-net-neutrality-comments/.

[168] Susan Decker. FCC Rules Out Delaying Net Neutrality Repeal Over Fake Comments. https://www.bloomberg.com/news/articles/2018-01-05/fcc-rules-out-delaying-net-neutrality-repeal-over-fake-comments.

[169] Tom Simonite. This AI Will Craft Tweets That Youll Never Know Are Spam. https://www.technologyreview.com/s/602109/this-ai-will-craft-tweets-that-youll-never-know-are-spam/.

[170] Zach Emmanuel. Security experts air concerns over hackers using AI and machine learning for phishing attacks. https://www.computerweekly.com/news/450427653/Security-experts-air-concerns-over-hackers-using-AI-and-machine-learning-for-phishing-atttacks/.

[171] The Cylance Team. Black Hat Attendees See AI as Double-Edged Sword. https://threatmatrix.cylance.com/en_us/home/black-hat-attendees-see-ai-as-double-edged-sword.html.

[172] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293, 2011.

[173] Annelie Heuser and Michael Zohner. Intelligent machine homicide. *COSADE*, 7275:249–264, 2012.

[174] Zdenek Martinasek and Vaclav Zeman. Innovative method of the power analysis. *Radioengineering*, 22(2):586–594, 2013.

[175] Zdenek Martinasek, Jan Hajny, and Lukas Malina. Optimization of power analysis using neural network. In *International Conference on Smart Card Research and Advanced Applications*, pages 94–107. Springer, 2013.

[176] Tony Beltramelli and Sebastian Risi. Deep-spying: Spying using smartwatch and deep learning. *arXiv preprint arXiv:1512.05616*, 2015.

[177] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

[178] Ian Goodfellow and Nicolas Papernot. Is attacking machine learning easier than defending it? http://www.cleverhans.io/security/privacy/ml/2017/02/15/why-attacking-machine-learning-is-easier-than-defending-it.html.

[179] Marco Chiappetta. Quickhpc. https://github.com/chpmrc/quickhpc, 2015.

[180] Pedro Tabacof and Eduardo Valle. Exploring the space of adversarial images. *CoRR*, abs/1510.05328, 2015.

[181] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. *CoRR*, abs/1511.07528, 2015.

[182] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.

[183] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox v0.8.0: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[184] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian Molloy, and Ben Edwards. Adversarial robustness toolbox v0.4.0. *CoRR*, 1807.01069.

[185] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv preprint arXiv:1607.04311*, 2016.