Cross-core Microarchitectural Side Channel Attacks and Countermeasures

by

Gorka Irazoqui

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 2017

APPROVED:

Professor Thomas Eisenbarth Dissertation Advisor ECE Department	Professor Berk Sunar Dissertation Committee ECE Department
Professor Craig Shue Dissertation Committee CS Department	Professor Engın Kirda Dissertation Committee Northeastern University

Abstract

In the last decade, multi-threaded systems and resource sharing have brought a number of technologies that facilitate our daily tasks in a way we never imagined. Among others, cloud computing has emerged to offer us powerful computational resources without having to physically acquire and install them, while smartphones have almost acquired the same importance desktop computers had a decade ago. This has only been possible thanks to the ever evolving performance optimization improvements made to modern microarchitectures that efficiently manage concurrent usage of hardware resources. One of the aforementioned optimizations is the usage of shared Last Level Caches (LLCs) to balance different CPU core loads and to maintain coherency between shared memory blocks utilized by different cores. The latter for instance has enabled concurrent execution of several processes in low RAM devices such as smartphones.

Although efficient hardware resource sharing has become the de-facto model for several modern technologies, it also poses a major concern with respect to security. Some of the concurrently executed co-resident processes might in fact be malicious and try to take advantage of hardware proximity. New technologies usually claim to be secure by implementing sandboxing techniques and executing processes in isolated software environments, called Virtual Machines (VMs). However, the design of these isolated environments aims at preventing pure software-based attacks and usually does not consider hardware leakages. In fact, the malicious utilization of hardware resources as covert channels might have severe consequences to the privacy of the customers.

Our work demonstrates that malicious customers of such technologies can utilize the LLC as the covert channel to obtain sensitive information from a co-resident victim. We show that the LLC is an attractive resource to be targeted by attackers, as it offers high resolution and, unlike previous microarchitectural attacks, does not require core-colocation. Particularly concerning are the cases in which cryptography is compromised, as it is the main component of every security solution. In this sense, the presented work does not only introduce three attack variants that can be applicable in different scenarios, but also demonstrates the ability to recover cryptographic keys (e.g. AES and RSA) and TLS session messages across VMs, bypassing sandboxing techniques.

Finally, two countermeasures to prevent microarchitectural attacks in general and LLC attacks in particular from retrieving fine-grain information are presented. Unlike previously proposed countermeasures, ours do not add permanent overheads in the system but can be utilized as preemptive defenses. The first identifies leakages in cryptographic software that can potentially lead to key extraction, and thus, can be utilized by cryptographic code designers to ensure the sanity of their libraries before deployment. The second detects microarchitectural attacks embedded into

innocent-looking binaries, preventing them from being posted in official application repositories that usually have the full trust of the customer.

Acknowledgements

The attacks described in Chapter 4 are based on collaborative work with Mehmet Sinan İnci and resulted in peer-reviewed publications [IIES14b, IIES15].

The content presented in Chapter 5 has been published as [IES16].

Parts of the results introduced in Chapter 6 are based joint work with Mehmet Sinan İnci and Berk Gulmezoglu and resulted in publication [İGI+16a]. The rest of the content of Chapter 6 has been published in [IES15a, IES15b]

The vulnerability analysis described in Chapter 7 was performed jointly with Intel employes Xiaofei Guo, Hareesh Khattri, Arun Kanuparthi and Kai Cong, and it is under submission for a peer-review venue. The remaining contributions in the chapter have also been submitted to a peer-review conference.

This work was supported by the National Science Foundation (NSF), under grants CNS-1318919, CNS-1314770 and CNS-1618837.

This thesis is the result of several years of effort that would not have been successful without the collaboration, both in a professional and personal way, of certain people to whom I would like to express my gratitude.

My two advisors Thomas Eisenbarth and Berk Sunar have given me confidence and freedom to achieve the results presented in this thesis. Without their knowledge, advise and personal treatment this journey would have been much more complicated. They were able to take the best out of my skills, and sometimes even believed in me more than I did. I will always feel that part of the achievements that I will accomplish during my career will belong to them.

As part of this thesis I had the chance of working closely to Yuval Yarom and Xiaofei Guo. Besides the pleasure that it was working with them, with both of them I established a relationship that goes beyond our professional collaborations. I wish them the best of the lucks in both their professional careers and their personal lives.

I would like to thank the members of the Vernam lab, for being such a great support in the most difficult moments. Every single person in the lab was helpful at some specific point during the last 4 years. I am looking forward to maintain the good relationships we built in the upcoming years.

I also would like to express my gratitude to the One United and E soccer team members, who gave me the distraction that every person needs to accomplish professional success. Besides the three tournaments that we won together, they provided me an invaluable personal support which produced friendship relationships that I will never forget. They can feel proud of being one of the most positive things I take from this experience.

My parents and my sister have always been the first ones believing in me, and have a great influence in any success I achieve in my personal career. Although they already know what they mean to me, I would still like to thank them for everything they do to take always the best of me. I cannot imagine a better source of strength for the upcoming challenges.

Finally, I would like to thank Elena Gonzalez for being the best person I could ever find to stay by my side. Despite all the obstacles she never gave up on us. I will always be in debt with her for all the support she gave me to complete this thesis. She has shown me personal values that I hardly find in other people, and that I am willing to enjoy everyday from now on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid increase in transistor densities over the past few decades brought numerous computing applications, previously thought impossible, into the realm of everyday computing. With dense integration and increased clock rates, heat dissipation in single core architectures has become a major challenge for processor manufacturers. The design of multi-core architectures has been the method of choice to profit from further advances in integration, a solution that has shown to substantially improve the performance over single-core architectures. Lately, the design of multi-CPU sockets (each allocating multi-core systems) has taken even more advantage of the benefits of having several Central Processing Units executing different tasks in parallel.

Despite its many advantages, multi-core and multi-CPU architectures are susceptible to suffering under bandwidth bottlenecks if the architecture is not designed properly, especially when more cores are packed into a high-performance system. Parallelism can only be effective if shared resources are correctly managed, ensuring their fair distribution. Several components have been designed and added to modern microarchitecture designs to achieve this goal. A good example are inclusive Last Level Caches (LLCs), which are shared across multiple CPU cores and aid the maintenance of cache coherency protocols within the same CPU socket by ensuring that copies in the upper level caches exist in the LLC.

In fact, it is the aforementioned parallelism that has enabled many of the technologies we use in a daily basis. For instance, in the last decade we have witnessed the cloud revolution, which allocates several customers (and their corresponding workloads) in a single physical machine. The concurrent execution of these number of processes in the same computer would not be possible without the multi-core/CPU designs that are common nowadays. Similarly, smartphones are now able to execute several processes at the same time by running some of them in the background. This has been possible due to the large adoption that embedded devices have done of multi-core architectures; modern smartphones not only have several cores in the same device, but also they started allocating more than one CPU socket.

Although parallelism and resource sharing help to improve the performance,

they also poses a big risk when it comes to execute untrusted processes alongside trusted processes. For example, a malicious attacker can take advantage of being co-resident within a potential victim and execute malicious code in commercial clouds. Similarly, a malicious application can try to steal sensitive information from a benign security-critical application, e.g., an online banking application. To cope with these issues both trusted and untrusted processes are usually executed in a software isolated environment. IaaS clouds for instance rent expensive hardware resource to multiple customers by offering guest OS instances sandboxed inside virtualized machines (VMs). PaaS cloud services go one step further and allow users to share the application space while sandboxed, e.g. using containers, at the OS level. Similar sandboxing techniques are used to isolate semi-trusted apps running on mobile devices. Even browsers use sandboxing to execute untrusted code without the risk of harming the local host. All these mechanisms have a clear goal in mind: taking advantage of resource sharing among several applications/users while isolating each of them to prevent malicious exploitation of the said shared resources.

Despite the widespread adoption of the aforementioned technologies, the robustness of the resource sharing scenarios that they provide have, prior to this work, yet not being tested against malicious users exploiting hardware covert channels. While the resistance of these technologies to pure software attacks is usually guaranteed, their response to hardware leakage based attacks still remains as an open question. In the following we will refer to these attacks, which utilize hardware resources like the cache and the Branch Prediction Unit (BPU) to gain information, as microarchitectural attacks.

By 2014, only [RTSS09, ZJRR12] were able to succeed on recovering information from co-resident users in realistics cloud environments utilizing microarchitectural attacks the first recovers keystrokes across VMs in Amazon EC2, the latter recovers an El Gamal decryption key in a lab virtualization environment. The problem mostly comes from the fact that these and the rest of the works (except for [YF14]) prior to this thesis only consider core-private covert channels like the L1 cache and the BPU. Therefore these attacks are only applicable when victim and attacker are core co-resident. With multi-core systems being the de-facto architectures utilized not only in high-end servers but also in smarpthone devices, the core co-residency requirement significantly reduces the applicability of previously known microarchitectural attacks. Further, constant optimization and penalty reduction make core-private resources difficult to utilize with the amount of noise that regular workloads introduce. For instance, L1 cache misses and L2 cache hits only differ by a few cycles while misspredicted branches do not add substantial overhead. It is questionable whether these covert channels would support the typical amount of noise in sandboxed scenarios.

It is therefore necessary to investigate whether more powerful and applicable covert channels can be utilized for unauthorized information extraction. For instance, considering covert channels that do not require core co-location increments the attack probabilities, as only CPU socket co-residency is needed. This becomes a

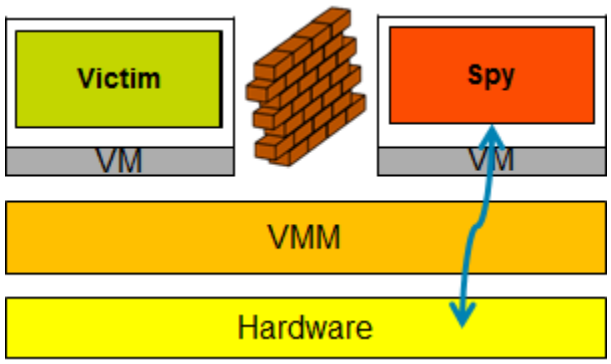


Figure 1.1: Malicious VMs can try to bypass the isolation provided by the hypervisors and steal information from co-resident VMs

crucial fact in, e.g., commercial clouds where core co-residence is hardly unlikely. In fact, several microarchitectural components are connected not only across cores but also across CPU sockets. These include, among others, the Last Level Cache (LLC), the memory bus or the DRAM. All these possible covert channels have to be thoroughly investigated to understand their capabilities and to develop the appropriate countermeasures to prevent their utilization for information theft.

Contributions

This thesis focuses in one of the aforementioned potentially powerful covert channels, i.e., the LLC. We will show its exploitability under different assumptions and scenarios. We present three attacks that can take advantage of such a resource, namely *Flush and Reload*, *Invalidate and Transfer* and *Prime and Probe*. The first is only applicable across cores and under memory deduplication features, the second is capable of reaching victims across CPU sockets under the same memory deduplication assumption, and the latter is able to recover information across cores without any special requirement.

We show how and where these three attacks can be applied, specifically in scenarios where previous attacks had proven to behave poorly. For instance, we show for the first time how to recover information across VMs located in different cores with the *Flush and Reload* attack in VMware. Further, we show that these attacks can be taken to multi CPU socket machines by applying the *Invalidate and Transfer* attack in a 4 CPU socket school server. In addition, this thesis presents the LLC *Prime and Probe* attack, applicable in any hypervisor without any special requirement (even VMware with updated security features). In fact, we utilize it to recover a RSA key from a co-resident VM in Amazon EC2, demonstrating the big threat that cache attacks pose in real world scenarios.

Lastly, we develop two tools that can help at preventing these leakages from being exploited. However, unlike other proposed countermeasures that add perma-

nent performance overheads in the system that private companies do not want to afford, our solutions take a preemptive approach. The first, aims at identifying LLC leakages in cryptographic code to ensure they are caught before the software reaches the end users. In particular, we use taint analysis, cache trace analysis and Mutual Information (MI) to derive whether a cryptographic code leaks or not. We found alarming results, as 50% of the implementations leaked information for AES, RSA and ECC algorithms. The second tool, MASCAT, performs a different approach. With official application stores like Google Play or Microsoft Store raising their popularity, it is important that the binaries offered by these repositories are malware safe. Unlike regular malware, e.g. shell code, which might be easily detectable by modern antivirus tools, microarchitectural attacks are different as they do not look malicious. MASCAT serves as a microarchitectural attack antivirus, detecting them inside binaries without having to inspect the source code manually.

Our work resulted in the discovery of several vulnerabilities in existing products that attackers can take advantage from to steal information belonging to co-resident victims. As part of our responsibility as researchers, we notified the corresponding software designers about the vulnerabilities of their solutions. These conversations lead to several security updates; VMware decided to implement a new salt-based deduplication mechanism (CVE-2014-3566), Intel modified the RSA implementation of its cryptographic library (CVE-2016-8100), the Bouncy Castle cryptographic library re-designed the AES implementation (2016-10003323) and WolfSSL closed leakages in all AES, RSA and ECC implementations (CVEs 2016-7438, 2016-7439 and 2016-7440). Thus, our investigation played a key role on improving and updating several up-to-date security solutions that, otherwise, could have compromised the privacy of their customers.

In summary, this work:

- Demonstrates the applicability of the *Flush and Reload* attack in virtualized environments by recovering AES keys across cores in less than a minute
- Shows that cache attacks can go beyond cryptographic algorithms by attacking three TLS implementations and re-implementing the closed Lucky 13 attack.
- Presents *Invalidate and Transfer*, a new attack targeting the cache coherency protocol that works across CPU sockets. We demonstrate its viability by recovering AES and RSA keys.
- Introduces the LLC *Prime and Probe* attack that does not require memory deduplication to succeed. In order to successfully apply the *Prime and Probe* attack, a thorough investigation on the architecture of LLCs is performed, in particular on how cache slices are distributed. We also show how the *Prime and Probe* attack succeeds in hypervisors where *Flush and Reload* was not able to succeed, e.g., the Xen hypervisor.

- Shows, for the first time, the applicability of microarchitectural attacks in commercial clouds. More precisely, we demonstrate how to recover a RSA key across co-resident VMs in Amazon EC2.

- Presents a cache leakage analysis tool to prevent the design of cryptographic algorithms form leaking information. We found alarming results, as 50% of the implementations showed cache leakages that could lead to full key extraction.

- Introduces MASCAT, a tool to detect microarchitectural attacks embedded in apparently innocent looking binaries. MASCAT serves as a verification process for official application distributors that want to ensure the sanity of the binaries being offered in their repositories.

- proposes fixes (that have been adopted) to all the vulnerabilities discovered in commercial software due to the previously mentioned contributions.

The rest of the thesis is distributed as follows. Chapter 2 describes the necessary background to understand the attacks and defenses later developed. Related work, both prior and concurrent to this thesis is presented in Chapter 3. Chapters 4, 5 and 6 describe the deployment of the *Flush and Reload*, *Invalidate and Transfer* and *Prime and Probe* respectively. Finally Chapter 7 describes the aforementioned countermeasures.

Chapter 2

Background

This thesis includes a large description on how cache attacks can be applied across cores to recover sensitive information belonging to a co-resident victim. In order to aid the reader understand the attacks that will later be presented, this chapter gives an introduction on the typical microarchitecture layout found in modern processors and the attacks that were proposed prior to this thesis. Further, we give a description of the most widely used cryptographic algorithms, as they will be under the attack radar of our microarchitectural attacks in subsequent chapters.

2.1 Side Channel Attacks

Side channel Attacks (SCA) are attacks that take advantage of the leakage coming from a covert channel during a secure communication. Typical attacks on a direct channel involve brute forcing the key or social phishing. Instead, side channel attacks observe additional information stemming from side channels that carry information about the key. These side channel traces are then processed and correlated to either obtain the full key or to reduce its search space significantly. Figure 2.1 shows the overall idea, where two parties are trying to establish an encrypted communication over an insecure channel. Due to the leakage stemming from unintended side channels, an attacker can at least try to use that information to obtain the key.

This leakage can come in many forms. Power and Electro Magnetic (EM) leakages are common in embedded devices and smartcards and usually imply having physical access to the device. Timing attacks can deduce information about a secret distinguishing the overall process time, but they suffer from the huge amount of noise in modern communication channels (e.g., the internet). microarchitectural attacks, in general, try to obtain information from the usage of some microarchitectural resource. Although they require physical co-residence with the targeted process, they do not need *physical access* to the device. In fact, several scenarios can arise in which malicious code is executed alongside a potential victim in the same physical machine. Other leakage forms are less common, e.g. sound or ther-

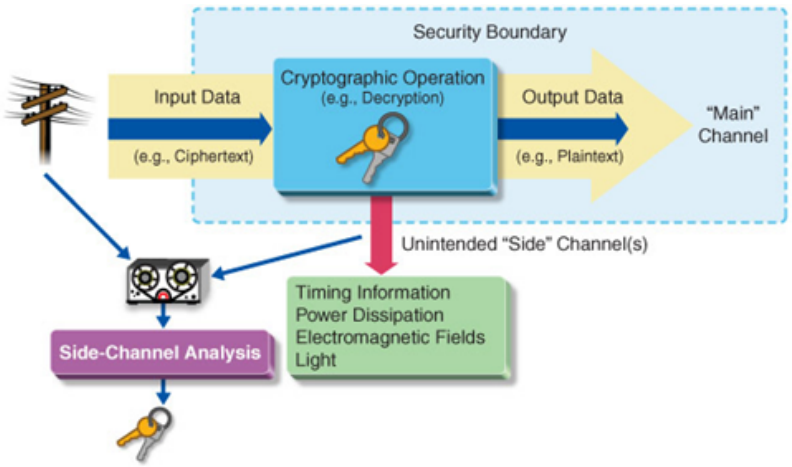


Figure 2.1: Side channel attack scenario. Instead of the direct channel, leakage coming from side channels are exploited to obtain information about the key. Figure from [Mic].

mal, but are constantly being studied to increase the applicability of side channel attacks [KJJ99, BECN+04, KJJR11].

2.2 Computer Microarchitecture

Modern computers execute user specified instructions and data in a Central Processing Unit (CPU), store the necessary memory to execute software in DRAM and interact with the outside world trough peripherals. With a constantly evolving market and strong competitors fighting for the same marketplace, efficiency has become one of the main goals of every microarchitecture designer. In fact, several hardware resources have been added to the basic microarchitectural components, e.g., caches or Branch Prediction Units (BPU), to provide a better performance. Further, modern microarchitectures now embed several processing units in the same processor, some of which are even capable of processing to threads concurrently. Even further, lately we have observed the rise of multi-CPU socket computers, in which more than one CPU sockets are embedded into the same piece of hardware. All these technological advances have the same goal: offer the end-user the best computing performance.

A typical microarchitecture design commonly found in modern processors can be observed in Figure 2.2. The example shows a dual socket CPU, each CPU having two cores. Each CPU core has private L1 and L2 caches, while they share the Last Level Cache (LLC). Further each core has its own BPU, in charge of predicting the outcome of the branches being executed. The communication between the cache hierarchy and the memory is done through the memory bus, which is also in charge of maintaining coherency between shared blocks across CPU sockets. Finally, the

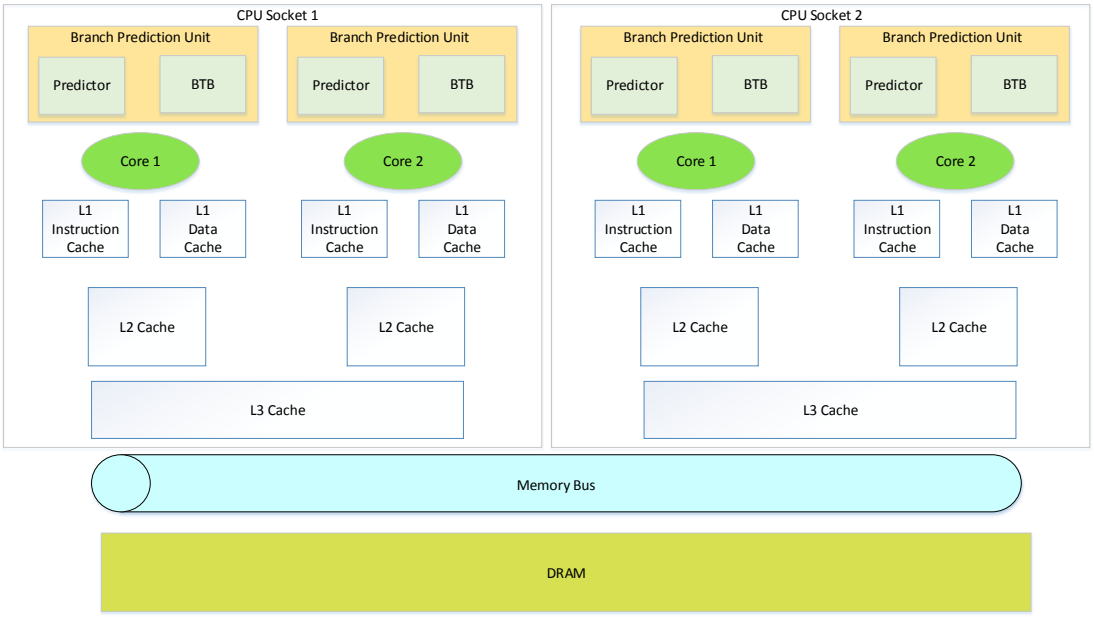


Figure 2.2: Typical microarchitecture layout in modern processors

DRAM stores the necessary instructions and data that the program being executed needs.

Although each component requires a exhaustive and thorough analysis, we put our focus in caches, as they are the core component being examined in this thesis. Nevertheless, some of the functionalities of the aforementioned components will indeed be described later in greater detail in the thesis when needed.

2.2.1 Hardware Caches

Hardware caches are small memories placed between the DRAM and the CPU cores to store data and instructions that are likely to be reused soon. When the software needs a particular memory block, the CPU first checks the cache hierarchy looking for that memory block. If found, the memory block is fetched from the cache and the access time is significantly faster. If the memory block was not found in the cache hierarchy, then it is fetched from the DRAM at the cost of a slower access time. These two scenarios are often called a *cache hit* and a *cache miss* respectively.

At this point the main question is probably how much faster cache accesses are over DRAM accesses. This can be observed in Figure 2.3, for which we performed consecutive timed accesses to a L1 and L3 cached memory block and an uncached memory block in an Intel i5-3320M. The access time for the L1 cache is about 3 cycles, for the L3 cache is around 7 cycles, while an access to the memory takes around 25 cycles. Thus, an access to the DRAM is about 3 times slower than an access to the lowest cache level, which gives an idea of the performance improvement

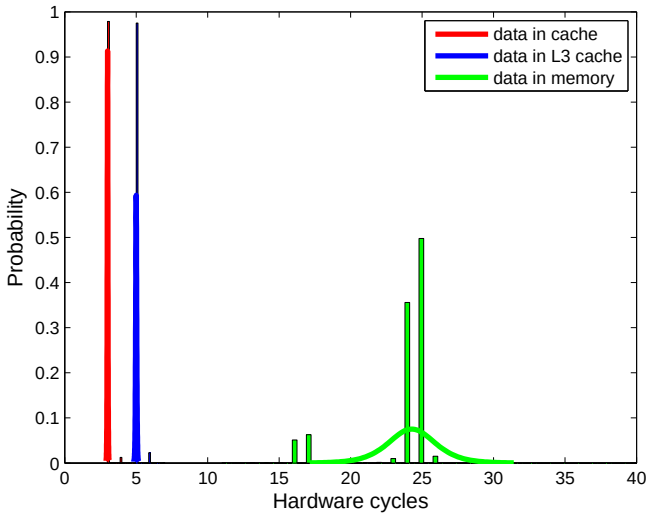


Figure 2.3: Reload timing distribution when a memory block is loaded from the: L1 cache, L3 cache and memory. The experiments were performed in an Intel i5-3320M.

that the cache hierarchy offers to software execution.

In particular, caches base their functionality in two main principles: *spatial locality* and *temporal locality*. While the first principle states that data residing close to the data being accessed is likely to be accessed soon, the latter assumes that recently accessed data is also likely to be accessed soon. Caches accomplish *temporal locality* by storing recently accessed memory blocks, and *spatial locality* by working with cache lines that load both the needed and neighbor data to the cache. In consequence, the cache is usually divided into several fixed size cache lines.

However, caches can have very different design characteristics. In the following we describe how their functionality changes for different design choices.

2.2.1.1 Cache Addressing

One of the most important decisions in the design of a cache is how is this distributed and addressed. In this sense there are three main design policies:

- **Direct mapped caches:** In this case, each memory block has a fixed location in the cache, i.e., they can occupy only one specific cache line.
- **Fully associative cache:** In this case a memory block can occupy *any* of the cache lines in the cache.
- **N-way set associative cache:** This is the most common design in modern processors. This design splits the cache into equally sized partitions called

cache sets, each allocating n cache lines. In this case, a memory block is constrained to occupy one of the n cache lines of a fixed set.

Each of the designs has advantages and disadvantages. For instance, the memory block search in a direct mapped cache is very efficient as the CPU has to check only one location. In contrast, in a fully associative cache the CPU will have to search for a memory block in *all* the cache lines. However, direct mapped caches suffer from collisions, as they always trigger cache misses for consecutive accesses to cache line colliding memory blocks. Fully associative caches do not suffer from this problem, as any block can occupy any location in the cache. Thus, it is understandable that the most common choice in modern processors is the n-way set associative cache, as it balances the advantages and disadvantages of the first two designs.

2.2.1.2 Cache Replacement policy

Another fundamental aspect in the design of n-way set associative caches is the algorithm that selects the block being evicted within a set. Recall that each set has n possible memory blocks that can be evicted to make room for a new one. The most common algorithms designed for this purpose are:

- **Least Recently Used:** This algorithm evicts the memory block that has been longer time without being accessed by taking count of the number of accesses being made to each n ways in the set.
- **Round Robin:** Also known as First-In First-Out (FIFO), which evicts the memory block that has reside for longer in the cache.
- **Pseudo-random:** The selection of the memory block to be evicted is done with a pseudo-random algorithm.

Note that, in the case of cache attacks, the knowledge of the algorithm might be crucial for the success of the attack. In fact, some cache attacks like the *Prime and Probe* can be challenging to implement with random replacement policies, as the occupancy of the memory blocks in the cache cannot be predicted. In this thesis, we focus on x86_64 architectures which implement a LRU eviction policy.

2.2.1.3 Inclusiveness Property

The last but perhaps most important property that cache designers need to take into account is whether they feature inclusive, non-inclusive or exclusive caches. This has severe implications, among others, in the cache coherency protocol:

- **Inclusive caches**: Inclusive caches are those that require that any block that exists in the upper level caches (i.e., L1 or L2) have to also exists in the LLC. Note that this has several simplifications when it comes to maintain the

cache coherency across cores, since the LLC is shared. Thus, the inclusiveness property itself is in charge of keeping a coherency between shared memory blocks across CPU cores. The drawback of inclusive caches is the additional cache lines wasted with several copies of the same memory block.

- **Exclusive caches**: Exclusive caches require that a memory block only resides at one cache level at a time. In contrast to inclusive caches, here the cache coherency protocol has to be implemented with upper level caches as well. However, they do not waste cache space to maintain several copies of the same block.

- **Non-inclusive caches**: This type of caches do not have any requirement, but a memory block can reside in one or more cache levels at a time.

Whether the cache features the inclusiveness property might also be a key factor when implementing cache attacks. In fact, as it will be explained in later sections, attacks like *Prime and Probe* might only be applicable in inclusive caches, while other attacks like *Invalidate and Transfer* are agnostic of the inclusiveness property.

2.3 The Cache as a Covert Channel

This thesis presents, among others, the utilization of the LLC as a new covert channel to obtain information from a co-resident user. It is thus important to see how the cache can be utilized as a covert channel to recover information, and in particular, how previous works have done it. The first thing we should clarify the reader is that the knowledge of which set a victim has utilized can lead to key extraction. For instance, key dependent data that always utilizes the same set can reveal the value of the key being processed if such a utilization of the set is observed. Based on this, research previous to this thesis that has demonstrated in the past the application of spy processes in core-private resources like the L1 cache to obtain fine-grain information have being based in two main attacks:

2.3.1 The Evict and Time attack

The `Evict and Time` attack was presented by [OST06] as a methodology to recover a complete AES key. In particular, the work demonstrated that key dependent T-table accesses in the AES implementation (see Section 2.4) can lead to knowledge of the key. The methodology utilized can be described as:

1. The attacker performs a full AES encryption with a known fixed plaintext and an unknown fixed key. After this step, all the data utilized by the AES encryption resides in the cache.

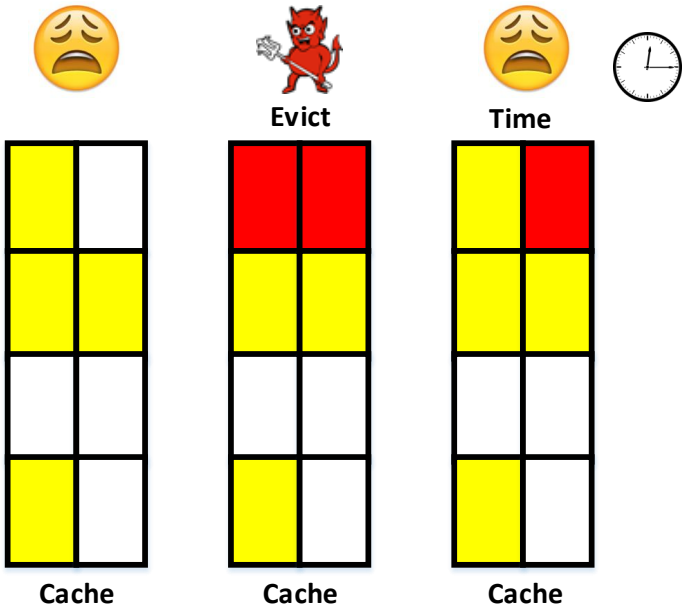


Figure 2.4: Evict and Time procedure

2. The attacker evicts a specific set in the cache. Note that, if this specific set was used by the AES process the data utilized by it will no longer reside in the cache.

3. The attacker performs the same encryption with the same plaintext and key, and measures the time to complete the encryption. The time spent to perform the encryption will highly depend on whether the attacker evicted a set utilized by AES in step 1. If he did, then the encryption will take longer as the data has to be fetched from the memory. If he did not, the attacker guesses that the AES encryption does not use the set he evicted, possibly discarding some key candidates.

Figure 2.4 graphically represents the states described. The victim first utilized the yellow memory blocks, while the attacker evicts one of them in the eviction process. When she times the victims process again, she sees the victim needed a block from the memory and infers the victim uses the set she evicted. Note that in this case the attacker has to record two encryptions with the same plaintext and key, which might not be entirely realistic. Further, as the leakage comes from the ability to measure the overall encryption time, the attack is not considered overly practical.

2.3.2 The *Prime and Probe* Attack

The *Prime and Probe* attack was again proposed in [OST06], but later was utilized by [Acı07, RTSS09, ZJRR12]. In a sense the attack is similar to `Evict and Time`, but only the monitorization of own memory blocks is required. This is a clear advantage as it brings a more realistics scenario. These are the main steps:

1. The attacker fills the L1 cache with her own junk data.
2. The attacker waits until the victim executes his vulnerable code snippet. The key dependent branch/memory access will utilize some of the sets filled by the attacker in the L1.
3. The attacker accesses back his own memory blocks, measuring the time it takes to access them. If the attacker observes high access times for some of the memory blocks it means that the targeted software utilized the set where those memory blocks resided. In contrast, if the attacker observes low access times it means that the set remain untouched during the software execution. This can be utilized to recover AES, RSA or El Gamal keys.

Once again the *Prime and Probe* steps can be graphically seen in Figure 2.5. The attacker first fill the set with her red memory blocks, then waits for the victim to utilize the cache (yellow block) and finally measures the time to reaccess her red memory blocks. In this case she will trigger a miss in the probe step, meaning the victim utilized the corresponding set. Note the the *Prime and Probe* attack shows a much more practical scenario than `Evict and Time`, as no measurement to the victims process is performed. However, as it was only applied in the L1 cache (mainly due to some complications that will later be discussed) the community still did not consider cache attacks practical enough to perform them in real world scenarios.

2.4 Funcionality of Commonly Used Cryptographic Algorithms

In this section we review the functionality of the three most widely used cryptographic algorithms in every secure system: AES, RSA and ECC. The goal is to give the reader enough background to understand the attacks that will later carried out on these ciphers.

2.4.1 AES

AES is one of the most widely used symmetric cryptographic ciphers, a family of ciphers that utilize the same key during the encryption and decryption. Symmetric

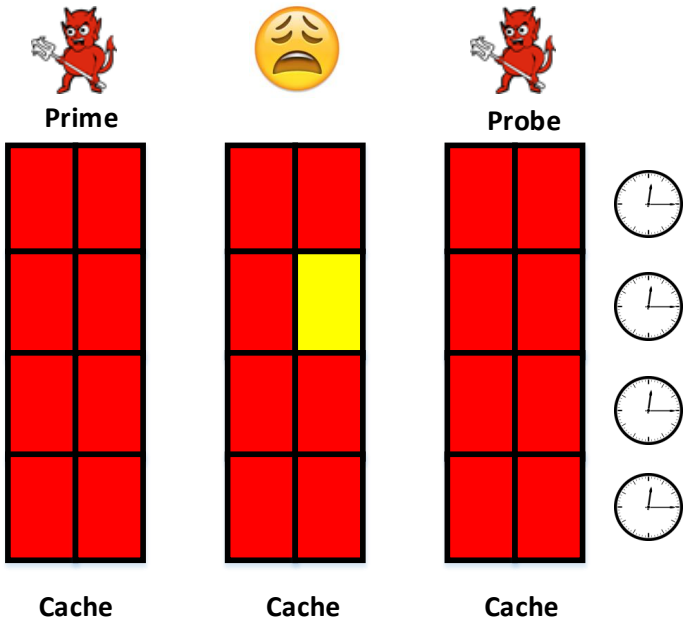


Figure 2.5: Prime and Probe procedure

cryptography ciphers can clearly provide **confidentiality**, as they can encrypt a message that only someone with access to the secret key is able to decrypt. Further, symmetric key ciphers can also be utilized in Cipher Block Chaining (CBC) mode for message **integrity** check, as symemtric key algoritm based Message Authentication Codes (MACs) can be built [Dwo04]. These ensure that the message does not get modified while being transmitted between two parties.

In particular, AES is part of the Rijndael cipher family, and as a block cipher, it processes packages in blocks of 16 bytes. AES is composed of 4 main operations, namely SubBytes, ShiftRows, Mixcolumns and AddRoundKey. The key can be of 128 ,192 and 256 bits, and depending on the length of the key, AES implements 10, 12 and 14 rounds of executions of the 4 main operation stages. The description of these stages is:

- AddRoundKey: In this stage, the round key is `XOR`ed with the intermediate state of the cipher.
- SubBytes: A table look up operation with 256 8 bit entry S-box.
- ShiftRows: In this stage the last three rows of ths state are shifted a given number of steps.
- MixColumns: A combination of the columns of the state.

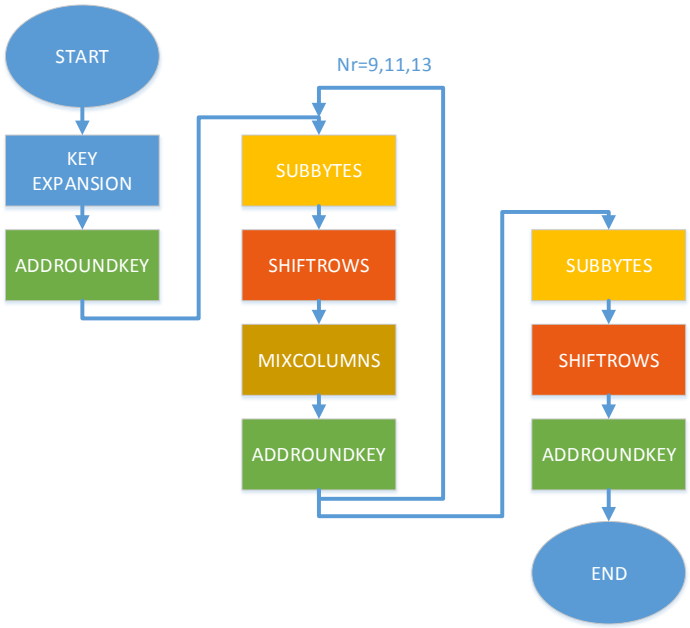


Figure 2.6: AES 128 state diagram with respect to its 4 main operations

Every round implements these 4 operations except the last one, in which a MixColumns operation is not issued. Figure 2.6 represents the state diagram of a 10 round AES with respect to these operations.

However, often cryptographic library designers decide to merge the SubBytes, ShiftRows and Mixcolumns operation into a Table look up operation and xor additions. The reason behind it is that, at the cost of a bigger table look up operation, the AES encryption is computed faster, as more operations are precomputed. Nevertheless the memory cost is usually affordable in smartphones, laptops, desktop computers and servers. These tables allocate 256 32 bit values, and usually 4 tables are utilized. We called these the T-tables. Some implementations even utilize a difference table for the last round. Figure 2.7 shows an example of the output of 4 bytes in the last round utilizing 4 T-tables. Observe that the T-table entry utilized directly depends on the key and the ciphertext byte, a detail that we will later utilize to perform an attack on the last round.

In general, for the scope of microarchitectural attacks, we do not consider those implementations based on the utilization of Intel AES hardware instructions (AES-NI). These instructions are built in modern processors to perform all the AES operations in pure hardware, i.e., without utilizing the cache hierarchy. Thus, any microarchitectural attack applied to AES-NI would not succeed on obtaining the key.

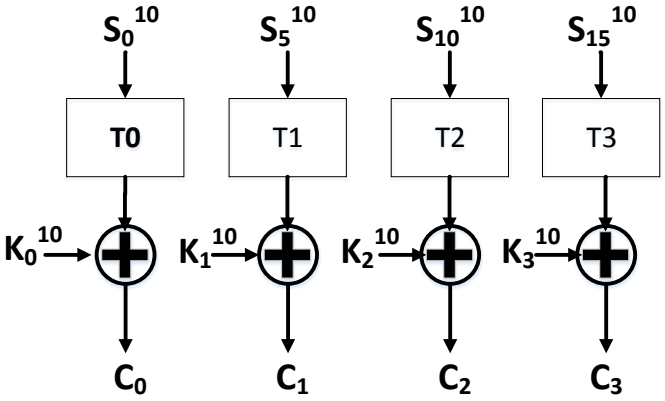


Figure 2.7: Last round of a T-table implementation of AES

2.4.2 RSA

RSA is the most widely used public key cryptographic algorithm. Public key cryptosystems usually try to solve the symmetric cryptography key distribution problem. Note that, symemtric key cryptography assumes that both parties share the same secret, but it does not explain how that secret can indeed be shared. And that is exactly where public key comes to place.

In public key (or asymmetric) crypto systems, in contrast to symmetric key cryptography, each agent in the communication is assumed to have two keys; a public key e and a private key d. The public key is publicly known, while the private key is kept secret. The two keys are related in the sense that one decodes what the other encoded, i.e., $D(E(M)) = M$ and $E(D(M)) =,$. It is assumed that revealing the public key does not reveal an easy information to compute D, and therefore, only the user holding the private key can decrypt mesages encrypted with his public key.

Assume the two agents willing to establish a secure communication are Alice and Bob. With public key cryptography, if Alice wants to send an encrypted message to Bob, she will have to encrypt the message with Bobs public key, as the message is only decryptable by Bobs private key. That is, public key cryptography ensures **confidentiality** between two comunicating agents.

The properties of public key cryptosystems also allow a user to digitally sign a message. This signature proves that the sender is indeed the one who he claims he is, i.e., public key cryptosystems also provide **authenticity**. Note that this is a feature that symmetric key algorithms were not able to claim. For instance assumes that Alice does not only wish to ensure confidentialy but also authenticity of their sent messages. Alice in this case would first sign the message with her own private key, and later encrypt it with Bobs public key $C = E_b(D_a(M))$. Upon the reception of this ciphertext Bob would decrypt the message with his own private key (which he

only knows) and verify the signature with Alice's public key (proving that only Alice could have signed the message), i.e., $M = E_a(D_b(C))$. Therefore the communication was not only encrypted but also authenticated.

In practice public key cryptosystems are usually utilized to distribute a shared symmetric key between two agents, such that these can later ensure confidentiality in their communication, since symmetric key cryptography is orders of magnitude faster thatn public key cryptography.

In particular, RSA takes advantage of the practical difficulty of the factorization of two large prime numbers to build a public key crypto system. Its operation is based on modular exponentiations. An overview of the key generation algorithm is presented in Algorithm 1 which starts by picking two distinct prime numbers p and q, and calculating $n = p * q$ and $\phi(n)$ as $(p-1) * (q-1)$. The public key e is chosen such that the greatest common divisor between e and $\phi(n)$ is equal to one. Finally, the private key is chosen as the modular inverse of e with respect to $\phi(n)$. The resistance of RSA comes from the fact that, even if the attacker knows n, it is computationally infeasible for him to recover the primes p and q that generated it.

Algorithm 1 RSA key generation given prime numbers p and q

Input : Prime numbers p and q
Output: Public key e and private key d
$n = p * q$;
$\phi(n) = (p-1) * (q-1)$;
//`Choose` e `s.t:`
1. $1 < e < \phi(n)$;
2. $gcd(e, \phi(n)) = 1$;
$d = e^{-1} \mod \phi(n)$;
return e, d, n

As we previously explained, messages are encrypted with public keys, performing the $c = m^e \mod n$ operation, while the decryption involves a modular exponentation with the private keys, i.e., $m = c^d \mod n$. Message signatures, in the other hand, are performed with the private key $s = m^d \mod (n)$ and verified with the public key $m = s^e \mod n$.

The modular exponentiation can indeed be performed in several ways. One way is to process the key bit by bit, and perform square and multiply operations accordingly. A description of such an implementation is presented in Algorithm 2, in which a square is always performed, and a multiplication with the base is performed only when a set bit is found.

But processing the key bit by bit is not the only possibility to perform modular exponentiations. Several implementations pre-compute some of the multiplicants in a table such that less multiplications have to be issued during the key processing. An example can be observed in Algorithm 3, in which values $b, b^2, b^3, \ldots, b^{2^w-1}$ are pre-computed and stored in a table, being w the window size. The key is then processed

Algorithm 2 Square and Multiply Exponentiation

Input : base b, modulus N, secret $E = (e_{k-1}, ..., e_1, e_0)$
Output: $b^E \bmod N$
$R = b$;
for $i = k - 1$ **downto** 0 **do**
  $R = R^2 \bmod N$;
  if e_i==*1* **then**
    $R = R * b \bmod N$;
  end
end
return R;

in chunks of w bits, w squares are issued together with a multiplication with the appropriate table entry. This procedure is called fixed windowed exponentiation.

Algorithm 3 Fixed window exponentiation

function modpow (a, b);
Input : base b, modulus N, secret $E = (e_{k-1}, ..., e_1, e_0)$
Output: $b^E \bmod N$
$T[0] = 1$;
$v = b$;
for $i = 1$ **to** $2^w - 1$ **do**
  $T[i] = T[i-1] * v$;
end
$R = 1$, $i = k - 1$;
while $i > 0$ **do**
  $S = e_i e_{i-1} \dots e_{i-w}$;
  $R = R^{2^w} \bmod N$;
  $R = R * T[S] \bmod N$;
  $i = i - w$;
end
return R;

Typical sizes for RSA are 2048 and 4096 bits, while typical window sizes range between 4 and 6. Indeed this is one of the main disadvantages of RSA; a good security level is only achieved with very large keys. The following section describes Elliptic Curve Cryptography (ECC), another public key crypto system that provides the same level of security with much lower key sizes.

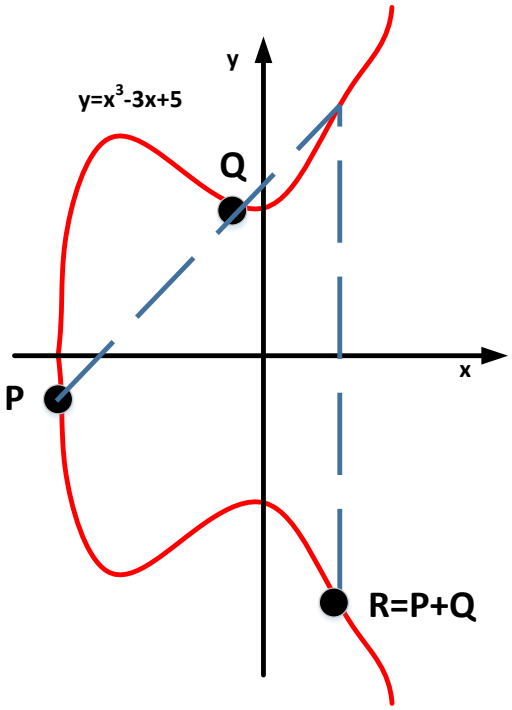


Figure 2.8: ECC elliptic curve in which R=P+Q

2.4.3 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) also belongs to the category of public key cryptographic algorithms. As with RSA, each user has a public and a private key pair. However, while the security of RSA relied mainly in the large prime factorization problem, ECC relies on the *elliptic curve discrete logarithm problem*: finding the discrete logarithm of an element in an elliptic curve with respect to a generator is computationally infeasible [HMV03, Mil86, LD00]. In fact, ECC achieves the same level of security as RSA with *lower key sizes*. Typical sizes for ECC keys are 256 or 512 bits.

The communication peers have first to agree on the ECC curve that they are going to utilize. A curve is just the set of points defined by an equation, e.g., $y^2 = x^3 + ax + b$. This equation is called the Weirstrass normal form of elliptic curves. In elliptic curves we define addition arithmetic as follows:

- if P is a point in the curve, $-P$ is just its reflection over the x axis.

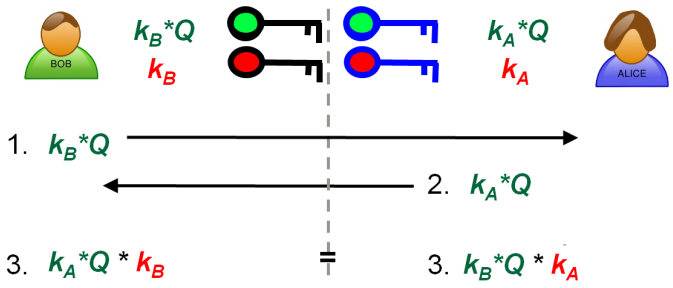


Figure 2.9: ECDH procedure. $k_a * k_b * Q$ is the shared secret key

- if two points P and Q are distinct, then result of adding $P+Q = R$ is computed by drawing a line crossing P and Q, which will intersect in a third point $-R$ in the curve. R is computed by taking the reflection of $-R$ with respect to the x axis.

- $P + P$ is computed by drawing a tangent line to the curve at P, which again will intersect in a third point in the curve $-2P$. $2P$ is just the reflect-ion over the x axis.

An example of an addition of two distinct points P and Q is presented in Figure 2.8, in which the line crossing P and Q intersects in a third point $-R$, for which we calculate the negated value by taking its reflection over the x axis.

With the new ECC point arithmetic we can define, as with RSA, cryptosystems based on the elliptic curve discrete logarithm problem. For all of them we assume the two communication peers agree on a curve, a base point Q in the curve. Each of them choose a scalar that will be their private key, while they while compute the public key as $k * Q$, being k the scalar and Q the base point. Note that,the resistance of ECC cryptosystems relies on the fact that, knowing Q and $k * Q$ it is very difficult to obtain k.

With these parameters, the Elliptic Curve Diffie Hellman (ECDH) is easily computable. In fact, both peers can agree on a shared key by performing $k_a * P_b$ and $k_b * P_a$, where k_i and P_i are private and public keys of peer i. The entire procedure for ECDH can be observed in Figure 2.9. With similar usage of the ECC properties digital signatures can also be performed.

Chapter 3

Related Work

The following section provides the state-of-the-art of classic as well as microarchitectural side channel attack literature. Publications are grouped and discussed to establish a comprehensive overview of scenarios and covert channels that have been exploited to retrieve sensitive information.

3.1 Classical Side Channel Attacks

Classical Side Channel attacks were introduced by Kocher et al. [Koc96, KJJ99] almost two decades ago and introduced a new era of cryptography research. Before the discovery of side channel attacks, the security of cryptographic algorithms was studied by assuming that adversaries only have *black box access* to cryptographic devices. As it turned out, this model is not sufficient for real world scenarios, because adversaries have additionally access to a certain amount internal state information which is leaked by side channels. Typical side channels include the power consumption, electromagnetic emanations and timing of cryptographic devices. In the following we give an overview on how these have been implemented with respect to timing and power side channel attacks.

3.1.1 Timing Attacks

Timing side channel attacks take advantage of variations in the execution time of security related processes to obtain information about the secret being utilized. These variations can arise due to several factors, e.g., cache line collisions or non-constant execution flows. For instance, Kocher [Koc96] demonstrated that execution time variations due to non-constant execution flow in a RSA decryption can be utilized to get information about the private key. Similar instruction execution variations were exploited by Brumley and Boher [BB03] in a more realistic scenario, i.e., an OpenSSL-based web server Later, Bernstein [Ber04] on the contrary exploited timing variations stemming from cache set access time differences to recover AES keys. In 2009 Crosby et al. [CWR09] expanded on [Koc96] reducing the jitter effect

from the measurements and therefore succeeded recovering the key with significantly less traces. [BB03] was also further expanded in 2011 by Brumley et al. [BT11] showing the ECC implementations can also suffer from timing variation leakage. Timing attacks have also shown to recover plaintext messages sent over security protocols like TLS, as demonstrated by Al Fardan and Paterson [FP13].

3.1.2 Power Attacks

During the last years, several attacks exploiting side channels were introduced. In the following, the required steps in order to perform a successful attack are described.

Measurement

In the first step, the acquisition of one or more side channel traces is performed, which results in discrete time series. The resulting traces represent physical properties of the device during execution such as the power consumption or electromagnetic field strength on a certain position.

Pre-processing

In the second optional step, the raw time series can be enhanced for further processing. In this step several techniques in order to reduce noise, remove redundant information by compressing the data, transformations into different representations like the frequency domain, and trace alignment are performed.

Analysis

The actual analysis step extracts information from the acquired traces in order to support the key recovery. There are many more or less complex methods available, which can be categorized with two orthogonal criteria. The first one is based upon the requirement if multiple side channel traces are needed or a single trace is sufficient to perform the attacks. The second criterion is based upon the question, whether a training device is available to obtain the leakage characteristics or if the attack is based on assumptions about the leakage characteristics. Table 3.1 shows the four possible combinations of these criteria.

Table 3.1: Side channel attack classification according to utilized data analysis method

	Single Measurement	**Multiple Measurements**
Non-Profiling	Simple Analysis (SPA, SEMA)	Differential Analysis (DPA, DEMA, CPA, MIA, KSA)
Profiling	Template Attacks (TA)	Stochastic Approach (SA)

The most basic analysis is the simple power analysis (SPA) which only requires one side channel trace and does not require profiling. The common way to perform this kind of attack is by visual inspection of a plotted trace. The goal is to find patterns in the trace caused by conditional branching with dependency on the secret key. This attack works well with computations on large numbers as used by public key cryptography.

The differential analysis method is very powerful by using statistics on multiple side channel traces. These method does not need a profiling step and uses leakage assumptions based on hypothetical internal values dependent on a small part of the key. This way, an attacker can try out different assumptions about a part of the key and compare the leakage assumptions with an actual observable leakage. The statistical methods utilized to quantify the accuracy of the assumption are referred as distinguishers in the side channel literature. The historically first distiguisher was the distance of means test [KJJ99] that is used by the differential power attack (DPA) and differential electromagnetic attack (DEMA). Later, more powerful ones including Pearson correlation (for CPA) [BCO04], mutual information (MIA) [GBTP08], and the Kolmogorov-Smirnov test (KSA) [WOM11] were introduced.

Attacks with a profiling step are based upon the assumption that the leakage characteristics of the different devices of the same type are similar. Using a training device, an attacker can model the leakage characteristics of the device and can use this model for the actual attack. The historically first profiled attack was the template attack (TA) [CRR03]. This method is based on multivariate Gaussian models for all possible sub-keys. Template attacks allow key extractions based on only one side channel trace after proper characterization of a training device.

The stochastic approach (SA) [SLP05, KSS10] approximates the leakage function of the device using a linear model. The actual key extraction can be performed by different methods. For example, in [SLP05] the maximum likelihood principle is used. The linear model parameters can also be used to identify leakage sources in a design as described by De Santis et al. [DSKM$^+$13].

3.2 Microarchitectural Attacks

Microarchitectural attacks try to exploit specific features of computer microarchitectures *without requiring physical access to the device* to recover cryptographic keys, chasing the pattern of other processors and also threading the cloud users. The focus of this section is put on hardware components that have been exploited in microarchitectural attacks, which have exhibited a significant tendency change on multi-core/CPU systems.

A rough description of the basic components in modern microarchitectures was presented in Figure 2.2. Modern computers usually consist on one or more CPU sockets, each containing several CPU cores. Processes are executed in one or several cores at the same time. The Branch Prediction Unit (BPU) is in charge of mak-

ing predictions on possible outcomes of branches inside the code being executed. L1/L2 and L3 caches, in order to avoid subsequent DRAM accesses, store data and instructions that have *recently* being used, because they are very likely to be utilized again. The memory bus is in charge of the communication between the cache hierarchy and the DRAM, and further can possibly used to communicate the state of shared memory across CPU sockets. Finally, the DRAM holds the memory pages that are necessary for the execution of the program.

Particularly important for microarchitectural attacks is to identify which of these components can serve as a covert channel to perform single-core, cross-core and cross-CPU attacks. In fact, from the figure we can identify many of the covert channels that will later be explained. For instance, it can be observed that an attacker trying to exploit Branch Prediction Units (BPUs) or L1/L2 caches has to co-reside in the same core as the victim, as they are core-private resources. A cross-core attack can be implemented if the L3 cache is utilized as the covert channel, since it is shared across cores. Finally, attacks across CPU sockets can be achieved exploiting, among other components, the memory bus and the DRAM. All these have been exploited in very different manners that will be explained in the following section.

3.2.1 Hyper-threading

Hyper threading technology was introduced to perform multiple computations in parallel by Intel in 2002. Each processor core has two virtual threads and they share the work load of a process. The main purpose of hyper-threading is to increase the number of independent instruction in the pipeline. In 2005, Percival et al. [Per05] exploited this technology to establish a cache covert channel in L1-data cache. In the same core but in different threads spy and crpytographic libraries are executed and the spy code is able to recover some bits of the secret RSA exponent. In 2007, Aciicmez et al. [AS07] proposed a new method to exploit the leakage in the hyper-threading technology of Intel processors. To do this, they take advantage of shared ALU's large parallel integer multiplier between two threads in a processor core. Even if they did not introduce a new vulnerability in OpenSSL library, they showed that it is possible to use ALU as a covert channel in the secret exponent computation.

3.2.2 Branch Prediction Unit Attacks

Control hazards have an increasing impact on the performance of a CPU as the pipeline depth increases. Efficient handling of speculative execution of instructions becomes a critical solution against control hazards. This efficiency is usually achieved by predicting the most likely execution path, i.e., by predicting the most likely outcome of a branch. Specifically, Branch Prediction Units (BPU) are in charge of predicting the most likely path that a branch will take. BPUs are usually divided into two main pieces: Branch Target Buffers (BTB) and the predictor. The

BTB is a buffer that stores the addresses of the most recently processed branches, while the predictor is in charge of making the most likely prediction of the path.

As BPUs are accessible by any user within the same core, the BTB has become a clear target to perform microarchitectural attacks. Imagine a BPU in which branches that are not observed in the BTB are always predicted as not taken, and will only be loaded into the TBT once the branch is taken. If a piece of code has a security critical branch to be predicted, a malicious user can interact with the BTB (i.e., by filling it) to ensure that the branch will be predicted as not taken. If the attacker is able to measure the execution time of the piece of code, he will be able to say whether the branch was misspredicted (i.e. the branch was taken) or the branch was correctly predicted (i.e. the branch was not taken).

This is only one possible attack vector that can be implemented against the BPU. A more realistic scenario is the one in which the attacker fills the TBT with always taken branches (which evicts any existing branches from the TBT), then waits for the victim to execute his security critical branch, and finally measure the time to execute his always taken branches again. If the security critical branch was taken, then the branch will be loaded into the TBT and one of the attackers branches will be evicted, causing a misspredict that he will observe in the measured time. In the other hand, if the security critical branch was not taken, it will not be loaded into the TBT and the attacker will predict correctly all his branches. The two attack models discussed have been proposed in [AKKS07, AKS07].

However, BPU microarchitectural attacks have a clear disadvantage when compared to other microarchitectural attacks: BPU units are core-private resources. Thus, these attacks are only applicable if attacker and victim co-reside in the same core. Nevertheless, new scenarios arise in which core co-residency is easily achievable, as TEE attacks (discussed in Section 3.2.12). Malicious OSs can control the scheduling of the CPU affinity of each process. In fact, attacks utilizing BPU have already been proposed in that scenario [LSG+16].

3.2.3 Out-of-order Execution Attacks

A recent microarchitectural side channel that was discovered and proved to establish communication between co-resident VMs is the exploitation of out of order executed instructions [D'A15]. Out of order execution of instructions is an optimization present in almost all the processors that allows the finalization of a future instruction while waiting for the result of a previous instruction. A first intuition of out-of-order instructions finishing earlier than in-order instructions was made in [CVBS09]. As with other microarchitectural attacks, optimizations can indeed open new covert channels to be exploited by malicious attackers.

Assume two threads execute interdependent load and store operations into a shared variable (i.e., one threads stores the value loaded by the other one). If these threads run in parallel, three cases might arise: both threads are executed in-order and at the same time, both threads are executed in-order but one is executed

faster or both threads execute out of order instructions. Note that the result of the operations will be different for each of the outputs. Thus, a covert channel can be established by transmitting a 0 when instructions are executed out of order, and a 1 in any other case. Indeed, an attacker can ensure to transmit a 1 by utilizing memory barrier instructions, which keep the execution of the instructions in-order.

3.2.4 Performance Monitoring Units

Performance Monitoring Units (PMU) are a set of special-purpose registers to store the count of hardware and software related activities. In 1997, the first study was conducted by Ammons et al. [ABL97] where the logical structure of hardware counters are explained to profile different benchmarks. In 2008, Uhsadel et al. [UGV08] showed that hardware performance counters can be used to perform cache attacks by looking at the L1 and L2 D-cache misses. In 2013, Weaver et al. [WTM13] investigated x86_64 systems to analyze deterministic counter events, concluding that non-x86 hardware has more deterministic counters. In 2015, Bhattacharya et al. [BM15] showed how to use the branch prediction events in hardware performance counters to recover 1024 bit RSA keys in Intel processors.

3.2.5 Special Instructions

The set of instructions that a CPU can understand and execute is refered as instruction set architectures (ISA). Although usually composed of common instructions (e.g. mov or add), some ISAs include a set of special instructions that aim at implementing system functionalities that the system cannot implement by itself. One of these examples are CPUs with a lack of memory coherence protocols. In these cases, special instructions are needed to handle the situation of conflicted (thus incoherent) values between the DRAM and the cache hierarchy.

For instance, the Intel x86 64 ISA provides the clflush instruction to solve the problem. The `clflush` instruction evicts the desired data from the cache hierarchy, thus making sure that the next data fetch will come from the DRAM. Since ARMV8, ARM processors started including similar instructions in their ISA. Although these instructions might become crutial for certain processors, they also serve as helpers to implement microarchitectural attacks. In fact many times an attacker is willing to evict a shared variable from the cache, either to reload it later [YF14, LGS+16, IES16], to measure the flushing time [GMWM16] or to access the DRAM continuously [KDK+14a].

A similar situation can be observed with instructions added to utilize hardware random number generators. In particular, it has recently been shown that, due to the low throughput of the special `rdseed` instruction, a covert channel can be implemented by transmitting different bits when the `rdseed` is exhaustively used/not used [EP16].

3.2.6 Hardware Caches

Modern cache structures, as shown in Figure 2.2, are usually divided into core-private L1 instruction and data caches, and at least one level of unified core-shared LLC. The instruction cache (I-cache) is the part of the L1 cache responsible for storing recently executed instructions by the CPU. In 2007, Aciicmez [Acı07] showed the applicability of I-cache attacks by exploiting the cipher accesses to recover an 1024-bit RSA secret key. The monitored I-cache sets are filled with dummy instructions by a spy process whose access is later timed to recover a RSA decryption key. After one year, Aciicmez et al. [AS08] demonstrated the power of the I-cache attacks on OpenSSL RSA decryption processese that use Chinese Remainder Theorem (CRT), Montgomery's multiplication algorithm and blinding to modular exponentiation. In 2010, Aciicmez et al. [ABG10] revisited I-cache attacks. In this work, the attack is automated by using Hidden Markov Models (HMM) and vector quantization to attack OpenSSL's DSA implementation. In 2012, Zhang et al. [ZJRR12] published a paper on cross-VM attacks in the same core using L1-data cache. The attack targets Libgcrypt's ElGamal decryption process to recover the secret key. To eliminate the noise Support Vector Machine algorithm is applied to classify square, multiplication and modular reduction from the L1-data cache accesses. The output of SVM is given to HMM to reduce the noise and increase the reliability of the method. This paper demonstrates the first study in cross-VM setup using L1-data cache. Finally, in 2016, Zankl et al. [ZHS16] proposed an automated method to find the I-cache leakage in RSA implementations of various cryptographic libraries. The correlation technique shows that there are still many libraries which are vulnerable to I-cache attacks.

The data cache stores recently accessed data, and as with the I-cache, it has been widely exploited to recover sensitive information. As soon as 2005, Percival et al. [Per05] demonstrated the usage of the L1-data cache as a covert-channel to exploit information from core co-resident processes. To demonstrate the efficiency of the side channel, the OpenSSL RSA implementation is targeted. The main reason of the leakage is that the different precomputed multipliers are loaded into different L1-data cache lines. By profiling each corresponding set it is possible to recover more than half of the bits of the secret exponent. A year later, Osvik et al. [OST06] presented the *Prime and Probe* and `Evict and Time` attacks, which were utilized to recover AES cryptographic keys. In 2007, Neve et al. [NS07] showed that it is possible to use L1-data cache as a side channel in single-threaded processors by exploiting the OS scheduler. This technique was applied to the last round of OpenSSL's AES implementation where only one T-table is used. The authors recovered the last round key with 20 ciphertext/accesses pairs. In 2009, a new type of attack is presented by Brumley et al. [BH09] where L1-data cache templates are implemented to recover OpenSSL ECC decryption keys. The goal of this method is to automate cache attacks by applying HMM and vector quantization in Pentium 4 and Intel Atom. Finally in 2011, Gullasch et al. [GBK11] presented the *Flush and Reload*

attack, although the attack would acquire its name later. The study demonstrated that it is possible to affect the Linux' Completely Fair Scheduler (CFS) to interrupt the AES thread and recover the AES encryption key with very few encryptions.

LLC attacks

In 2014, Yarom et al. [YF14] implemented, for the first time, the *Flush and Reload* attack across cores/VMs to recover sensitive information aided by memory deduplication mechanisms. The attack is applied to the GnuPG implementation of RSA. With this work the *Flush and Reload* attack became popular and more scenarios in which it could be applied arised. For instance, Benger et al. [BvdPSY14] presented the *Flush and Reload* attack on the ECDSA implementation of OpenSSL. In the same year, Irazoqui et al. [IIES14b] applied *Flush and Reload* to recover AES encryption keys across VMs. This attack is implemented in VMware platforms to show the strength of the attack in virtualized environments. Shortly later, Zhang et al. [ZJRR14] showed the applicability of the *Flush and Reload* attack to verify the co-location in PaaS clouds and obtain the number of items of a co-resident users shopping cart. In 2015, Irazoqui et al. [IIES15] showed that it is possible to recover sensitive data from incorrectly CBC-padded TLS packets. In the same year, Gruss et al. [GSM15] presented an automated way to catch LLC cache patterns applying *Flush and Reload* method and consequently detect keys strokes pressed by the victim.

Finally, hypervisor providers disabled the de-duplication feature to prevent *Flush and Reload* attacks in their platforms. Therefore, there was a need to find a new way to target the LLC. Concurrently, Irazoqui et al. [IES15a] and Liu et al. [Fan15] described how to apply *Prime and Probe* attack in LLC on deduplication-free systems. While Irazoqui et al. [IES15a] applied the method to recover OpenSSL AES last round key in VMware and Xen hypervisors, Liu et al. [Fan15] demonstrated how to recover El Gamal decryption key from the recent GnuPG implementation. In 2016, Inci et al. [IGI+16b] showed that commercial clouds are not immune to these attacks and applied the same technique in the commercial Amazon EC2 cloud to recover 2048 bit RSA key from Libgcrypt implementation. Moreover, Oren et al. [OKSK15] presented the feasibility of implementing cache attacks trough javascript executions to profile incognito browsing.

In the same year, Lipp et al [LGS+16] applied three different methods (*Prime and Probe*, *Flush and Reload* and Evict+Reload) to attack ARM processors typically used in mobile devices. The success of the work showed that it is possible to implement cache attacks in mobile platforms.

3.2.7 Cache Internals

With novel an more complex cache designs, which include many undocumented features, it become increasingly difficult to obtain the necessary knowledge to use it as

a covert channel. Aiming at correctly characterizing the usage of the caches as an attack vector, researchers started investigating their designs. An example of these features are the LLC slices, selected by an undocumented hash functions in Intel processors. In 2015, the first LLC slice reverse engineering was applied by Irazoqui et al. [IES15b] utilizing timing information to recover slice selection algorithms of a total of 6 different Intel processors. Later, Maurice et al. [MSN+15] presented a more efficient method using performance counters to recover slice selection algorithm. Later,Inci et al. [IGI+16b] and Yarom et al. [YGL+15] again utilized timing information to recover non-linear slice selection algorithms. Finally, in 2016, Yarom et al. [YGH16] exploited the cache-bank conflicts on the Intel Sandy Bridge processors. The study shows that cache-banks in L1 cache can be used to recover an RSA key if the hyper-threading is implemented in the core.

3.2.8 Cache Pre-Fetching

Pre-fetching is a commonly used method in computer architecture to provide better performance to access instructions or data from local memory. It consists on predicting the utilization of a cache line and fetching it to the cache prior to its utilization. In 2014, Liu et al. [LL14] suggested that previous architectural countermeasures do not provide sufficient prevention against demand-fetch policy of shared cache. Thus, to prevent the pre-fetching attacks they propose a randomization policy for the LLC where a cache miss is not sent to the CPU, but instead perform randomized fetches to neighborhood of the missing memory line. In 2015, Rebeiro et al. [RM15] presented an analysis of sequential and arbitrary-stride pre-fetching on cache timing attacks. They conclude that ciphers with smaller tables leak more than ciphers with large tables due to data pre-fetching. In 2016, Gruss et al. [GMF+16] utilized the pre-fetching instructions to obtain the address information by defeating SMAP, SMEP and kernel ASLR.

3.2.9 Other Attacks on Caches

Other attacks have exploited different characteristics of a process rather than execution time and cache accesses . In 2010, Kong et al. [KJC+10] presented a thermal attack on I-cache by checking the cache thermal sensors. They showed that dynamic thermal management (DTM) is not sufficient to prevent the thermal attacks if the malicious codes target a specific section of I-cache. In 2015, Masti et al. [MRR+15] expanded the previous work to multi-core platforms. They used core temperature as a side channel to communicate with other processes. In the same year, Riviere et al. [RNR+15] targeted I-cache by electromagnetic fault injection (EMFI) on the control flow. They showed that the applicability of the fault injection is trivial against cryptographic libraries even if they have countermeasures against fault attacks. In 2016, researchers focused on Spin-Transfer Torque RAM (STTRAM), a promising feature for cache applications. Rathi et al. [RDNG16] proposed three new tech-

niques to handle magnetic field and temperature of processor. The new techniques are stalling, cache bypass and check-pointing to establish the data security and the performance degradation is measured with SPLASH benchmark suite. In the same year, Rathi et al. [RNG16] exploited read/write current and latency of STTRAM architecture to establish a side channel inside LLC.

3.2.10 Memory Bus Locking Attacks

So far it has been shown how an attacker can create contention at different cache hierarchy levels to retrieve secret information from a co-resident user. Memory bus locking attacks are different in this sense, since they do not have the ability to recover fine-grain information. Yet, they can be utilized to perform a different set of malicious actions or as a pre-step to the performance of more powerful side channel attacks.

When two different threads located in different cores operate on the same shared variable, the value of the shared variable retrieved by each thread might be different, since modifications might only be performed to core-private caches. This is called a data race. In order to solve this, atomic operations operate in shared variables in such a way that no other thread can see the modification half-complete. For that purpose, atomic operations usually utilize lock prefixes to ensure the particular shared variable in the cache hierarchy is locked until one of the threads has finished updating the value.

The lock instructions work well when the data to be locked fits into a single cache line. However, if the data to be locked spans more than one cache line, modern CPUs cannot issue two lock instructions in separate cache lines at the same time [IGES16]. Instead, modern CPUs adopt the solution of flushing any memory operation from the pipeline, incurring in several overheads that can be utilized by users with malicious purposes.

In fact, there are several examples of malicious usage of memory bus locking overheads. For instance, Varadarajan et al. [VZRS15], Xu et al. [XWW15] and Inci et al. [IGES16] utilized this mechanism to detect co-residency in IaaS clouds by observing the performance degradation in http queries to the victim. Finally, Inci et al. [IIES16] and Zhang et al. [ZZL17] utilized memory bus locking effects as a Quality of Service (QoS) degradation mechanism in IaaS clouds.

3.2.11 DRAM and Rowhammer Attacks

The DRAM, often also called the main memory, is the hardware component in charge of providing memory pages to executed programs. There are two major attacks targeting the DRAM: the rowhammer attack and DRAMA side channel attacks. Since both attacks utilize different concepts and have different goals, we proceed to explain them separately.

The DRAM is usually divided into several categories, i.e., channels (physical links between DRAM and memory controller), Dual Inline Memory Modules (DIMMS, physical memory modules attached to each channel), ranks (back and front of DIMMS), banks (analogous to cache sets) and rows (analogous to cache lines). Two addresses are physical adjacent if they share the same channel, DIMM, rank and bank. Additionally each bank contains a row buffer array that holds the most recently accessed row. DRAMA side channel attacks take advantage of the fact that accessing a DRAM row from the row buffer is faster in time that having a row buffer miss. Thus, similar to the fact observed in cache attacks, an attacker can create collisions in a banks row buffer to infer information about a co-resident victim.

The rowhammer attack takes advantage of the influence that accesses to a row in a particular bank of the DRAM have in adjacent rows of the same bank. In fact, continuous accesses can increase the charge leak rate of adjacent rows. Typically processors have a refresh rate (at which they refresh their values) to avoid the complete charge leak of cells in a DRAM row. However, if the charge leak rate is faster than the refresh rate, then some cells completely loose their charge before their value can be refreshed, incurring in bit flips. These bit flips can be particularly dangerous if adjacent rows contain security critical information.

The rowhammer attack has been more exploited than the DRAMA side channel attack. In fact, since it was discovered [KDK+14a], rowhammer has been shown to be executable from a javascript extension [GMM16], in mobile devices [vdVFL+16], or even from a IaaS VM to break the isolation provided by the hypervisor [XZZT16]. Further, it has been shown that it can be utilized as a mechanism to inject faults into cryptographic keys that can lead to their full exposure [BM16].

3.2.12 TEE Attacks

Trusted Execution Environmnets (TEE) are designed with the goal of executing processes securely in isolated environments, including powerful adversaries such as malicious Operating Systems. In order to achieve this goal, TEEs usually work, among others, with encrypted DRAM memory pages. However, TEEs still utilize the same hardware components as untrusted execution environments, and thus most of the above explained microarchitectural attacks are still applicable to processes executed inside a TEE. Examples of TEEs are the well known Intel SGX [Sch16] and the ARM TrustZone [ARM09].

Perhaps the distinguishing fact when discussing the applicability of microarchitectural attacks to TEE is the DRAM memory encryption. While in fact TEE DRAM memory pages are encrypted, this data is decrypted when placed in the cache hierarchy for faster access by the CPU. This implies that any of the above discussed cache hierarchy attacks is still applicable against TEE environments, as nothing stops an attacker from creating cache contention and obtaining *decrypted* TEE information. More than that, a malicious OS can schedule processes in any

way it wants, interrupt the victim process after every small number of cycles or prevent it from using side channel free hardware resources (e.g. AES-NI [TNM11]). Thus, compared to the low resolution that microarchitectural attackers obtain from a commercial OS, a malicious OS obtains a much higher resolution as it can observe every single memory access made by the victim.

A similar issue is experienced with BPU microarchitectural attacks. BPU attacks were largely dismissed due to their core co-residency limitation (BPU are core-private resources) and their low access distinguishability. However, the whole picture changes when a malicious OS comes into play. Once again, malicious OS are in control of the scheduling of the processes being executed in the system. Thus, they are in control of where and when each process executes, i.e., they can schedule malicious processes in the same core as the victim process. The situation becomes even more dangerous when special instructions are available from an OS to better control the outcome of the branches executed [LSG+16].

Although DRAM row access and memory bus locking attacks have still not been shown to be applicable in the TEE scenario, the theoretical characteristics of these attacks suggest that both attacks will be soon demonstrated to be applicable in TEEs. In fact there is nothing theoretically solid that suggests TEE would prevent memory bus locking attacks. As for the DRAM access attacks our assumption is that it will depend on how the DRAM rows are placed in the row buffer array. If these are placed unencrypted, then the attack would be equally viable. Rowhammer, on the contrary, is assumed to be defeated by memory authentication and integrity mechanisms being utilized by TEEs.

3.2.13 Cross-core/-CPU Attacks

Multiple microarchitectural attacks have been reviewed in the previous paragraphs, but it has not been discussed how their applicability improved over the years. In fact, microarchitectural attacks have improved their practicality since they were implemented for the first time in 2005. Not in vain, the first microarchitectural attacks (i.e. those targeting the L1 and BPU) were largely dismissed almost for several years, as they were only applicable if victim and attacker shared the same CPU core. In a multi-core system world, this restriction seemed to be the main limitation that prevented microarchitectural attacks from being considered as a threat.

It was in 2013 when the first cross-core cache attack was presented, even though it required that victim and attacker shared memory pages. Later this requirements would be eliminated, making cross-core cache attacks applicable in almost every system. However, there was a requirement that microarchitectural attacks did not accomplish yet, i.e., to be applicable even when victim and attacker share the underlying hardware but are located in different CPU sockets.

The microarchitectural attack cross-CPU applicability would later be accomplished by cache coherency protocol attacks, rowhammer, DRAMA and memory

bus locking attacks. Indeed all of them target resources that are shared not only by every core, but farther by every CPU socket in the system. As explained in the previous paragraphs, the first targets the cache coherence protocol characteristics, the next two target the DRAM characteristics, while the latter targets the memory bus characteristics.

In short, microarchitectural attacks have experienced a huge popularity increase in the last years, specially for their improved characteristics and applicability. While they were largely dismissed at the beginning for being applicable in private core-resources, current microarchitectural attacks are applicable even across CPU sockets.

Chapter 4

The *Flush and Reload* attack

Until 2013 microarchitectural attacks have been largely disregarded by the community, mainly due to their lack of applicability in real world scenarios. In particular, they have suffered from the following limitations:

- Microarchitectural attacks were only shown to be successful in core-private resources, like the L1 caches and the BPUs. With the wide utilization of multi-core systems, the restriction of having to be located in the same core with the victim seems a huge obstacle when it comes to the applicability of microarchitectural attacks.
- Core-private resources usually exhibit low resolution to execute microarchitectural attacks. For instance, L1 and L2 accesses only differ in a few cycles. BPU misspredictions are also hardly distinguishable as their penalties have been largely optimized. Therefore, core-private resources are less likely to resist the amount of noise typically observed in real world scenarios.
- Before 2013 microarchitectural attacks lacked of real world scenarios in which they could be applied, as they usually demanded a highly controlled environment.

In order to increase the applicability of microarchitectural attacks it is necessary to investigate and discover new covert channels that can be exploited across cores and be resistant to the typical amount of noise seen in modern technology usage environments. The only work prior to ours that investigated cross-core covert channels is [YF14], in which a RSA key is recovered through the Last Level Cache (LLC). This sections expands on [YF14], demonstrating that LLC attacks can recover a wider range of information in a number of scenarios.

The first attack that we discuss is the *Flush and Reload* attack, first applied in the L1 in [GBK11], which acquired its name in [YF14]. The *Flush and Reload* attack utilizes the LLC as the covert channel to recover information, and prior to this work, was only shown to be applicable across processes being executed in the

same OS. In this chapter we demonstrate how such an attack can be utilized to recover fine-grain information from a co-resident VM placed in a different core. In particular,

- We demonstrate that *Flush and Reload* can recover information from co-resident VMs in hypervisors as popular as VMware.
- We expand on the capabilities of *Flush and Reload* by recovering an AES key in less than a minute.
- We show that microarchitectural attacks can be applied against high level protocols by recovering TLS session messages.

4.1 *Flush and Reload* Requirements

We cannot start discussing the functionality of the *Flush and Reload* attack without first mentioning the requirements that are needed to successfully apply it. In particular the *Flush and Reload* attack has four very important pre-requisites that have to be met in the targeted system to succeed:

- **Shared memory with the victim:** The *Flush and Reload* attack assumes that attacker and victim share at least the targeted memory blocks in the system, i.e., both access the same physical memory address. Although this requirement might seem difficult to achieve, we discuss in Section 4.2 the scenarios in which memory sharing approaches are implemented.
- **CPU socket co-residency:** The *Flush and Reload* attack is only applicable if attacker and victim co-reside in the same CPU socket, as the LLC is only shared across cores and not across CPU sockets. Note that, unlike prior microarchitectural attacks, core co-residency is not necessary.
- **Inclusive LLC:** The *Flush and Reload* attack requires the inclusiveness property in the LLC. This, as we will see in section 4.4.1, is necessary to be able to manipulate memory blocks in the upper level caches. The vast majority of intel processors feature an inclusive LLC. However, as we will see in Section 5, similar technical procedures can exploit non-inclusive caches through cache coherence covert channels.
- **Access to a flushing instruction:** The *Flush and Reload* attack needs of a very specific instruction in the Instruction Set Architecture (ISA) capable of forcing memory blocks to be removed from the entire cache hierarchy. In `x86-64` systems, this is provided trough the `clflush` instruction.

Without any of these four requirements the *Flush and Reload* attack is not able to successfully recover data from a victim. Two of them can be assumed to be easily achievable in Intel processors, as they feature inclusive LLCs and the `clflush` instruction is accessible from userspace. The CPU socket co-residency should also be easily accomplished, as modern computers do not usually feature more than two CPU sockets. However, the shared memory requirement might be harder to achieve. In the following section we describe mechanisms under which different processes/users share the same physical memory.

4.2 Memory Deduplication

Although the idea of different processes/users sharing the same physical memory might seem threatening, the truth is we encounter mechanisms that permit it in popular OS and hypervisors. In particular, all linux OS implement the so called Kernel Samepage Merging (KSM) mechanism, which merges duplicate read-only memory pages belonging to different processes. Consequently KVM, a linux based hypervisor, features the same mechanisms across different VMs. Furthermore, VMware implements Transparent Page Sharing (TPS), a similar mechanism to KSM that also allows different VMs to share memory.

Even though the deduplication optimization method saves memory and thus allows more virtual machines to run on the host system, it also opens a door to side channel attacks. While the data in the cache cannot be modified or corrupted by an adversary, parallel access rights can be exploited to reveal secret information about processes executed in the target VM. We will focus on the Linux implementation of Kernel Samepage Merging (KSM) memory deduplication feature and on the TPS mechanism implemented by VMware. We describe in detail the functionality of KSM, but the same procedure is implemented by TPS.

KSM is the Linux memory deduplication feature implementation that first appeared in Linux kernel version 2.6.32 [Jon10, KSM]. In this implementation, KSM kernel daemon `ksmd`, scans the user memory for potential pages to be shared among users [AEW09]. Also, since it would be CPU intensive and time consuming, instead of scanning the whole memory continuously, KSM scans only the potential candidates and creates signatures for these pages. These signatures are kept in the deduplication table. When two or more pages with the same signature are found, they are cross-checked completely to determine if they are identical. To create signatures, KSM scans the memory at 20 msec intervals and at best only scans the 25% of the potential memory pages at a time. This is why any memory disclosure attack, including ours, has to wait for a certain time before the deduplication takes effect upon which the attack can be performed. In our case, it usually took around 30 minutes to share up to 32000 pages. During the memory search, KSM analyzes three types of memory pages [SIYA12];

- **Volatile Pages**: Where the contents of the memory change frequently and

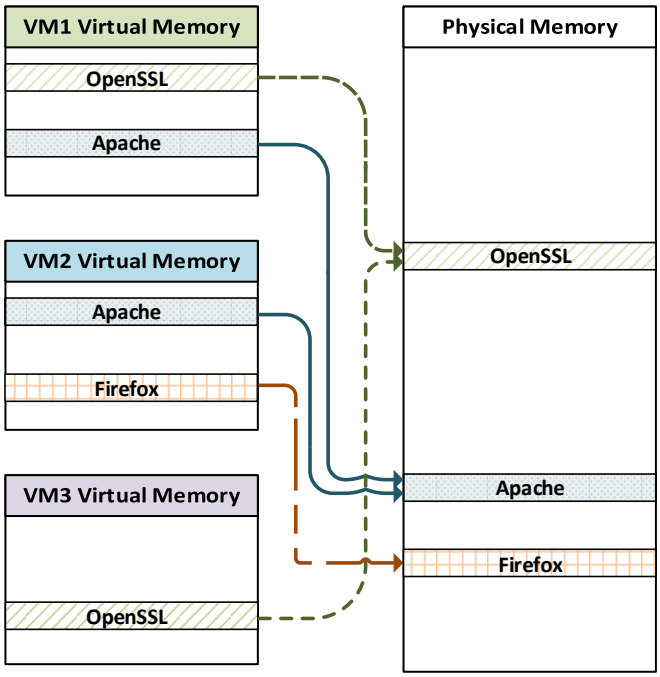


Figure 4.1: Memory Deduplication Feature

should not be considered as a candidate for memory sharing.

- **Unshared Pages**: Candidate pages for deduplication. The *madvise* system call advises to the `ksmd` to be likely candidates for merging.
- **Shared Pages**: Deduplicated pages that are shared between users or processes.

When a duplicate page signature is found among candidates and the contents are cross-checked, `ksmd` automatically tags one of the duplicate pages with copy-on-write (COW) tag and shares it between the processes/users while the other copy is eliminated. Experimental implementations [KSM] show that using this method, it is possible to run over 50 Windows XP VMs with 1GB of RAM each on a physical machine with just 16GB of RAM. As a result of this, the power consumption and system cost is significantly reduced for systems with multiple users.

4.3 *Flush and Reload* Functionality

We described the pre-requisites that the system in which the *Flush and Reload* attack will be applied needs to fullfill. If they are satisfied, and assuming victim and attacker share a memory block b, the *Flush and Reload* attack can be applied performing the following steps:

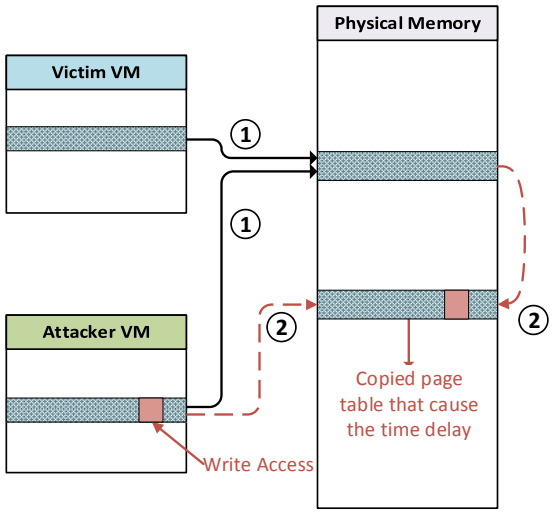


Figure 4.2: Copy-on-Write Scheme

Flushing stage: In this stage, the attacker uses the `clflush` command to flush b from the cache hence making sure that it has to be retrieved from the main memory next time it needs to be accessed. We have to remark here that the `clflush` command does not only flush the memory block from the cache hierarchy of the corresponding working core, but it is flushed from all the caches of all the cores due to the inclusiveness LLC property. This is an important point: if it only flushed the corresponding core's caches, the attack would only work if the attacker and victim's processes were co-residing on the same core. This would have required a much stronger assumption than just being in the same physical machine.

Target accessing stage: In this stage the attacker waits until the target runs a fragment of code, which might use the memory block b that has been flushed in the first stage.

Reloading stage: In this stage the attacker reloads again the previously flushed memory block b and measures the time it takes to reload. We perform this with the popular (and userspace accessible) `rdtsc` instruction, which counts the hardware cycles spent doing a process. Before reading the cycle counter we issue memory barrier instructions (`mfence and lfence`) to make sure all load and store operations have finished before reading the memory block b. Depending on the reloading time, the attacker decides whether the victim accessed the memory block (in which case the memory block would be present in the cache) or if the victim did not access the corresponding memory block (in which case the memory block will be fetched from memory).

The timing difference between a cache hit and a cache miss makes the aforemen-

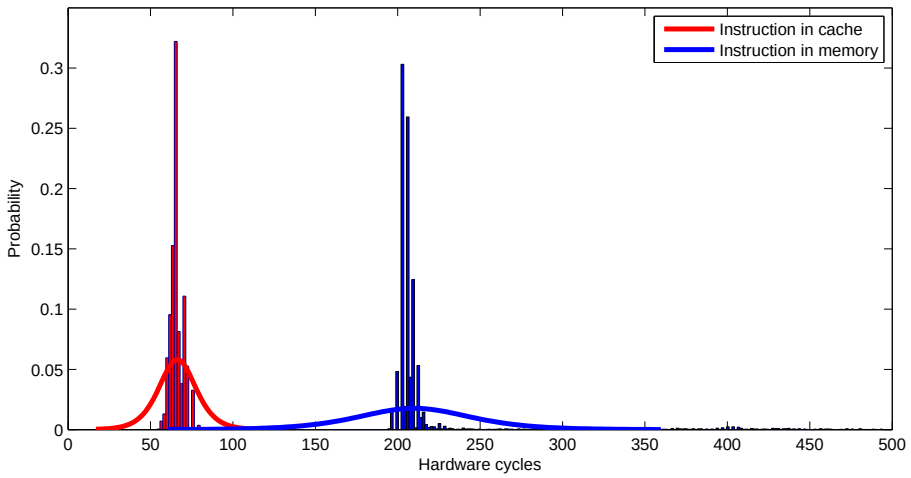


Figure 4.3: Reload time in hardware cycles when a co-located VM uses the memory block b (red, LLC accesses) and when it does not use the targeted memory block b (blue, memory accesses) using KVM on an Intel XEON 2670

tioned access easily detectable by the attacker. In fact this is one of the big advantages of targeting the LLC as a covert channel. Figure 4.3 shows the reload times of a memory block being retrieved from the LLC and from the memory, represented as red and blue histograms respectively. We observe that LLC accesses utilizing the *Flush and Reload* technique usually take around 70 cycles, while memory accesses usually take around 200 cycles. Thus, an attacker using the *Flush and Reload* technique can easily distinguish when a victim uses a shared memory block.

***Flush and Reload* memory targets:**

As we explained *Flush and Reload* targets memory blocks that are shared between victim and attacker, and mechanisms like KSM can help this phenomenon happen. However, this also means that *Flush and Reload* can only target very specific memory blocks, as KSM only shares *read-only* memory pages. Functions and data declared globally usually belong to the kind of memory that an attacker can target with *Flush and Reload*. However, dynamically allocated data, as it is modifiable, cannot be targeted by *Flush and Reload*. Thus, an attacker needs to pick the application to attack carefully, taking this consideration into account.

4.4 *Flush and Reload* Attacking AES

AES has been one of the main targets of cache attacks. For instance, Bernstein demonstrated that table entries in different cache lines can have different L1 access times, while Osvik et al. applied the `evict and time` and the *Prime and Probe*

attack to the L1 cache. In this section we will describe the principles of our *Flush and Reload* attack on the C-implementation of AES in `OpenSSL`. In [GBK11] Gullasch et al. described a *Flush and Reload* attack on AES implementation of the `OpenSSL` library. However in this study, we are going to use the *Flush and Reload* method with some modifications that from our point of view, have clear advantages over [GBK11]. We consider two scenarios: the attack as a spy process running in the same OS instance as the victim (as done in [GBK11]), and the attack running as a cross-VM attack in a virtualized environment.

4.4.1 Description of the Attack

As in prior *Flush and Reload* attacks, we assume that the adversary can monitor accesses to a given cache line. However, unlike the attack in [GBK11], this attack

- only requires the monitoring of a *single* memory block; and
- flushing can be done before encryption, reloading after encryption, i.e. the adversary does not need to interfere with or interrupt the attacked process.

More concretely, the Linux kernel features a completely fair scheduler which tries to evenly distribute CPU time to processes. Gullasch et al. [GBK11] exploited Completely Fair Scheduler (CFS) [CFS], by overloading the CPU while a victim AES encryption process is running. They managed to gain control over the CPU and suspend the AES process thereby gaining an opportunity to monitor cache accesses of the victim process. Our attack is agnostic to CFS and does not require time consuming overloading steps to gain access to the cache.

We assume the adversary monitors accesses to a single line of one of the T tables of an AES implementation, preferably a T table that is used in the last round of AES. Without loss of generality, let's assume the adversary monitors the memory block corresponding to the first positions of table T, where T is the lookup table applied to the targeted state byte s_i, where s_i is the i-th byte of the AES state before the last round. Let's also assume that a cache line can hold n T table values, e.g, the first n T table positions for our case. If s_i is equal to one of the indices of the monitored T table entries in the memory block (i.e. $s_i \in \{0, \ldots, n\}$ if the memory block contains the first n T table entries) then the monitored memory block will with very high probability be present in the cache (since it has been accessed by the encryption process). However, if s_i takes different values, the monitored memory block is not loaded in this step. Nevertheless, since each T table is accessed l times (for AES-128 in `OpenSSL`, $l = 40$ per T_j), there is still a probability that the memory block was loaded by any of the other accesses. In both cases, all that happens after the T table lookup is a possible reordering of bytes (due to AES's `Shift_Rows`), followed by the last round key addition. Since the last round key is always the same for s_i, the n values are mapped to n specific and constant ciphertext byte values. This means that for n out of 256 ciphertext values, the monitored memory block

will *always* have been loaded by the AES operation, while for the remaining $256 - n$ values the probability of having been reloaded is smaller. In fact, the probability that the specific T table memory block i has not been accessed by the encryption process is given as:

$$\Pr\left[\text{no access to } T[i]\right] = \left(1 - \frac{t}{256}\right)^{l}$$

Here, l is the number of accesses to the specific T table. For `OpenSSL 1.0.1` AES-128 we have $l = 40$. If we assume that each memory block can hold $t = 16$ entries per cache line, we have $\Pr\left[\text{no access to } T[i]\right] = 7.6\%$. However, if the T-tables start at the middle of a cache line, an attacker can be smart enough to target those memory blocks, for which $t = 8$ and $\Pr\left[\text{no access to } T[i]\right] = 28.6\%$. Therefore it is easily distinguishable whether the memory block is accessed or not. Indeed, this turns out to be the case as confirmed by our experiments.

In order to distinguish the two cases, all that is necessary is to measure the timing for the *reload* of the targeted memory block. If the line was accessed by the AES encryption, the reload is quick; else it takes more time. Based on a threshold that we will empirically choose from our measurements, we expect to distinguish main memory accesses from L3 cache accesses. For each possible value of the ciphertext byte c_i we count how often either case occurs. Now, for n ciphertext values (the ones corresponding to the monitored T table memory block) the memory block has always been reloaded by AES, i.e. the reload counter is (close to) zero. These n ciphertext values are related to the state as follows:

$$c_i = k_i \oplus T\left[s_{[i]}\right] \tag{4.1}$$

where the $s_{[i]}$ can take n consecutive values. Note that Eq. (4.1) describes the last round of AES. The brackets in the index of the state byte $s_{[i]}$ indicate the reordering due to the `Shift_Rows` operation. For the other values of c_i, the reload counter is significantly higher. Given the n values of c_i with a low reload counter, we can solve Eq. (4.1) for the key byte k_i, since the indices $s_{[i]}$ as well as the table output values $T\left[s_{[i]}\right]$ are known for the monitored memory block. In fact, we get n possible key candidates for each c_i with a zero reload counter. The correct key is the only one that all n valid values for c_i have in common.

A general description of the key recovery algorithm is given in Algorithm 4, where key byte number 0 is recovered from the ciphertext values corresponding to n low reload counter values that were recovered from the measurements. Again, n is the number of T table positions that a cache line holds. The *reload vector* $X_i = [x(0), x(1), \ldots, x(255)]$ holds the reload counter values $x(j)$ for each ciphertext value $c_i = j$. Finally K_0 is the vector that, for each key byte candidate k, tracks the number of appearances in the key recovery step.

Algorithm 4 Recovery algorithm for key byte k_0

```
Input  : X_0                    //Reload vector for ciphertext byte 0
Output: k_0                     //Correct key byte 0
forall x_j ∈ X_0 do
    //Threshold for values with low reload counter.
    if x_j < Low_counter_threshold then
        for s = 0 to n do
            //xor with each value of the targeted T table memory
              block
            K_0[j ⊕ T[s]]++;
        end
    end
end
return argmax_k(K_0[k]);
```

Example

Assume that the cache linee can hold $n = 4$ T table values and we want to recover key byte k_0. There are four ciphertext values detected with a low reload counter. Assume further that each c_0 has been xored with the T table values of the monitored memory block (the first 4 if we are working with the first positions), giving $k_0^{(i)} = c_0^i \oplus T\left[s_{[0]}\right]$. For each of the four possibilities of c_0, there are $n = 4$ possible solutions for k_0. If the results are the following:

$$k_0^{(0)}\begin{cases}43\\ ba\\ 91\\ 17\end{cases} k_0^{(1)}\begin{cases}8b\\ 91\\ f3\\ 66\end{cases} k_0^{(2)}\begin{cases}91\\ 45\\ 22\\ af\end{cases} k_0^{(3)}\begin{cases}cd\\ 02\\ 51\\ 91\end{cases}$$

And since there is only one common solution between all of them, which is 91, we deduce that the correct key value is $k_0 = 91$. This also means that $K_0[91] = 4$, since $k = 91$ appeared four times as possible key candidate in the key recovery step.

Note that this is a *generic attack* that would apply virtually to any table-based block cipher implementation. That is, our attack can easily be adapted to other block ciphers as long as their last round consists of a table look-up with a subsequent key addition.

4.4.2 Recovering the Full Key

To recover the full key, the attack is expanded to all tables used in the last round, e.g. the 4 T tables of AES in `OpenSSL 1.0.1`. For each ciphertext byte it is known which T table is used in the final round of the encryption. This means that

Algorithm 5 *Flush and reload* algorithm extended to 16 ciphertext bytes

```
Input  : T0_0, T1_0, T2_0, T3_0          //Addresses of each T table
Output: X_0, X_1, ...X_15                //Reload vectors for ciphertexts
                                         //Each X_k holds 256 counter values
while iteration < total number of measurements do
    clflush(T0_0, T1_0, T2_0, T3_0);     //Flush data to the main memory
    ciphertext=Encryption(plaintext);    //No need to store plaintext!
    for i ← T0_0 to T3_0 do
        time=Reload(i);
        if time > AccessThreshold then
            Addcounter(T_i,X_i);         //Increase counter of X_i using T_i
        end
    end
end
return X_0, X_1, ..., X_15
```

the above attack can be repeated on each byte, by simply analyzing the collecting ciphertexts and their timings for each of the ciphertext bytes individually. As before, the timings are profiled according to the value that each ciphertext byte c_i takes in each of the encryptions, and are stored in a ciphertext byte vector. The attack process is described in Algorithm 5. In a nutshell, the algorithm monitors the first T table memory block of all used tables and hence stores four reload values per observed ciphertext. Note that, this is a *known ciphertext attack* and therefore all that is needed is a flush of one memory block before one encryption. There is no need for the attacker to gain access to plaintexts.

Finally the attacker should apply Algorithm 4 to each of the obtained ciphertext reload vectors. Recall that each ciphertext reload vector uses a different T table, so the right corresponding T table should be applied in the key recovery algorithm.

Performing the Attack. In the following we provide the details about the process followed during the attack.

Step 1: Acquire information about the offset of T tables The attacker has to know the offset of the T tables with respect to the beginning of the library. With that information, the attacker can refer and point to any memory block that holds T table values even when the ASLR is activated. This means that some reverse engineering work has to be done prior to the attack. This can be done in a debugging step where the offset of the addresses of the four T tables are recovered.

Step 2: Collect Measurements In this step, the attacker requests encryptions and applies *Flush and Reload* between each encryption. The information

gained, i.e. Ti_0 was accessed or not, is stored together with the observed ciphertext. The attacker needs to observe several encryptions to get rid of the noise and to be able to recover the key. Note that, while the reload step must be performed and timed by the attacker, the flush might be performed by other processes running in the victim OS.

Step 3: Key recovery In this final step, the attacker uses the collected measurements and his knowledge about the public T tables to recover the key. From this information, the attacker applies the steps detailed in Section 7.4.2.1 to recover the individual bytes of the key.

4.4.3 Attack Scenario 1: Spy Process

In this first scenario we will attack an encryption server running in the same OS as the spy process. The encryption server just receives encryption requests, encrypts a plaintext and sends the ciphertext back to the client. The server and the client are running on different cores. Thus, the attack consists in distinguishing accesses from the LLC, i.e. L3 cache, which is shared across cores. and the main memory. Clearly, if the attacker is able to distinguish accesses between LLC and main memory, it will be able to distinguish between L1 and main memory accesses whenever server and client co-reside in the same core. In this scenario, both the attacker and victim are using the same shared library. KSM is responsible for merging those pages into one unified shared page. Therefore, the victim and attacker processes are linked through the KSM deduplication feature.

Our attack works as described in the previous section. First the attacker discovers the offset of the addresses of the T tables with respect to the begining of the library. Next, it issues encryption requests to the server, and receives the corresponding ciphertext. After each encryption, the attacker checks with the *Flush and Reload* technique whether the chosen T table values have been accessed. Once enough measurements have been acquired, the key recovery step is performed. As we will see in our results section, the whole process takes less than half a minute.

Our attack significantly improves on previous cache side channel attacks such as *evict + time* or *prime and probe* [OST06]. Both attacks were based on spy processes targeting the L1 cache. A clear advantage of our attack is that —since it is targeting the last shared level cache— it works across cores.

A more realistic attack scenario was proposed earlier by Bernstein [Ber04] where the attacker targets an encryption server. Our attack similarly works under a realistic scenario. However, unlike Bernstein's attack [Ber04], our attack does not require a profiling phase that involves access to an identical implementation with a known-key. Finally, with respect to the previous *Flush and Reload* attack in AES, our attack does not need to interrupt the AES execution of the encryption server. We will compare different attacks according to the number of encryptions needed in Section 4.4.6.

4.4.4 Attack Scenario 2: Cross-VM Attack

In our second scenario the victim process is running in one virtual machine and the attacker in another one but on the physical server, possibly on different cores. For the purposes of this study it is assumed that the co-location problem has been solved using the methods proposed in [RTSS09]. The attack exploits memory overcommitment features that some hypervisors such as VMware provide. In particular, we focus in memory deduplication. The hypervisor will search periodically for identical pages across VMs to merge both pages into a single page in the memory. Once this is done (without the intervention of the attacker) both the victim and the attacker will access the same portion of the physical memory enabling the attack. The attack process is the same as in Scenario 1. Moreover, we later show that the key is recovered in less than a minute, which makes the attack *practical*.

We discussed the improvements of our attack over previous proposals in the previous scenario except the most important one: We believe that the `evict+time`, `prime and probe` and time collision attacks will be rather difficult to carry out in real cloud environment. The first two, as we know them so far, are targeting the L1 cache, which is not shared across cores. The attacker would have to be in the same core as the victim, which is a much stronger assumption than being just in the same physical machine. Finally, targeting the CFS [GBK11] to evict the victim process, requires for the attacker's code to run in the same OS, which will certainly not be possible in a virtualized environment.

4.4.5 Experiment Setup and Results

We present results for both a spy process within the native machine as well as the cross-VM scenario. The target process is executed in Ubuntu 12.04 64 bits, kernel version 3.4, using the C-implementation of AES in `OpenSSL 1.0.1f` for encryption. This is used when `OpenSSL` is configured with `no-asm` and `no-hw` option. We want to remark that this *is not* the default option in the installation of `OpenSSL` in most of the products. All experiments were performed on a machine featuring an Intel i5-3320M four core clocked at 3.2GHz. The Core i5 has a three-level cache architecture: The L1 cache is 8-way associative, with 2^{15} bytes of size and a cache line size of 64 bytes. The level-2 cache is 8-way associative as well, with a cache line width of 64 bytes and a total size of 2^{18} bytes. The level-3 cache is 12-way associative with a total size of 2^{22} bytes and 64 bytes cache line size. It is important to note hypervisor each core has private L1 and L2 caches, but the L3 cache is shared among all cores. Together with the deduplication performed by the VMM, the shared L3 cache allows the adversary to learn about data accesses by the victim process.

The *attack scenario* is as follows: the victim process is an encryption server handling encryption requests through a socket connection and sends back the ciphertext, similar to Bernstein's setup in [Ber04]. But unlike Bernstein's attack, where packages of at least 400 bytes were sent to deal with the noise, our server

only receives packages of 16 bytes (the plaintext). The encryption key used by the the server is unknown to the attacker. The attack process sends encryption queries to the victim process. All measurements such as timing measurements of the reload step are done on the attacker side. In our setup, each cache line holds 16 T table values, which results in a 7.6% probability for not accessing a memory block per encryption. All given attack results target only the first cache line of each T table, i.e. the first 16 values of each T table for *Flush and Reload*. Note that in the attack any memory block of the T table would work equally well. Both native and cross-VM attacks establish the threshold for selecting the correct ciphertext candidates for the working T table line by selecting those values which are below half of the average of overall timings for each ciphertext value. This is an empirical threshold that we set up after running some experiments as follows

$$\text{threshold} = \sum_{i=0}^{256} \frac{t_i}{2 \cdot 256} \ .$$

Spy process attack setup:

The attack process runs in the same OS as the victim process. The communication between the processes is carried out via localhost connection and measures timing using Read Time-Stamp Counters (`rdtsc`). The attack is set up to work across cores; the encryption server is running in a different core than the attacker. We believe that distinguishing between L3 and main memory accesses will be more susceptible to noise than distinguishing between L1 cache accesses and main memory accesses. Therefore while working with the L3 cache gives us a more realistic setting, it also makes the attack more challenging.

Cross-VM attack setup:

In this attack, we use VMware ESXI 5.5.0 build number 1623387 running Ubuntu 12.04 64-bits guest OSs. We know that VMware implements TPS with large pages (2 MB) or small pages (4 KB). We decided to use the latter one, since it seems to be the default for most systems. Furthermore, as stated in [VMWb], even if the large page sharing is selected, the VMM will still look for identical small pages to share. For the attack we used two virtual machines, one for the victim and one for the attacker. The communication between them is carried out over the local IP connection.

The results are presented in Figure 4.4 which plots the number of correctly recovered key bytes over the number of timed encryptions. The dash-dotted line shows that the spy-process scenario completely recovers the key after only 2^{17} encryptions. Prior to moving to the cross-VM scenario, a single VM scenario was performed to gauge the impact of using VMs. The dotted line shows that due to the noise introduced by virtualization we need to nearly double the number of encryptions to

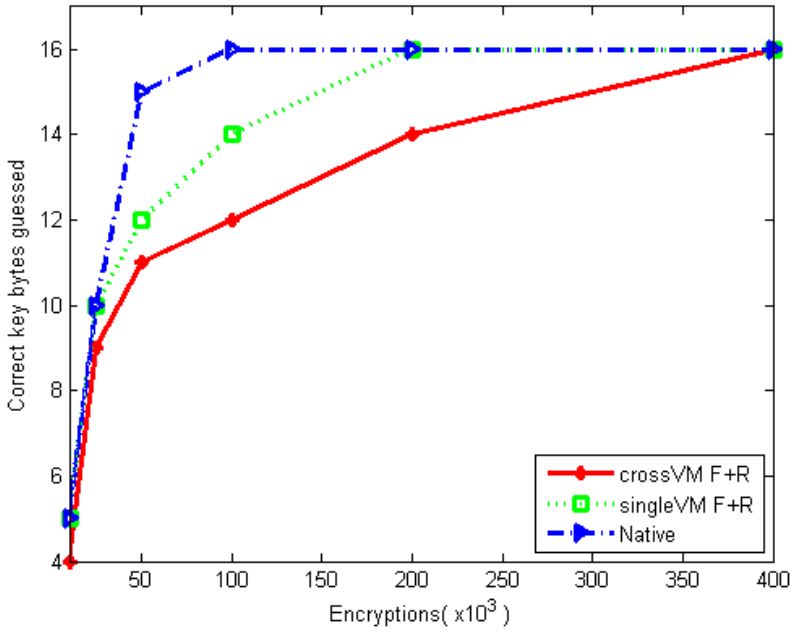


Figure 4.4: Number of correct key bytes guessed of the AES-128 bit key vs. number of encryption requests. Even 50,000 encryptions (i.e. less than 5 seconds of interaction) result in significant security degradation in both the native machine as well as the cross-VM attack scenario.

match the key recovery performance of the native case. The solid line gives the result for the cross-VM attack: 2^{19} observations are sufficient for stable full key recovery. The difference might be due to cpuid like instructions which are emulated by the hypervisor, therefore introducing more noise to the attack. In the worst case, both the native spy process and the single VM attack took around 25 seconds (for 400.000 encryptions). We believe that this is due to communication via the localhost connection. However, when we perform a cross-VM attack, it takes roughly twice as much time as in the previous cases. In this case we are performing the communication via local IPs that have to reach the router, which is believed to add the additional delay. This means that *all of the described attacks —even in the cross VM scenario— completely recover the key in less than one minute*!

4.4.6 Comparison to Other Attacks

Next we compare the most commonly implemented cache-based side channel attacks to the proposed attack. Results are shown in Table 4.1. It is difficult to compare the attacks, since most of them have been run on different platforms. Many of the prior attacks target `OpenSSL 0.9.8` version of AES. Most of these attacks exploit the fact that AES has a separate T Table for the last round, significantly reducing

Table 4.1: Comparison of cache side channel attack techniques against AES

Attack	Platform	Methodology	`OpenSSL`	Traces
Spy-Process based Attacks:				
Collision timing [BM06]	Pentium 4E	Time measurement	`0.9.8a`	300.000
Prime+probe [OST06]	Pentium 4E	L1 cache prime-probing	`0.9.8a`	16.000
Evict+time [OST06]	Athlon 64	L1 cache evicting	`0.9.8a`	500.000
Flush+Reload (CFS) [GBK11]	Pentium M	*Flush+reload* w/CFS	`0.9.8m`	100
Flush+Reload [IIES14b]	i5-3320M	L3 cache *Flush+reload*	`0.9.8a`	8.000
Bernstein [AE13]	Core2Duo	Time measurement	`1.0.1c`	2^{22}
Flush+Reload [IIES14b]	i5-3320M	L3 cache *Flush+reload*	`1.0.1f`	100.000
Cross-VM Attacks:				
Bernstein [IIES14a][1]	i5-3320M	Time measurement	`1.0.1f`	2^{30}
Our attack(VMware)	i5-3320M	L3 cache *Flush and reload*	`1.0.1f`[2]	400.000

[1] Only parts of the key were recovered, not the whole key.
[2] The AES implementation was not updated for the recently released `OpenSSL` 1.0.1g and 1.0.2 beta versions. So the results for those libraries are identical.

the noise introduced by cache miss accesses. Hence, attacks on `OpenSSL 0.9.8` AES usually succeed much faster, a trend confirmed by our attack results. Note that our attack, together with [AE13] and [IIES14a] are the only ones that have been run on a 64 bit processor. Moreover, we assume that due to undocumented internal states and advanced features such as hardware prefetchers, implementation on a 64 bit processor will add more noise than older platforms running the attack. With respect to the number of encryptions, we observe that the proposed attack has significant improvements over most of the previous attacks.

Spy process in native OS:

Even though our attack runs in a noisier environment than Bernstein's attack, *Evict and Time*, and cache timing collision attacks, it shows better performance. Only *Prime and Probe* and *Flush and Reload* using CFS show either comparable or better performance. The proposed attack has better performance than *Prime and Probe* even though their measurements were performed with the attack and the encryption being run as one unique process. The *Flush and Reload* attack in [GBK11] exploits a much stronger leakage, which requires that attacker **to interrupt the target AES between rounds** (an unrealistic assumption). Furthermore, *Flush and Reload* with CFS needs to monitor the entire T tables, while our attack only needs to monitor a single line of the cache, making the attack much more lightweight and subtle.

Cross-VM attack:

So far there is only one publication that has analyzed cache-based leakage across VMs for AES [IIES14a]. Our attack shows dramatic improvements over [IIES14a], which needs 2^{29} encryptions (hours of run time) for a partial recovery of the key. Our attack only needs 2^{19} encryptions to recover the full key. Thus, while the attack

presented in [IIES14a] needs to interact with the target for several hours, our attack succeeds in under a minute and recovers the entire key. Note that the CFS enabled *Flush and Reload* attack in [GBK11] will not work in the cross-VM setting, since the attacker has no control over victim OS's CFS.

4.5 *Flush and Reload* Attacking Transport Layer Security: Re-viving the Lucky 13 Attack

Although cache attacks are usually applied to cryptographic algorithms, virtually any security critical software that has non-constant execution flow can be targeted by the *Flush and Reload* attack. In this section, we show an example of such an application. In particular, we show for the first time that cache attacks can be utilized to attack security protocols like the Transport Layer Security (TLS) protocol, by re-implementing the Lucky 13 attack that was believed to be closed by the security community.

The Lucky 13 attack targets a vulnerability in the TLS (and DTLS) protocol design. The vulnerability is due to MAC-then-encrypt mode, in combination with the padding of the CBC encryption, also referred to as MEE-TLS-CBC. In the following, our description focuses on this popular mode. Vaudenay [Vau02] showed how the CBC padding can be exploited for a message recovery attack. AlFardan et al. [FP13] showed—more than 10 years later—that the subsequent MAC verification introduces timing behavior that makes the message recovery attack feasible in practical settings. In fact, their work includes a comprehensive study of the vulnerability of several TLS libraries. In this section we give a brief description of the attack. For a more detailed description, please refer to the original paper [FP13].

4.5.1 The TLS Record Protocol

The TLS record protocol provides encryption and message authentication for bulk data transmitted in TLS. The basic operation of the protocol is depicted in Figure 4.5. When a payload is sent, a sequence number and a header are attached to it and a MAC tag is generated by any of the available HMAC choices. Once the MAC tag is generated, it is appended to the payload together with a padding. The payload, tag, and pad are then encrypted using a block cipher in CBC mode. The final message is formed by the encrypted ciphertext plus the header.

Upon receiving an encrypted packet, the receiver decrypts the ciphertext with the session key that was negotiated in the handshake process. Next, the padding and the MAC tag need to be removed. For this, first the receiver checks whether the size of the ciphertext is a multiple of the block size and makes sure that the ciphertext can accommodate minimally a zero-length record, a MAC tag, and at least one byte of padding. After decryption, the receiver checks if the recovered padding matches one of the allowed patterns. A standard way to implement this

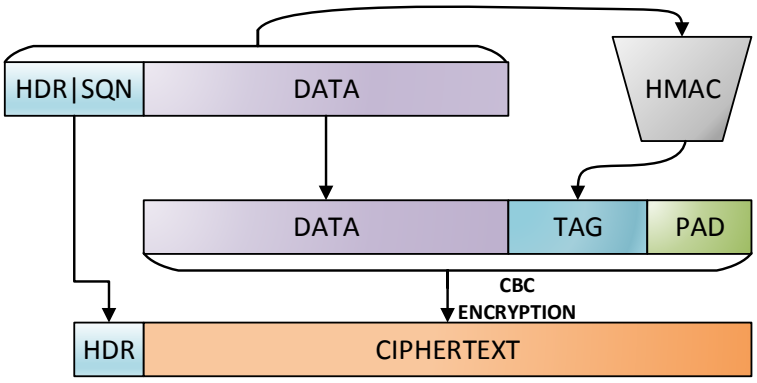


Figure 4.5: Encryption and authentication in the TLS record protocol when using HMAC and a block cipher in CBC mode.

decoding step is to check the last byte of the plaintext, and to use it to determine how many of the trailing bytes belong to the padding. Once the padding is removed, and the plain payload is recovered, the receiver attaches the header and the sequence number and performs the HMAC operation. Finally, the computed tag is compared to the received tag. If they are equal, the contents of the message are concluded to be securely transmitted.

4.5.2 HMAC

The TLS record protocol uses the HMAC algorithm to compute the tag. The HMAC algorithm is based on a hash function H that performs the following operations:

$$\mathsf{HMAC}(K, m) = H((K \oplus opad)||H((K \oplus ipad)||M)$$

Common choices in TLS 1.2 for H are SHA-1, SHA-256 and the now defunct MD5. The message M is padded with a single 1 bit followed by zeros and an 8 byte length field. The pad aligns the data to a multiple of 64 bytes. $K \oplus opad$ already forms a 64 byte field, as well as $K \oplus ipad$. Therefore, the minimum number of compression function calls for a HMAC operation is 4. This means that depending on the number of bytes of the message, the HMAC operation is going to take more or less compression functions. To illustrate this, we are repeating the example given in [FP13] as follows. Assume that the plaintext size is 55 bytes. In this case an 8 byte length field is appended together with a padding of size 1, so that the total size is 64 bytes. Here in total the HMAC operation is going to take four compression function calls. However if the plaintext size is 58, an 8 byte length field is attached and 62 bytes of padding are appended to make the total size equal to 128 bytes. In this case, the total compression function calls are going to be equal to five. Distinguishing the

number of performed compression function calls is the basic idea that enables the Lucky 13 attack.

4.5.3 CBC Encryption & Padding

Until the support of the Galois Counter Mode in TLS 1.2, block ciphers were always used in cipher block chaining (CBC) mode in TLS. The decryption of each block of a ciphertext C_i is performed as follows:

$$P_i = D_k(C_i) \oplus C_{i-1}$$

Here, P_i is the plaintext block and $D_k(\cdot)$ is the decryption under key k. For the prevalent AES, the block size is 16 bytes. The size of the message to be encrypted in CBC mode has to be indeed a multiple of the cipher block size. The TLS protocol specifies a padding as follows: the last padding byte indicates the length of the padding; the value of the remaining padding bytes is equal to the number of padding bytes needed. This means that if 3 bytes of padding is needed, the correct padding has to be 0x02|0x02|0x02. Possible TLS paddings are: 0x00, 0x01|0x01, 0x02|0x02|0x02, up to 0xff—0xff| . . . |0xff. Note that there are several valid paddings for each message length.

4.5.4 An Attack On CBC Encryption

We now discuss the basics of the Lucky 13 attack. For the purposes of this study the target cipher is AES in CBC mode, as described above. Again, we use the same example that AlFardan et al. gave in [FP13]. Assume that the sender is sending 4 non-IV blocks of 16 bytes each, one IV block, and the header number. Let's further assume that we are using SHA-1 to compute the MAC tag, in which case the digest size is 20 bytes. The header has a fixed length of 5 bytes and the sequence number would have a total size of 8 bytes. The payload would look like this:

$$\mathsf{HDR}|C_{IV}|C_1|C_2|C_3|C_4$$

Now assume that the attacker masks Δ in C_3. The decryption of C_4 is going to be as follows:

$$P_4^* = D_k(C_4) \oplus C_3 \oplus \Delta = P_4 \oplus \Delta$$

Focusing on the last two bytes $P^*_{4(14)}|P^*_{4(15)}$ three possible scenarios emerge:

Invalid padding

This is the most probable case, where the plaintext ends with an invalid padding. Therefore, according to TLS protocol, this is treated as 0 padding. 20 bytes of MAC (SHA-1) are removed and the corresponding HMAC operation in the client side is

performed on 44 bytes +13 bytes of header, in total 57 bytes. Therefore the HMAC evaluates 5 compression function calls.

Valid 0x00 padding

If $P^*_{4(15)}$ is 0x00, this is considered as valid padding, and a single byte of padding is removed. Then the 20 bytes of digest are removed, and the HMAC operation in client side is done in 43+13 bytes, 56 in total, which takes 5 compression function calls.

Any other valid padding

For instance, if we consider a valid padding of two bytes, the valid padding would be 0x01|0x01 and 2 bytes of padding are removed. Then 20 bytes of digest are removed, and the HMAC operation is performed over $42 + 13 = 55$ bytes, which means four compression function calls.

The Lucky 13 attack is based on detecting this difference between 4 and 5 compression function calls. Recall that if an attacker knows that a valid 0x01|0x01 padding was achieved, she can directly recover the last two bytes of P_4, since

$$0x01|0x01 = P_{4(14)}|P_{4(15)} \oplus \Delta_{(14)}|\Delta_{(15)}$$

Note that the successful padding $0x01|0x01$ is more likely to be achieved than any other longer valid padding, and therefore, the attacker can be confident enough that it is the padding that was forged. Furthermore, she can keep on trying to recover the remaining bytes once she knows the first 2 bytes. The attacker needs to perform at most 2^{16} trials for detecting the last two bytes, and then up to 2^8 messages for each of the bytes that she wants to recover.

4.5.5 Analysis of Lucky 13 Patches

The Lucky 13 attack triggered a series of patches for all major implementations of TLS [FP13]. In essence, all libraries were fixed to remove the timing side channel exploited by Lucky 13, i.e. implementations were updated to handle different CBC-paddings in constant time. However, different libraries used different approaches to achieve this:

- Some libraries implement *dummy functions* or processes,
- Others use *dummy data* to process the maximum allowed padding length in each MAC checking.

In the following, we discuss these different approaches for some of the most popular TLS libraries.

4.5.6 Patches Immune to *Flush and Reload*

In this section we will analyze those libraries that are secure against the *Flush and Reload* technique.

- **OpenSSL:** The Lucky 13 vulnerability was fixed in OpenSSL versions 1.0.1, 1.0.0k, and 0.9.8y by February 2013 without the use of a time consuming dummy function and by using dummy data. Basically, when a packet is received, the padding variation is considered and the maximum number of HMAC compression function evaluations needed to equalize the time is calculated. Then each compression function is computed directly, without calling any external function. For every message, the maximum number of compression functions are executed, so that no information is leaked through the time channel in case of the incorrect padding. Furthermore, the OpenSSL patch removed any data dependent branches ensuring a fixed data independent execution flow. This is a generic solution for microarchitectural leakage related attacks, i.e. cache timing or even branch prediction attacks.

- **Mozilla NSS:** This library is patched against the Lucky 13 attack in version 3.14.3 by using a constant time HMAC processing implementation. This implementation follows the approach of OpenSSL, calculating the number of maximum compression functions needed for a specific message and then computing the compression functions directly. This provides not only a countermeasure for both timing and cache access attacks, but also for branch prediction attacks.

- **MatrixSSL:** MatrixSSL is fixed against the Lucky 13 with the release of version 3.4.1 by adding timing countermeasures that reduce the effectiveness of the attack. In the fix, the library authors implemented a decoding scheme that does a sanity check on the largest possible block size. In this scheme, when the received message's padding length is incorrect, Matrix SSL runs a loop as if there was a full 256 bytes of padding. When there are no padding errors, the same operations are executed as in the case of an incorrect padding to sustain a constant time. Since there are no functions that are specifically called in the successful or unsuccessful padding cases, this library is not vulnerable to our *Flush and Reload* attack. In addition, Matrix SSL keeps track of all errors in the padding decoding and does the MAC checking regardless of valid or invalid padding rather than interrupting and finalizing the decoding process at the first error.

4.5.7 Patches Vulnerable to *Flush and Reload*

There are some patches that ensure constant time execution and therefore are immune to the original Lucky 13 attack [FP13] which are vulnerable to *Flush and*

Reload. This implies a dummy function call or a different function call tree for valid and invalid paddings. Furthermore, if these calls are preceded by branch predictions, these patches might also be exploitable by branch prediction attacks. Some examples including code snippets are given below.

- **GnuTLS:** uses a `dummy_wait` function that performs an extra compression function whenever the padding is incorrect. This function makes the response time constant to fix the original Lucky 13 vulnerability. Since this function is only called in the case of incorrect padding, it can be detected by a co-located VM running a *Flush and Reload* attack.

```
if (memcmp (tag, &ciphertext->data[length],
tag_size) != 0 || pad_failed != 0)
 //HMAC was not the same.
{dummy_wait(params, compressed,pad_failed,
 pad,length+preamble_size);}
```

- **PolarSSL:** uses a dummy function called `md_process` to sustain constant time to fix the original Lucky 13 vulnerability. Basically the number of extra runs for a specific message is computed and added by `md_process`. Whenever this dummy function is called, a co-located adversary can learn that the last padding was incorrect and use this information to realize the Lucky 13 attack.

```
for( j = 0; j < extra_run; j++ )
 \\We need an extra run
md_process( &ssl->transform_in->
md_ctx_dec,ssl->in_msg );]*
```

- **CyaSSL:** was fixed against the Lucky 13 with the release of 2.5.0 on the same day the Lucky 13 vulnerability became public. In the fix, CyaSSL implements a timing resistant pad/verify check function called `TimingPadVerify` which uses the `Padcheck` function with dummy data for all padding length cases whether or not the padding length is correct. CyaSSL also does all the calculations such as the HMAC calculation for the *incorrect* padding cases which not only fixes the original Lucky 13 vulnerability but also prevents the detection of incorrect padding cases. This is due to the fact that the `Padcheck` function is called for both correctly and incorrectly padded messages which makes it impossible to detect with our *Flush and Reload* attack.

  However, for the correctly padded messages, CyaSSL calls the `CompressRounds` function which is detectable with *Flush and Reload* .Therefore, we monitor the correct padding instead of the incorrect padding cases.

  Correct padding case:

```
PadCheck(dummy, (byte)padLen,
MAX_PAD_SIZE - padLen - 1);
ret = ssl->hmac(ssl, verify, input,
pLen - padLen - 1 - t, content, 1);
CompressRounds(ssl, GetRounds(pLen,
padLen, t), dummy);
ConstantCompare(verify, input +
(pLen - padLen - 1 - t), t) != 0)
```

Incorrect padding case:

```
CYASSL_MSG("PadCheck failed");
PadCheck(dummy, (byte)padLen,
MAX_PAD_SIZE - padLen - 1);
ssl->hmac(ssl, verify, input,
pLen - t, content, 1);
 //still compare
ConstantCompare(verify, input +
pLen - t, t);
```

4.5.8 Reviving Lucky 13 on the Cloud

As the cross-network timing side channel has been closed (c.f. Section 4.5.5), the Lucky 13 attack as originally proposed no longer works on the recent releases of most cryptographic libraries. In this work, we revive the Lucky 13 attack to target some of these (fixed) releases by gaining information through co-located VMs (a leakage channel not considered in the original paper) rather than the network timing exploited in the original attack.

4.5.8.1 Regaining the Timing Channel

Most cryptographic libraries and implementations have been largely fixed to yield an *almost* constant time when the MAC processing time is measured over the network. As discussed in Section 4.5.5, although there are some similarities in these patches, there are also subtle differences which—as we shall see—have significant implications on security. Some of the libraries not only closed the timing channel but also various cache access channels. In contrast, other libraries left an open door to implement access driven cache attacks on the protocol. In this section we analyze how an attacker can gain information about the number of compression functions performed during the HMAC operation by making use of leakages due to shared memory hierarchy in VMs located on the same machine. This is sufficient to re-implement the Lucky 13 attack.

More precisely, during MAC processing depending on whether the actual MAC check terminates early or not, some libraries call a dummy function to equalize the processing time. Knowing if this dummy function is called or not reveals whether the received packet was processed as to either having a invalid padding, zero length

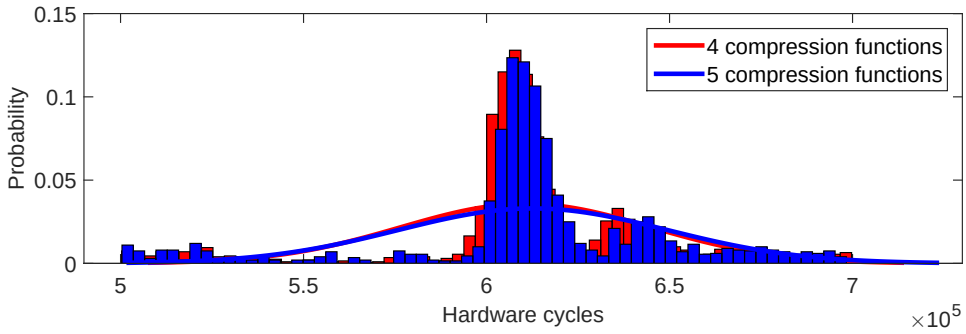


Figure 4.6: Histogram of network time measured for sent packages with valid (4 compression functions) and invalid (5 compression functions) paddings.

padding or any other valid padding. In general, any difference in the execution flow between handling a well padded message, a zero padded message or an invalid padded message enables the Lucky 13 attack. This information is gained by the *Flush and Reload* technique if the cloud system enables deduplication features.

To validate this idea, we ran two experiments:

- In the first experiment we generated encrypted packets using PolarSSL client with valid and invalid paddings and measured the network time as shown in Figure 4.6. Note that, the network time in the two distributions obtained for valid and invalid paddings are essentially indistinguishable as intended by the patches.

- In the second experiment we see a completely different picture. Using PolarSSL we generated encrypted packets with valid and invalid paddings which were then sent to a PolarSSL server. Here instead, we measured the time it takes to load a specifically chosen PolarSSL library function running inside a co-located VM. Figure 4.7 shows the probability distributions for a function reloaded from L3 cache vs. a function reloaded from the main memory. The two distributions are clearly distinguishable and the misidentification rate (the area under the overlapping tails in the middle of the two distributions) is very small. Note that, this substitute timing channel provides much more precise timing that the network time. To see this more clearly, we refer the reader to Figure 2 in [FP13] where the network time is measured to obtain two overlapping Gaussians by measurements with OpenSSL encrypted traffic. This is not a surprise, since the network channel is significantly more noisy.

In conclusion, we regain a much more precise timing channel, by exploiting the discrepancy between L3 cache and memory accesses as measured by a co-located attacker. In what follows, we more concretely define the attack scenario, and then precisely define the steps of the new attack.

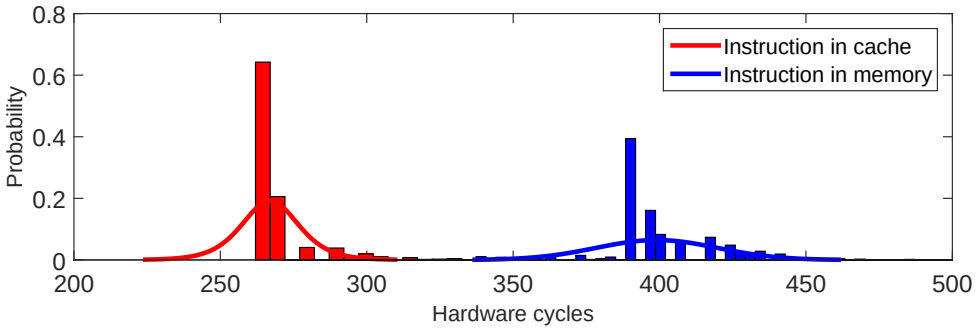


Figure 4.7: Histogram of access time measured for function calls from the L3 cache vs. a function called from the main memory.

4.5.8.2 New Attack Scenario

In our attack scenario, the side channel information will be gained by monitoring the cache in a co-located VM. In the same way as in [FP13] we assume that the adversary captures, modifies, and replaces any message sent to the victim. However, TLS sessions work in such a way that when the protocol fails to decrypt a message, the session is closed. This is the reason why we focus in multi-session attacks where the same plaintext in the same place is being sent to the victim e.g. an encrypted password sent during user authentication.

The fact that we are working with a different method in a different scenario gives us some advantages and disadvantages over the previous Lucky 13 work:

Advantages:

- Recent patches in cryptographic libraries mitigate the old Lucky 13 attack, but are still vulnerable in the new scenario.
- In the new scenario, no response from the server is needed. The old Lucky 13 attack needed a response to measure the time, which yielded a noisier environment in TLS than DTLS.
- The new attack does not suffer from the network channel noise. This source of noise was painful for the measurements as we can see in the original paper, where in case of TLS as many as 2^{14} trials were necessary to guess a single byte value.

Disadvantages:

- Assumption of co-location: To target a specific victim, the attacker has to be co-located with that target. However the attacker could just reside in a

physical machine and just wait for some potential random victim running a TLS operation.

- Other sources of noise: The attacker no longer has to deal with network channel noise, but still has to deal with other microarchitectural sources of noise, such as instruction prefetching. This new source of noise is translated in more traces needed, but as we will see, much less than in the original Lucky 13 attack. In Section 7.5 we explain how to deal with this new noise.

4.5.8.3 Attack Description

In this section we describe how an attacker uses *Flush and Reload* technique to gain access to information about the plaintext that is being sent to the victim.

- **Step 1 Function identification:** Identify different function calls in the TLS record decryption process to gain knowledge about suitable target functions for the spy process. The attacker can either calculate the offset of the function she is trying to monitor in the library, and then add the corresponding offset when the Address Space Layout Randomization (ASLR) moves her user address space. Another option is to disable the ASLR in the attackers VM, and use directly the virtual address corresponding to the function she is monitoring.

- **Step 2 Capture packet, mask and replace:** The attacker captures the packet that is being sent and masks it in those positions that are useful for the attack. Then she sends the modified packet to the victim.

- **Step 3 Flush targeted function from cache:** The *Flush and Reload* process starts after the attacker replaces the original version of the packet and sends it. The co-located VM flushes the function to ensure that no one but the victim ran the targeted function. Any subsequent execution of the targeted function will bear a faster reload time during the reload process.

- **Step 4 Reload target function & measure:** Reload the corresponding function memory line again and measure the reload time. According to a threshold that we set based on experimental measurements, we decide whether the dummy function was loaded from the cache (implying that the victim has executed the dummy function earlier) or was loaded from the main memory (implying the opposite).

Since the attacker has to deal with instruction prefetching, she will be constantly running *Flush and Reload* for a specified period of time. The attacker therefore distinguishes between functions preloaded and functions preloaded *and executed*, since the latter will stay for a longer period of time in the cache.

4.5.9 Experiment Setup and Results

In this section we present our test environment together with our detection method to avoid different cache prefetching techniques that affect our measurements. Finally we present the results of our experiments for the PolarSSL, GnuTLS and CyaSSL libraries.

4.5.9.1 Experiment Setup

The experiments were run on an Intel i5-650 dual core at 3.2 GHz. Our physical server includes 256 KB per core L2 cache, and a 4 MB L3 cache shared between both cores. We used VMware ESXI 5.5.0 build number 162338 for virtualization. TPS is enabled with 4 KB pages. In this setting, our *Flush and Reload* technique can distinguish between L3 cache and main memory accesses.

For the TLS connection, we use an echo server which reads and re-sends the message that it receives, and a client communicating with it. Client and echo server are running in different virtual machines that use Ubuntu 12.04 guest OS. We modify the echo server functionality so that it adds a jitter in the encrypted reply message, modeling the Man in the Middle Attack. Once the message is sent, the echo server uses *Flush and Reload* to detect different function calls and concludes if the padding was correct or not.

4.5.9.2 Dealing with Cache Prefetching

Modern CPUs implement cache prefetching in a number of ways. These techniques affect our experiments, since the monitored function can be prefetched to cache, even if it was not executed by the victim process. To avoid false positives, it is not sufficient to detect *if* the monitored functions were loaded to cache, but also for *how long* they have resided in the cache. This is achieved by counting the number of subsequent detections for the given function in one execution. Therefore, the attack process effectively distinguishes between *prefetched* functions and *prefetched and executed* functions.

We use experiments to determine a threshold (which differs across the libraries) to distinguish a prefetch and execute from a mere prefetch. For PolarSSL this threshold is based on observing three *Flush and Reload* accesses in a row. Assume that n is the number of subsequent accesses required to conclude that the function was executed. In the following, we present the required hits for different libraries, i.e. the number of n-accesses required to decide whether the targeted function was executed or not.

4.5.9.3 Attack on PolarSSL1.3.6

Our first attack targets PolarSSL 1.3.6, with TLS 1.1. In the first scenario the attacker modifies the last two bytes of the encrypted message until she finds the

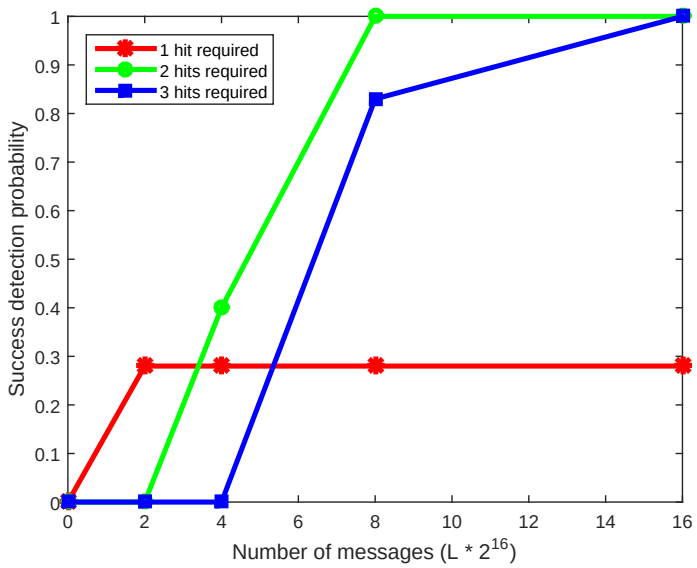


Figure 4.8: (PolarSSL 1.3.6) Success probability of recovering P_{14} and P_{15} vs. L, for different number of hits required. L refers to the number of 2^{16} traces needed, so the total number of messages is $2^{16} * L$.

Δ that leads to a 0x01|0x01 padding. Recall that 2^{16} different variations can be performed in the message. The first plot shows the success probability of guessing the right Δ versus L, where L refers to the number of 2^{16} traces needed. For example $L = 4$ means that $2^{16} * 4$ messages are needed to detect the right Δ. Based on experimental results, we set the access threshold such that we consider a hit whenever the targeted function gets two accesses in a row.

The measurements were performed for different number of required hits. Figure 4.8 shows that requiring a single hit might not suffice since the attacker gets false positives, or for small number of messages she may miss the access at all. However when we require two hits, and if the attacker has a sufficient number of messages (in this case $L = 2^3$), the probability of guessing the right Δ is comfortably close to one. If the attacker increases the limit further to ensure an even lower number of false positives, she will need more messages to see the required number of hits. In the case of 3 hits, $L = 2^4$ is required to have a success probability close to one.

Figure 4.9 shows the success probability of correctly recovering P_{13}, once the attacker has recovered the last two bytes. Now the attacker is looking for the padding 0x02|0x02|0x02. We observed a similar behavior with respect to the previous case where with $L = 8$ and with a two hits requirement we will recover the correct byte with high probability. Again if the attacker increases the requirement to 3 hits, she will need more measurements; about $L = 16$ is sufficient in practice.

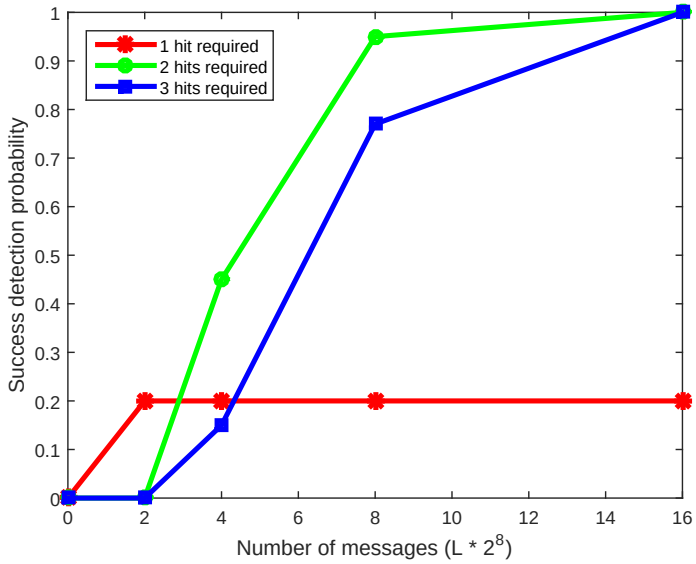


Figure 4.9: (PolarSSL 1.3.6) Success probability of recovering P_{13} assuming P_{14}, P_{15} known vs L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages is $2^8 * L$.

4.5.9.4 CyaSSL 3.0.0

Recall that the attack is much more effective if the attacker knows any of the preceding bytes of the plaintext, for example the last byte P_{15} of the plaintext. This would be the case in a Javascript/web setting where adjusting the length of an initial HTTP request an attacker can ensure that there is only one unknown byte in the HTTP plaintext. In this case, the attacker would not need to try 2^{16} possible variations but only 2^8 variations for each byte that she wants to recover. This is the scenario that we analyzed in CyaSSL TLS 1.2, where we assumed that the attacker knows P_{15} and she wants to recover P_{14}. Now the attacker is again trying to obtain a 0x01|0x01 padding, but unlike in the previous case, she knows the Δ to make the last byte equal to 0x01. The implementation of CyaSSL behaves very similarly to the one of PolarSSL, where due to the access threshold, a one hit might lead to false positives. However, requiring two hits with a sufficient number of measurements is enough to obtain a success probability very close to one. The threshold is set as in the previous cases, where a hit is considered whenever we observe two *Flush and Reload* accesses in a row.

4.5.9.5 GnuTLS 3.2.0

Finally we present the results confirming that GnuTLS3.2.0 TLS 1.2 is also vulnerable to this kind of attack. Again, the measurements were taken assuming that

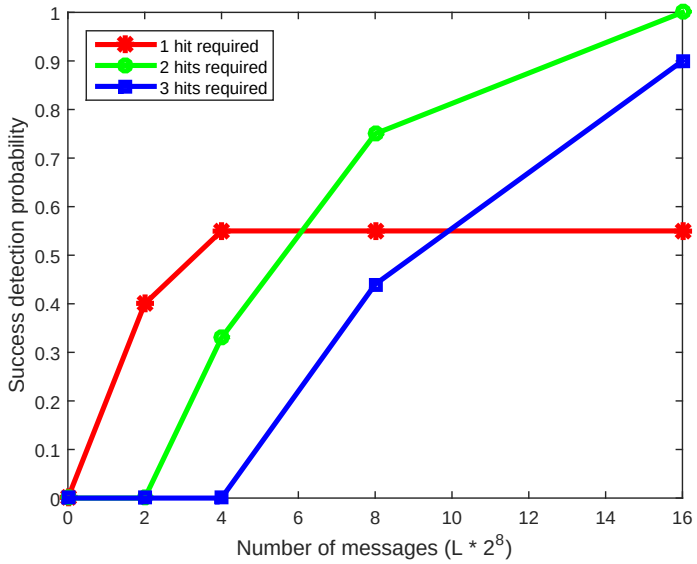


Figure 4.10: (CyaSSL3.0.0) Success Probability of recovering P_{14} assuming P_{15} known vs L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages would be $2^8 * L$.

the attacker knows the last byte P_{15} and she wants to recover P_{14}, i.e., she wants to observe the case where she injects a 0x01|0x01 padding. However, GnuTLS's behavior shows some differences with respect to the previous cases. For the case of GnuTLS, we find that if we set an access threshold of three accesses in a row (which would yield our desired hit), the probability of getting false positives is very low. Based on experimental measurements we observed that only when the dummy function is executed we observe such a behavior. However the attacker needs more messages to be able to detect one of these hits. Observing one hit indicates with high probability that the function was called, but we also consider the two hit case in case the attacker wants the probability of having false positives to be even lower. Based on these measurements, we conclude that the attacker recovers the plaintext with very high probability, so we did not find it necessary to consider the three hit case.

4.6 *Flush and Reload* Outcomes

In short, we demonstrated that if the memory deduplication requirement is satisfied, *Flush and Reload* can have severe consequences to processers/users co-residing in the same CPU socket, even if they are located in different CPU cores. We have demonstrated that such an attack can be utilized to recover cryptographic keys and TLS messages from CPU co-resident users. More than that, we demonstrated that

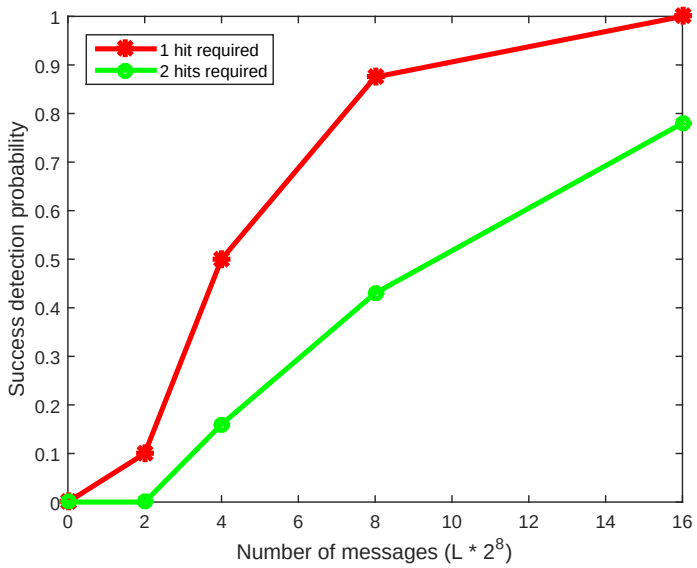


Figure 4.11: (GnuTLS3.2.0) Success Probability of recovering P_{14} assuming P_{15} known vs. L, for different number of hits required. L refers to the number of 2^8 traces needed, so the total number of messages would be $2^8 * L$.

Flush and Reload can bypass the isolation techniques implemented by commonly used hypervisors to avoid cross-VM leakage.

Despite all these advantages, we observed two major hurdles that *Flush and Reload* cannot overcome:

- *Flush and Reload* cannot attack victims located in a different CPU, but it is restricted to target victims located in the same CPU socket.
- *Flush and Reload* cannot be applied in systems in which memory deduplication does not exist, as the attacker does not get access to the victim's data. This fact also restricts *Flush and Reload* to attack *only* statically allocated data.

In the following chapters we explain how to overcome the two obstacles that *Flush and Reload* is not able to bypass.

Chapter 5

The First Cross-CPU Attack: *Invalidate and Transfer*

In previous sections we have presented *Flush and Reload* as a cross-core side channel attack introduced to target Intel processors. However, the utilized covert channels makes use of specific characteristics that Intel processors feature. For example, the proposed LLC attack takes advantage of their inclusive cache design. Furthermore, it also relies on the fact that the LLC is shared across cores. Therefore the *Flush and Reload* attack succeeds only when the victim and the attacker are co-located on the same CPU.

These characteristics are not observed in other CPUs, e.g. AMD or ARM. Aiming at solving these issues, this chapter presents *Invalidate and Transfer*, an attack that expands deduplication enabled LLC attacks to victims residing in different CPU sockets with any LLC characteristics. We utilize AMD as an example, but the same technique should also succeed in ARM processors. In this sense, AMD servers present two main complications that prevents application of existing side channel attacks:

- AMD tends to have more CPU sockets in high end servers compared to Intel. This reduces the chance of being co-located in the same CPU, and therefore, to apply the aforementioned *Flush and Reload* attack.
- LLCs in AMD are usually *exclusive* or *non-inclusive*. The former does not allocate a memory block in different level caches at the same time. That is, data is present in only one level of the cache hierarchy. Non-inclusive caches show neither inclusive or exclusive behavior. This means that any memory access will fetch the memory block to the upper level caches first. However, the data can be evicted in the outer or inner caches independently. Hence, accesses to L1 cache cannot be detected by monitoring the LLC, as it is possible on Intel machines.

Hence, to perform a side channel attack on AMD processors, both of these challenges need to be overcome. Here we present a covert channel that is immune to both

complications. The proposed *Invalidate and Transfer* attack is the first side channel attack that works across CPUs that feature non-inclusive or exclusive caches.

Invalidate and Transfer presents a new covert channel based on cache coherency technologies implemented in modern processors. In particular, we focus on AMD processors, which have exclusive caches that in principle are invulnerable to cache side channel attacks although the results can be readily applied to multi-CPU Intel processors as well. In summary,

- We present the first cross-CPU side channel attack, showing that CPU co-location is not needed in multi-CPU servers to obtain fine grain information.
- We present a new deduplication-based covert channel that utilizes directory-based cache coherency protocols to extract sensitive information.
- We show that the new covert channel succeeds in those processors where cache attacks have not been shown to be possible before, e.g. AMD exclusive caches.
- We demonstrate the feasibility of our new side channel technique by mounting an attack on a T-table based AES and a square and multiply implementation of El Gamal schemes.

5.1 Cache Coherence Protocols

In order to ensure coherence between different copies of the same data, systems implement cache coherence protocols. In the multiprocessor setting, the coherency between shared blocks that are cached in different processors (and therefore in different caches) also needs to be maintained. The system has to ensure that each processor accesses the most recent value of a shared block, regardless of where that memory block is cached. The two main categories of cache coherence protocols are *snooping based protocols* and *directory based protocols*. While snooping based protocols follow a decentralized approach, they usually require a centralized data bus that connects all caches. This results in excessive bandwidth need for systems with an increasing number of cores. Directory-based protocols, however, enable point-to-point connections between cores and directories, hence follow an approach that scales much better with an increasing number of cores in the system. We put our focus in the latter one, since it is the prevailing choice in current multiprocessor systems. The directory keeps track of the state of each of the cached memory blocks. Thus, upon a memory block access request, the directory will decide the state that the memory block has to be turned into, both in the requesting node and the *sharing* nodes that have a cached copy of the requested memory block. We analyze the simplest cache coherence protocol, with only 3 states, since the attack that is implemented in this study relies on read-only data. Thus, the additional states applied in more complicated coherency protocols do not affect the flow of our attack.

We introduce the terms `home node` for the node where the memory block resides, `local node` for the node requesting access to the memory block, and `owner node` referring a node that has a valid copy of the memory block cached. This leads to various communication messages that are summarized as follows:

- The memory block cached in one or more nodes can be in either `uncached` state, `exclusive/modified` or `shared`.
- Upon a read hit, the `local node's` cache services the data. In this case, the memory block maintains its state.
- Upon a read miss, the `local node` contacts the home node to retrieve the memory block. The directory knows the state of the memory block in other nodes, so its state will be changed accordingly. If the block is in `exclusive` state, it goes to `shared`. If the block is in `shared` state, it maintains it. In both cases the `local node` then becomes an owner and holds a copy of the shared memory block.
- Upon a write hit, the `local node` sets the memory block to exclusive. The `local node` communicates the nodes that have a cached copy of the memory block to invalidate or to update it.
- Upon a write miss, again the `home node` will service the memory block. The directory knows the nodes that have a cached copy of the memory block, and therefore sends them either an update or an invalidate message. The `local node` then becomes an owner of the `exclusive` memory block.

In practice, most cache coherency protocols have additional states that the memory block can acquire. The most studied one is the MESI protocol, where the `exclusive` state is divided into the `exclusive` and `modified` states. Indeed, a memory block is `exclusive` when a single node has a clean state of the memory block cached. However, when a cached memory block is modified, it acquires the `modified` state since it is not consistent with the value stored in the memory. A write back operation would set the memory block back to the `exclusive` state.

The protocols implemented in modern processors are variants of the MESI protocol, mainly adding additional states. For instance, the Intel i7 processor uses a MESIF protocol, which adds the additional `forward` state. This state will designate the sharing processor that should reply to a request of a shared memory block, without involving a memory access operation. The AMD Opteron utilizes the MOESI protocol with the additional `owned` state. This state indicates that the memory block is owned by the corresponding cache and is out-of-date with the memory value. However, contrary to the MESI protocol where a transition from `modified` to `shared` involves a write back operation, the node holding the `owned` state memory block can service it to the sharing nodes without writing it back to memory. Note that both the MESIF and MOESI protocol involve a cache memory

block forwarding operation. Both the `owned` and the `forward` state suggest that a cache rather than a DRAM will satisfy the reading request. If the access time from cache differs from regular DRAM access times, this behavior becomes an exploitable covert channel.

5.1.1 AMD HyperTransport Technology

Cache coherency plays a key role in multi-core servers where a memory block might reside in many core-private caches in the same state or in a modified state. In multiple socket servers, this coherency does not only have to be maintained within a processor, but also across CPUs. Thus, complex technologies are implemented to ensure the coherency in the system. These technologies center around the cache directory protocols explained in section 5.1. The *HyperTransport* technology implemented by AMD processors serves as a good example. We only focus on the features relevant to the new proposed covert channel. A detailed explanation can be found in [CKD+10, AMD].

The *HyperTransport* technology reserves a portion of the LLC to act as directory cache in the directory based protocol. This directory cache keeps track of the cached memory blocks present in the system. Once the directory is full, one of the previous entries will be replaced to make room for a new cached memory block. The directory *always* knows the state of any cached memory block, i.e., if a cache line exists in any of the caches, it must also have an entry in the directory. Any memory request will go first through the home node's directory. The directory knows the processors that have the requested memory block cached, if any. The home node initiates in parallel both a DRAM access and a *probe filter*. The probe filter is the action of checking in the directory which processor has a copy of the requested memory block. If any node holds a cached copy of the memory block, a directed probe against it is initiated, i.e., the memory block will directly be fast forwarded from the cached data to the requesting processor. A `directed probe` message does not trigger a DRAM access. Instead, communications between nodes are facilitated via HyperTransport links, which can run as fast as 3 GHz. Figure 5.1 shows a diagram of how the HyperTransport links directly connect the different CPUs to each other avoiding memory node accesses. Although many execution patterns can arise from this protocol, we will only explain those relevant to the attack, i.e. events triggered over read-only blocks which we will elaborate on later. We assume that we have processors A and B, refereed to as P_a and P_b, that share a memory block:

- If P_a and P_b have the same memory block cached, upon a modification made by P_a, *HyperTransport* will notify P_b that P_a has the latest version of the memory block. Thus, P_a will have to update its version of the block to convert the `shared` block into a `owned` block. Upon a new request made by P_b, *HyperTransport* will transfer the updated memory block cached in P_a.

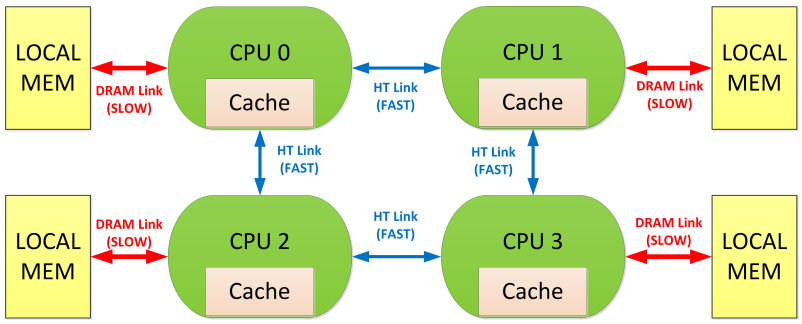


Figure 5.1: DRAM accesses vs Directed probes thanks to the HyperTransport Links

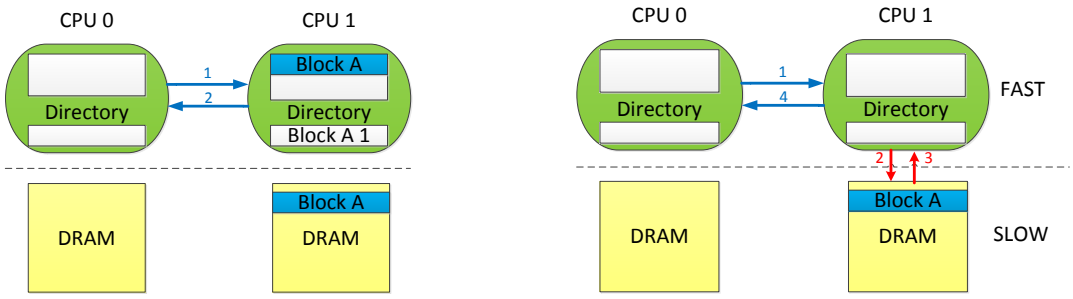
- Similarly, upon a cache miss in P_a, the home node will send a probe message to the processors that have a copy of the same shared memory block, if any. If, for instance, P_b has it, a `directed probe` message is initiated so that the node can service the cached data through the hypertransport links. Therefore, *HyperTransport* reduces the latency of retrieving a memory block from the DRAM by also checking whether someone else maintains a cached copy of the same memory block. Note that this process does not involve a write-back operation.

- When a new entry has to be placed in the directory of P_a, and the directory is full, one of the previously allocated entries has to be evicted to make room for the new entry. This is referred as a `downgrade probe`. In this case, if the cache line is dirty a writeback is forced, and an invalidate message is sent to all the processors (P_b) that maintain a cached copy of the same memory block.

In short, *HyperTransport* reduces latencies that were observed in previously implemented cache coherency protocols by issuing `directed probes` to the nodes that have a copy of the requested memory block cached. The HyperTransport links ensure a fast transfer to the requesting node. In fact, the introduction of HyperTransport links greatly improved the performance and thus viability of multi-CPU systems. Earlier multi-CPU systems relied on broadcast or directory protocols, where a request of a exclusive cached memory block in an adjacent processor would imply a writeback operation to retrieve the up-to-date memory block from the DRAM.

5.1.2 Intel QuickPath Interconnect Technology

In order to maintain cache coherency across multiple CPUs Intel implements a similar technique to AMD's *HyperTransport* called *Intel QuickPath Interconnect* (QPI) [Intd, IQP]. Indeed, the later one was designed five years latter than the first one to compete with the existing technology in AMD processors. Similar to *HyperTransport*, QPI connects one or more processors through high speed point-to-point links as fast as 3.2GHz. Each processor has a memory controller on the same



(a) HTLink: fast access

(b) DRAM: slow access

Figure 5.2: Comparison of a directed probe access across processors: probe satisfied from CPU 1's cache directly via HTLink (a) vs. probe satisfied by CPU 1 via a slow DRAM access (b).

die to make to improve the performance. As we have already seen with AMD, among other advantages, this interface efficiently manages the cache coherence in the system in multiple processor servers by transferring shared memory blocks through the QPI high speed links. In consequence, the proposed mechanisms that we later explain in this paper are also applicable in servers featuring multi-CPU Intel processors.

5.2 *Invalidate and Transfer* Attack Procedure

We propose a new spy process that takes advantage of leakages observed in the cache coherency protocol with memory blocks shared between many processors/cores. The spy process does not rely on specific characteristics of the cache hierarchy, like inclusiveness. In fact, the spy process works even across co-resident CPUs that do not share the same cache hierarchy. From now on, we assume that the victim and attacker share the same memory block and that they are located in different CPUs or in different cache hierarchies in the same server.

The spy process is executed in three main steps, which are:

- **Invalidate step:** In this step, the attacker invalidates a memory block that is in his own cache hierarchy. If the invalidation is performed in a shared memory block cached in another cache processors cache hierarchy, the *HyperTransport* will send an invalidate message to them. Therefore, after the invalidation step, the memory block will be invalidated in all the caches that have the same memory block, and this will be uncached from them. This invalidation can be achieved by specialized instructions like `clflush` if they are supported by the targeted processors, or by priming the set where the memory block resides in the cache directory.

- **Wait step:** In this step, the attacker waits for a certain period of time to

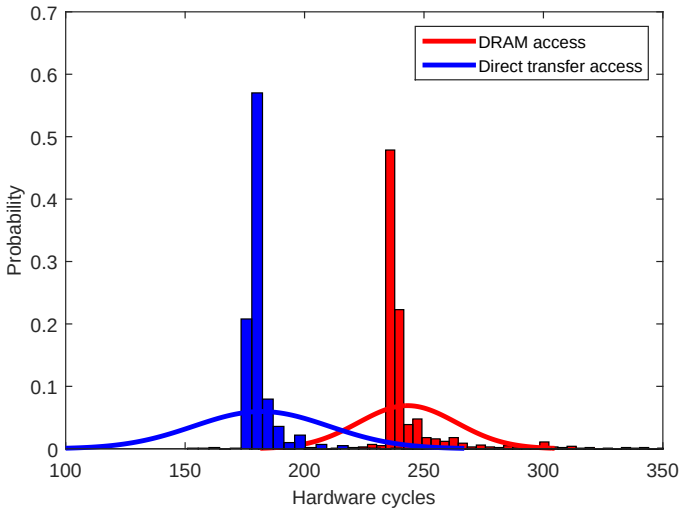


Figure 5.3: Timing distribution of a memory block request to the DRAM (red) vs a block request to a co-resident processor(blue) in a AMD opteron 6168. The measurements are taken from different CPUs. Outliers above 400 cycles have been removed

let the victim do some computation. The victim might or might not use the invalidated memory block in this step.

- **Transfer step:** In the last step, the attacker requests access to the shared memory block that was invalidated. If any processor in the system has cached this memory block, the entry in the directory would have been updated and therefore a `direct probe` request will be sent to the processor. If the memory block was not been used, the home directory will request a DRAM access to the memory block.

The system experiences a lower latency when a `direct probe` is issued, mainly because the memory block is issued from another processors cache hierarchy. This is graphically observed in Figure 5.2. Figure 5.2(a) shows a request serviced by the *HyperTransport* link from a CPU that has the same memory block cached. In contrast, Figure 5.2(b) represents a request serviced by a DRAM access. This introduces a new leakage if the attacker is able to measure and distinguish the time that both actions take. This is the covert channel that will be exploited in this work. We use the `RDTSC` function which accesses the time stamp counter to measure the request time. In case the `RDTSC` function is not available from user mode, one can also create a parallel thread incrementing a shared variable that acts as a counter. We also utilize the `mfence` instruction to ensure that all memory load/store operations have finished before reading the time stamp counter.

The timing distributions of both the DRAM access and the directed transfer access are shown in Figure 5.3, where 10,000 points of each distribution were taken

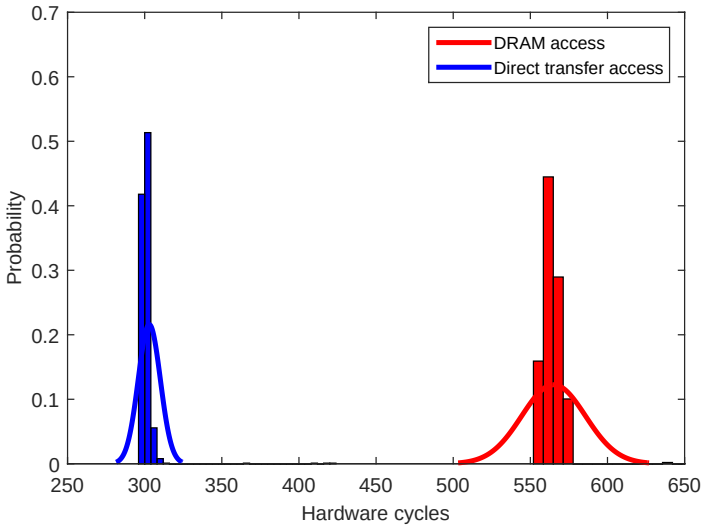


Figure 5.4: Timing distribution of a memory block request to the DRAM (red) vs a block request to a co-resident core(blue) in a dual core Intel E5-2609. The measurements are taken from the same CPU. Outliers above 700 cycles have been removed

across CPUs in a 48-core 4 CPU AMD Opteron 6168. The x-axis represents the hardware cycles, while the y-axis represents the density function. The measurements are taken across processors. The blue distribution represents a `directed probe` access, i.e., a co-resident CPU has the memory block cached, whereas the red distribution represents a DRAM access, i.e., the memory block is not cached anywhere. It can be observed that the distributions differ in about 50 cycles, fine grain enough to be able to distinguish them. However, the variance in both distributions is very similar, in contrast to LLC covert channels. Nevertheless, we obtain a covert channel that works across CPUs and that does not rely on the inclusiveness property of the cache.

We also tested the viability of the covert channel in a dual socket Intel Xeon E5-2609. Intel utilizes a similar technique to the *HyperTransport* technology called `Intel Quick Path Interconnect`. The results for the Intel processor are shown in Figure 5.4, again with processes running in different CPUs. It can be observed that the distributions are even more distinguishable in this case.

5.3 Exploiting the New Covert Channel

In the previous section, we presented the viability of the covert channel. Here we demonstrate how one might exploit the covert channel to extract fine grain information. More concretely, we present two attacks:

- a symmetric cryptography algorithm, i.e. table based `OpenSSL` implementation of AES, and
- a public key algorithm, i.e. a square-and-multiply based `libgcrypt` implementation of the El Gamal scheme.

5.3.1 Attacking Table Based AES

We test the granularity of the new covert channel by mounting an attack in a software implementation of AES, as in 4.4.We again use the C OpenSSL reference implementation, which uses 4 different T-tables along 10 rounds for AES-128.

To recap, we monitor a memory block belonging to each one of the T-tables. Each memory block contains 16 T-Table positions and it has a certain probability, 8% in our particular case, of not being used in any of the 10 rounds of an encryption. Thus, applying our *Invalidate and Transfer* attack and recording the ciphertext output, we can know when the monitored memory block *has not* been used. For this purpose we `invalidate` the memory block before the encryption and try to probe it after the encryption. In a noise free scenario, the monitored memory block will not be used for 240 ciphertext outputs with 8% probability, and it will not be used for the remaining 16 ciphertext with 0% probability (because they directly map through the key to the monitored T-table memory block). Although microarchitectural attacks suffer from different microarchitectural sources of noise, we expect that the *Invalidate and Transfer* can still distinguish both distributions.

Once we know the ciphertext values belonging to both distributions, we can apply the equation:

$$K_i = T[S_j] \oplus C_i$$

to recover the key. Since the last round of AES involves only a Table look up and a XOR operation, knowing the ciphertext and the T-table block position used is enough to obtain the key byte candidate that was used during the last AES round. Since a cache line holds 16 T-table values, we XOR each of the obtained ciphertext values with all the 16 possible T-table values that they could map to. Clearly, the key candidate will be a common factor in the computations with the exception of the observed noise which is eliminated via averaging. As the AES key schedule is revertible, knowing one of the round keys is equivalent to knowing the full encryption key.

5.3.2 Attacking Square and Multiply El Gamal Decryption

We test the viability of the new side channel technique with an attack on a square and multiply `libgcrypt` implementation of the public key ElGamal algorithm, as in [ZJRR12]. An ElGamal encryption involves a cyclic group of order p and a generator g of that cyclic group. Then Alice chooses a number $a \in \mathbb{Z}_p^*$ and computes her public key as the 3-tuple (p, g, g^a) and keeps a as her secret key.

To encrypt a message m, Bob first chooses a number $b \in \mathbb{Z}_p^*$ and calculates $y_1 = g^b$ and $y_2 = ((g^a)^b) * m$ and sends both to Alice. In order to decrypt the message, Alice utilizes her secret key a to compute $((y_1)^{-a}) * y_2$. Note that, if a malicious user recovers the secret key a he can decrypt any message sent to Alice.

Our target will be the y_1^{-a} that uses the square and multiply technique as the modular exponentiation method. It bases its procedure in two operations: a square operation followed by a modulo reduction and a multiplication operation followed by a modulo reduction. The algorithm starts with the intermediate state $R = b$ being b the base that is going to be powered, and then examines the secret exponent a from the most significant to the least significant bit. If the bit is a 0, the intermediate state is squared and reduced with the modulus. If in the contrary the exponent bit is a 1, the intermediate state is first squared, then it is multiplied with the base b and then reduced with the modulus. Algorithm 2, presented in the background section and shown below, shows the entire procedure.

Algorithm 6 Square and Multiply Exponentiation

Input : base b, modulus N, secret $E = (e_{k-1}, ..., e_1, e_0)$
Output: $b^E \bmod N$
$R = b$;
for $i = k - 1$ **downto** 0 **do**
    $R = R^2 \bmod N$;
    if e_i==*1* **then**
        $R = R * b \bmod N$;
    end
end
return R;

As it can be observed the algorithm does not implement a constant execution flow, i.e., the functions that will be used directly depend on the bit exponent. If the square and multiply pattern is known, the complete key can be easily computed by converting them into ones and zeros. Indeed, our *Invalidate and Transfer* spy process can recover this information, since functions are stored as shared memory blocks in cryptographic libraries. Thus, we mount an attack with the *Invalidate and Transfer* to monitor when the square and multiplication functions are utilized.

5.4 Experiment Setup and Results

In this section we present the test setup in which we implemented and executed the *Invalidate and Transfer* spy process together with the results obtained for the AES and ElGamal attacks.

5.4.1 Experiment Setup

In order to prove the viability of our attack, we performed our experiments on a 48-core machine featuring four 12-core AMD Opteron 6168 CPUs. This is an university server which has not been isolated for our experiments, i.e., other users are utilizing it at the same time. Thus, the environment is a realistic scenario in which non desired applications are running concurrently with our attack.

The machine runs at 1.9GHz, featuring 3.2GHz HyperTransport links. The server has 4 AMD Opteron 6168 CPUs, with 12 cores each. Each core features a private 64KB 2-way L1 data cache, a private 64KB L1 instruction cache and a 16-way 512KB L2 cache. Two 6MB 96-way associative L3 caches—each one shared across 6 cores—complete the cache hierarchy. The L1 and L2 caches are core-private resources, whereas the L3 cache is shared between 6 cores. Both the L2 and L3 caches are non-inclusive, i.e., data can be allocated in *any* cache level at a time. This is different to the inclusive LLC where most of the cache spy processes in literature have been executed.

The attacks were implemented in a RedHat enterprise server running the linux 2.6.23 kernel. The attacks do not require root access to succeed, in fact, we did not have sudo rights on this server. Since ASLR was enabled, the targeted functions addresses were retrieved by calculating the offset with respect to the starting point of the library. All the experiments were performed across CPUs, i.e., attacker and victim do not reside in the same CPU and do not share any LLC. To ensure this, we utilized the `taskset` command to assign the CPU affinity to our processes.

Our targets were the AES C reference implementation of OpenSSL and the ElGamal square and multiply implementation of libgcrypt 1.5.2. The libraries are compiled as shared, i.e., all users in the OS will use the same shared symbols (through KSM mechanism). In the case of AES we assume we are synchronized with the AES server, i.e., the attacker sends plaintext and receives the corresponding ciphertexts. As for the ElGamal case, we assume we are not synchronized with the server. Instead, the attacker process simply monitors the function until valid patterns are observed, which are then used for key extraction.

5.4.2 AES Results

As explained in Section 5.3, in order to recover the full key we need to target a single memory block from the four T-tables. However, in the case that a T-table memory block starts in the middle of a cache line, monitoring only 2 memory blocks is enough to recover the full key. In fact, there exists a memory block that contains both the last 8 values of T0 and the first 8 values of T1. Similarly there exists a memory block that contains the last 8 values of T2 and the first 8 values of T3. Unlike in section 4.4, we make use of this fact and only monitor those two memory blocks to recover the entire AES key.

We store both the transfer timing and the ciphertext obtained by our encryption

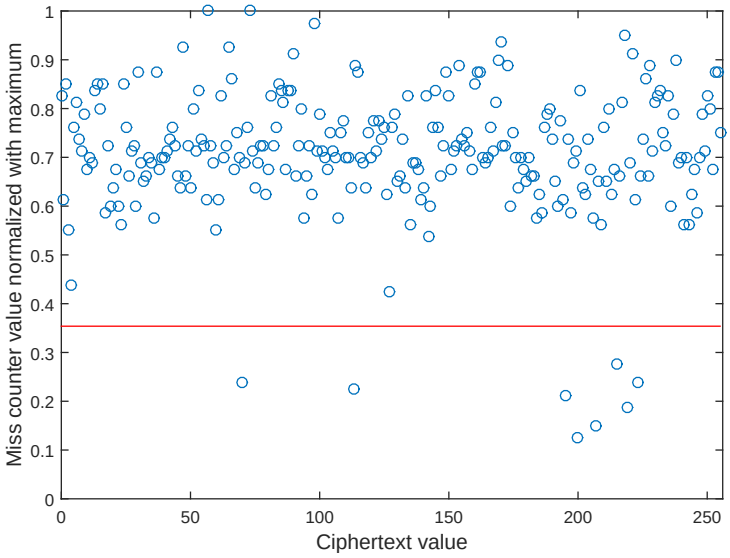


Figure 5.5: Miss counter values for each ciphertext value, normalized to the average

server. In order to analyze the results, we implement a miss counter approach: we count the number of times that each ciphertext value sees a miss, i.e. that the monitored cache line was not loaded for that ciphertext value. An example of one of the runs for ciphertext number 0 is shown in Figure 5.5. The 8 ciphertext values that obtain the lowest scores are the ones corresponding to the cache line, thereby revealing the key value.

In order to obtain the key, we iterate over all possible key byte values and compute the last round of AES only for the monitored T-table values, and then group the miss counter values of the resulting ciphertexts in one set. We group in another set the miss counter of the remaining 248 ciphertext values. Clearly, for the correct key, the distance between the two sets will be maximum. An example of the output of this step is shown in Figure 5.6, where the y-axis represents the miss counter ratio (i.e., ratio of the miss counter value in both sets) and the x-axis represents the key byte guess value. It can be observed that the ratio of the correct key byte (180) is much higher than the ratio of the other guesses.

Finally, we calculate the number of encryptions needed to recover the full AES key. This is shown in Figure 5.7, where the y-axis again represents the ratios and the x-axis represents the number of encryptions. As it can be observed, the correct key is not distinguishable before 10,000 traces, but from 20,000 observations, the correct key is clearly distinguishable from the rest. We conclude that the new method succeeds in recovering the correct key from 20,000 encryptions.

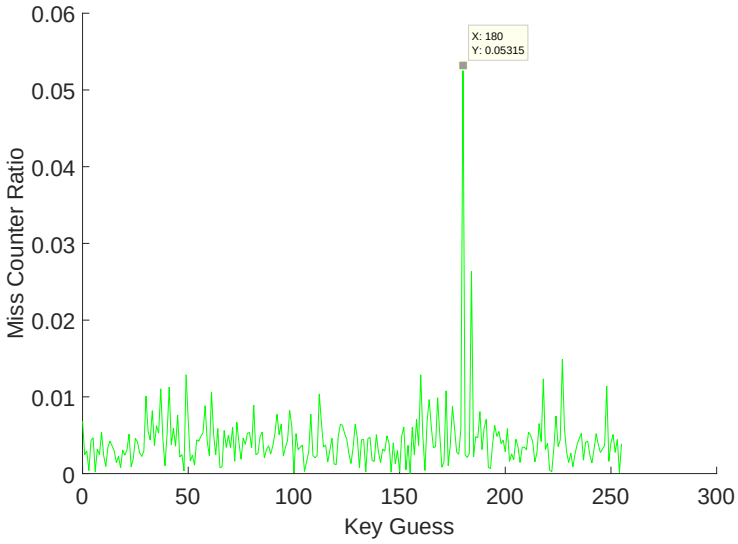


Figure 5.6: Correct key byte finding step, iterating over all possible keys. The maximum distance is observed for the correct key

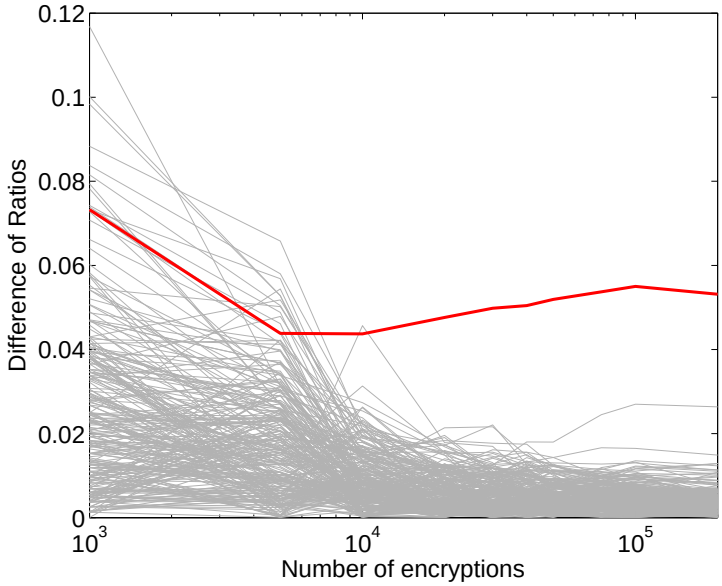


Figure 5.7: Difference of Ratios over the number of encryptions needed to recover the full AES key. The correct key (bold red line) is clearly distinguishable from 20,000 encryptions.

5.4.3 El Gamal Results

Next we present the results obtained when the attack aims at recovering an ElGamal decryption key. We target a 2048 bit ElGamal key. Remember that, unlike in the

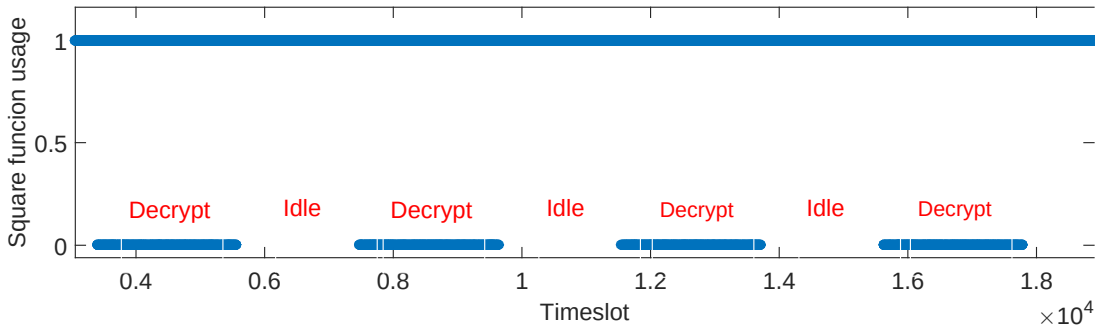


Figure 5.8: Trace observed by the *Invalidate and Transfer*, where 4 decryption operations are monitored. The decryption stages are clearly visible when the square function usage gets the 0 value

case of AES, this attack does not need synchronization with the server, i.e., the server runs continuous decryptions while the attacker continuously monitors the vulnerable function. Since the modular exponentiation creates a very specific pattern with respect to both the square and multiply functions, we can easily know when the exponentiation occurred in the time. We only monitor a single function, i.e., the square function. In order to avoid speculative execution, we do not monitor the main function address but the following one. This is sufficient to correctly recover a very high percentage of the ElGamal decryption key bits. For our experiments, we take the time that the invalidate operation takes into account, and a minimum waiting period of 500 cycles between the *invalidate* and the *transfer* operation is sufficient to recover the key patterns. Figure 5.8 presents a trace where 4 different decryptions are caught. A 0 in the y-axis means that the square function is being utilized, while a 1 the square function is not utilized, while the x-axis represents the time slot number. The decryption stages are clearly observable when the square function gets a 0 value.

Recall that the execution flow caused by a 0 bit in the exponent is `square+reduction`, while the pattern caused by a 1 bit in the exponent is `square+reduction+multiply+reduction`. Since we only monitor the square operation, we reconstruct the patterns by checking the distance between two square operations. Clearly, the distance between the two square operations in a 00 trace will be smaller than the distance between the two square operations in a 10 trace, since the latter one takes an additional multiplication function. With our waiting period threshold, we observe that the distance between two square operations without the multiplication function varies from 2 to 4 *Invalidate and Transfer* steps, while the distance between two square operations varies from 6 to 8 *Invalidate and Transfer* steps. If the distance between two square operations is lower than 2, we consider it part of the same square operation. An example of such a trace is shown in Figure 5.9. In the figure, S refers to a square operation, R refers to a modulo reduction operation

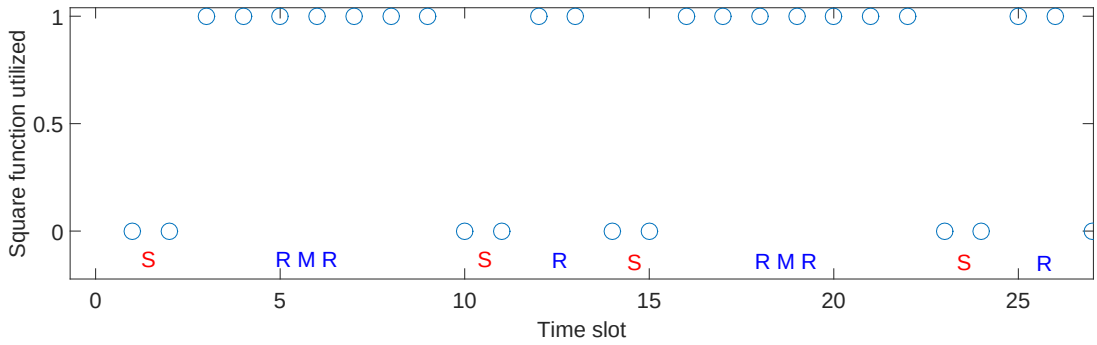


Figure 5.9: Trace observed by the *Invalidate and Transfer*, converted into square and multiply functions. The y-axis shows a 0 when the square function is used and a 1 when the square function is not used

Table 5.1: Summary of error results in the RSA key recovery attack

Traces analysed	20
Maximum error observed	3.47%
Minimum error observed	1.9%
Average error	2.58%
Traces needed to recover full key	5

and M refers to a multiply operation. The x-axis represents the time slot, while the y axis represents whether the square function was utilized. The 0 value means that the square function was utilized, whereas the 1 value means that the square function was not utilized. The pattern obtained is $SRMRSRSRMRSR$, which can be translated into the key bit string 1010.

However, due to microarchitectural sources of noise (context switches, interrupts, etc.) the recovered key has still some errors. In order to evaluate the error percentage obtained, we compare the obtained bit string with the real key. Any insertion, removal or wrong guessed bit is considered a single error. Table 5.1 summarizes the results. We evaluate 20 different key traces obtained with the *Invalidate and Transfer* spy process. On average, they key patterns have an error percentage of 2.58%. The minimum observed error percentage was 1.9% and the maximum was 3.47%. Thus, since the errors are very likely to occur at different points in order to decide the correct pattern we analyse more than one trace. On average, 5 traces are needed to recover the key correctly.

5.5 *Invalidate and Transfer* Outcomes

We presented the *Invalidate and Transfer* attack, capable of recovering, for the first time, information across CPU sockets in systems that provide memory deduplication. The new attack utilizes the cache coherency protocol as a covert channel and its effectiveness was proved by recovering both AES and RSA keys. Further, the attack is inclusiveness property agnostic. On the downsides, the attack still requires attacker and victim to share memory, and thus, is only applicable in VMMs with memory deduplication or smartphones. In response, the next chapter introduces an attack that is not reliant upon the memory deduplication feature, and thus is applicable in virtually every system in which attacker and victim processes can co-reside.

Chapter 6

The *Prime and Probe* Attack

In previous chapters, we presented two side channel attacks that used hardware cache properties to retrieve information across cores/CPUs. However, both attacks worked under the assumption of shared memory between attacker and victim, which was achievable through mechanisms like KSM. Although we demonstrated to recover fine grain information with them, due to the shared memory requirement, we observe the following challenges associated to them:

- Some real world scenarios might not implement memory deduplication features. For instance, some commercial clouds have cross-VM memory deduplication disabled, as it is the case for Amazon EC2. Furthermore, memory page sharing is not allowed between trusted and untrusted worlds in Trusted Execution Environments (TEEs).
- *Flush and Reload* and *Invalidate and Transfer*, since they rely on memory sharing, cannot get information dynamically allocated memory, as every user gets its own copy of it. This limits the applicability of both attacks.

These two inconveniences restrict the applicability of *Flush and Reload* and *Invalidate and Transfer*. Thus, it is necessary to know whether an attacker can bypass these obstacles and still gain information about a victim's activity across cores. This chapter explains a new approach to utilize the Last Level Cache (LLC) as a covert channel without relying on memory deduplication features. In particular, we take the already known *Prime and Probe* attack and make it successful on the LLC. This is not a straightforward process, as it still requires to solve some technical issues when targeting the LLC.

6.1 Virtual Address Translation and Cache Addressing

In this chapter we present an attack that takes advantage of some known information in the virtual to physical address mapping process. Thus, we give a brief overview

about the procedure followed by modern processors to access and address data in the cache [HP11].

In modern computing, processes use *virtual memory* to access the different requested memory locations. Indeed processes do not have direct access to the physical memory, but use virtual addresses that are then mapped to physical addresses by the Memory Management Unit (MMU). This virtual address space is managed bv the Operating System. The main benefits of virtual memory are security (processes are isolated from *real memory*) and the usage of more memory than physically available due to paging techniques.

The memory is divided into fixed length continuous blocks called memory pages. The virtual memory allows the usage of these memory pages even when they are not allocated in the main memory. When a specific process needs a page not present in the main memory, a page fault occurs and the page has to be loaded from the auxiliary disk storage. Therefore, a translation stage is needed to map virtual addresses to physical addresses prior to the memory access. In fact, cloud systems have two translation processes, i.e, guest OS to VMM virtual address and VMM virtual address to physical address. The first translation is handled by shadow page tables while the second one is handled by the MMU. This adds an abstraction layer with the physical memory that is handled by the VMM.

During translation, the virtual address is split into two fields: the offset field and the page field. The length of both fields depends directly on the *page size*. Indeed, if the page size is p bytes, the lower $\log_2(p)$ bits of the virtual address will be considered as the *page offset*, while the rest will be considered as the *page frame number* (PFN). Only the PFN is processed by the MMU and needs to be translated from virtual to physical page number. The page offset remains untouched and will have the same value for both the physical and virtual address. Thus, the user still knows some bits of the physical address. Modern processors usually work with 4 KB pages and 48 bit virtual addresses, yielding a 12 bit offset and the remaining bits as virtual page number.

In order to avoid the latency of virtual to physical address translation, modern architectures include a Translation Lookaside Buffer (TLB) that holds the most recently translated addresses. The TLB acts like a small cache that is first checked prior to the MMU. One way to avoid TLB misses for large data processes is to increase the *page size* so that the memory is divided in less pages [CJ06, Inte, WW09]. Since the possible virtual to physical translation tags have been significantly reduced, the CPU will observe less TLB misses than with 4 KB pages. This is the reason why most modern processors include the possibility to use huge size pages, which typically have a size of at least 1 MB. This feature is particularly effective in virtualized settings, where virtual machines are typically rented to avoid the intensive hardware resource consumption in the customers private computers. In fact, most well known VMMs support the usage of huge size pages by guest OSs to improve the performance of those heavy load processes [VMwc, KVM, Xenb].

Cache Addressing: Caches are physically tagged, i.e, the physical address is used to decide the position that the data is going to occupy in the cache. With b bytes size cache lines and m-way set associative caches (with n number of sets), the lower $\log_2(b)$ bits of the physical address are used to index the byte in a cache line, while the following $\log_2(n)$ bits select the set that the memory line is mapped to in the cache. A graphical example of the procedure carried out to address the data in the cache can be seen in Figure 6.2. Recall that if a page size of 4 KB is used, the offset field is 12 bits long. If $\log_2(n) + \log_2(b)$ is not bigger than 12, the set that a cache line is going to occupy can be addressed by the offset. In this case we say that the cache is virtually addressed, since the position occupied by a cache line can be determined by the virtual address. In contrast, if more than 12 bits are needed to address the corresponding set, we say that the cache is physically addressed, since only the physical address can determine the location of a cache line. Note that when huge size pages are used, the offset field is longer, and therefore *bigger* caches can be virtually addressed. As we will see, this information can be used to mount a cross-VM attack in the L3 cache in deduplication free systems. Note that, this information was not necessary with *Flush and Reload* and *Invalidate and Transfer* as we assumed that we shared ownership of the attacked memory blocks with the victim.

6.2 Last Level Cache Slices

Recent SMP microarchitecures divide the LLC into slices with the purpose of reducing the bandwidth bottlenecks when more than one core attempts to retrieve data from the LLC at the same time. The number of slices that the LLC is divided into usually matches the number of physical cores. For instance, a processor with s cores divides the LLC into s slices, decreasing the probability of resource conflict while accessing it. However, each core is still able to access the whole LLC, i.e., each core can access every slice. Since the data will be spread into s "smaller caches" it is less likely that two cores will try to access the same slice at the same time. In fact, if each slice can support one access per cycle, the LLC does not introduce a bottleneck on the data throughput with respect to the processors as long as each processor issues no more than one access per cycle. The slice that a memory block is going to occupy directly depends on its own physical address and a non-public hash function, as in Figure 6.1.

Performance optimization of sliced caches has received a lot of attention in the past few years. In 2006, Cho et al. [CJ06] analyzed a distributed management approach for sliced L2 caches through OS page allocation. In 2007, Zhao et al. [ZIUN08] described a design for LLC where part of the slice allocates core-private data. Cho et al [JC07] describe a two-dimensional page coloring method to improve access latencies and miss rates in sliced caches. Similarly, Tam et al. [TASS07] also proposed an OS based method for partitioning the cache to avoid cache contention.

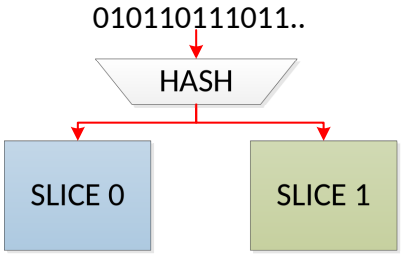


Figure 6.1: A hash function based on the physical address decides whether the memory block belongs to slice 0 or 1.

In 2010 Hardavellas et al. [HFFA10] proposed an optimized cache block placement for caches divided into slices. Srikantaiah et al. [SKZ+11] presented a new adaptive multilevel cache hierarchy utilizing cache slices for L2 and L3 caches. In 2013 Chen et al. [CCC+13] detail on the approach that Intel is planning to take for their next generation processors. The paper shows that the slices will be workload dependent and that some of them might be dynamically switched off for power saving. In 2014 Kurian et al. [KDK14b] proposed a data replication protocol in the LLC slices. Ye et al. [YWCL14] studied a cache partitioning system treating each slice as an independent smaller cache.

However, only very little effort has been put into analyzing the slice selection hash function used for selecting the LLC slice. A detailed analysis of the cache performance in Nehalem processors is described in [MHSM09] without an explanation of cache slices. The LLC slices and interconnections in a Sandy Bridge microarchitecture are discussed in [Bri], but the slice selection algorithm is not provided. In [hen], a cache analysis of the Intel Ivy Bridge architecture is presented and cache slices are mentioned. However, the hash function describing the slice selection algorithm is again not described, although it is mentioned that many bits of the physical address are involved. Hund et al. [HWH13] were the only ones describing the slice selection algorithm for a specific processor, i.e, the i7-2600 processor. They recover the slice selection algorithm by comparing Hamming distances of different physical addresses. This information was again not needed with the *Flush and Reload* and *Invalidate and Transfer* as we could evict a memory block from the cache with the `clflush` instruction in systems where deduplication is enabled.

6.3 The Original *Prime and Probe* Technique

The new attack proposed in this work is based on the methodology of the known *Prime and Probe* technique. *Prime and Probe* is a cache-based side channel attack technique used in [OST06, ZJOR11, ZJRR12] that can be classified as an access driven cache attack. The spy process ascertains which of the sets have been accessed in the cache by a victim. The attack is carried out in 3 stages:

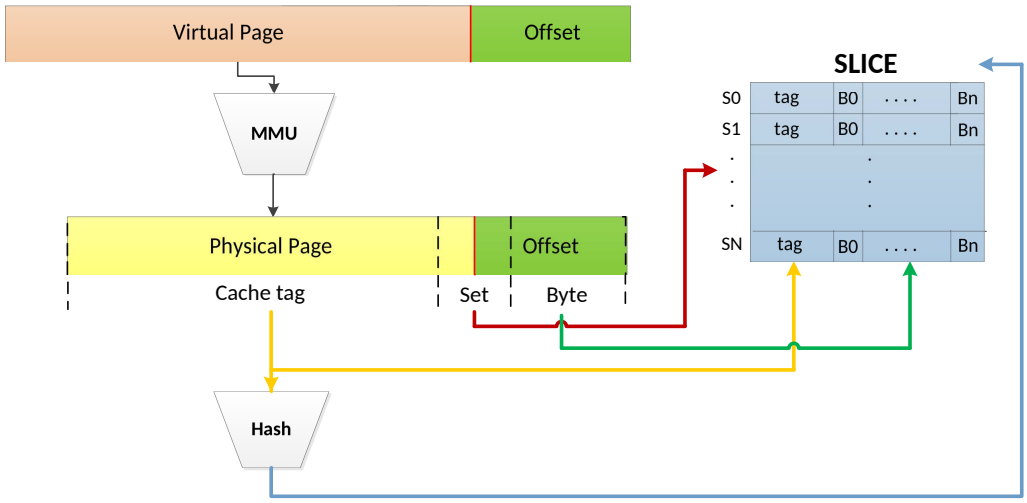


Figure 6.2: Last level cache addressing methodology for Intel processors. Slices are selected by the tag, which is given as the MSBs for the physical address.

- **Priming stage:** In this stage, the attacker fills the monitored cache with own cache lines. This is done by reading own data.
- **Victim accessing stage:** In this stage the attacker waits for the victim to access some positions in the cache, causing the eviction of some of the cache lines that were primed in the first stage.
- **Probing stage:** In this stage the attacker accesses the priming data again. When the attacker reloads data from a set that has been used by the victim, some of the primed cache lines have been evicted, causing a higher probe time. However if the victim did not use any of the cache lines in a monitored set, all the primed cache lines will still reside in the cache causing a low probe time.

6.4 Limitations of the Original *Prime and Probe* Technique

The original *Prime and Probe* technique was successfully applied to L1 caches to recover cryptographic keys. It is therefore an open question why, with multi-core systems and shared LLCs, no work prior to this has applied it into the LLC to recover information across cores. Here we summarize the three main problems of taking the *Prime and Probe* attack to the LLC:

- The L1 *Prime and Probe* attack fills the whole L1 cache with attackers data. As the LLC is usually at least two orders of magnitude bigger than the L1, filling the entire LLC does not seem a realistic approach.

- As memory pages are usually 4KB pages and the page offset remains untouched in the virtual address translation, the attacker gains enough bits of information to fully know the location of a memory block in the L1. However, as the LLC has more sets, the attacker is unable to predict the set that his memory addresses will occupy.

- The LLC in Intel, as previously described, is divided into slices after the release of the Sandy Bridge architecture. The slice that a memory block occupies is decided by an undocumented hash algorithm. Thus, an attacker that might be willing to fill one of the sets in the LLC might observe how his addresses get distributed across several slices and do not fill the set.

In the following sections we solve the aforementioned challenges to successfully apply the *Prime and Probe* attack in the LLC.

6.5 Targeting Small Pieces of the LLC

The first obstacle when executing thew original *Prime and Probe* attack in the LLC is the vast number of memory addresses needed to fill it. Instead, we will only target *smaller* pieces of it that will give us enough information of the secret process being executed by a victim.

Recall algorithm 2 from Section 2.4.2. The modular exponentiation leaked information due to the usage of the multiplication function when a "1"' bit was found. Instead of filling the entire cache, we can just perform the *Prime and Probe* attack in the LLC set where the multiplication resides. This gives us enough information about when the multiplication function is used, and therefore, when the victim is processing a "1"' or a "'0"' bit. We will cover in future sections how to know where this multiplication function resides.

6.6 LLC Set Location Information Enabled by Huge Pages

The second obstacle that the original *Prime and Probe* attack encounters when targeting the LLC is that the set occupied by the attackers memory blocks is unknown in the LLC, due to the lack of control on the physical address bits. The LLC *Prime and Probe* attack proposed in this work, is enabled by making use of Huge pages and thereby eliminating a major obstacle that normally restricts the *Prime and Probe* attack to target the L1 cache. As explained in Section 6.1, a user does not use the physical memory directly, but he is assigned a virtual memory so that a translation stage is performed from virtual to physical memory at the hardware level. The address translation step creates an additional challenge to the attacker since real addresses of the variables of the target process are unknown to him. However

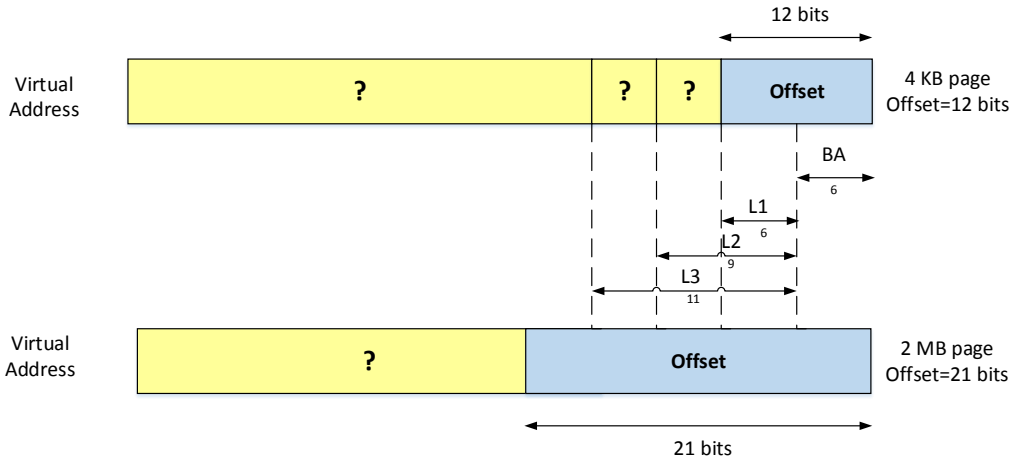


Figure 6.3: Regular Page (4 KB, top) and Huge Page (256 KB, bottom) virtual to physical address mapping for an Intel x86 processor. For Huge pages the entire L3 cache sets become transparently accessible even with virtual addressing.

this translation is only performed in some of the higher order bits of the virtual address, while a lower portion, named the *offset*, remains untouched. Since caches are addressed by the physical address, if we have cache line size of b bytes, the lower $\log_2(b)$ bits of the address will be used to resolve the corresponding byte in the cache line. Furthermore if the cache is set-associative and for instance divided into n sets, then the next $\log_2(n)$ bits of the address will select the set that each memory data is going to occupy in the cache. The $\log_2(b)$-bits that form the byte address within a cache line, are contained within the offset field. However, depending on the cache size the following field which contains the set address may exceed the offset boundary. The offsets allow addressing within a memory page. The OS's Memory Management Unit (MMU) keeps track of which page belongs to which process. The page size can be adjusted to better match the needs of the application. Smaller pages require more time for the MMU to resolve.

Here we focus on the default 4 KB page size and the larger page sizes provided under the common name of Huge pages. As we shall see, the choice of page size will make a significant difference in the attackers ability to carry out a successful attack on a particular cache level:

- **4 KB pages:** For 4 KB pages, the lower 12-bit offset of the virtual address is not translated while the remaining bits are forwarded to the Memory Management Unit. In modern processors the memory line size is usually set as 64 bytes. This leaves 6 bits untouched by the Memory Management Unit while translating regular pages. As shown in the top of Figure 6.3 the page offset is known to the attacker. Therefore, the attacker knows the 6-bit byte address

plus 6 additional bits that can only resolve accesses to small size caches (64 sets at most). This is the main reason why techniques such as *Prime and Probe* have only targeted the L1 cache, since it is the only one permitting the attacker to have full control of the bits resolving the set. Therefore, the small page size indirectly prevents attacks targeting big size caches like the L2 and L3 caches.

- **Huge pages:** The scenario is different if we work with huge size pages. Typical huge page sizes are at 1 MB or even greater. This means that the offset field in the page translation process is bigger, with 21 bits or more remaining untouched during page translation. Observe the example presented in Figure 6.3. For instance, assume that our computer has 3 levels of cache, with the last one shared, and that 64, 512 and 2048 are the number of sets the L1, L2 and L3 caches are divided into, respectively. The first lowest 6-bits of the offset are used for addressing the 64 byte long cache lines. The following 6 bits are used to resolve the set addresses in the L1 cache. For the L2 and L3 caches this field is 9 and 11-bits wide, respectively. In this case, a huge page size of 2 MB (21 bit offset) will give the attacker full control of the set occupied by his data in all three levels of cache, i.e. L1, L2 and L3 caches. The significance of targeting last level cache becomes apparent when one considers the access time gap between the last level cache and the memory, which is much more pronounced compared to the access time difference between the L1 and L2 caches. Therefore, using huge pages makes it possible to reach a higher resolution *Prime and Probe* style attack.

6.7 Reverse Engineering the Slice Selection Algorithm

The last inconvenience that we observe when executing *Prime and Probe* in the LLC is the fact that the LLC is divided into slices, whose assignment is decided by an undocumented hash function. This means that the attacker cannot control which slice is targeted. Although knowing the slice selection algorithm implemented is not crucial to run a *Prime and Probe* attack (since we could calculate the eviction set for every set s that we want to monitor), the knowledge of the non-linear slice selection algorithm can save significant time, specially when we have to profile a big number of sets. Indeed, in the attack step, we can select a range of sets/slices $s_1, s_2, ...s_n$ for which, thanks to the knowledge of the non-linear slice selection algorithm, we know that the memory blocks in the eviction set will not change. This section describes the methodology applied to reverse engineer the slice selection algorithm for specific Intel processors. Note that the method can be used to reverse engineer slice selection algorithms for other Intel processors that have not been analyzed in this work. To the best of our knowledge, this slice selection hash function is not public. We solves

the issue by:

- Generating data blocks at slice-colliding addresses to fill a specific set. Access timings are used to determine which data is stored in the same slice.
- Using the addresses of data blocks identified to be co-residing in slices to generate a system of equations. The resulting equation systems can then be solved to identify the slice selection algorithm implemented by the processor.
- Building a scalable tool, i.e, proving its applicability for a wide range of different architectures.

6.7.1 Probing the Last Level Cache

As stated in Section 6.2, the shared last level caches in SMP multicore architectures are usually divided into slices, with an unknown hash function that determines the slice. In order to be able to reverse engineer this hash function, we need to recover addresses of data blocks co-residing in a set of a specific slice. The set where a data block is placed can be controlled, even in the presence of virtual addressing, if huge size pages are used. Recall that by using 2MB huge size pages we gain control over the least significant 21 bits of the physical address, thereby controlling the set in which our blocks of data will reside. Once we have control over the set a data block is placed in, we can try to detect data blocks co-residing in the same slice. Co-residency can be inferred by distinguishing LLC accesses from memory accesses.

6.7.2 Identifying m Data Blocks Co-Residing in a Slice

We need to identify the m memory blocks that fill each one of the slices for a specific set. Note that we still do not know the memory blocks that collide in a specific set. In order to achieve this goal we perform the following steps:

- **Step 1:** Access one memory block b_0 in a set in the LLC
- **Step 2:** Access several additional memory blocks $b_1, b_2, \ldots, b_n$ that reside in the same set, but may reside in a different slice, in order to fill the slice where b_0 resides.
- **Step 3:** Reload the memory block b_0 to check whether it still resides in the last level cache or in the memory. A high reload time indicates that the memory block b_0 has been evicted from the slice, since intel utilizes a Pseudo Last Recently Used (PLRU) cache eviction algorithm. Therefore we know that the required m memory blocks to evict $b0$ from the slice are part of the accessed additional memory blocks $b_1, b_2, \ldots, b_n$.

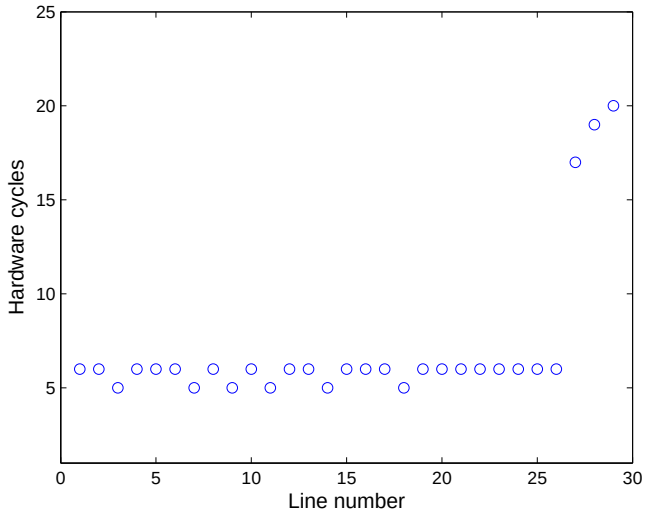


Figure 6.4: Generating additional memory blocks until a high reload value is observed, i.e., the monitored block is evicted from the LLC. The experiments were performed in an Intel i5-3320M.

- **Step 4:** Subtract one of the accessed additional memory blocks b_i and repeat the protocol. If b_0 still resides in memory, b_i does not reside in the same slice. If b_0 resides in the cache, it can be concluded that b_i resides in the same cache slice as b_0.

Steps 2 and 3 can be graphically seen in Figure 6.4, where additional memory blocks are generated until a high reload time is observed, indicating that the monitored block b_0 was evicted from the LLC cache after memory block b_{26} was accessed. Step 4 is also graphically presented in Figure 6.5 where each additional block is checked to see whether it affects the reload time observed in Figure 6.4. If the reload time remains high when one of the blocks b_i is no longer accessed, b_i does not reside in the same slice as the monitored block b_0. In our particular case, we observe that the slice colliding blocks are $b_3, b_4, b_7, b_9, b_{10}, b_{13}, b_{14}, b_{17}, b_{18}, b_{21}, b_{22}$ and b_{24}

6.7.3 Generating Equations Mapping the Slices

Once m memory blocks that fill one of the cache slices have been identified, we generate additional blocks that reside in the same slice to be able to generate more equations. The approach is similar to the previous one:

- Access the m memory blocks $b_0, b_1, \ldots, b_m$ that fill one slice in a set in the LLC

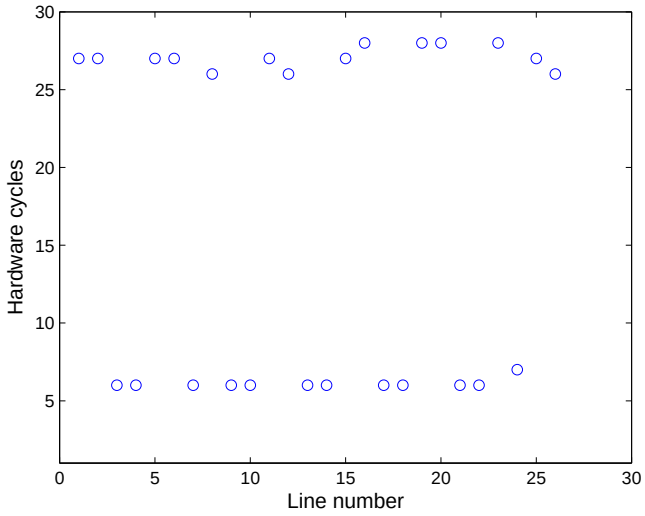


Figure 6.5: Subtracting memory blocks to identify the m blocks mapping to one slice in an Intel i5 3320-M. Low reload values indicate that the line occupies the same slice as the monitored data.

- Access, one at a time, additional memory blocks that reside in the same set, but may reside in a different slice
- Reload the memory block b_0 to check whether it still resides in the LLC or in memory. Again, due to the PLRU algorithm, a high reload time indicates that b_0 has been evicted from the slice. Hence, the additional memory block also resides in the same cache slice.
- Once a sufficiently large group of memory blocks that occupy the same LLC slice has been identified, we get their physical addresses to construct a matrix P_i of equations, where each row is one of the physical addresses mapping to the monitored slice.

The equation generation stage can be observed in Figure 6.6, where 4000 additional memory blocks occupying the same set were generated. Knowing the m blocks that fill one slice, accessing an additional memory block will output a higher reload value if it resides in the same slice as b_0 (since it evicts b_0 from the cache).

Handling Noise: We choose a detection threshold in such a way that we most likely *only* deal with false negatives, which do not affect correctness of the solutions of the equation system. As it can be observed in Figure 6.6, there are still a few values that are not clearly identified (i.e, those with reload values of 10-11 cycles). By simply not considering these, false positives are avoided and the resulting equation system remains correct.

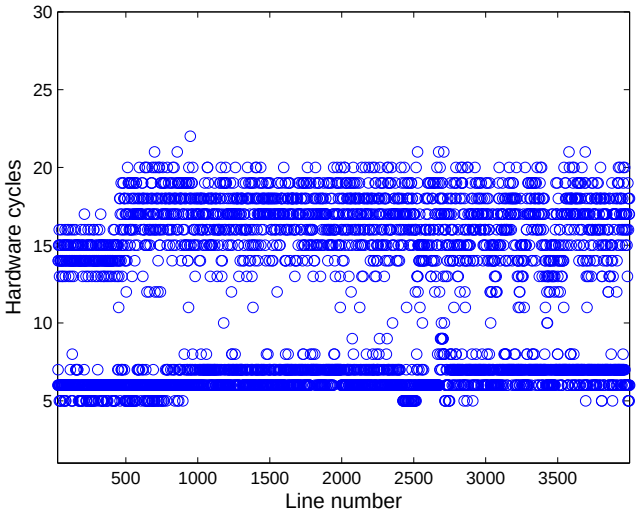


Figure 6.6: Generating equations mapping one slice for 4000 memory blocks in an Intel i5 3320-M. High reload values indicate that the line occupies the same slice as the monitored data.

6.7.4 Recovering Linear Hash Functions

The mapping of a memory block to a specific slice in LLC cache is based on its physical address. A hash function $H(p)$ takes the physical address p as an input and returns the slice the address is mapped to. We know that H maps all possible p to s outputs, where s is the number of slices for the processor.

$$H : \{0,1\}^{\lceil \log_2 p \rceil} \xrightarrow{h} \{0,1\}^{\lceil \log_2 s \rceil}$$

The labeling of these outputs is arbitrary. However, each output should occur with a roughly equal likelihood, so that accesses are balanced over the slices. We model H as a function on the address bits. In fact, as we will see, the observed hash functions are linear in the address bits p_i. In such a case we can model H as a concatenation of linear Boolean functions $H(p) = H_0(p)|| \ldots ||H_{\lceil \log_2 s \rceil - 1}(p)$, where $||$ is concatenation. Then, H_i is given as

$$H_i(p) = h_{i,0}p_0 + h_{i,1}p_1 + \ldots h_{i,l}p_l = \sum_{0}^{l} h_{i,j}p_j.$$

Here, $h_{i,j} \in \{0,1\}$ is a coefficient and p_j is a physical address bit. The steps in the previous subsections provide addresses p mapped to a specific slice, which are combined in a matrix P_i, where each row is a physical address p. The goal is to recover the functions H_i, given as the coefficients $h_{i,j}$. In general, for linear systems,

the H_i can be determined by solving the equations

$$P_i \cdot \hat{H}_i = \hat{0}, \tag{6.1}$$

$$P_i \cdot \hat{H}_i = \hat{1} \tag{6.2}$$

where $\hat{H}_i = (h_{i,0}, h_{i,1}, \ldots, h_{i,l})^T$ is a vector containing all coefficients of H_i. The right hand side is the ith bit of the representation of the respective slice, where $\hat{0}$ and $\hat{1}$ are the all zeros and all ones vectors, respectively. Note that finding a solution to Equation (6.1) is equivalent to finding the kernel (null space) of the matrix P_i. Also note that any linear combination of the vectors in the kernel is also a solution to Equation (6.1), whereas any linear combination of a particular solution to Equation (6.2) and any vector in the kernel is also a particular solution to Equation (6.2). In general:

$$\hat{h} \in \ker P_i \iff P_i \cdot \hat{h} = \hat{0}$$
$$\forall \hat{h_1}, \hat{h_2} \in \ker P_i \quad \hat{h_1} + \hat{h_2} \in \ker P_i$$
$$\forall \hat{h_1}, \hat{h_2} \mid P_i \cdot \hat{h_1} = \hat{1}, \hat{h_2} \in \ker P_i : \quad P_i \cdot (\hat{h_1} + \hat{h_2}) = \hat{1}$$

Recall that each equation system should map to $x = \lceil \log_2(s) \rceil$ bit selection functions H_i. Also note that we cannot infer the labeling of the slices, although the equation system mapping to slice 0 will never output a solution to Equation (6.2). This means that there is more than one possible solution, all of them valid, if the number of slices is greater than 2. In this case, $\binom{2^x-1}{x}$ solutions will satisfy Equation (6.1). However, any combination of x solutions is valid, differing only in the label referring to each slice.

We have only considered linear systems in our explanation. In the case of having a nonlinear system (i.e., the number of slices is not a power of two), the system becomes non-linear and needs to be re-linearized. This can be done by expanding the above matrix P_i with the non-linear terms $P_{i_{linear}} | P_{i_{nonlinear}}$ and solve Equations (6.1) and (6.2). Note that the higher the degree of the non-linear terms, the bigger number of equations that are required to solve the system. For that reason, later in Section 6.7.7 we present an example on an alternative (more intuitive) approach that can be taken to recover non-linear slcie selection algorithms.

This tool can also be useful in cases where the user cannot determine the slice selection algorithm, e.g., due to a too low number of derived equations. Indeed, the first step that this tool implements is generating the memory blocks that co-reside in each of the slices. This information can already be used to mount a side channel attack.

6.7.5 Experiment Setup for Linear Hash Functions

In this section we describe our experiment setup. In order to test the applicability of our tool, we implemented our slice selection algorithm recovery method in a wide

Table 6.1: Comparison of the profiled architectures

Processor	Architecture	LLC size	Associativity	Slices	Sets/slice
Intel i5-650 [i65]	Nehalem	4MB	16	2	2048
Intel i5-3320M [inta]	Ivy Bridge	3MB	12	2	2048
Intel i5-4200M [iM]	Haswell	3MB	12	2	2048
Intel i7-4702M [i74]	Haswell	6MB	12	4	2048
Intel Xeon E5-2609v2 [intb]	Ivy bridge	10MB	20	4	2048
Intel Xeon E5-2640v3 [intc]	Haswell	20MB	20	8	2048

range of different computer architectures. The different architectures on which our tool was tested are listed in Table 6.1, together with the relevant parameters. Our experiments cover a wide range of linear (power of two) slice sizes as well as different architectures.

All architectures except the Intel Xeon e5-2640 v3, were running Ubuntu 12.04 LTS as the operating system, whereas the last one used Ubuntu 14.04. Ubuntu, in root mode, allows the usage of huge size pages. The huge page size in all the processors is 2MB [Lin]. We also use a tool to obtain the physical addresses of the variables used in our code by looking at the `/proc/PID/pagemap` file. In order to obtain the slice selection algorithm, we profiled a single set, i.e, set 0. However, in all the architectures profiled in this paper, we verified that the set did not affect the slice selection algorithm. This might not be true for all Intel processors.

The experiments cover wide selection of architectures, ranging from Nehalem (released in 2008) to Haswell (released in 2013) architectures. The processors include laptop CPUs (i5-3320, i5-4200M and i7-4200M), desktop CPUs (i5-650) and server CPUs (Xeon E5-2609v2, Xeon E5-2640v3), demonstrating the viability of our tool in a wide range of scenarios.

As it can be seen, in the entire set of processors that we analyzed, each slice gets 2048 sets in the L3 cache. Apparently, Intel designs all of its processors in such a way that every slice gets all 2048 sets of the LLC. Indeed, this is not surprising, since it allows to use cache addressing mechanisms independent from the size or the number of cores present in the cache. This also means that 17 bits are required from the physical address to select the set in the last level cache. This is far from the 21 bit freedom that we obtain with the huge size pages.

6.7.6 Results for Linear Hash Functions

Table 6.2 summarizes the slice selection algorithm for all the processors analyzed in this work.

The Intel i650 processor is the oldest one that was analyzed. Indeed, the slice selection algorithm that it implements is much simpler than the rest of them, involving only one single bit to decide between the two slices. This bit is the 17th bit, i.e., the bit consecutive to the last one selecting the set. This means that an

Table 6.2: Slice selection hash function for the profiled architectures

Processor	Architecture	Solutions	Slice selection algorithm
Intel i7-2600 [HWH13]	Sandy Bridge		$p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31}$ $p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{29} \oplus p_{31}$
Intel i650	Nehalem	1	p_{17}
Intel i5-3320M	Ivy Bridge	1	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$
Intel i5-4200M	Haswell	1	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$
Intel i7-4702M	Haswell	3	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
Intel Xeon E5-2609 v2	Ivy Bridge	3	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{18} \oplus p_{19} \oplus p_{21} \oplus p_{23} \oplus p_{25} \oplus p_{27} \oplus p_{29} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
Intel Xeon E5-2640 v2	Haswell	35	$p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32}$ $p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{26} \oplus p_{27} \oplus p_{30} \oplus p_{31}$ $p_{17} \oplus p_{20} \oplus p_{21} \oplus p_{24} \oplus p_{27} \oplus p_{28} \oplus p_{29} \oplus p_{30}$

attacker using *Prime and Probe* techniques can fully control both slices, since all the bits are under his control.

It can be seen that the Intel i5-3320M and Intel i5-4200M processors implement a much more complicated slice selection algorithm than the previous one. Since both processors have 2 slices, our method outputs one single solution, i.e, a single vector in the kernel of the system of equations mapping to the zero slice. Note the substantial change between these processors and the previous one, which evaluate many bits in the hash function. Note that, even though both processors have different microarchitectures and are from different generations, they implement the same slice selection algorithm.

We next focus on the 4 slice processors analyzed, i.e., the Intel i7-4702MQ and the Intel Xeon E5-2609. Again, many upper bits are used by the hash function to select the slice. We obtain 3 kernel vectors for the system of equations mapping to the zero slice (two vectors and the linear combination of them). From the three solutions, any combination of two solutions (one for h0 and one for h1) is a valid solution, i.e, the 4 different slices are represented. However, the labeling of the slices is not known. Therefore, choosing a different solution combination will only affect the labeling of the non-zero slices, which is not important for the scope of the tool. It can also be observed that, even when we compare high end servers (Xeon-2609) and laptops (i7-4702MQ) with different architectures, the slice selection algorithm implemented by both of them is the same. Further note that one of the functions is equal to the one discussed for the two core architectures. We can therefore say that the hash function that selects the slice for the newest architectures only depends on the number of slices.

Finally, we focus on the Intel Xeon E5-2640v3, which divides the last level cache in 8 slices. Note that this is a new high end server, which might be commonly found in public cloud services. In this case, since 8 slices have to be addressed, we need 3 hash functions to map them. The procedure is the same as for the previous

processor: we first identify the set of equations that maps to slice 0 (remember, this will not output any possible solution to 1) by finding its kernel. The kernel gives us 3 possible vectors plus all their linear combinations. As before, any solution that takes a set of 3 vectors will be a valid solution for the equation system, differing only in the labeling of the slices. Note also that some of the solutions have only up to 6 bits of the physical address, making them suitable for side channel attacks.

In summary, we were able to obtain all the slice selection hash functions for all cases. Our results show that the slice selection algorithm was simpler in Nehalem architectures, while newer architectures like Ivy Bridge or Haswell use several bits of the physical address to select the slice. We also observed that the slice selection algorithm mostly depends on the number of slices present, regardless of the type of the analyzed CPU (laptop or high end servers).

6.7.7 Obtaining Non-linear Slice Selection Algorithms

It is important to note that *Prime and Probe* attacks become much simpler when architectures with linear slice selection algorithms are targeted, because the memory blocks to create an eviction set for different set values do not change. This means that we can calculate the eviction set for, e.g, set 0 and the memory blocks will be the same if we profile a different set s. As we will see in this section, this is not true for non-linear slice selection algorithms (where the profiled set also affects the slice selected).

We utilize the Intel Xeon E5-2670 v2 as an example, the most widely used EC2 instance type, which has a 25MB LLC distributed over 10 LLC slices. By just performing some small tests we can clearly observe that the set field affects the slice selection algorithm implemented by the processor. Indeed, it is also clearly observable that the implemented hash function is a *non-linear* function of the address bits, since the 16 memory blocks mapped to the same set in a huge memory page cannot be evenly distributed over 10 slices. Thus we describe the slice selection algorithm as

$$H(p) = h_3(p) \| h_2(p) \| h_1(p) \| h_0(p) \tag{6.3}$$

where each $H(p)$ is a concatenation of 4 different functions corresponding to the 4 necessary bits to represent 10 slices. Note that $H(p)$ will output results from 0000 to 1001 if we label the slices from 0-9. Thus, a non-linear function is needed that excludes outputs 10-15. Further note that p is the physical address and will be represented as a bit string: $p = p_0 p_1 \dots p_{35}$. In order to recover the non-linear hash function implemented by the Intel Xeon E5-2670 v2, we perform experiments in a fully controlled machine featuring the Intel Xeon E5-2670 v2. We first generate ten equation systems based on addresses colliding in the same slice by applying the same methodology explained in Section 6.7.4 and generating up to 100,000 additional memory blocks. We repeat the same process 10 times, changing the primed memory block b_0 in each of them to target a different slice. This outputs 10 different systems

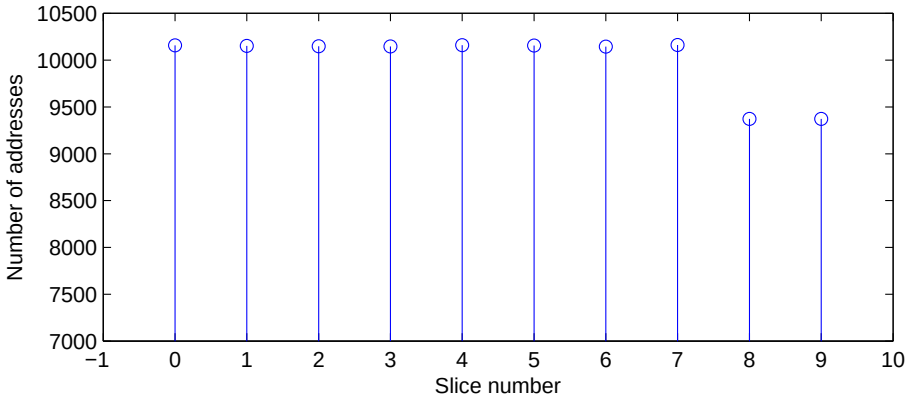


Figure 6.7: Number of addresses that each slice takes out of 100,000. The non-linear slices take less addresses than the linear ones.

of addresses, each one referring to a different slice.

The first important observation we made on the 10 different systems is that 8 of them behave differently from the remaining 2. In 8 of the address systems recovered, if 2 memory blocks *in the same huge memory page* collide in the same slice, they only differ in the 17th bit. This is not true for the remaining two address systems. We suspect, at this point, that the 2 systems behaving differently are the 8th and 9th slice. We will refer to these two slices as the non-linear slices.

Up to this point, one can solve the non-linear function after a re-linearization step given sufficiently many equations. However, one may not be able to recover enough addresses. Recall that the higher the degree of the non-linear term the more equations are needed. In order to keep our analysis simpler, we decided to take a different approach. The second important observation we made is on the distribution of the addresses over the 10 slices. It turns out that the last two slices are mapped to with a lower number of addresses than the remaining 8 slices. Figure 6.7 shows the distribution of the 100,000 addresses over the 10 slices. The different distributions seen for the last two slices give us evidence that a non-linear slice selection function is implemented in the processor. Even further, it can be observed that the linear slices are mapped to by 81.25% of the addresses, while the non-linear slices get only about 18.75%. The proportion is equal to 3/16. We will make use of this uneven distribution later.

We proceed to first solve the first 8 slices and the last 2 slices separately using linear functions. For each we try to find solutions to the equations 6.1 and 6.2. This outputs two sets of a linear solutions both for the first 8 linear slices and the last 2 slices separately.

Given that we can model the slice selection functions separately using linear functions, and given that the distribution is non-uniform, we suspect that the hash function is implemented in two levels. In the first level a non-linear function chooses between either of the 3 linear functions describing the 8 linear slices or the linear

Table 6.3: Hash selection algorithm implemented by the Intel Xeon E5-2670 v2

Vector	**Hash function** $H(p) = h_0(p)\|\neg(nl(p)) \cdot h'_1(p)\|\neg(nl(p)) \cdot h'_2(p)\|nl(p)$
h_0	$p_{18} \oplus p_{19} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{30} \oplus p_{32} \oplus p_{33} \oplus p_{34}$
h'_1	$p_{18} \oplus p_{21} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{30} \oplus p_{31} \oplus p_{32}$
h'_2	$p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{26} \oplus p_{28} \oplus p_{30}$
v_0	$p_9 \oplus p_{14} \oplus p_{15} \oplus p_{19} \oplus p_{21} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{29} \oplus p_{32} \oplus p_{34}$
v_1	$p_7 \oplus p_{12} \oplus p_{13} \oplus p_{17} \oplus p_{19} \oplus p_{22} \oplus p_{23} \oplus p_{24} \oplus p_{25} \oplus p_{27} \oplus p_{31} \oplus p_{32} \oplus p_{33}$
v_2	$p_9 \oplus p_{11} \oplus p_{14} \oplus p_{15} \oplus p_{16} \oplus p_{17} \oplus p_{19} \oplus p_{23} \oplus p_{24} \oplus p_{25} \oplus p_{28} \oplus p_{31} \oplus p_{33} \oplus p_{34}$
v_3	$p_7 \oplus p_{10} \oplus p_{12} \oplus p_{13} \oplus p_{15} \oplus p_{16} \oplus p_{17} \oplus p_{19} \oplus p_{20} \oplus p_{23} \oplus p_{24} \oplus p_{26} \oplus p_{28} \oplus p_{30} \oplus p_{31} \oplus p_{32} \oplus p_{33} \oplus p_{34}$
nl	$v_0 \cdot v_1 \cdot \neg(v_2 \cdot v_3)$

functions describing the 2 non-linear slices. Therefore, we speculate that the 4 bits selecting the slice look like:

$$H(p) = \begin{cases} h_0(p) = h_0(p) \\ h_1(p) = \neg(nl(p)) \cdot h'_1(p) \\ h_2(p) = \neg(nl(p)) \cdot h'_2(p) \\ h_3(p) = nl(p) \end{cases}$$

where h_0,h_1 and h_2 are the hash functions selecting bits 0,1 and 2 respectively, h_3 is the function selecting the 3rd bit and nl is a nonlinear function of an unknown degree. We recall that the proportion of the occurrence of the last two slices is 3/16. To obtain this distribution we need a degree 4 nonlinear function where two inputs are negated, i.e.:

$$nl = v_0 \cdot v_1 \cdot \neg(v_2 \cdot v_3) \tag{6.4}$$

Where nl is 0 for the 8 linear slices and 1 for the 2 non-linear slices. Observe that nl will be 1 with probability 3/16 while it will be zero with probability 13/16, matching the distributions seen in our experiments. Consequently, to find v_0 and v_1 we only have to solve Equation (6.2) for slices 8 and 9 together to obtain a 1 output. To find v_2 and v_3, we first separate those addresses where v_0 and v_1 output 1 for the linear slices $0-7$. For those cases, we solve Equation (6.2) for slices $0-7$. The result is summarized in Table 6.3. We show both the non-linear function vectors v_0, v_1, v_2, v_3 and the linear functions h_0, h_1, h_2. These results describe the behavior of the slice selection algorithm implemented in the Intel Xeon E5-2670 v2. It can be observed that the bits involved in the set selection (bits 6 to 16 for the LLC) are also involved in the slice selection process, unlike with linear selection algorithms. This means that for different sets, different memory blocks will map to the same slice.

Note that the method applied here can be used to reverse engineer other machines that use different non-linear slice selection algorithms. By looking at the distribution of the memory blocks over all the slices, we can always get the shape of the non-linear part of the slice selection algorithm. The rest of the steps are generic, and can be even applied for linear slices selection algorithms.

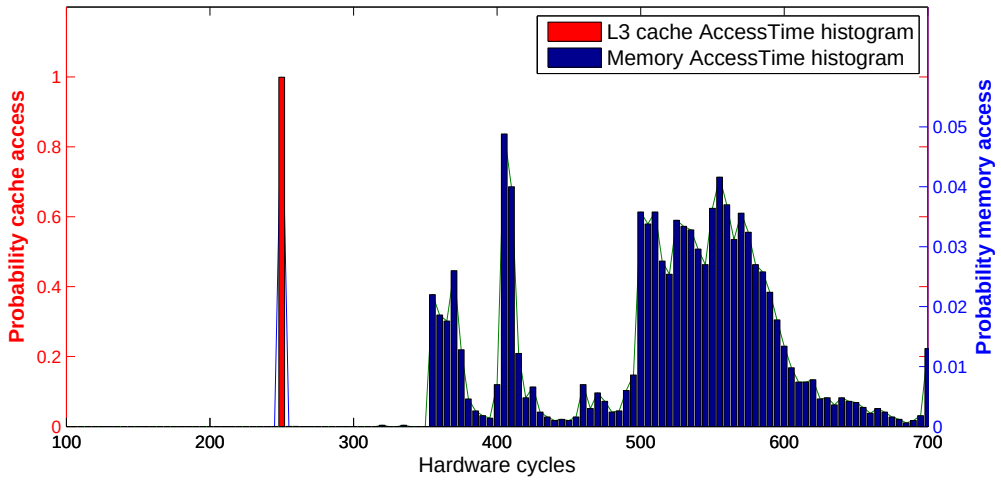


Figure 6.8: Histograms of 10,000 access times in the probe stage when all the lines are in the L3 cache and when all except one are in the cache (and the other one in the memory).

6.8 The LLC *Prime and Probe* Attack Procedure

With the previously discussed challenges solved, we can proceed to apply our LLC *Prime and Probe* attack. Our LLC *Prime and Probe* technique takes advantage of the control of the lower k bits in the virtual address that we gain with the huge size pages and the knowledge of the slice selection algorithm. These are the main steps that our spy process will follow to detect accesses to the last level cache:

- **Step 1 Allocation of huge size pages if available:** The spy process is based on the control that the attacker gains on the virtual address when using huge size pages. Therefore the spy process has to have access to the available huge pages, which requires administrator rights. Recall that this is not a problem in the cloud scenario where the attacker has administrator privileges to his guest OS. If huge size pages are not available, an attacker can start from step 2 at the cost of a more time-consuming eviction set discovery.

- **Step 2 Find eviction-sets:** Due to the slice selection algorithm, an attacker has first to deduce which of his memory blocks collide in the same set-slice. To do this, he can be aided by the slice selection algorithm knowledge acquired previously in Section 6.7.4, speeding up the attack process. If the slice selection algorithm is not know, an attacker can also create eviction sets "on the fly", again at the cost of a more challenging and complicated process.

- **Step 3 Prime the desired set-slice in the last level cache:** In this step the attacker creates data that will occupy one of the sets-slices in the last level

cache. By controlling the virtual address (and, if known, the slice selection algorithm), the attacker knows the set-slice that the created data will occupy in the last level cache. Once sufficiently many blocks are created to occupy the set and slice, the attacker primes the set-slice and ensures it is filled. Typically the last level caches are inclusive. Thus we will not only fill the shared last level cache set but also the corresponding sets in the upper level caches.

- **Step 4: Victim process runs:** After the priming stage, the victim runs the target process. Since one of the sets-slices in the last level cache is already filled, if the targeted process uses the monitored it, one of the primed blocks is going to be evicted. Remember we are priming the last level cache, so evictions will cause memory blocks to reside in the memory. If the monitored set-slice is not used, all the primed blocks are still going to reside in the cache hierarchy after the victim's process execution.

- **Step 5: Probe and measure:** Once the victim's process has finished, the spy process probes the primed memory blocks and measures the time to probe them all. If one or more blocks have been evicted by the targeted process, they will be loaded from the memory and we will see a higher probe time. However if all the blocks still reside in the cache, then we will see a shorter probe time.

The last step can be made more concrete with the experiment results summarized in Figure 6.8. The experiment was performed in native execution (no VM) on an Intel i5-650 that has a 16-way associative last level cache. It can be seen that when all the blocks reside in the last level cache we obtain very precise probe timings with average around 250 cycles and with very little variance. However when one of the blocks is evicted from last level cache and resides in memory, both the access time and the variance are higher. We conclude that both types of accesses are clearly distinguishable.

6.8.1 *Prime and Probe* Applied to AES

In this section we proceed to explain how the *Prime and Probe* spy process can be applied to attack AES. Again, we use the C reference implementation of `OpenSSL1.0.1f` library which uses 4 different T-tables during the AES execution. Recovering one round key is sufficient for AES-128, as the key scheduling is invertible.

We use the last round as our targeted round for convenience. Since the 10^{th} round does not implement the MixColumns operation, the ciphertext directly depends on the T-table position accessed and the last round key. Assume S_i to be the value of the ith byte prior to the last round T-table look up operation. Then the ciphertext byte C_i will be:

$$C_i = T_j[S_i] \oplus K_i^{10} \tag{6.5}$$

where T_j is the corresponding T-table applied to the i^{th} byte and K_i^{10}. It can be observed that if the ciphertext and the T-table positions are known, we can guess the key by a simple XOR operation. We assume the ciphertext to be always known by the attacker. Therefore the attacker will use the *Prime and Probe* spy process to guess the T-table position that has been used in the encryption and consequently, obtain the key. Thus, if the attacker knows which set each of the T-table memory lines occupies, *Prime and Probe* will detect that the set is not accessed 8% of the time, and once he has gotten enough measurements, the key can be obtained from Equation 6.5.

Locating the Set of the T-Tables: The previous description implicitly assumes that the attacker knows the location, i.e. the sets, that each T-table occupies in the shared cache. A simple approach to gain this knowledge is to prime and probe every set in the cache, and analyze the timing behavior for a few random AES encryptions. The T-table based AES implementation leaves a distinctive fingerprint on the cache, as T-table size as well as the access frequency (92% per line per execution) are known. Once the T-tables are detected, the attack can be performed on a single line per table. Nevertheless, this locating process can take a significant amount of time when the number of sets is sufficiently high in the outermost shared cache.

An alternative, more efficient approach is to take advantage of the shared library page alignment that some OSs like Linux implement. Assuming that the victim is not using huge size pages for the encryption process, the shared library is aligned at a 4 KB page boundary. This gives us some information to narrow down the search space, since the lower 12 bits of the virtual address will not be translated. Thus, we know the offset f_i modulo 64 of each T-table memory line and the T-table location process has been reduced by a factor of 64. Furthermore, we only have to locate one T-table memory line per *memory page*, since the remaining table occupies the consecutive sets in the last level cache.

Attack stages: Putting all together, these are the main stages that the we follow to attack AES with *Prime and Probe*

- **Step 1: Last level cache profile stage:** The first stage to perform the attack is to gain knowledge about the structure of the last level cache, the number of slices, and the lines that fill one of the sets in the last level cache.
- **Step 2: T-table set location stage:** The attacker has to know which set in the last level cache each T-table occupies, since these are the sets that need to be primed to obtain the key.
- **Step 3: Measurement stage:** The attacker primes the desired sets-slices, requests encryptions and probes again to check whether the monitored sets have been used or not.

- **Step 4: Key recovery stage:** Finally, the attacker utilizes the measurements taken in Step 3 to derive the last round key used by the AES server.

6.8.2 Experiment Setup and Results for the AES Attack

In this section we analyze our experiment setup and the results obtained in native machine, single VM and in the cross-VM scenarios, specifically with deduplication-free hypervisors where *Flush and Reload* and *Invalidate and Transfer* could not succeed.

6.8.2.1 Testbed Setup

The machine used for all our experiments is a dual core Nehalem Intel i5-650 [inta] clocked at 3.2 GHz. This machine works with 64 byte cache lines and has private 8-way associative L1 and L2 caches of size 2^{15} and 2^{18} bytes, respectively. In contrast, the 16-way associative L3 cache is shared among all the cores and has a size of 2^{22} bytes, divided into two slices. Consequently, the L3 cache will have 2^{12} sets in total. Therefore 6 bits are needed to address the byte address in a cache line and 12 more bits to specify the set in the L3 cache. The huge page size is set to 2 MB, which ensures a set field length of 21 bits that are untouched in the virtual to physical address translation stage. All the guest OSs use Ubuntu 12.04, while the VMMs used in our cloud experiments are Xen 4.1 fully virtualized and VMware ESXI 5.5. Both allow the usage of huge size pages by guest OSs [BDF+03, Xenb, xena]. None of them implements memory deduplication mechanisms; Xen because it lacks of such a feature, VMware because we disabled it manually. Recall that, under this settings, *Flush and Reload* and *Invalidate and Transfer* would not be able to recover any meaningful information. We will observe how this is not the case for *Prime and Probe*.

The target process uses the C reference implementation of `OpenSSL1.0.1f`, which is the default if the library is configured with `no-asm` and `no-hw` options. We would like to remark that these are not the default OpenSSL installation options in most of the products.

The attack scenario is going to be the same one as in Section 4.4, where one process/VM is handling encryption requests with an secret key. The attacker's process/VM is co-located with the encryption server, but on different cores. We assume synchronization with the server, i.e, the attacker starts the *Prime and Probe* spy process and then sends random plaintexts to the encryption server. The communication between encryption server and attacker is carried out via socket connections. Upon the reception of the ciphertext, the attacker measures the L3 cache usage by the *Prime and Probe* spy process. All measurements are taken by the attackers process/VM with the `rdtscp` function, which not only reads the time stamp counters but also ensures that all previous processes have finished before its execution [rdt].

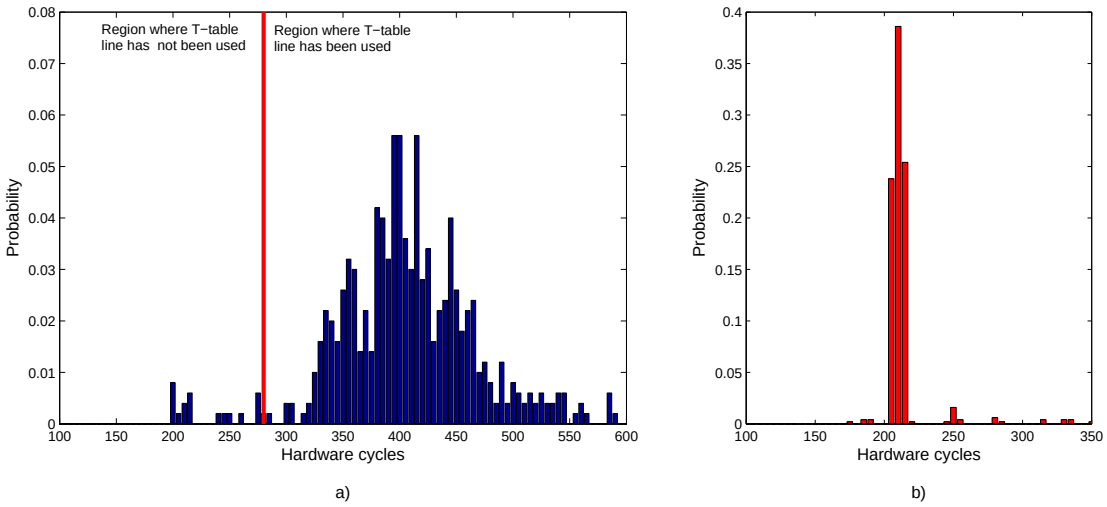


Figure 6.9: Histograms of 500 access times monitored in the probe stage for a) a set used by a T-table memory line and b) a set not used by a T-able memory line. Measurements are taken in the Xen 4.1 cross-VM scenario.

6.8.2.2 The Cross-Core Cross-VM Attack

We perform the attack in three different scenarios: native machine, single VM and cross-VM. In the native and single VM scenarios, we assume that the huge size pages are set to be used by any non-root process running in the OS. Recall that in the cross-VM scenario, the attacker has administrator rights in his own OS.

The first step is to recognize the access pattern of the L3 cache in our Intel i5-650. Making use of the knowledge of the slice selection algorithm in Section 6.7.4, we observe that odd blocks (17th bit of the physical address equal to 1) and even blocks (17th bit of the physical address equal to 0) are allocate in different slices. Thus we need 16 odd blocks to fill a set in the odd slice, whereas we need 16 even blocks to fill a specific set in the even slice.

The second step is to recognize the set that each T-table cache line occupies in the L3 cache. For that purpose we monitor each of the possible sets according to the offset obtained from the linux shared library alignment feature. Recall that if the offset modulo 64 f_0 of one of the T-tables is known, we only need check the sets that are 64 positions apart, starting from f_0. By sending random plaintexts the set holding a T-table cache line is used around 90% of the times, while around 10% of the times the set will remain unused. The difference between a set allocating a T-table cache line and a set not allocating a T-table cache line can be graphically seen in Figure 6.9, where 500 random encryptions were monitored with *Prime and Probe* for both cases in a cross-VM scenario in Xen 4.1. It can be observed that monitoring an unused set results in more stable timings in the range of 200-300 cycles. However

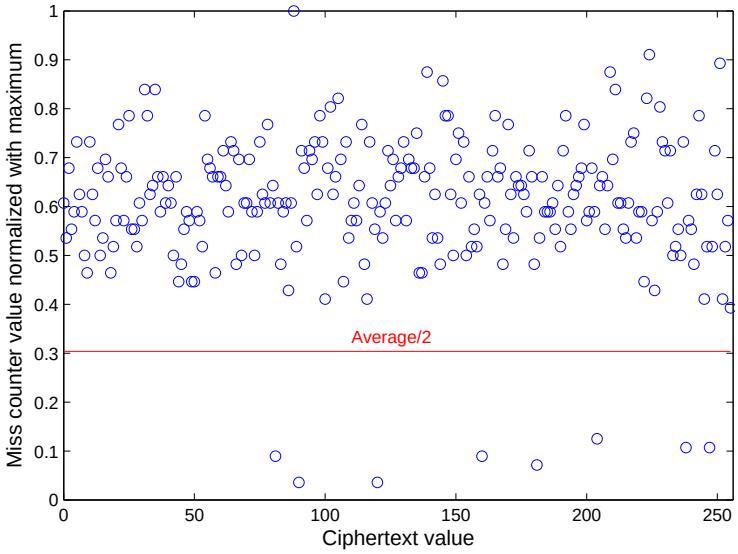


Figure 6.10: Miss counter values for ciphertext 0 normalized to the maximum value. The key is e1 and we are monitoring the last 8 values of the T-table (since the table starts in the middle of a memory line).

monitoring a set used by the T-tables outputs higher time values around 90% of the time, whereas we still see some lower time values below 300 around 10% of the times. Note that the key used by the AES server is irrelevant in this step, since the set used by the T-table cache lines is going to be independent of the key.

The last step is to run *Prime and Probe* to recover the AES key used by the AES server. We consider as valid ciphertexts for the key recovery step those that are at least below half the average of the overall timings. This threshold is based on empirical results that can be seen in Figure 6.10, and is calculated as the overall counter average divided by 2. The figure presents the miss counter value for all the possible ciphertext values of C_0, when the last line in the corresponding T-table is monitored. The key in this case is $0xe1$ and the measurements are taken in a cross-VM scenario in Xen 4.1. In this case only 8 values take low miss counter values because the T-table finishes in the middle of a cache line. These values are clearly distinguishable from the rest and appear in opposite sides of the empirical threshold.

Results for the three scenarios are presented in Figure 6.11, where it can be observed that the noisier the scenario is, e.g. in the cross-VM scenario, the more monitored encryptions are needed to recover the key. The plot shows the number of correctly guessed key bytes vs. the number of encryptions needed. Recall that the maximum number of correctly guessed key bytes is 16 for AES-128. The attack only needs 150.000 encryptions to succeed on recovering the full AES key in the native OS scenario. Due to the higher noise in the cloud setting, the single VM recovers

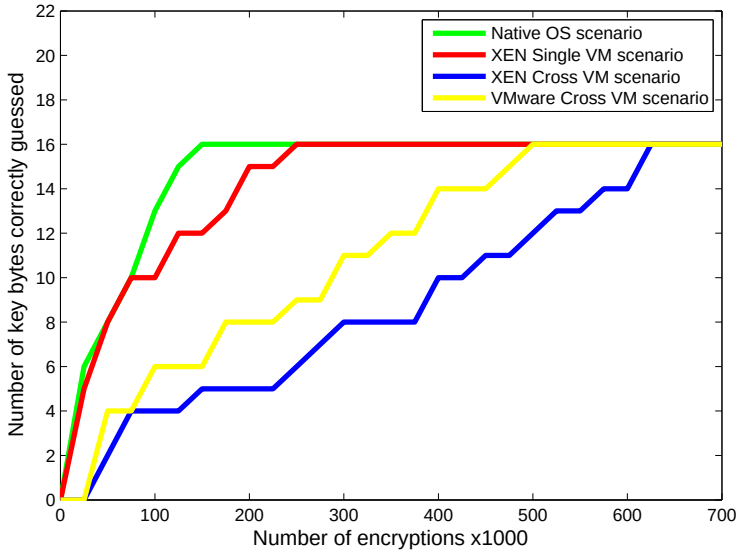


Figure 6.11: Number of key bytes correctly recovered vs number of encryptions needed for native OS, single VM and cross-VM scenarios.

the full key with 250.000 encryptions. The cross-VM scenario was analyzed in two popular hypervisors, Xen and VMware, requiring 650.000 and 500.000 encryptions to recover the 16 key bytes respectively. We believe that Xen requires a higher number of encryptions due to the higher noise caused by the usage of a fully virtualized hypervisor. It is important to remark that the attack is completed in only 9 and 35 seconds, respectively, for the native and single VM scenarios. In the cross VM scenario, the attack succeeds in recovering the full key in 90 and 150 seconds in VMware and Xen, respectively. Recall that in the cross-VM scenario the external IP communication adds significant latency.

In short, compared to the *Flush and Reload* attack earlier presented, the *Prime and Probe* attack needs more encryptions to succeed in recovering the key. This is an expected behavior, as the *Prime and Probe* attack suffers more from noise. Indeed, *Flush and Reload* needs w accesses from an unrelated process to create a noisy measurement, being w the associativity of the cache. However, as the *Prime and Probe* attack fills the whole set, a simple access from an unrelated process to the set being primed create a noisy measurement. Nevertheless, *Prime and Probe* succeeds to recover the key in hypervisors where *Flush and Reload* and *Invalidate and Transfer* cannot, including commercial clouds, which do not have memory deduplication enabled. We present a full real world scenario attack using *Prime and Probe* in the next section.

6.9 Recovering RSA Keys in Amazon EC2

As we said, the *Prime and Probe* attack does not assume any special requirements but that attacker and victim are co-resident in the same CPU socket. Thus, there is no reason why it cannot succeed in commercial clouds, like Amazon EC2. Amazon EC2 mostly uses Intel-based servers, for wich we know that the LLC is inclusive. More than that, the market share that Intel has on desktop and server processors was more than 80% at the beginning of 2016 [Alp09], and since Intel does not seem to offer non-inclusive caches in these devices (at least we have not observed any), our attack would work in big amount of current computers. In the cloud scenario, of course, co-residency has first to be achieved with the target. We assume co-residency is achieved (in fact, with the methodologies described in [VZRS15, IGES16]), and that the attacker utilizes the same hardware resources as a RSA decryption server.

To prove the viability of the *Prime and Probe* attack in Amazon EC2 across co-located VMs, we introduce how it can be utilized to steal RSA cryptographic keys. It is important to remark that the attack *is not* processor specific, and can be implemented in any processor with inclusive last level caches.

The attack targets a sliding window implementation of RSA-2048. Note that attacking such an implementation is far more difficult than the one described in Section 5.4, since the multiplication function by itself does not give us enough information about the key. In this case, contrary to the case of AES and as explained in Section 2.4.2, we put our focus in the multiplicants, which are dynamically allocated. Thus *Flush and Reload*, even in deduplication enabled systems would be able to recover such information.

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 7. The message blinding process is performed as a side channel countermeasure for `chosen-ciphertext` attacks, in response to studies such as [GST14, GPPT15]. However, note that this does not prevent our attack, as we only focuses on the key management without requiring any particular ciphertext shape. Further, the CRT implementations makes us have to recover d_p and d_q separately, but once the knowledge of both is acquired, the full key can be retrieved [HDWH12, Ham13].

The modular exponentiation in Libgcrypt uses the sliding window approach explained in Algorithm 8. In particular, libgcrypt pre-computes the values $c^3, c^5, c^7, \ldots, c^{2^W-1}$ in a table. Then, the key is processed in windows that are required to start and finish with a set bit, and have a maximum length of W, being W the window size. Until the set bits are found, squares are issued normally. When the set bits and the window of length l ($l <= W$) is found, l squares are issued and a multiplication with the appropriate table entry is performed. Clearly, the accesses to the precomputed table leak information about the window being processed, and therefore, the key bit values of the windows.

Our attack uses the *Prime and 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 is the

Algorithm 7 RSA with CRT and Message Blinding

Input : Ciphertext $c \in \mathbb{Z}_N$, Exponents d, e, Modulus $N = pq$
Output: m

$r \xleftarrow{\$} Z_N$ with $\gcd(r, N) =$
1//`Message Blinding`
$c^* = c \cdot r^e \mod N$ $d_p = d \mod (p-1)$
//`CRT conversion`
$d_q = d \mod (q-1)$ $m_1 = (c^*)^{d_p}$
$\mod p$//`Modular Exponentiation`
$m_2 = (c^*)^{d_q} \mod q$ $h = q^{-1} \cdot (m_1 - m_2)$
$\mod p$//`Undo CRT`
$m^* = m_2 + h \cdot q$ $m = m^* \cdot r^{-1} \mod N$
//`Undo Blinding`
return m

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. 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 [Fan15] 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 noise.

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, as it is dynamically allocated. 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 offset 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 utilized by Amazon EC2, the LLC is divided in 2048 sets and 10 slices. Therefore, knowing the lowest

Algorithm 8 RSA Sliding-Window Exponentiation

Input : Ciphertext $c \in \mathbb{Z}_N\%$, Exponent d, Window Size w
Output: $c^d \mod N$
//`Table Precomputation step`
$T[0] = c^3 \mod N$ $v = c^2 \mod N$ **for** i ***from*** 1 ***to*** $2^{w-1} - 1$ **do**
  $T[i] = T[i-1] \cdot v \mod N$
end
//`Exponentiation step:`
$b = 1, j = len(d)$ **while** $j > 0$ **do**
  if $e_j == 0$ **then**
    $b = b^2 \mod N$ $j = j - 1$
  else
    Find $e_j e_{j-1} \dots e_l \mid j - l + 1 \leq w$ with $e_l = 1$ **for** k ***from*** 1 ***to*** l **do**
      $b = b^{2^{j-l+1}} \mod N$
    end
    if $e_j == 1$ **then**
      $b = b \cdot c \mod N$
    else
      $b = b \cdot T[(e_i - 3)/2] \mod N$
    end
    $j = j - l - 1$
  end
end
return b

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.

Recall that we reverse engineered the slice selection algorithm for Intel Xeon E5-2670 v2 processors in Section 6.7.7. 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. Thanks to the knowledge of the non linear slice selection algorithm, we will select a range of sets/slices $s_1, s_2, \dots, s_n$ for which the memory blocks that create the eviction sets do not change, and that allow us to profile all the precomputed table entries. 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 [IGES16]. 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.

Among the 4,000 observations for each monitored set, 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 6.12(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 6.12 shows measurements where no correct pattern was detected (Fig. 6.12(a)), and where a correct pattern was measured (Fig. 6.12(b)).

In general, after the elimination step, there are $8-12$ correct traces left per 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 6.13(a) shows the table

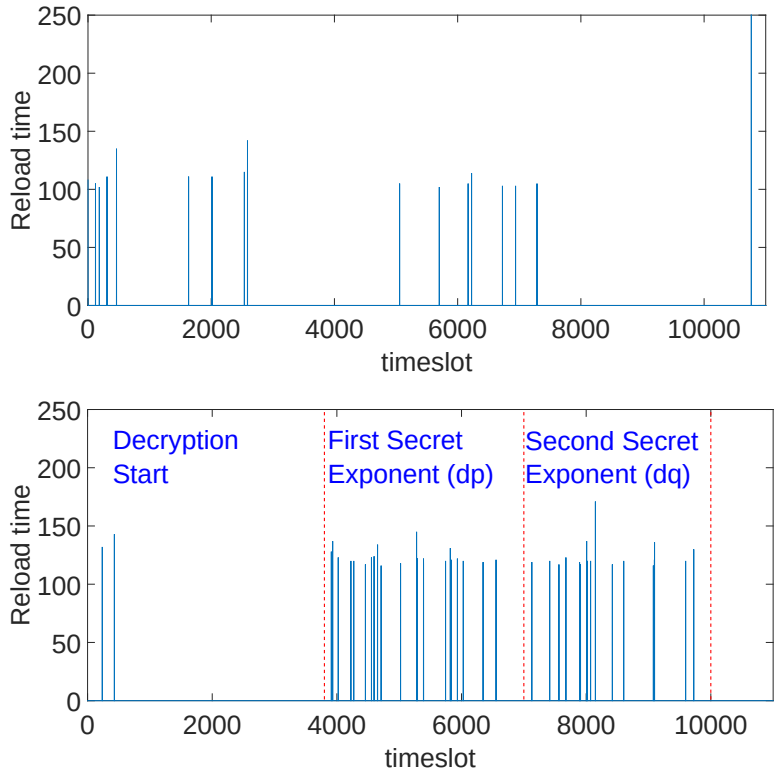


Figure 6.12: Different sets of data where we find a) trace that does not contain information b) trace that contains information about the key

access position indexes aligned with the table creation. In the figure, the top graph shows the true 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.

In consequence, we apply alignment and noise reduction techniques, typically utilized in DPA attacks [KJJ99] to obtain a clean cache trace. The result after such an alignment process can be observed in Figure 6.14, which shows the real indices of a particular table entry with those retrieved from our analysis.

Despite the good results after these noise reduction techniques, we still do not end up with a perfect trace. The overall results are presented in Table 6.4, where a 0.65% of the peaks are missdetected. Thus, the key recovery algorithm described in [Ham13] is applied to establish relationships between noisy d_p and d_q to recover a full clean RSA key. In order to apply such an algorithm we need information about the public key. We distinguish two different scenarios in which the attacker is able to utilize public keys for the error correction algorithm. In both, we assume the attacker has already retrieved the leakage from the RSA decryption key of a (potentially known) server:

- **Targeted Co-location: The Public Key is Known** In this case we assume

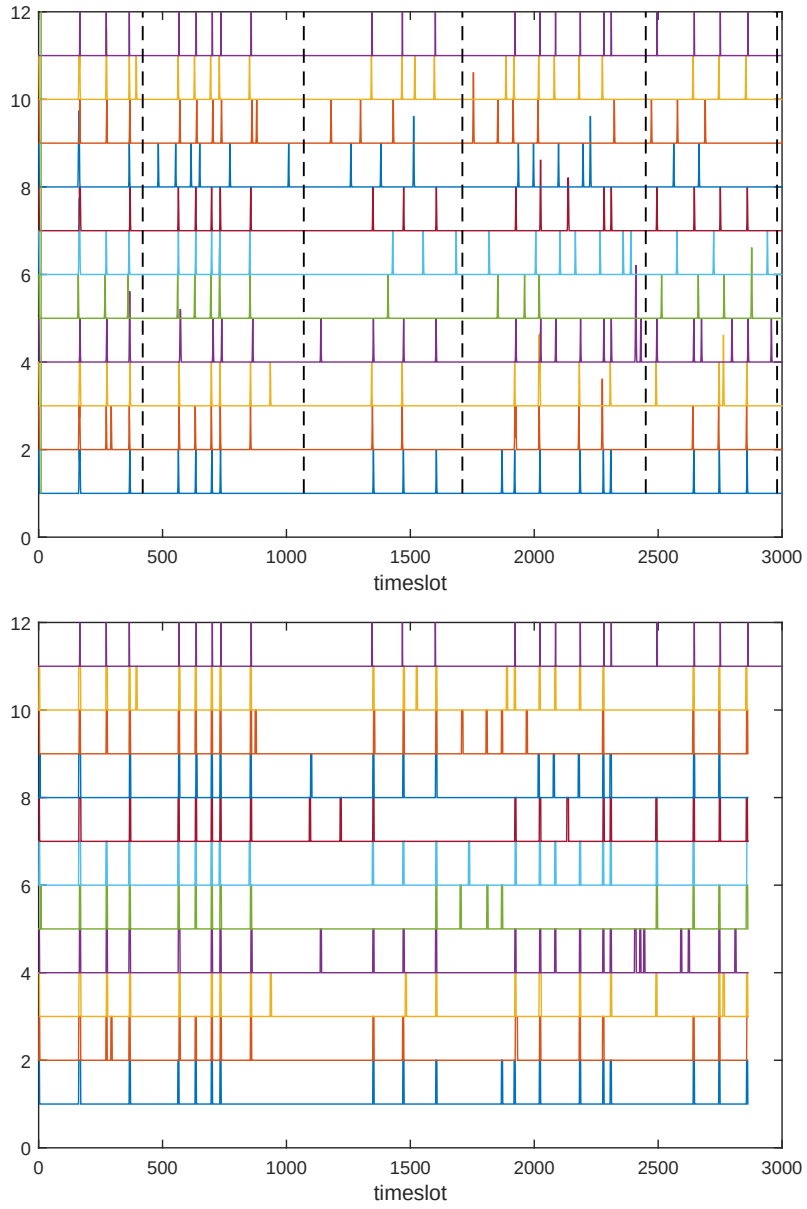


Figure 6.13: 10 traces from the same set where a) they are divided into blocks for a correlation alignment process b) they have been aligned and the peaks can be extracted

that the attacker implemented a targeted co-location against a known server, and that she has enough information about the public key parameters of the target.

- **Bulk Key Recovery: The Public Key is Unknown** In this scenario, the attacker can build up a database of public keys by mapping the entire IP range

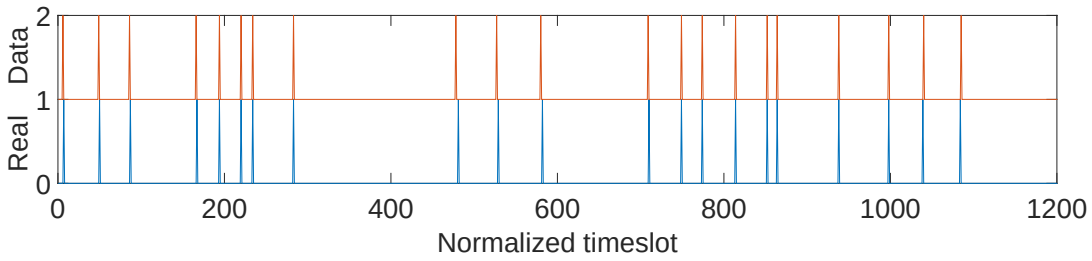


Figure 6.14: Comparison of the final obtained peaks with the correct peaks with adjusted timeslot resolution

Table 6.4: 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%

of the targeted Amazon EC2 region and retrieve all the public keys of hosts that have the TLS port open. The attacker then runs the above described algorithm for each of the recovered private keys and the entire public key database. Having the list of 'neighboring' IPs with an open TLS port also allows the attacker to initiate TLS handshakes to make the servers use their private keys with high frequency.

In the first scenario, the attacker has information about the public key and thus he can apply the error correction algorithm that we will describe directly. In the second scenario, the attacker has a database of public keys, and he does not know which public key the private key leakage belongs to. In this case, the attacker implements the error correction algorithm with each of the public keys in the database until he finds a successful correlation. We proceed next to explain the algorithm to recover the noise-free decryption key.

The leakage analysis described recovers information on the CRT version of the secret exponent d, namely $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$. A noise-free version of either one can be used to trivially recover the factorization of $N = pq$, since $\gcd(m - m^{ed_p}, N) = p$ for virtually any m [CS04]. Thus, our goal is to retrieve noise-free d_p or d_q from noisy $d*_p$ and $d*_q$. Note that this is not our case, as even with alignment and noise filtering techniques we still had miss-detected and wrongly detected peaks. In such cases, we can exploit their relation to the known public key can if the used public exponent e is small [Ham13]. Note that this will be our case, as almost all RSA implementations currently use $e = 2^{16} + 1$ due to the heavy

performance boost over a random and full size e. For CRT exponents we calculated d_p and d_q as $d_p = d \mod (p-1)$ and $d_q = d \mod (q-1)$. Multiplying both sides by e we obtain

$$ed_p = 1 \mod (p-1) \tag{6.6}$$

$$ed_q = 1 \mod (q-1) \tag{6.7}$$

As $d * e \mod ((p-1) * (q-1)) = 1$. This means that exist integers k_p and k_q such that $ed_p = k_p(p-1) + 1$ for some $1 \leq k_p < e$ and $ed_q = k_q(q-1) + 1$ for some $1 \leq k_q < e$. This yields $k_p p = ed_p + k_p - 1$ and $k_q p = ed_q + k_q - 1$. If we multiply this equations together we obtain the main equation we will utilize for our noise correction error algorithm:

$$k_p k_q N = (ed_p + k_p - 1)(ed_q + k_q - 1). \tag{6.8}$$

Note that there are nice tricks that help solving this equation. First, we can take the equation modulo e such that the terms $e * d_p$ and $e * d_q$ disappear, giving us the equation $k_p k_q N = (k_p - 1)(k_q - 1) \pmod{e}$. As e is usually $2^{16} + 1$ and $1 \leq k_p < e$ we can obtain k_p and k_q with this equation considering all possible 2^{16} set of k_p (and their corresponding values k_q).

Next, assume we are given the first t bits of d_p and d_q, e.g. $a = d_p \mod 2^t$ and $b = d_q \mod 2^t$. For each k_p we check whether $\delta(a, b, t) = 0$ where

$$\delta(a, b, t) = k_p k_q N - (ea + k_p - 1)(eb + k_q - 1) \pmod{2^t} \tag{6.9}$$

which indeed will indicate us whether for a choice of k_p, the least-significant t bits of d_p, d_q are correct. We can

- **Check parts** of d_p and d_q by verifying if the test $\delta(d_p(t), d_q(t), t) = 0$ holds for $t \in [1, \lceil \log(p) \rceil]$.
- **Fix alignment and minor errors** by shifting and varying $d_p(t)$ and $d_q(t)$, and then sieving working cases by checking if $\delta(d_p(t), d_q(t), t) = 0$,
- **Recover parts** of d_q given d_p (and vice versa) by solving the error equation $\delta(d_p(t), d_q(t), t) = 0$ in case the data is missing or too noisy to correct.

Note that the algorithm may need to try all 2^{16} values of k_p in a loop. Further, in the last case where we recover a missing data part using the checking equation we need to speculatively continue the iteration for a few more steps to see whether this leads to many mistakes in the information we recovered of d_p and d_q. If we observe too many discrepancies between the solution given by the Equation 6.9 and the recovered key information for a given value t we may early terminate the execution thread without having to iterate over the entire size end of d_p and d_q.

Algorithm 9 Windowed RSA Key Recovery with Noise

Input : Cache noisy window information $w_p[0], \ldots, w_p[n]$ and $w_q[0], \ldots, w_q[n]$, public key e modulus N

Output: Noise free d_p and d_q

for k_p ***from*** 1 ***to*** $e-1$ **do**
  Compute $k_q = (1-k_p)(k_pN - k_p + 1)^{-1} \pmod{e}$;
  while $i < |wp|$ **do**
    Process windows $wp[i], wp[i+1]$;
    Introduce shifts; vary $ip[i]$] up max_{zeros};
    for *each* d_p *variation* **do**
      Compute $X = \sum_{j=0}^{i+1} wp[j]2^{ip[j]}$;
      Identify wq that overlap with $wp[i], wp[i+1]$;
      Compute $Y = \sum_{j=0}^{i+1} wq[j]2^{iq[j]}$;
      if $\delta(X, Y, t)$=*0* **then**
        Update wp, ip, wq, iq;
        Create thread for $i+1$;
      end
      if *if no check succeeded* **then**
        Too many failures: abandon thread;
      end
      if max_{zeros} *achieved* **then**
        $i = i - 1$;
      end
      Update ip, wq, iq;
      Create thread for i;
    end
  end
end
return correct d_p and d_q;

To see how this approach can be adapted into our setting, we need to consider the error distribution observed in d_p and d_q as recovered by cache timing. Furthermore, since the sliding window algorithm was used in the RSA exponentiation operation, we are dealing with variable size (1-5 bit) *windows* with contents wp, wq, and window positions ip, iq for d_p and d_q, respectively.

The windows are separated by 0 strings. We observed:

- The window wp contents for d_p had no errors and were in the correct order. There were slight misalignments in the window positions ip with extra or missing zeros in between.
- In contrast, d_q had not only alignment problems but also few windows with incorrect content, extra windows, and missing windows (overwritten by zeros). The missing windows were detectable since we do not expect unusually long zero strings in a random d_q.
- Since the iterations proceed from the most significant windows to the least we observed more errors towards the least significant words, especially in d_q.

Algorithm 9 shows how one can progressively error correct d_p and d_q by processing groups of consecutive ℓ windows of d_p. For each guess of k_p, the algorithm creates new execution threads and compares the windows with those recovered from the cache measurements. The algorithm kills a thread when too many checks fail to produce any matching on different windows. However, then the kill threshold has to be increased and the depth of the computation threads and more importantly the number of variations that need to be tested increases significantly. In this case, the algorithm finds the correct private key in the order of microseconds for a noise-free d_p and needs 4 seconds for our recovered d_p.

In the South America Amazon EC2 region, we have found 36000+ IP addresses with the TLS port open using `nmap`. We used such a region to improve our co-location chances, as the number of servers is lower compared to the US region. With a public key database of that size, our algorithm takes between less than a second (for noise-free d_ps) and 30 CPU hours (noisy d_ps) to check each private key with the public key database. This approach recovers the public/private key pair, and consequently, the identity of the key owner.

6.10 *Prime and Probe* Outcomes

This chapter introduced *Prime and Probe*, an attack that is memory deduplication agnostic and that is applicable in those systems where *Flush and Reload* and *Invalidate and Transfer* failed, e.g., commercial clouds like Amazon EC2. *Prime and Probe* works by creating set collisions in the LLC, overcoming several difficulties inherited to LLC like slices or the lack of knowledge of sufficient physical bits. We utilized reverse engineering tools and huge pages to overcome this issues, and

demonstrated its applicability by recovering AES and RSA keys. In systems where both *Flush and Reload* and *Prime and Probe* can be applied, the first will give a cleaner channel, as it is more resistant to LLC noise.

Chapter 7

Countermeasures

In the previous chapters we covered the effectiveness of three cache based side channel attacks, *Flush and Reload*, *Invalidate and Transfer* and *Prime and Probe* to recover sensitive information from co-resident users in a number of scenarios. All have distinct requirements that make them applicable in very different scenarios. For instance, we observed how *Flush and Reload* and *Invalidate and Transfer* succeed in systems with memory deduplication features, as it is the case of most PaaS clouds, some IaaS clouds and smartphones. *Prime and Probe* does not require any special assumption but CPU co-residency, and thus, it is not only applicable in those scenarios in which *Flush and Reload* is, but also can be executed in every IaaS cloud, as Javascript extensions [OKSK15] or in TEEs (e.g. SGX or ARM Trustzone) [SWG+17, BMD+17]. Observe how some of these scenarios belong to the category of "'everyday use technology". Therefore, the aforementioned cache attacks are applicable in realistic scenarios and in response, appropriate countermeasures have to be taken to defend against them.

Indeed, despite the big popularity that cross-core microarchitectural attacks acquired, countermeasures are not being deployed in commercial software. To the best of our knowledge, neither commercial clouds, OS designers, software developers nor hardware manufacturers are developing countermeasures against them. In this chapter, we propose two novel countermeasures that can aid all these agents prevent cache attacks in particular (and microarchitectural attacks in general) from succeeding. We first revisit countermeasures that have been proposed by other works to avoid the leakage exploitation through the LLC, categorizing them as hardware, software and application layer countermeasures. Next, we present a LLC leakage finder tool for cryptographic libraries, that evaluates whether cryptographic algorithms are designed properly and that substantially improves on previous approaches, demonstrated by the 5 CVE numbers (and at least 4 more being studied) that we obtained. Finally, we develop a static microarchitectural malware analyzer that can be utilized by online repository distributors verify the sanity of the binaries being offered (which remain undetected to the best antivirus software) before a end-user gets to execute them.

7.1 Existing Countermeasures

Alongside ever improving cache and microarchitectural attacks, countermeasures have been proposed in literature as well. We divide them in three basic categories: hardware, software and application layer countermeasures.

Page [Pag05] proposed a hardware countermeasure that partitions the cache such that cache conflicts between different users do not occur. Wang et al. [WL07] proposed the design of new caches with different characteristics, which involve the ability of locking security-related ways in the cache or enabling cache interference randomization. These implementations were later improved by Kong et al. [KASZ09]. Although previous approaches might give solution to *Prime and Probe* based attacks, it is still an open question as to how these solutions would prevent *Flush and Reload* and *Invalidate and Transfer* attacks. Recently, Intel added the Cache Allocation Technology (CAT) feature to their CPUs, which was investigated by Liu et al. [LGY+16] as a countermeasure to *Prime and Probe* attacks, while considering deduplication-free systems to defeat *Flush and Reload* and *Invalidate and Transfer*.

Further mechanisms have been proposed at the software level. For instance, Page Coloring was proposed by Kim et al. [KPMR12] as a solution in which each user gets pages that do no collide with others in cache. Later Zhang et al. [ZZL16] expanded the work by Chiapetta et al. [CSY15] and proposed a cache attack detector mechanism, cloud radar, which monitors the performance counters looking for patterns observed in cache attacks. Recently, Zhang et al. [ZRZ16] developed a smart access-based memory page management to defeat both *Prime and Probe* and *Flush and Reload*.

Another approach is the design of code that does not exhibit cache leakage at the application level. Almeida et al. [ABB+16] developed a compiler based methodology to verify constant-time code, which for instance prevents attacks on the execution flow via the instruction cache. Zankl et al. [ZHS16] proposed a tool based on dynamic binary instrumentation that automatically detects instruction cache leaks in modular exponentiation software.

In the following we provide a more detailed explanation of two of such countermeasures, mainly page coloring and cloud radar. The goal is to help the reader understand some of the existing countermeasure functionalities.

7.1.1 Page Coloring

As explained in Section 6.1, the location that a memory block will occupy in the cache directly depends on some of its physical address bits. The main idea behind page coloring is that the OS controls the assignment of those bits such that memory blocks belonging to different users/processes/VMs do not collide in the cache. Page Coloring was implemented in [KPMR12] as a mechanisms to avoid cache leakages.

The OS/hypervisor colors DRAM pages by assigning a different color to each different bit combination for the bits that select the position of a memory block in

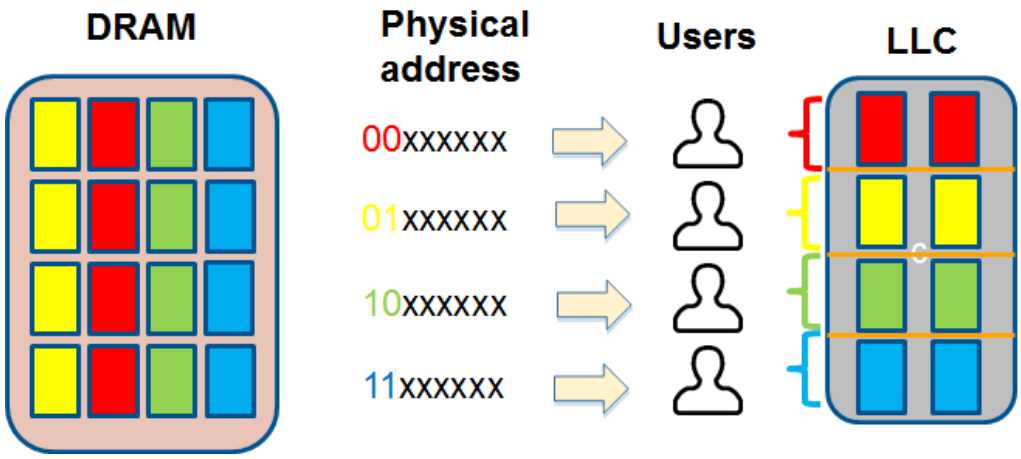


Figure 7.1: Page Coloring implementation, where DRAM is divided into different colors that are assigned to different users that will not collide in the cache

the cache. Figure 7.1 shows the main idea in which the cache is partitioned in 4 different colors, and where 2 bits of the physical address decide the color. Upon DRAM allocation requests made by processes/users, the OS assigns each user *only* pages of one color, and different to any other user in the system. As bits 12 and 13 of the physical address are also involved in the selection of the set that a memory block occupies, the coloring ensures that pages belonging to different users do *not collide in the cache*. Figure 7.1 shows the complete process and how users do not collide in the cache utilization. If memory pages do not collide, *Prime and Probe* like attacks are not possible, as the attacker cannot evict the victims data. Further, page coloring and memory deduplication are antagonic concepts, and therefore neither *Flush and Reload* nor *Invalidate and Transfer* would be applicable if the first is implemented. However, page coloring might be difficult to implement in systems with inconsistent loads, as the number of colors would have to be re-calculated for different loads.

7.1.2 Performance Event Monitorization to Detect Cache Attacks

The LLC attacks that we (and other concurrent works) exploited take advantage of very specific characteristics in the cache, and implement very specific memory accesses. Thus, a mechanisms that constantly monitors the LLC trying to detect malicious utilization would most likely succeed on detecting LLC attacks. Zhang et al. [ZZL16] proposed such a mechanism, in which performance counters are utilized to monitor the hardware usage.

This mechanisms is implemented at the OS or hypervisor level, depending on the system to defend. In particular, very specific hardware events are monitored

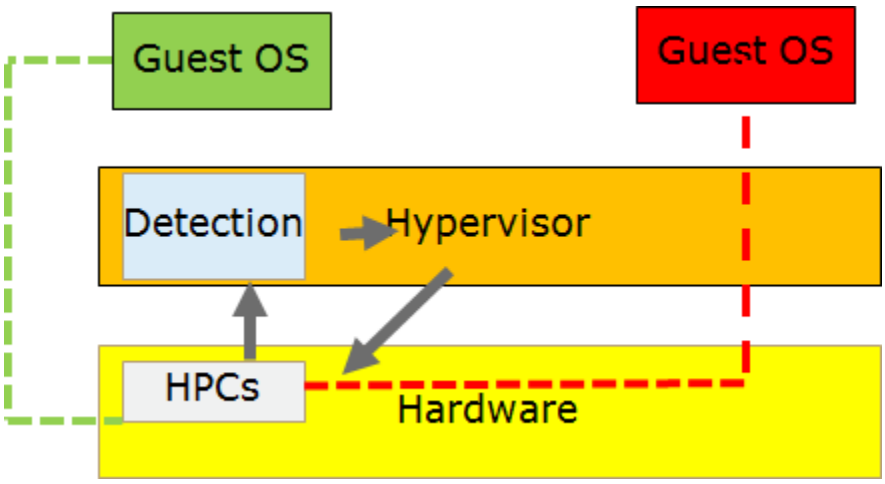


Figure 7.2: HPCs as a methodology to find whether cache attacks are being executed by the processes in the system

looking for cache attack patterns. Obviously, the most affected hardware events are cache misses and hits, but additional perhaps less obvious hardware events are also monitored. In particular, this mechanisms usually do not constantly monitor every process in the system, but monitor random processes for a specific period of time. The detection system can also be aided by machine learning algorithms that are beforehand trained. When a cache attack evidence is observed, the process can be either isolated or prevented from being executed again. Figure 7.2 represents graphically the process. The main downside of this kind of mechanisms is the big overhead they impose to the defending system.

7.2 Problems with Previously Existing Countermeasures

Even though the aforementioned studies have investigated solutions to stop the exploitation of hardware caches as a covert channel, we observe some concerns that might explain why these have not been adopted by commercial products.

- **Application level countermeasures:** One of the main problems that we observe in previously proposed application level countermeasures is the high number of false positives that they usually exhibit. In fact, cryptographic algorithms usually make secret dependent memory accesses, but this does not mean they leak information about the key, as they might happen outside the main loop or within the same cache line. Furthermore, some of the proposed countermeasures do not take into account the entire memory leakage spectrum but only a subset of it. For instance, Zankl et al. [ZHS16] utilize the

PIN instrumentation tool to find distinct *instruction* execution flow (static memory) for different keys, and correlate it with the key. Extracting the same information for dynamically allocated data is not an option, as their memory addresses change from execution to execution.

- **OS level countermeasures:** Although some of the proposed OS level countermeasures would indeed prevent *any* leakage stemming from cache covert channels, we have not observed any of them being implemented in commercial products. We believe there are several reasons preventing their adoption, the main one being the performance overhead associated with these countermeasures. Further, the balanced design of such countermeasures is not an easy task. For instance, page coloring might lead to a miss-utilization of the cache when sufficient workloads are not being executed, while it might lead to leakage exploitation when the number of workloads is higher than the number of colors. We hardly believe these facts take a key role on the decision of the implementation of OS level countermeasures.

- **Hardware level countermeasures:** Among our three countermeasure categories, we believe the most difficult one to be adopted are the hardware level countermeasures. These involve re-designing the hardware, testing it against microarchitectural attack vectors and producing a full new processor family. This entire process usually takes several years, therefore making them unlikely to be utilizable in the short run.

The problems inherited to OS/hardware countermeasures and their non-adoption leaves any end user or application designer alone in the response to microarchitectural attacks in general and cache attacks in particular. In such a situation, the only solution is to create effective and easily implementable application level countermeasures. In the following sections, we aim at giving response to such concerns; first, by proposing a leakage sanity verification tool for cryptographic libraries that can help at achieving leakage free code design, second, by proposing a static analysis tool to be utilized by online binary distributors to ensure they deliver microarchitectural attack free applications. Note that even though our countermeasures are intended to be mainly used by specific agents (i.e. cryptographic code designers and online repository distributors), they can also be utilized by any end user willing to improve the security of his system.

7.3 Detecting Cache Leakages at the Source Code

We demonstrated how LLC attacks can recover a wide range of information, including AES and RSA cryptographic keys. Furthermore, we demonstrated that LLC attacks can be successfully applied in a commercial IaaS clouds like Amazon EC2. In addition, these can also be a applied as Javascript execution in web

browsers [OKSK15] or even as malicious smartphone apps [LGS+16]. In short, cache attacks have demonstrated to pose a severe threat applicable in a wide range of scenarios. Particularly dangerous are attacks on cryptographic software, as cryptography is the core component that every security solution in built upon.

We have discussed how these attacks can be stopped at three different layers; at the hardware [WL07], the OS/hypervisor [LGY+16] or at the application layer [ZHS16]. While the first two involve overheads that hardware/OS designers might not be willing to pay, little study has been made at catching and fixing cache leakages before deployment. Not in vain, preventing leakage in security critical applications (and more specifically in cryptographic software) can be achieved by aware code designers through proper implementation techniques. The few exceptions that aimed at fixing this leakages in the source code either did not consider the entire broad of memory types that can leak information [ZHS16] or lead to a high number of false positives [ABB+16]. Thus, we observe a need for a tool that can catch the vast majority of the subtle cache leakages that can arise before the cryptographic software leaves the lab. We observe three main challenges in the prevention of cache leakage exploitation:

- **Stealthy:** Leakage attacks are extremely hard to detect. The effects of an attack may only be felt through performance degradation for a short duration while leakage traces are being collected. After the attack is performed no digital footprint is left. Hence standard detection techniques such as traffic, access and privilege monitoring are completely blind to leakage attacks. For instance, any application submitted to the app store is checked for access violations to devices. However, any attack code exploiting hardware leakage would only monitor legitimate memory access time variations.

- **Hard to Prevent:** It is difficult to design leakage proof code, especially if good performance is also an objective. Leakages are quite subtle and may not be detected after deployment for a long time. Even if the code is not leaky on one platform, with gradual optimizations implemented at the microarchitectural level, new leakage channels may emerge with newer releases of a platform.

- **Difficult to Test:** Verification of leakage resistant code is painfully slow and typically requires inspection by experts well versed in cache attacks.

In this section, we introduce a tool that helps cryptographic code designers to analyze and expose any leakages in cryptographic implementations. We demonstrate the viability of the technique by analyzing cache leakages in a number of popular cryptographic libraries with respect to three main cryptographic primitives we rely on every day to securely communicate over the internet: AES, RSA and ECC. In order to achieve this goal, we first find secret dependent instructions/data with dynamic taint analysis, introduce cache evictions inside code routines, record cache

traces and then verify the existing dependence with respect to the secret. Our results show that, despite the efforts of cryptography library designers in reaction to the multitude of recently published LLC cache attacks several popular libraries are still vulnerable. In particular in this section:

- We present a proactive tool to analyze leakage behavior of security critical code. Unlike other approaches, our tool can be used to find real exploitable leakage in the design of cryptographic implementations.
- The tool identifies secret dependent data, obtains cache traces obtained from the code execution on the microarchitecture and uses the generic Mutual Information Analysis (MIA) as a metric to determine the existence of cache leakage.
- The detection technique is agnostic to the implementation, i.e. the testing code can be run across all target platforms without having to redesign it, yet pinpoints parts of code that cause found leakages.
- We perform the first big scale analysis of popular cryptographic libraries on three of the most commonly used primitives: AES, RSA and ECC.
- We demonstrate that several cache leakages are still present in up-to-date cryptographic libraries (i.e. 50% still leak information) and need to be fixed.

7.3.1 Preliminaries

The proposed tool employs techniques from modern cache attacks to monitor cache activity and measures information leakage using Mutual Information. We take into account only *Flush and Reload* and *Prime and Probe*, as *Invalidate and Transfer* exploits the same data types ans *Flush and Reload*.

7.3.1.1 LLC Attacks

The *Flush and Reload* was presented in Section 4 as a LLC side channel attack. In particular attack assumes the following;

- The attacker and the victim are executing processes in the same CPU but in different cores.
- Attacker and victim share read-only memory pages.

The *Flush and Reload* attack can only succeed when attacking memory blocks shared with the potential victim. These usually imply static code and global variables, but never dynamically allocated variables.

***Prime and Probe* Attack** The *Prime and Probe* attack was introduced as an approach to overcome those situations in which the *Flush and Reload* was unsuccessful Both attacks differ in the following way:

- *Prime and Probe* does not require any memory sharing between victim and attacker.
- *Prime and Probe* can increase the attack vector to dynamically allocated variables.
- *Prime and Probe* implies some reverse engineering to know which sets in the cache the attacker is using, while *Flush and Reload* does not.
- *Prime and Probe* is noisier than *Flush and Reload*. In particular, a noisy access in the first one can be caused by a single unrelated access to the monitored set, while in the second a minimum of `w` unrelated accesses are needed, being `w` the associativity of the cache.

Taking these two attacks into account, we explore the cryptographic libraries looking for leakages exploitable by *either* attack. Thus, we examine both statically and dynamically allocated memory in our analysis. Note that, although these attacks have been mainly used to attack the shared LLC, these leakages also appear and can be exploited at any level of the cache hierarchy. Thus, our analysis will detect leakages that can be exploited by either of these attacks (or variations of them) in the entire cache hierarchy.

7.3.1.2 Mutual Information Analysis

This section gives a brief overview on Mutual Information Analysis and its use to derive correlation, which will later be used to detect leakages. Assume we are given a discrete variable X with probability distribution P_X. Then the entropy of P_X is defined as

$$H(X) = -\sum_{x \in X} \Pr[X = x] \log_2(\Pr[X = x]) \ . \tag{7.1}$$

$H(X)$ expresses the entropy or uncertainty of a variable in bits. The entropy is 0 when the variable always takes the same value; $\Pr[X = X_i] = 1$ and $\Pr[X = X_k] = 0$, $\forall k \neq i$. The maximum entropy is achieved when the variable X is uniformly distributed. We are now ready to define a metric that captures how much the entropy of X is reduced given Y. Or in other words, how much information is leaked on X by Y. This metric is called Mutual Information $I(X, Y)$ and is defined as

$$I(X;Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X,Y)$$

Where $H(X)$ is the entropy of random variable X and $H(X|Y)$ is the entropy of the random variable X given the knowledge of the random variable Y. In a way $I(X;Y)$ gives us how related variables X and Y are. Note that if X and Y are independent, $I(X;Y) = 0$ since $H(X|Y) = H(X)$. In contrast, if X and Y are fully dependent we obtain maximum MI, i.e., $I(X;Y) = H(X)$, since $H(X|Y) = 0$ (Y fully determines X).

MI has been used in prior work for side channel attacks and leakage quantification. Gierlichs et al. [GBTP08] utilize MI as a side channel distinguisher to mount differential side channel attacks. Standaert et al. [SMY09] utilized MI as a leakage quantifier to evaluate side channel attack security. Prouff et al. [PR09] further expanded on the limitations and strengths of MI as a side channel attack metric. Recently Zhang and Lee [ZL14] used MI to measure the security level of several cache architectures which they modeled as finite state machines.

7.3.2 Methodology

Our approach involves the monitorization of the cache usage for all the memory addresses considered susceptible of carrying information of the secret to hide. In order to achieve this goal we utilize common known techniques from cache attacks, like the usage of cache flushing or cycle counter instructions.

We illustrate our approach on a toy example shown in Figure 7.3. The `Hello` code snippet simply returns different messages depending on whether the caller is a female or a male. We assume that the designer of this simple code does not want a potential attacker to know the gender of the caller. Assume that the code designers would like to know whether they did a good job, i.e., whether the cache traces do not reveal the gender of the user. Here, it is easy to observe that there is a *gender* dependent branch that could reveal whether the user is a male or female. We call $B1$ and $B2$ the two possible outputs of the branch. To test the sanity of

```
1 void Hello(char *gender){
2   if (gender=="Male"){
3     male=1;
4   }
5   else if (gender=="Female"){
6     female=1;
7   }
8 }
```

Figure 7.3: Vulnerable code snippet example

the code, we insert forced evictions at the beginning of the code routine for those memory references susceptible of leaking information about the gender, and time the re-access after its execution, as shown in Figure 7.4. Then we re-execute the code routine for several input values, retrieving the cache traces for all those potentially leaky references.

The cache traces for this simple example are shown in Figure 7.5. The x-axis shows the actual gender while the y-axis represents accesses either to the cache as a 1 or to memory as 0. The branch outputs are represented with different colors. Observe that the cache traces match perfectly the gender that the code designer is trying to hide. Thus, this code snippet would be classified as vulnerable. This

```
1 evict(B1, B2);
2 Hello(&gender);
3 time(access(B1, B2));
```

Figure 7.4: Code snippet wrapped in cache tracer

is because $B1$ and $B2$ were located in different cache lines. However, due to their proximity, it could have happened that both fall in the same cache line, in which case the code would be classified as non-leaky.

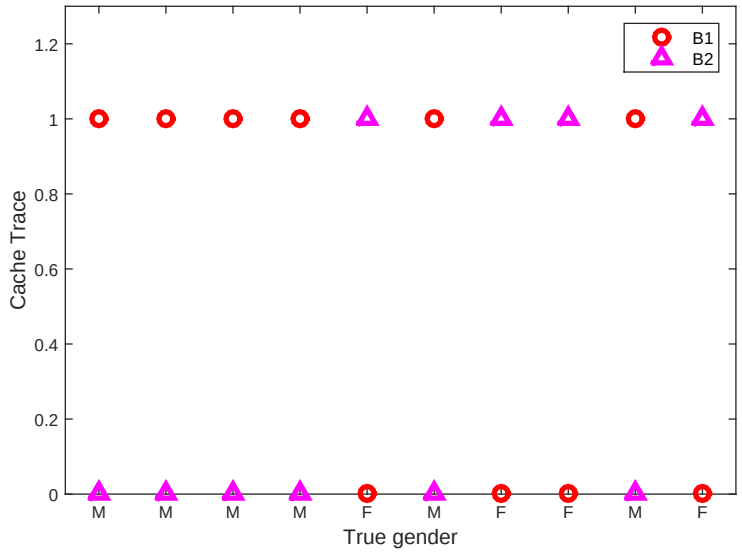


Figure 7.5: Results for cache traces obtained from invocations of Hello for varying inputs. The x-axis shows the actual gender whereas the y-axis represents cache hits/misses (1 indicates hit, 0 indicates miss). The cache trace has full correlation with $B1$ and $B2$ traces

.

Although identifying leakages in the toy example did not seem overly challenging, as the code starts getting more and more complicated, it becomes increasingly difficult and time consuming both to find secret dependent memory and to detect the leakages using manual code inspection. Particularly difficult to detect are cases in which the code does not seem to leak, but subtle architectural optimizations at the hardware level make the leakage appear [IIES15]. In order to cope with this issues we introduce a tool that automatically finds secret dependent memory thanks to the properties of dynamic taint analysis and that automates the analysis of cache leakages through the usage of statistical properties like Mutual Information for number of different inputs. We proceed to explain the metric and the methodology to identify secret dependent memory locations and to uncover hidden cache leakages.

7.3.2.1 Our Approach: MI Based Leakage Detection

Informally we can summarize our technique as follows: We first identify, thanks to dynamic taint analysis, potentially leaky pieces in the target code that may compromise the secret. For instance, *variables* and *branch instructions* that can reveal information to the attacker. Then we insert forced cache hierarchy evictions on those secret related memory addresses prior to the execution of the targeted piece of code, and accesses after it. These accesses determine a clear cache fingerprint of the potentially leaky memory addresses. Finally, we correlate the observed cache trace with the secret to be extracted by the attacker using Mutual Information. We summarize the proposed approach in four steps:

1. Identify, through a dynamic taint analysis, *suspect instructions and variables* that might reveal information related to a secret in the targeted code.
2. Insert forced cache hierarchy evictions just before the vulnerable targeted code, and accesses after it, for all the variables/instructions susceptible of leaking secret information.
3. Execute the targeted code under different secret values, and record the cache traces for every possible vulnerable variable.
4. Correlate, with MI, the observed cache traces to the secret value to determine leakage existence.

7.3.2.2 The Detection Algorithm

We formalize the detection process as shown in Algorithm 10. The input is the fragment of code that we want to analyze. First, the first step involves a dynamic taint analysis to retrieve the potential variables/instructions that are dependent on the secret the code routine is hiding. For each of them we generate access traces obtained through the execution of the code for a large number of secret values. We then compute the mutual information between the secret value used and access trace obtained through during an execution of the code. We characterize the code as to leak information if the average mutual information is above a predefined baseline *threshold*. We should note that two critical steps require code inspection: identification of the suspect variables/instructions and how the secret will be processed during MIA. Splitting the key requires understanding of the cryptographic algorithm. To improve detection rates we split the key in a way that mimics key is processing in a particular library (more details in Section 7.3.3).

7.3.2.3 Secret Dependent Memory Detection Using Dynamic Taint Analysis

In this work, we use dynamic taint analysis to effectively identify suspect instructions and variables. In the past decade, dynamic taint analysis has been widely explored

Algorithm 10 MI based leakage test

```
function DoesItLeak (C);
Input  : Code C
Output: True: the code leaks;
        False: the code does not leak
L = ListSecretTaintedMemory(C);
for each vi in L do
    trace = [ ], M = [ ];
    for each key = ⟨k_0, k_1, ..., k_ℓ⟩ value do
        evict(vi);
        execute(C, key);
        trace.append(time(vi));
        M.append(MI(trace, ⟨k_0, k_1, ..., k_ℓ⟩));
    end
    if M.average> threshold then
        return True;
    end
end
return False;
```

in many research areas, including malware analysis, vulnerability detection, program debugging and data leak protection [ZJS+11, MWX+15]. Dynamic taint analysis is a powerful approach to track the propagation of sensitive program data such as user inputs.

To detect key dependent memory for crypto libraries, our dynamic taint analysis retrieves two kinds of information. First, we collect all branch instructions affected by the taint inputs. Second, we collect all memory object information that is accessed with a taint value as index.

We illustrate our approach on a toy example shown in Figure 7.6. The `encrypt` function takes user input and encryption keys, and returns `out` including encrypted data. The `encrypt` function checks `key` and decides which variable is used for encryption and invokes `compute` function to compute output. In the `encrypt` function, variable `key` is marked as taint inputs. Then we execute the program using our taint analysis engine. The engine can easily identify that Line 8, variable index (Line 13) and variable A (Line 14) are our desired information. This does not mean that all three memory addresses will leak information; in fact, the variable index will not leak information as it is used in both cases. Our taint analysis outputs secret dependent data that might not leak information, and thus has to be verified with cache trace analysis.

We employ a symbolic execution engine KLEE [CDE08] as our taint analysis engine. The target program is compiled into a LLVM intermediate representation, and then executed by the taint analysis engine to collect suspect instructions and

```
1 static int A[] = {0x1, 0x2, 0x3, 0x4};
2 static int B = 0x5;
3
4 void encrypt(int *in, int *out, int *k)
5 {
6 int index
7   for(int i = 0; i < 64; ++i) {
8    if(k[i]>=32) {
9       index = 0;
10      compute(in, out, B);
11    }
12    else {
13      index = k[i] % 4;
14      compute(in, out, A[index]);
15    }
16  }
17 }
18
19 void main()
20 {
21   int key[64], *in, *out;
22   get_input(&key, in, out);
23   set_taint(&key, 64, "key");
24
25   encrypt(in, out, key);
26 }
```

Figure 7.6: Taint analysis example

variables.

7.3.2.4 Collecting the Cache Traces

We collect the cache usage for those memory locations that our dyncamic taint analysis outputs during different executions for varying secret inputs of the targeted code. In order to mimic the attacker ability to insert evictions, and due to its high resolution, we utilize the common cache flushing and timing approach typically utilized in LLC attacks. In short, we insert forced flushes (with the `clflush` instruction) for those variables that have a dependency with the key before the targeted code, and timed accesses after it. These timing measurements are later converted to 0s and 1s, compared to a pre-defined memory access threshold, indicating whether these memory addresses were retrieved from the cache or from the memory. These cache traces are later correlated using MI statistical properties to derive dependencies with the secret to hide.

7.3.2.5 Dealing With Noise

Although our methodology works best in an environment with minimal unintended noise coming from co-resident processes, our measurements will still suffer from the omni-present microarchitectural/ OS noise. In that sense, the *threshold* (see Algorithm 10) for which an implementation will be considered to be leaky has to be established taking into account these unintended noise sources. Aiming at correctly categorizing the leakage, we perform a set of identical measurements to those carried out for the cryptographic primitives to an *always used* variable under microarchitectural noise. Note that the variable is always cached and thus it should not present any leakage. Our goal is to see how minimum microarchitectural noise affects to the measurements that will later be taken for the cryptographic primitives. The results of 100 MI calculations under minimal microarchitectural/OS noise on datasets containing 10^5 cache access times can be observed in Figure 7.7. Our noise threshold will follow the famous *three-sigma rule of thumb* [Puk94], although one can use any statistical method that fits best to the approach taken. In our case, if the leakage observed for the cryptographic primitives is above $\mu_{noise} + 3\sigma$, then the implementation is considered to leak information. On the other hand, if the leakage tests fall bellow $\mu_{noise} + 3\sigma$, then cryptographic algorithm is classified as to not exhibit any visible leakage. Finally we note the threshold we set via experimentation is rather conservative. This stems from the fact that cache access timing measurements are susceptible to have false negatives, while memory accesses timing measurements do not show false negatives. Therefore, microarchitectural noise tends to affect timing measurements by *increasing the measured time*, but rarely reducing it.

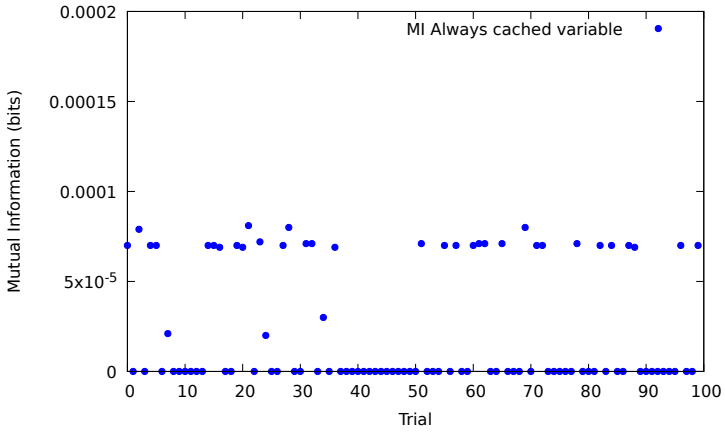


Figure 7.7: 100 MI measurements each composed of 10^5 time accesses to a cached variable. Our noise threshold is set to $\mu_{noise} + 3\sigma$

7.3.2.6 Threat Model

The threat model under consideration for our leakage tests is a key factor to be taken into account before implementing the analysis. We restrict our threat model to a software adversary capable of exploiting set collisions in any cache in the cache hierarchy and use it as a covert channel. Furthermore, we assume that the attacker is a regular user without root privileges. This means that we do not evaluate the possible leakages coming from L1 bank contentions or more powerful attackers such as malicious code with root level privileges. Although we restrict our leakage model to the most common attack scenario, our methodology can be expanded to cover more restrictive threat models. For instance, for an adversary capable of finer grain timing of crypto libraries, e.g. by timing AES rounds by manipulating the completely fair scheduler [SCNS16], then we would also have to adapt the detection algorithm to profile the rounds of AES individually instead of the AES entire execution. In any case, although time consuming, a finer grain analysis over the code space will improve the detection rate.

7.3.3 Evaluated Crypto Primitives

We restrict our evaluation to three of the most widely used crypto primitives: AES, RSA and ECC. Note that this methodology can be extended to any cryptographic software code. However, for brevity we only focus on these three primitives. Note that each cryptographic primitive has its own leakage behavior. Different implementations of the same cryptographic primitive may also result in very different leakage behavior. Our analysis excludes the quantization of the number of measurements needed by an attacker to exploit the leakage. Indeed, we believe code designers should aim at designing leakage free code without taking into account the attacker exploitation effort. Our analysis was performed on both Intel Core i7-6700K Sky-

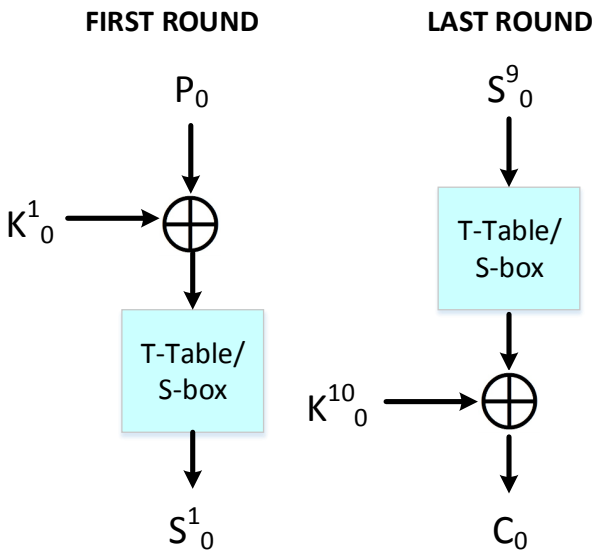


Figure 7.8: First and Last round of an AES encryption. In both cases, the entry to the T-table/S-box is influenced by the key and the plaintext/ciphertext

lake and Intel Core i5-650 Nehalem processors. However, the leakage we found does not seem to be architecture dependent, except those for which the cache line size affected the leakage. Since the cache line size is 64 bytes in the three most widely used processors, i.e., Intel, ARM and AMD, our analysis should detect the same leakages in other processors.

7.3.3.1 AES

Across the cryptographic libraries evaluated, we found three different AES implementations. The AES-NI implementation will be excluded from the analysis, since it is a pure hardware based implementation. We are left with two techniques to implement AES:

- **T-table implementation:** In this case, the entire AES algorithm (SubBytes, ShiftRows, MixColumns, AddRoundKey) implements *only* XOR and Table lookup operations. This implementation involves a maximum of 40 accesses to each table per encryption. In previous Sections (e.g. Section 4.4) we showed that leakages come from the fact that some of the table entries might not be used by the algorithm.

- **S-box implementation:** In this case, the algorithm implements all four steps of the algorithm (SubBytes, ShiftRows, MixColumns, AddRoundKey) for every byte processed [DR99]. Thus the implementation is slower (160 S-

box reads per encryption) and the leakage comes from an unbalanced usage of the S-box during the encryption.

Some implementations also might use a combination of both approaches, e.g., T-tables in the first rounds and S-box in the last. We will evaluate each of these implementations separately. Our goal is to observe whether all key dependent accesses (mostly table look ups) are in the cache or not after an AES encryption. In particular, we calculate the MI with respect to the ciphertext values, which will present different cache-memory probability distributions if the implementation leaks information. This is similar to the method outlined in [GBTP08]. We choose this methodology to simplify our measurement collection and avoid calculations of the last round key internally. Note that these represent only the implementations we encountered across the libraries we evaluated, but are not the only implementation options of AES. For instance, AES routines may utilize S-box/ T-table free implementations, which compute the inverse in Galois field mode without pre-computed tables.

Customizing for AES Leakage Detection: For AES, the algorithm presented in Algorithm 10 is customized to include several encryptions in a MI cache trace vector. However, the main concept remains the same:

- the plaintext is fixed;
- for different key byte values we record different cache traces of our potentially leaky variables; and
- the traces and the key bytes are correlated

However, note that we can also fix the key and randomize the plaintext input byte, since in the first round the T-table entry access is influenced by both the plaintext and the key bytes (see Figure 7.8). Similarly, we can apply the same concept in the last round and compute the MI of our cache traces with the last round key bytes or with the ciphertext bytes, since again the last round T-table access is influenced by both the ciphertext and last round key bytes (Figure 7.8). This might be convenient in cases where the last round is handled differently than the first 9 rounds and leaks more information than the first. In this work, we compute the MI test in both the first and last rounds, and select the one that leaks more information.

We restrict our analysis to 10^5 and 10^7 encryptions. This restriction allows us to test AES leakages in at most a few hours. Furthermore, most cache side channel attacks performed against AES have succeeded with fewer number of traces than those we require in our analysis, as it can be observed in Table 4.1 in Section 4.4.6. Therefore, we expect a leakage to be already visible with 10^7 traces. However, the analysis could be expanded to further encryptions, resulting in a much higher and time consuming analysis.

7.3.3.2 RSA

The second cryptographic primitive that we evaluate is RSA, the most widely used public key algorithm. RSA can be implemented in several ways, including by using the Chinese Remainder Theorem, with and without ciphertext blinding or exponent blinding. The only implementation that would completely stop cache and other side channel attacks from recovering meaningful information is exponent blinding, since a different exponent is used for every decryption. However, we have not seen any cryptographic library using exponent blinding. RSA leakages can come from any key dependent execution or memory usage during the modular exponentiation.

Customizing for RSA Leakage Detection: Across all the libraries evaluated in this work, we found that there are two main (and distinct) ways in which crypto libraries implement modular exponentiation in practice:

- **Bitwise modular exponentiation**: The algorithm processes the key and executes all the squaring and multiplication operations bit by bit. This means, in our detection algorithm we can split the key into bits and then correlate with MIA with traces collected over the entire RSA encryption. Details on the square and multiply implementations can be found in [Paa15], while details on montgomery ladder implementations were further studied in [JY03].

- **Fixed window algorithm**: The algorithm first precomputes the set of values $b^0, b^1, \ldots, b^{2^w-1}$ in a table, w being the value of the window. Then, the algorithm processes the key in chunks of w bits. For each chunk of w bits, w squares are executed and a multiplication with the appropriate value of the window in the precomputed table. Our detection algorithm therefore, is customized to split the key into fixed windows which are then correlated via MIA with access traces collected over the entire windowed RSA encryption. Further details on this implementations can be found in [Koc94, Paa15].

- **Sliding window algorithm**: The algorithm usually starts a window with a MSB one bit. Similar to fixed windowed exponentiation, the algorithm first precomputes the set of values $b^{2^{w-1}}, b^{2^{w-1}} - 1, \ldots, b^{2^w-1}$ in a table, w being the width of the largest window. Then, the algorithm processes the key in chunks of w bits *only* when the 1 is found in the key bit string. Until then, squares are computed for each of the zeros processed. Once the starting 1 of the window is found, w squares are executed and a multiplication with the appropriate value of the window in the precomputed table. Note that in this case we assumed the condition that the window starts with a 1. Other implementations establish that the window start and finish with a 1.Further details on this implementations can be found in [Koc94, Paa15].

Our analysis consists of observing leakage at the decryption process for 100 different 2048 bit decryption keys. Note that, unlike AES, RSA leakages are visible with much

fewer encryptions since the algorithm leaks serially bit by bit. AES on the contrary needs aggregated measurements to derive the table entries utilized.

7.3.3.3 ECC

The last cryptographic primitive we analyzed is Elliptic Curve Cryptography (ECC). We utilize Elliptic Curve Diffie Hellman (ECDH) as opposed to Elliptic Curve Digital Signature Algorithm (ECDSA) because in the latter the secret operation is performed on an ephemeral key, which limits our analysis capabilities. Note that this does not imply that the leakage observed for ECDH will not be present in ECDSA, since both algorithms use the same implementation techniques in all the libraries analyzed. As before, we utilize 100 different 256-bit private keys.

Customizing for ECC Leakage Detection: The scalar point multiplication implementations we found are very similar to those observed for modular exponentiation:

- **Double and Add with Montgomery ladder**: The algorithm processes the secret scalar bit by bit, executing a double and an add regardless of the bit value being processed. In the detection algorithm, the key is split into bits and then correlated with access traces obtained over a scalar point multiplication computation using MIA. More on bit by bit scalar multiplications can be found in [Riv11].

- **Fixed window algorithm**: In this case, the algorithm first precomputes the set of values $P, 2P, \ldots\ldots, (2^w - 1)P$ in a table, being w the window size. Then, the algorithm processes the secret scalar in chunks of w bits. For each chunk, w doublings are executed and an addition with the appropriate value of the window in the precomputed table. In the detection algorithm the key is split into fixed sized windows and then correlated with access traces obtained over a scalar point multiplication computation using MIA. More details windowed algorithms can be found in [YS06].

- **Sliding window algorithm**: In this case, the algorithm will set a condition for the window. For instance, a condition might be that the window has to start with a 1. Assuming this condition we have that the algorithm again first precomputes the set of values $(2^{w-1})P, (2^{w-1} + 1)P, \ldots, (2^w - 1)P$ in a table, w being the window size. Then, the algorithm processes the secret scalar in chunks of w bits *only* when a 1 is found in the key. Until then, doublings are issued for each of the zeros processed. Once the starting 1 of the window is found, w doublings are executed plus an addition with the appropriate value of the window in the precomputed table. In the detection algorithm similar to the fixed window case the key is split into bits and then correlated with access traces obtained over a scalar point multiplication computation using MIA.

Table 7.1: Cryptographic libraries evaluated

Cryptographic Library	Version
OpenSSL	1.0.1-t
WolfSSL	3.9.0
Intel Integrated Performance Primitives (IPP)	9.0.3
Bouncy Castle	1.8.1
LibreSSL	2.4.2
Mozilla Network Security Services (NSS)	4.13.0
Libgcrypt (NSS)	1.7.3
MbedTLS	2.3.0

- **wNAF**: Aiming at reducing the number of non-zero elements in a key, this algorithm works with both signed and unsigned digits, making sure there are at least $w-1$ zeroes between two non-zero digits, w being the window size. The algorithm precomputes the set of values $(P, 3P, \ldots, (2^{w-1}-1)P$ and performs a double operation for every digit, plus an addition or subtraction whenever for the non-zero digits. In the detection algorithm the key is split into recoded into wNAF windows, and then correlated with access traces obtained over a scalar point multiplication computation using MIA. More details of this kind of implementation are provided in [ML09].

7.3.4 Cryptographic Libraries Evaluated

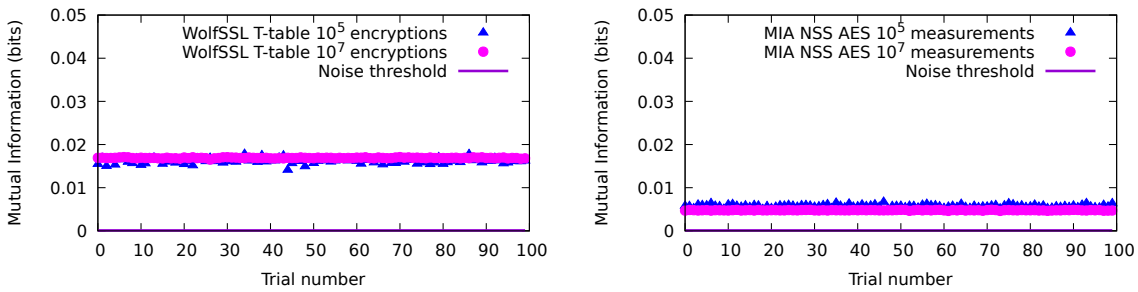
Our evaluation is restricted to 8 popular up-to-date (at the time of the leakage evaluation) cryptographic libraries, represented in Table 7.1. The analysis can be extended to other cryptographic libraries with mild effort.

7.3.5 Results for AES

In this section we present the results obtained for AES implementations for different libraries. For the sake of brevity, we do not include all the results analyzed, but rather examples of every kind of implementation. The complete results are summarized in Section 7.3.8.

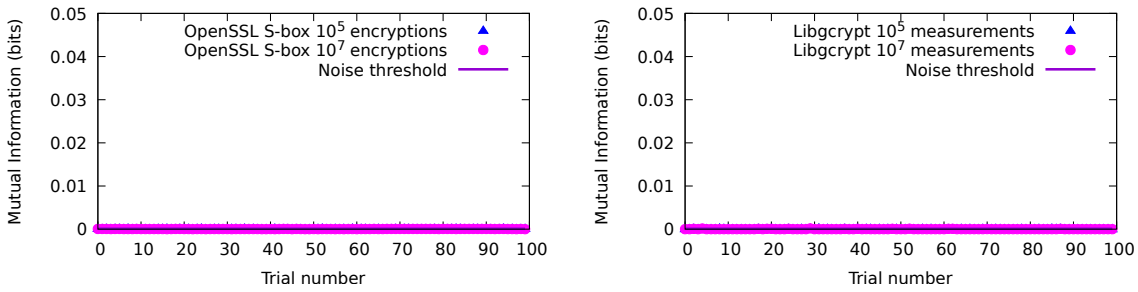
7.3.5.1 T-table Based AES (WolfSSL, MbedTLS, NSS & Bouncy Castle)

WolfSSL, MbedTLS, NSS and Bouncy Castle utilize T-table based implementations to perform AES encryptions. We only include examples for WolfSSL and NSS. Note that the results obtained for the other libraries are similar to the ones obtained for WolfSSL and NSS shown in Figure 7.9(a) and Figure 7.9(b), respectively. For both cases, the T-tables were classified as key dependent accesses and leaked information.



(a) (b)

Figure 7.9: AES MI leakage in a) WolfSSL and b) NSS. The leakage is observable at 10^5 encryptions in both cases.



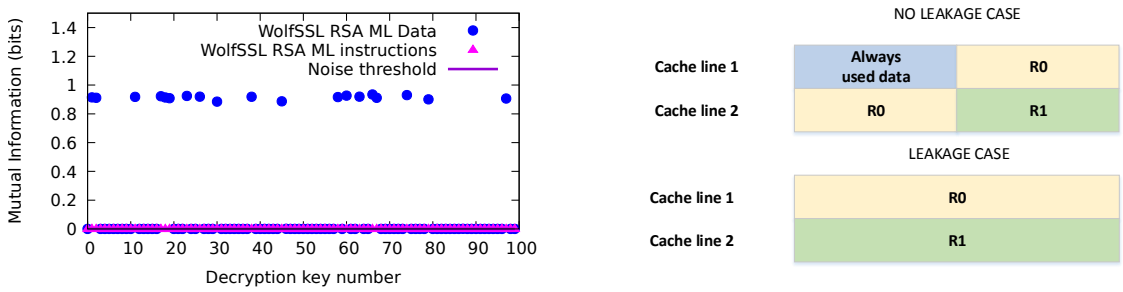
(a) (b)

Figure 7.10: AES S-box implementation MI leakage in a) OpenSSL and b) Libgcrypt. None of them leak information.

Observe that in both cases the leakage is already observable at 10^5 encryptions, and that WolfSSL seems to leak more information than NSS. This is because WolfSSL uses a different T-table for the last round, while NSS re-uses the first 4 T-tables in the last round. Consequently, the probabilities are more distinguishable in the case of WolfSSL. In both cases the leakage is above our noise threshold thus leading to key recovery attacks after a small number of encryptions.

7.3.5.2 S-Box Based AES (OpenSSL, LibreSSL and libgcrypt)

In this case we analyze those implementations that utilize a single 256 entry S-Box in the AES encryption. As before, we only show the results obtained for two of them, since the results obtained for the rest are very similar. In this case, the results for OpenSSL and Libgcrypt are included in Figure 7.10(a) and Figure 7.10(b), respectively. In both cases, the S-boxes accesses were flagged as key dependent, but the leakage results were not higher than our noise threshold, and as such are categorized as non-leaky implementations. In the case of OpenSSL, we observe that

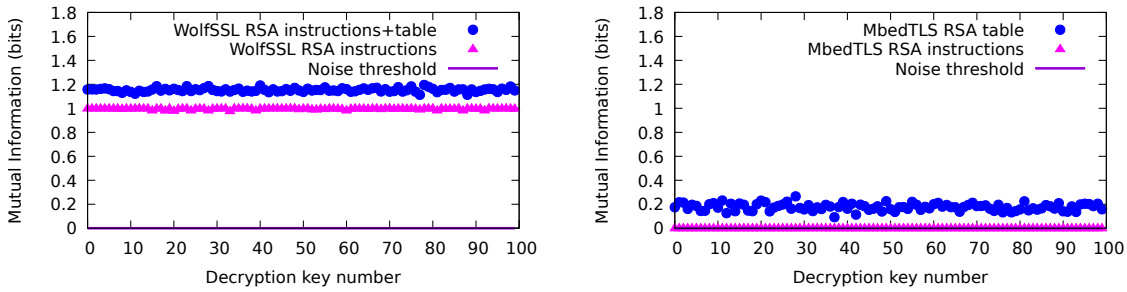


(a) (b)

Figure 7.11: Montgomery ladder RSA MI leakage for WolfSSL (a) Instruction leakage is shown in magenta, data leakage shown in blue. Register accesses leak for some keys due to cache alignment issues. b) Explanation for the varying leakage. When R0 is not stored at a cache line boundary, its cache line is always used and we cannot find leakage.

the S-Box entries are prefetched to the cache prior to the encryption. Libgcrypt utilizes the approach described in [Ham09], i.e., they use vector permute instructions not only to speed up the AES encryption but further to protect against cache attacks.

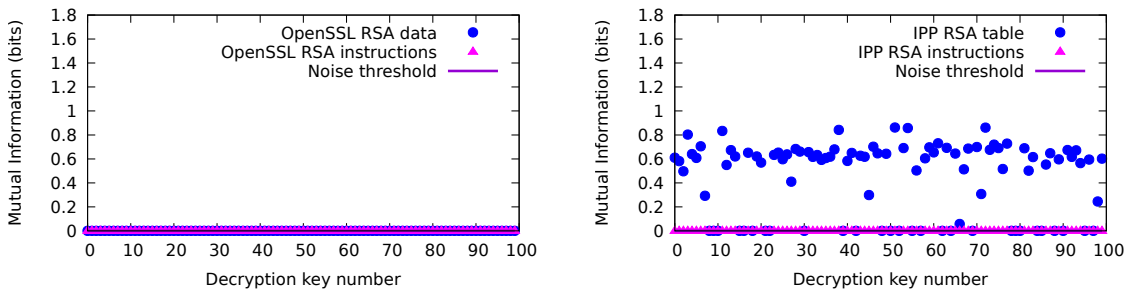
7.3.6 Results for RSA



(a) (b)

Figure 7.12: Sliding window RSA leakage for a) WolfSSL and b) MbedTLS. Magenta indicates instruction leakage, blue indicates instruction+data leakage. WolfSSL leaks from both possible sources (instructions reveal zeroes between windows, data accesses reveal window values) and MbedTLS only leaks from the precomputed table accesses.

Similar to the measurements taken for AES, we proceed to test the robustness of the RSA implementations observed across our evaluated libraries. The three main approaches found were:

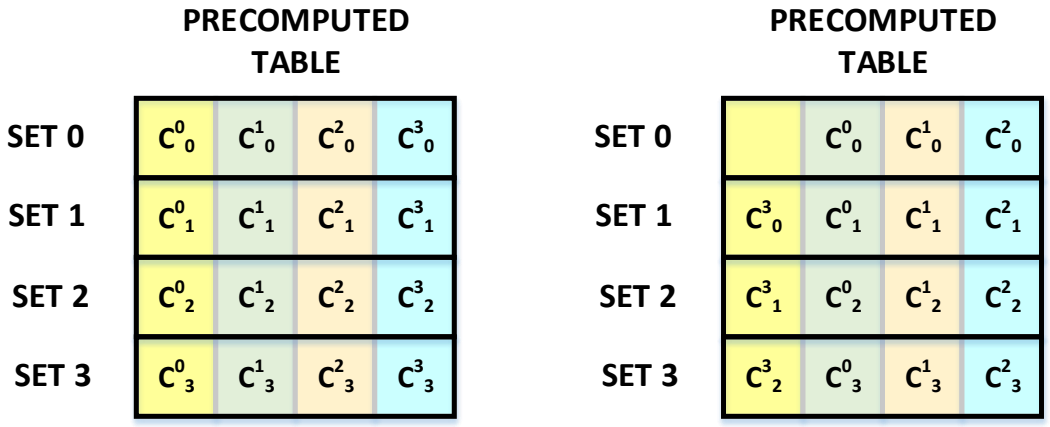


(a) (b)

Figure 7.13: Leakage for fixed window RSA in a) OpenSSL and b) IPP. Magenta indicates instruction leakage, blue indicates instruction+data leakage. OpenSSL does not leak information (scatter and gather approaches correctly used), while IPP leaks varying amount of leakage for different decryption keys (although scatter and gather is used)

7.3.6.1 Montgomery Ladder RSA (WolfSSL)

WolfSSL is the only library that utilizes Montgomery ladder to perform modular exponentiations. This implementation attempts to prevent the leakages coming from simple square and multiply approaches. Our results show that, among others, squaring or multiplying instructions and key storing registers are key dependent. The results for these are shown in Figure 7.11, where instruction leakage is presented in magenta and data leakage coming from the registers in blue. We observe 0 MI for these (or any) instructions involved in the modular exponentiation process leak information. However, in the case of the registers, we see that the MI is 0 for most of the keys, except for some for which it increases to almost 1, the maximum entropy. These measurements cannot be outliers, since even if a noisy measurement is observed, it should not output an almost maximum MI. Thus, after inspecting this particular behavior, we realized that the MI was high *only* when the variable R_0 started at a cache line boundary. If it does not, then there is data allocated in the same cache line that is used regardless of the key bit (fetching the R_0 cache line always to the cache). This behavior is represented in Figure 7.11(b). Note that the alignment of the register R_0 is controlled by the OS, and usually changes from execution to execution. In consequence, a potential attacker only needs to keep iterating until observing a trace for which R_0 is aligned with a cache line boundary. Our methodology was able to find this particular cache alignment dependent leakage, which would not be observable without empirical cache traces.



(a) (b)

Figure 7.14: Cache structure difference for aligned and misaligned precomputed table when using scatter and gather approaches. Observe that when misaligned, an attacker monitoring set 0 can retrieve when c^3 is being used.

7.3.6.2 Sliding Window (WolfSSL, MbedTLS, Libgcrypt & Bouncy Castle)

WolfSSL and MbedTLS utilize sliding window approaches, which require the window to start with a 1 bit. Sliding window approaches are quite common in RSA cryptographic implementations. Indeed, they are faster because less multiplications are used than when processing the key bit by bit. This comes at the cost of storing a table of several precomputed powers of the ciphertext. For these implementations, squaring and multiplications and the pre-computed table were flagged as key dependent. Results for these are shown in Figure 7.12(a) and Figure 7.12(b), where instruction leakage is shown in magenta and the instruction + data leakage is shown in blue. We found instruction leakage in WolfSSL due to the usage of the multiplication function (different from the square function), revealing the zeroes between windows. Accesses to the precomputed table also leak information, revealing the window values being used. In the case of MbedTLS we only found table entry accesses to be non-protected, which reveal the window values. In both cases (easier in the case of WolfSSL), an attacker monitoring the cache would be able to retrieve the key [IGI+16b, Fan15].

7.3.6.3 Fixed Window (OpenSSL and IPP)

OpenSSL and IPP utilize fixed window approaches to perform modular exponentiations, wich execute a constant number of squares and multiplications for every

key value. In contrast to sliding window approaches, fixed window exponentiations do not have requirements for processing windows. The results are presented in Figure 7.13(a) and 7.13(b). In OpenSSL, we observe that neither the key dependent instructions nor the data leakage is above our noise threshold, i.e., our test categorizes it as non-leaky implementation. One of the reason, among others, is that OpenSSL uses a scatter and gather approach [HGLS07], i.e., ciphertext powers are stored vertically in the cache and not horizontally. Thus, each cache line contains a little bit of information of each of the ciphertext powers (the multiplicants), as seen in Figure 7.14(a). This technique ensures that all the ciphertext powers are loaded whenever one is needed, and thus, an attacker is not able to distinguish which (among all) was accessed. However, other covert channels might exploit this approach [YGH16].

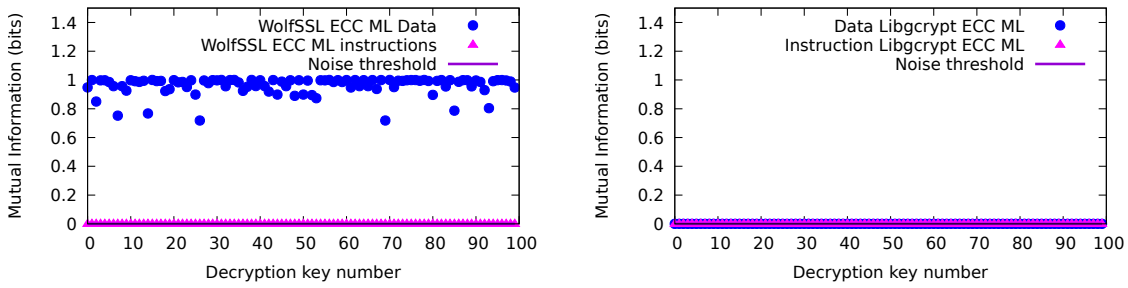
We found no instruction leakage in IPP. In contrast, varying data leakage coming from the pre-computed table was observed for different keys. To better understand this counter-intuitive behavior, we investigated the code. We realized that IPP uses a similar approach to OpenSSL, but the scatter-gather precomputed table is *not* always starting in a cache line boundary, i.e., there is no alignment check when the table was initialized. In consequence, the OS chooses where the table starts with respect to a cache line, i.e., it is not deterministic. Figure 7.14(b), in which the table starts at a 1/4 of a cache line, helps to illustrate the problem. An attacker monitoring set 0 obtains cache misses whenever c^0, c^1 and c^2 are utilized, but cache hits whenever c^3 is used. Thus, she can detect when window 3 has been utilized by the exponentiation algorithm. If the table is created at the middle of a cache line, she can recover when window 2 is being used (since she already knows when 3 is used). Thus, a divide and conquer approach leads to a full key recovery. In contrast, OpenSSL performs an alignment check in the precomputed table.

7.3.7 Results for ECC

The last cryptographic primitive that we analyze is ECC. we perform 100 ECDH private key operations for which we measure leakage. The following are the four main implementation styles that we encountered for ECC implementation

7.3.7.1 Montgomery Ladder ECC (WolfSSL & Libgcrypt)

Only WolfSSL and Libgcrypt implemented this approach. Results are presented in Figure 7.15(b). In the case of WolfSSL, we only found data leakage coming from the key dependent register accesses. Unlike the behavior observed in RSA (in which the cache lines could make the implementation not to leak), the ECC implementation presents leakage for every key. Thus, an attacker can recover the key with just a few traces if he primes the set that these key dependent register occupy in the cache. The results for Libgcrypt indicate no presence of leakage because it utilizes a temporary variable on which the computations are performed, and then the contents of it are



(a) (b)

Figure 7.15: Montgomery Ladder ECC leakage for a) WolfSSL and b) Libgcrypt. Magenta indicates instruction leakage while blue indicates instruction+data leakage. Instructions do not give us any information in neither case. Register accesses give us the full key in WolfSSL

placed in the corresponding registers in constant execution flow. This fully prevents any leakage coming from the state registers, as both registers are read when one needs to be used.

7.3.7.2 Sliding Window ECC (WolfSSL)

The default implementation in WolfSSL uses a sliding window approach, very similar to the RSA implementation. The results are shown in Figure 7.16, displaying in magenta the instruction leakage and the instruction plus data leakage in blue. We observed that the implementation leaks from both the add instruction, which reveals the zeroes between windows (for being distinct to the double function), and from the accesses to the precomputed table, which reveal the window values. Once again, the ECC implementation of WolfSSL leaks enough information to reveal the secret key.

7.3.7.3 Fixed Window ECC (MbedTLS, Bouncy Castle, NSS)

As with RSA, fixed window implementations process the key in fixed w sized windows. We present two examples, MbedTLS and Bouncy Castle, for which the results are shown in Figures 7.17(a) and 7.17(b). Surprisingly, we do not observe any instruction or data leakage in the case of MbedTLS, while we did observe for RSA. One of the differences with respect to the RSA implementation is that MbedTLS accesses all the precomputed table values in a loop before using one, and they choose it in constant execution flow. Thus, all table values are loaded into the cache while only one needs to be used. The fact that their ECC algorithm is well designed but their RSA algorithm is not is surprising. In the case of Bouncy Castle, we do not observe leakage from key dependent instructions, but we observe leakage in the

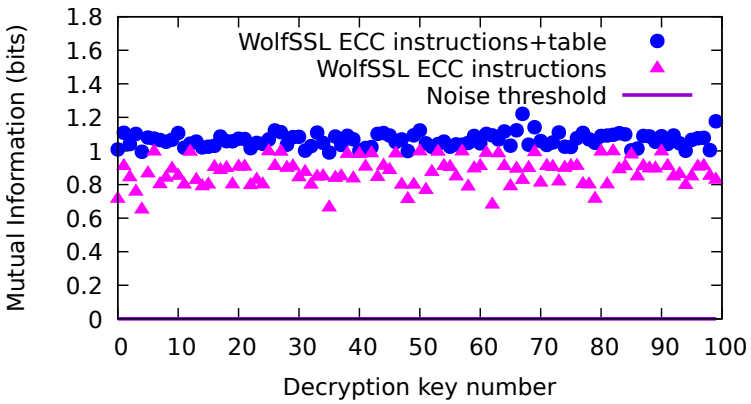
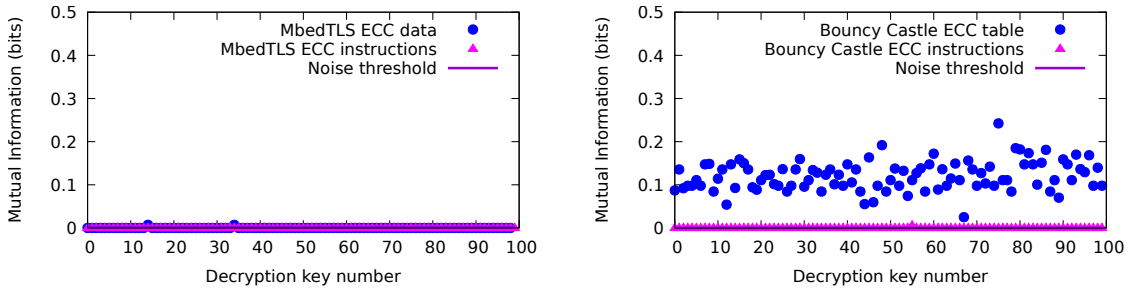


Figure 7.16: Sliding window ECC leakage in WolfSSL. Magenta indicates instruction leakage, blue indicates instruction+data leakage. Both sources leak information.



(a) (b)

Figure 7.17: ECC leakage in MbedTLS and Bouncy Castle. Magenta represents instruction leakage and blue represents instruction+data leakage. MbedTLS does not leak while Bouncy Castle leaks information from the precomputed table

accesses to the key dependent precomputed table that allows an attacker to recover the key.

7.3.7.4 wNAF ECC (OpenSSL & LibreSSL)

OpenSSL and LibreSSL utilize the wNAF scalar multiplication method. To reduce the number of non-zero digits, wNAF works with signed digits, ensuring there are at least $w + 1$ zeroes between two non-zero digits. Then the key is processed in a loop, doubling always and performing additions/subtractions whenever a non-zero digit is found. Note that, as the number of multiplications needed is reduced, the algorithm show better performance behavior compared to those analyzed in previous sections. The results for OpenSSL's implementation are shown in Figure 7.18, where the key dependent addition function instruction leakage is shown in blue, addition instruction + key dependent precomputed table entry leakage is shown in green

and the additional key dependent sign change instruction leakage effect is shown in magenta. As we can see, the addition instruction already gives us the zero values (again, as we can distinguish it from the double function), the leakage from the precomputed table accesses give us the window value and the sign change function gives us the sign of the window value. Thus, OpenSSL a big door for ECC key recovery attacks.

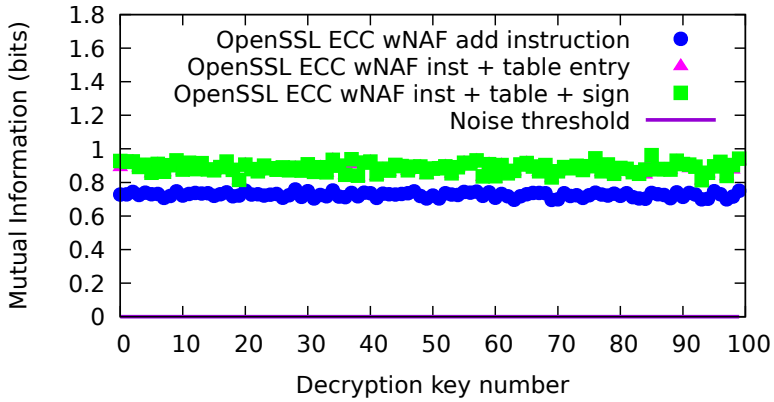


Figure 7.18: wNAF ECC OpenSSL results. Blue shows addition instruction leakage, magenta adds precomputed table entry leakage and green adds sign change leakage. All possible sources leak information

7.3.8 Leakage Summary

This section summarizes the leakages found in the cryptographic libraries analyzed. The results for all the default (marked in bold) and some non-default implementations are presented in Table 7.2. The third column indicates whether the implementation leaks the secret or not. We found that 50% of the default implementations were susceptible to key recovery attacks (i.e. 12 out of 24). We believe these numbers are alarming especially given the popularity cache attacks have gained over the last few years. These numbers underline the need for a methodology, like the one proposed in this work, that cryptographic code designers can utilize to detect leakages before the libraries are released to the public.

7.3.9 Comparison with Related Work

Our work is not the first attempt to detect cache leakage in software binaries. For instance, Zankl et al. [ZHS16] use the PIN instrumentation tool to detect instruction

[1]Due to Intellectual Property conflicts, we are not allowed to disclose the approach taken by the library

[2]We verified with OpenSSL that while the 1.0.1 branch leaks, the 1.1.0 branch does not leak

Table 7.2: Leakage summary for the cryptographic libraries. Default implementations are presented in bold

Cryptographic Primitive	Library	Outcome
AES	OpenSSL (T-table)	**Leaks**
	OpenSSL (S-box)	No leak
	WolfSSL	**Leaks**
	IPP (v1)[1]	No leak
	IPP (v2)[1]	No leak
	LibreSSL (S-box)	No leak
	NSS	**Leaks**
	Libgcrypt	No leak
	Bouncy Castle	**Leaks**
	MbedTLS	**Leaks**
RSA	OpenSSL (Sliding W)	**Leaks**
	OpenSSL (Fixed W)	No leak
	WolfSSL (Montgomery L)	**Leaks**
	WolfSSL (Sliding W)	**Leaks**
	IPP	**Leaks**
	LibreSSL	No leak
	NSS	No leak
	Libgcrypt	No leak
	Bouncy Castle (Sliding W)	**Leaks**
	MbedTLS (Sliding W)	**Leaks**
ECC	**OpenSSL (WNAF)**[2]	**Leaks**
	WolfSSL (Montgomery L)	**Leaks**
	WolfSSL (Sliding W)	**Leaks**
	IPP[1]	No leak
	LibreSSL	**Leaks**
	NSS	No leak
	Libgcrypt	No leak
	Bouncy Castle (Fixed W)	**Leaks**
	MbedTLS (Fixed W)	No leak

leakage in RSA implementations. Demme et al. [DMWS12] use simulated cache traces to define a correlation-based side channel vulnerability factor of different hardware characteristics. Almeida et al. [ABB+16] defined a generic constant time security definition and verifies constant time implementations. Our work differs from them in the following:

- We proposed a sophisticated tool that combines taint analysis with automatic code insertion to track cache leakages of analyzed code while being executed in a realistic environment. We make use of several modern techniques such as the `flush and reload` technique, taint analysis and mutual information to make the tool efficient and easy to use.

- The SVF proposed in [DMWS12] uses correlation between an ideal memory-access trace and an cache-observable trace, every time for the entire analyzed code. This means that constant execution flow code will still feature high correlation on leaky architectures, even though the code is not vulnerable. Hence, SVF is a metric for comparing hardware architectures, but cannot be used to find leaking code.

- Like [ZHS16], our tool looks for dependencies between secret variables and observed leakages to find vulnerabilities. We chose mutual information over correlation, as mutual information is more generic as it does not only look for linear relations and is as easy to implement. Furthermore our work evaluates both instruction and data leakage, while [ZHS16] only analyzes instruction leakage, mainly due to the limitation of the proposed PIN based approach to find dynamic memory leakage. Extending their approach to cover leakage of dynamically allocated data is difficult with using PIN, since memory references vary for each execution.

- Unlike [ZHS16, ABB+16] our methodology takes into account microarchitectural features such as cache line sizes that can affect the code's resistance to cache attacks. This is particularly important to expose leakages caused by OS data cache line missalignments, as those described in Section 7.3.6.3 and 7.3.6.1, which are not observable without empirical cache traces.

- Compared to [ZHS16, ABB+16], our methodology detects *exploitable* leakage, as demonstrated by our discovered 12 vulnerabilities that are being fixed. In contrast, [ABB+16] does not expose new vulnerabilities (it would simply mark all analyzed libraries as non-constant execution flow, i.e., generates lots of false positives) while [ZHS16] only exposed one.

7.3.10 Recommendations to Avoid Leakages in Cryptographic Software

Our work demonstrates that up to date cryptographic libraries are still vulnerable to cache side channel attacks. The leakages were primarily due to key dependent memory accesses which are divided into two classes:

1. Secret dependent non-constant execution flow.

2. Secret dependent data memory accesses.

The leakages found in this work can be avoided by designing cryptographic code with those two considerations in mind. Based on the observations made in this study, we provide the following recommendations More specifically, we provide the following recommendations and observations made in this study for the cryptographic primitives analyzed:

- **S-Box and T-table based AES**: Cryptographic code designers should at least protect against passive attackers by prefetching the T-table or the S-box into the cache before the encryption. To be protected against an attacker with higher resolution, the whole table has to be fetched for each table access. Alternatively designers can use vector permutation instruction based implementations [Ham09].

- **S-Box free AES implementation**: These implementations are protected since no key dependent look-up tables are used. AES-NI and implementations where the Affine Transformation is computed on the fly are examples of safe implementations.

- **Square and Multiply RSA**: Designers should avoid this implementation, since cache protection introduces large overheads.

- **Montgomery Ladder RSA**: Designers should use temporary registers and constant execution flow.

- **Sliding/Fixed window RSA**: Both sliding and fixed window approaches should protect accesses to the precomputed table, e.g., accessing all table entries in a loop. Additionally, if sliding window is used, designers should use the same instruction to perform squaring and multiplying operations.

- **Double and Add ECC**: *Do not use this implementation.* Protecting instructions in ECC is difficult, since double and add perform different operations.

- **Montgomery Ladder ECC**: Designers should use temporary registers with constant execution flow.

- **Sliding/Fixed Window ECC**: In both cases, the table has to be protected e.g. by accessing all entries in a loop. However, *we recommend using fixed window over sliding window for ECC*, since protecting double and add instruction is difficult and can reveal partial key values.

- **WNAF**: Although the fastest implementation, we found that its leakage its quite high. Indeed, prior attacks have shown to recover the key even just from the instruction leakage (e.g., [BvdPSY14]).

If implementers choose to go all the way and write code with a truly secret-independent execution flow, a tool like [ABB+16] can be very helpful. However, if designers depend on microarchitectural properties to hide secret-dependent indexing (see Section 7.3.6.3), we strongly recommend the methodology proposed in this work to verify that all assumptions are actually met.

7.3.11 Outcomes

We introduced a proactive tool to detect leakage in security critical code before deployment. The tool uses cache traces for secret dependent memory obtained from execution of the code on the actual microarchitecture and uses the generic Mutual Information Analysis (MIA) as a metric to determine cache leakage. The detection technique can be run across all target platforms without having to redesign it, yet pinpoints parts of code that cause found leakages. We used the detection technique to perform the first big scale analysis of popular cryptographic libraries on the three most popular crypto-primitives, namely AES, RSA and ECC. Our results show that about half of the implementations leak information through the LLC. Worse still, a tool like ours can also be used by malicious parties to find exploitable vulnerabilities with ease. This proves the need for cryptographic libraries to resist cache attacks, which can be aided by using the proposed tool in real-life security software development.

7.4 MASCAT: Preventing Microarchitectural Attacks from Being Executed

In the previous section, we presented a tool to help code designers avoid cache leakages from being exploited. However, we understand that the acquisition and execution of tools like the one we proposed might not be immediate, and that even if they do, cache attacks can still try to recover non-security critical data from benign applications, e.g. the number of items in a shopping cart [ZJRR14] or key strokes on a touch screen [GSM15]. Further, malicious attackers can try to utilize different covert channels than the proposed LLC. For instance, memory bus locking attacks are capable of acting as covert-channels and may be used for detecting hardware co-residency or for Quality of Service (QoS) degradation [VZRS15, IIES16]. In addition, rowhammer attacks pushed the envelope further by introducing cryptographic faults or by breaking memory isolation techniques [BM16, XZZT16].

Besides, one of the reasons why microarchitectural attacks are so dangerous is the wide range of situations they apply. We have proved in previous sections how they can be applied in commercial clouds (e.g. Amazon EC2), but they can also be applied in very common scenarios as in browsers (as Javascript extensions [OKSK15]) or even on mobile devices, e.g. as smartphone applications [LGS+16].

Due to the severity of the threat, it is important not only to design leakage-free code but also to roll out protection against microarchitectural attacks before they find widespread adoption. Solutions proposed so far, can be divided into preventive and reactive (online detection) categories. The implementation of preventive methodologies, e.g., page coloring or Intel Cache Allocation Technology [LGY+16], requires changes to current infrastructure (hardware or OS/hypervisor), and usually incurs noticeable performance overheads, making infrastructure providers unlikely

to deploy them (in fact we have not seen any of those applied in commercial use). Similarly, reactive approaches, such as monitoring of performance counter events by the OS [ZZL16] also incur overheads and thus make widespread adoption unlikely.

In some of the aforementioned cache applicable scenarios, e.g. smartphones or even play Operating Systems (like Windows) software is now commonly distributed through official digital app stores, e.g., Microsoft store or Android Google Play. Users can profit from a certain level of trust for the downloaded applications, as store operators review app binaries before posting them. However, unlike other kind of malware, microarchitectural attacks look harmless, as they access resources in a natural way like other harmless code and are thus harder to detect. A common solution is to utilize anti-virus software to detect malicious code inside binaries. Nevertheless, as it will be presented later in this document, these mis-detect the majority of microarchitectural attacks. In consequence, microarchitectural attacks can take advantage of the lack of proper detection mechanisms in official app stores and utilize them as a channel to obtain widespread distribution.

We present MASCAT (microarchitectural side channel attack trapper), a tool to detect microarchitectural attacks through static analysis of binaries. In short MASCAT is similar to an antivirus tool to catch microarchitectural attacks embedded in innocent looking software. MASCAT works by statically analyzing binary elf files, detecting implicit characteristics commonly exhibited by microarchitectural attacks. We have identified several characteristics that give strong evidence of the presence of certain attack classes. The importance factor of each characteristic is configurable to either detect all or a specific subset of microarchitectural attacks. Further, our approach is designed to easily add more characteristics to the range of attacks that MASCAT covers. The execution of our approach is fully automated and in comparison to other solutions, the outcome is easily understandable as it not only colors the location where the threat was found in the binary but also adds an explanation on the characteristics that triggered the alarm.

Due to its similarity to antivirus tools, MASCAT can be adopted by digital application distributors (like Androids GooglePlay, Microsoft Store or Apple app store) to ensure the applications being offered do not contain microarchitectural attacks. In fact, they can use MASCAT to detect the presence of microarchitectural attack characteristics, and decide whether the application needs to be removed. More than that, if such an attack is found, the submitter can be banned from the app store. Further, MASCAT can be easily adopted by antivirus software users who would otherwise find it technically challenging to adopt any of the other existing countermeasures.

In summary, this section

- Shows for the first time that app stores can stop microarchitectural attacks prior to their execution without the collaboration of OS/software designers;
- Introduces a novel static binary analysis approach looking for attributes implicit to microarchitectural attacks in apparently innocent binaries and APKs;

- Performs a full analysis of 32 attack codes designed by different research groups and identifies characteristics that are common to the various classes of attacks.
- Implements a configurable threat score based approach, that is not only easily changeable but also expandable. This approach yields a detection rate of 96.67%.
- Presents a false positive analysis on `x86-64` binaries and Android APKs to estimate an average of 0.75% false positive rates.

7.4.1 Microarchitectural Attacks

Microarchitectural attacks take advantage of shared hardware contention to infer information from a co-resident victim. The choice of the hardware piece to target usually becomes a trade-off between feasibility and threat exposure. In the last years several practical microarchitectural attacks with severe privacy/availability violation implications have been proposed, most of which we include in our static analysis approach. Although our thesis has mainly described cache attacks, MASCAT will consider an broader range of attacks. This section gives a brief description on the functionality of each of them.

7.4.1.1 Cache Attacks

Cache attacks take advantage of cache collisions occurring in some shared cache in the cache hierarchy. These collisions are detected by measuring access times, i.e., by distinguishing accesses between two levels in the same cache hierarchy or between accesses to the cache hierarchy and accesses to the DRAM. Although several attack designs have been proposed, two (and their derivations) stand out over the rest and have already been discussed in our work: the *Flush and Reload* and the *Prime and Probe* attacks. Once again, as the *Invalidate and Transfer* is consider a derivation of *Flush and Reload* for its similar characteristics:

***Flush and Reload*:** The *Flush and Reload* attack assumes memory sharing between victim and attacker and is performed in three major steps. In the first step, the attacker removes a shared memory block from the cache hierarchy (e.g., with the `clflush` instruction). In the second step, the attacker lets the victim interact with the shared memory block by waiting a specified number of cycles. In the last step, the attacker re-accesses the removed memory block. If the memory block comes from the cache (i.e. a fast access time observed) she derives that the victim utilized the memory block during the waiting period. On the contrary, if the memory block comes from the DRAM (i.e. a slow access observed) the attacker assumes the victim did not access the memory block during the waiting period. *Flush and Reload* attacks have been exploited in Section 4 and in [YF14, GSM15].

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3
4    asm __volatile__ (
5      "  mfence             \n"
6      "  lfence             \n"
7      "  rdtsc              \n"
8      "  lfence             \n"
9      "  movl %%eax, %%esi  \n"
10     "  movl (%1), %%eax   \n"
11     "  lfence             \n"
12     "  rdtsc              \n"
13     "  subl %%esi, %%eax  \n"
14     "  clflush 0(%1)      \n"
15     : "=a" (time)
16     : "c" (adrs)
17     :  "%esi", "%edx");
18   return time < threshold;
19 }
```

Figure 7.19: *Flush and Reload* code snippet from [YF14]

***Prime and Probe*:** The *Prime and Probe* attack does not make any special assumption between victim and attacker (rather than sharing the underlying hardware) and is also performed in three major steps. In the first step, the attacker fills a portion of the cache (usually a set) with his own data. Then the attacker waits again hoping to observe activity from the victim. In the third step, he re-accesses the data he used to fill part of the cache. If all his data comes from the cache (i.e. low access times) the attacker assumes the victim did not use the portion of the cache she filled (otherwise one of her data lines would have been evicted). On the contrary, if she observes that at least one of her data lines (i.e. high access times) comes from the DRAM, the she assumes the victims utilized that portion of the cache. *Prime and Probe* attacks were exploited in Sections 6 and in [Fan15].

Combinations of both techniques have also shown to be successful at executing attacks. For instance, a Evict and Reload approach would be successful in those systems in which users do not have access to a `clflush` like instruction.

7.4.1.2 DRAM Access Attacks

The DRAM is usually divided in channels (physical links between DRAM and memory controller), Dual Inline Memory Modules (DIMMS, physical memory modules attached to each channel), ranks (bank and front of DIMMS), banks (analogous to cache sets) and rows (analogous to cache lines). If two addresses are physically adjacent they share the same channel, DIMM, rank and bank. Additionally each bank contains a row buffer array that holds the most recently accessed row.

In fact DRAM access side channel attacks take advantage of collisions produced between addresses physically adjacent, i.e., in the same bank, rank, DIMM and channel. More precisely, retrieving a memory location from the row buffer yields

```
loop:
        mov (X), %eax
        mov (Y), %ebx
        clflush (X)
        clflush (Y)
        mfence
        jmp loop
```

Figure 7.20: Rowhammer code snippet from [KDK+14a]

faster accesses than retrieving it from the bank. In fact, the row buffer acts like a direct mapped cache holding the most recently accessed rows. In order to build a successful attack, an attacker would first have to prime the row buffer. Then he would have to evict the cache portion that the target memory location occupies, making sure that the next victim access will hit the DRAM. After the victim access, the attacker probes the row buffer to check whether the victim accessed memory locations within the same bank. If he did, he will observe row buffer misses, and thus an increase in the access time. On the contrary, if the victim did not access bank congruent locations, the attacker will obtain fast accesses for his primed data. The attack was proposed and exploited in [PGM+16].

7.4.1.3 Rowhammer Attacks

Rowhammer attacks take advantage of disturbance errors that occur in adjacent DRAM rows within the same DRAM bank. It has been demonstrated that continuous accesses to a DRAM row can indeed influence the charge of adjacent row cells, making them leak at a higher rate. If these cells loose charge faster than the charge refreshing interval, accesses to a DRAM row induce bit flips in adjacent rows in the same bank. Note that bit flips can have catastrophic consequences, e.g., cryptographic fault injections.

In order to execute a rowhammer attack, an attacker performs three essential steps. The attacker first opens a row that resides in the same bank as the row in which bit flips want to be induced. Then the attacker performs accesses to the DRAM row, trying to influence the charge leak rate of adjacent rows. In the third step, the attacker removes the accessed DRAM row from the cache hierarchy to ensure the next subsequent accesses will access the DRAM (e.g. with the clflush instruction). Note that steps 2 and three are continuously executed in a loop to try to induce bit flips in the victims DRAM row. Examples of studies in rowhammer attacks are [KDK+14a, GMM16, BM16].

7.4.1.4 Memory Bus Locking Covert Channels

Memory Bus Locking attacks take advantage of pipeline flushes that occur when atomic operations are performed on data that occupies more than one cache line. However, if the atomic operation is performed on data that fits within a single cache line the system locks the cache line for the atomic operation to happen. The pipeline flushing operations cause pipeline flushes that incur in performance overheads. Memory bus locking mechanisms have been used as a method to establish covert channels and derive co-residency in IaaS clouds and as a mechanism to perform Quality of Service degradation (QoS) [VZRS15, IIES16].

7.4.2 Implicit Characteristics of Microarchitectural Attacks

In this section we review those characteristics that well known microarchitectural attacks exhibit in their design. More in particular, we put our focus in three of the most dangerous microarchitectural threats presented in the last years: Cache/-DRAM attacks, memory bus locking and rowhammer attacks.

7.4.2.1 Cache Attacks

As already explained in the background section, cache attacks are implemented by creating cache contentions in any of the caches in the cache hierarchy. This effect, depending on the attack structure, can be accomplished in several ways. However, the majority of cache attacks have three main characteristics in common:

- **High resolution timers** Cache attacks rely on the ability of distinguishing different access patterns (e.g. L1 accesses from LLC accesses or even memory accesses). As these accesses differ at most by a few hundred cycles, the attacker needs to have access to a timer accurate enough to carry out an attack. These timers can be accessed in many different ways (e.g., the common `rdtsc` function or an incremental thread).

- **Memory Barriers** Another common factor of cache attacks is the utilization of memory barriers to ensure serialization before the targeted reads are executed (i.e. making sure any load and stores have finished before the target access). These memory barriers can sometimes be embedded in timer instructions (like the popular `rdtscp` instruction) or can come in the form of different instructions.

- **Cache evictions** The last factor that all cache attacks have in common is the implementation of an eviction routine that removes a target cache line from the cache hierarchy. This can be implemented with the popular flush instruction (in the case of shared memory) or with the creation of eviction sets.

7.4.2.2 DRAM Access Attacks

Similar to cache attacks, DRAM access attacks base their functionality on the capacity of colliding with an attacker in the same row buffer of the same bank, rank, Dual Inline Memory Module (DIMM) and channel. In order to achieve that purpose, DRAM attacks share common characteristics with cache attacks.

- **Cache evictions** DRAM access attacks exploit collisions in the DRAM row buffer, and as such, attackers need the victim to access the DRAM (instead of caches) for memory fetches. Since every time the victim makes an access the memory location would be brought to the cache, the attacker continuously needs to evict memory locations from the cache hierarchy. As with cache attacks, this can be achieved by utilizing specific instructions (i.e. the flush instruction) or with carefully designed eviction set mechanisms.

- **Fine grain timers** In order to retrieve meaningful information, attackers need to distinguish between DRAM row buffer hits and misses. This implicitly requires accesses to fine grain timers.

- **Memory barriers** To prevent out of order execution and obtain precise timing behavior.

7.4.2.3 Rowhammer Attacks

Rowhammer attacks are perhaps the microarchitectural attack with the most dangerous implications that has been discovered in the last decade, due to the ability of flipping bits from memory locations belonging to a victim. In contrast to cache or memory bus locking attacks, rowhammer attacks only have one clearly distinguishable characteristic:

- **Cache bypassing** In fact, rowhammer attacks rely on continuous access to a DRAM location that shares the DRAM bank with the victims memory location. Thus, attacker need to continuously bypass the cache, otherwise the CPU will bring the accessed DRAM portion to the cache for faster access. There are several methodologies to avoid the cache accesses, some of which are similar to those utilized in cache attacks (e.g. flush instructions or eviction sets).

7.4.2.4 Memory Bus Locking Covert Channels

Another popular microarchitectural attack is the ability to stall the memory bus to establish a covert channel between two co-resident processes. As with cache attacks, memory bus locking covert channels also have their own characteristics that need to be accomplished in their design:

- **High resolution timers** As with cache attacks, memory bus locking mechanisms require a fine grain timer to measures the transmission of a 1 or 0 bit depending on the time to execute atomic operations. Further, fine grain timers might also be utilized by attackers willing to ensure the effects of the memory bus lockdown prior to its execution.

- **Lock Instructions** In addition to the fine grain timer, these attacks also require of specific atomic instructions capable of locking the memory bus. In `x86-64` systems these instructions include, among others, the ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, DEC, FADDL, INC, NEG, NOT, OR, SBB, SUB, XADD, XOR instructions with the lock prefix. Further, the XCHG instruction executes atomically when operating on a memory location, regardless of the LOCK use.

7.4.3 Our Approach: MASCAT a Static Analysis Tool for Microarchitectural Attacks

After identifying the main characteristics of the attacks we aim to detect, we propose a novel technique to detect microarchitectural attacks by statically analyzing binaries. All countermeasures proposed to detect microarchitectural attacks so far are based on dynamically detecting specific microarchitectural patterns when the binary is executed. We observe three clear problems with this approach:

- The knowledge on how to install and use such a dynamic analysis tool might not be trivial for every end-user. Thus, although such an approach might well be adoptable by large scale systems (like IaaS clouds), a regular user might not have the technical capabilities to use it.

- The adoption of some of those mechanisms might not be in control of online app distributors, who might be blamed if an application offered in their repository succeeds attacking an end device.

- Cache attacks can be embedded into a binary to be executed only after some specific time/date or event. Thus, the system should be monitoring the microarchitectural patterns of the binary every time it is executed and not only once. Thus such tools can create significant overhead in the system.

Our approach solves these issues. Upon the reception of a potentially malicious binary, the most common approach to validate its sanity before it is executed is to utilize an antivirus software. Although antivirus tools might work well with certain kind of malware [SKH11], their success when detecting microarchitectural attacks is still an open question. Table 7.3 shows the outcome of such an analysis using well known antivirus software (the best in 2016, as stated in [Lib]). Indeed, we utilized all our examples of cache, DRAM, rowhammer and memory bus locking attacks.

Table 7.3: As of 12.28.2016, none of the major antivirus tools detects the majority of the attacks we analyzed (✗ means non-detected and ✓ means detected). The only exception is Drammer, which is detected by 7 out of 9 antiviruses

Antivirus software	**Output cache/ /DRAM attack**	**Output rowhammer**	**Output bus locking**	**Output Drammer [vdVFL+16]**
Avast	✗	✗	✗	✓
BitDefender	✗	✗	✗	✓
Emsisoft	✗	✗	✗	✗
ESET-NOD32	✗	✗	✗	✓
KasperSky	✗	✗	✗	✓
F-secure	✗	✗	✗	✓
McAfee	✗	✗	✗	✓
Symantec	✗	✗	✗	✓
TrendMicro	✗	✗	✗	✗

None of the antivirus softwares were able to detect that the binaries were malicious, except for the drammer APK [vdVFL+16], which was detected by 7 of 9 antivirus tools.

In order to cope with this problem, we propose MASCAT, a tool that detects intrinsic characteristics of microarchitectural attack code in potentially malicious binaries. MASCAT consists of several fully automated scripts that are executed by the well known IDA Pro static binary disassembler. Thus, MASCAT can be directly adopted by any person using IDA Pro or can be directly translated to any other static binary disassembler. Furthermore, it can be offered as an online scanner for microarchitectural attack code. The following section details on the methodology that our static analysis approach uses to detect malicious microarchitectural attacks.

7.4.3.1 Attributes Analyzed by MASCAT

This section summarizes the attributes that MASCAT detects within the binary code. Note that the goal of our design is not to detect all possible microarchitectural attack designs but rather to give a good framework on the detection of existing attacks. More attributes can be added to cover more sophisticated and intelligent designs of microarchitectural attacks. All the attributes below are critical only when used in a loop, with the exception of the affinity assignment. Thus, we mark them only if we observe them inside loops.

- **Cache flushing instructions:** Cache flushing instructions are included in the ISA to perform cache coherency operations often needed in parallel processing. These instructions are commonly used for both cache attacks as well as DRAM access and rowhammer attacks, as they allow the user to evict a memory block from the cache. Intel utilizes the `clflush` instruction for this purpose, while ARM utilizes the `DC CIVAC` instructions (only available from ARMv8 on).

Since these attacks require several accesses to succeed, our tool identifies both instructions when being used inside a loop.

- **Non-temporal instructions:** Non-temporal instructions permit the access to a memory location bypassing the cache hierarchy, i.e., making a direct DRAM access. Obviously, these instructions can be utilized as part of a rowhammer attack, which requires several accesses to the same DRAM bank. Examples of these instructions are the `monvnti` and `movntdq` for Intel or `STNP` for ARM. Again, the tool marks these instructions if they are utilized in a loop.

- **Monotonic timers, thread counters, performance counters, specialized fine-grain timer instructions:** Fine grain timers are utilized by cache and DRAM attacks and by bus locking covert channels. Such timers can be built utilizing specialized instructions like the `rdtscp` and `rdtsc` instructions on Intel or the c9 register on ARM. In contrast to `rdtsc`, `rdtscp` further implements memory barrier instructions before reading the clock cycle counter. Similarly, performance counters can be queried to get the number of elapsed clock cycles or the number of evictions. If none of these natural counters are accessible, a timer can be created by using a thread continuously incrementing a counter. Lastly, and by far the noisiest method, one can utilize access to monotonic timers offered by the OS. Our tool identifies any of these timers if they are repeatedly accessed in loops.

- **Memory barrier instructions:** These instructions are necessary to prevent out of order execution and to obtain accurate timings from the fine grain timers. Instructions that execute memory barrier operations in `x86-64` are `lfence` (to ensure all loads have finished) and `mfence` and `cpuid` loads (to ensure all loads and stores have finished). In ARM, this can be achieved by the `DSB`, `ISB` and `DMB` instructions.

- **Lock instructions:** Atomic instructions that can be issued to implement memory bus locking attacks. Our tool monitors for all the instruction with the lock prefix plus the XCHG instruction in `x86-64` binaries, as well as for `LDREX` and `STREX` instructions in ARM binaries.

- **Jump opcodes:** L1 instruction cache attacks are commonly designed utilizing jump opcodes to jump to set concurrent addresses. Our tool again analyzes whether any of the functions inside a binary assigns jump opcodes in a loop.

- **Pointer chasing & Page size jump approaches:** One of the approaches that can be taken to create cache eviction sets (which can be utilized for cache, DRAM access and rowhammer attacks) is the pointer chasing approach, in which the address of the next address is stored in contents of the previous address. Another approach is to introduce jumps of ℓ bytes within a vector,

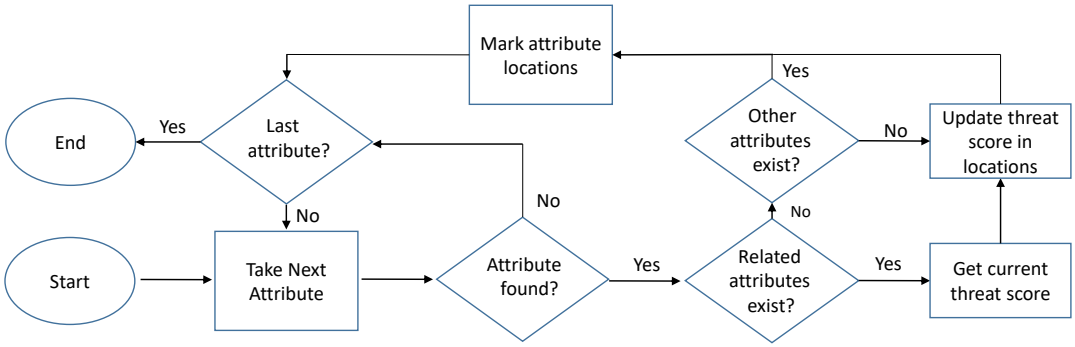


Figure 7.21: Attribute analysis and threat score update implemented by MASCAT.

where ℓ is the necessary number of bytes to find set-concurrent addresses. Our tool identifies both approaches when utilized in binary code.

- **Selfmap translation & slice mappings** Although non-available from user mode since linux kernel 4.0.0, prior kernels are still vulnerable to having an attacker looking at the physical address of his memory to facilitate eviction set creations. Aiming at avoiding these attacks, our tool also identifies accesses to this particular mapping. Further, in the case of `x86-64` binaries, our tool also finds whether the *known* slice selection algorithms are utilized in the binary code to guess the slice location of the memory addresses.

- **Affinity assignment** Some of the above mentioned attacks (like the L1 cache attacks or LLC attacks) only succeed when core co-residency or CPU cluster co-residency are achieved. Further, the timers (and especially the thread counter) exhibit better performance when scheduled on a single core. Thus, our tool also tries to identify whether there is any affinity assignment inside the inspected binary code.

7.4.3.2 Measuring the Threat Score

One of the most important challenges of the design of our tool is to determine, based on the attributes observed, whether a threat exists or not. Although we could use machine learning algorithms to design such a methodology, we believe there are several facts that have to be smartly taken into account (e.g. location of the attributes, location of the nested functions calling the functions where the attributes were found, etc.) that can make machine learning algorithms to perform poorly. In contrast, we design MASCAT to be intelligent enough to retrieve those facts, such that we do not have a necessity for utilizing an automated data analysis tool. In particular we design our tool to retrieve:

- **Location of an attribute and functions interacting with it:** Particularly useful is the knowledge of the location in which these attributes are observed.

MASCAT is designed to retrieve all locations from which the code can reach the location where an attribute is observed, i.e., all possible functions that can reach the location in which the attribute was found.

- **Interaction between attributes:** As stated before, a key factor of microarchitectural attacks is not only the attributes they exhibit, but also the relation between them. In that sense, MASCAT takes into account where and how attributes interact with each other to decide whether a microarchitectural attack exists.

In order to achieve the goals described above, we decide to implement a score system based on the combination of attributes observed. The main idea is represented in Figure 7.21. MASCAT analyzes, one by one, and *ordered by threat magnitude* (from maximum to minimum) all the characteristics we described in Section 7.4.3.1. For each of them, if the attribute is found in the binary, MASCAT first checks whether a related characteristic exists. If they do exist, the threat score is updated for that binary location. For instance, we know memory barrier instructions are not a threat if they are not issued together with timers, lock or eviction instructions. If no related attributes are found, our tool continues to detect whether any other attributes exist. Indeed, if no attributes exist at all, the threat score has to be updated (since this is the first attribute found in that location). However, if other attributes are found, then the thread score is not changed as a higher threat level has already been assigned. In any of the above cases, the characteristics are marked to be present (if found) in the appropriate locations. The relations between characteristics are defined according to the description given in Section 7.4.2. In order to detect the entire range of microarchitectural attacks discussed, we designed the following score based threat classification:

Maximum Threat Attributes:

Attributes from this category are immediately considered a threat when found by our algorithm. Specific flushing instructions, non-temporal stores and selfmap translation attributes fall into this category. The first two because they can directly imply a rowhammer attack, the latter one because having access to the physical address space can become a threat for both cache and DRAM access rowhammer attacks. The location where these attributes (and the functions calling them) are found will be marked in red.

Medium Threat Attributes:

Attributes from this category are not immediately considered threats but rather warnings. The locations in which these attributes are found will only be considered a threat if the rest of the necessary attributes (as described in Section 7.4.2) are found within the same location of the code. Examples of these attributes are `rdtscp`

and `lock` instructions, as well as pointer chasing, set-concurrent jumps and jump opcode assignments. A combination of two related medium threat attributes will immediately be considered a threat, while a combination of two non-related medium threat attributes will not change the score. For instance, `rdtscp` and `lock` instructions together would be considered a threat, while `lock` instructions with pointer chasing functions are not (see Section 7.4.2).

Low Threat Attributes:

Attributes from this category are not immediately considered threats but rather small warnings. Only if these attributes are observed within the same location of necessary medium or low threat attributes the code portion is considered a threat. Examples of these kind of attributes are memory barriers, timers (excluding `rdtscp`) and affinity assignments. Only if two low threat attributes are observed with a medium threat attribute the code location is classified as a threat.

7.4.3.3 Tool Framework

In order to implement the specified design we utilize the popular IDA PRO static binary analyzer. We chose IDA because it offers a high level programming language and several built-in functions that facilitate our attribute finding design. Note that developing our tool with another open source static binary analyzer (e.g. hooper) should also be feasible with additional work.

In summary, MASCAT works by analyzing elf files *without the need for the source code*. The tool detects the attributes described above, coloring those locations where microarchitectural attack related features are found. Our tool colors threats in red, warnings in orange and small warnings in magenta. Note that, as MASCAT finds new attributes in already colored locations, the colors of those locations might change. Once the analysis is finished, MASCAT outputs a summary of the locations in which threats and warnings were found (by scanning the color of those locations) and their root of cause (e.g., `rdtsc` instruction, `lock` instructions, etc.). Figure 7.22 shows a visual example of an analysis made to one of our microarchitectural attacks. The binary is considered a threat for utilizing both flush and fine grain timer instructions.

While our analysis on `x86-64` binaries is performed in a single step, our analysis on Android APKs needs to be performed in a dual step. First, the attributes that we look for are scanned within the existing native code in the APK, as with `x86-64` binaries. However, we find this non-sufficient, as although the necessary attributes might be found, they might only interact together in the Dalvik Executable (DEX) code. For that reason, the DEX file is analyzed in a second step to check its interaction with the native code, trying to find whether the attributes interact with each other. This approach prevents an attacker from creating different microarchitectural attack functions that interact and are called in the DEX file.

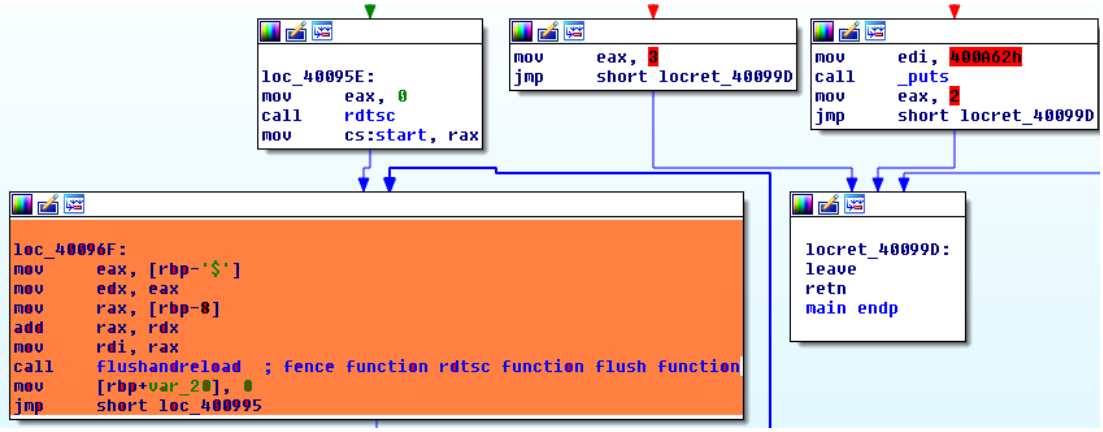


Figure 7.22: Visual example output of MASCAT, in which a flush and reload attack is detected

7.4.4 Experiment Setup

Our analysis includes the Mastik tool (which performs all range of L1 and LLC attacks) designed by Yarom et al. [Yar16], LLC attacks and memory bus locking attacks designed by Inci et al. [IGES16] and Irazoqui et al. [IIES14b, IES15a, IGI+16b], LLC, cache prefetching, DRAM access and rowhammer attacks designed by Gruss et al. [GSM15, PGM+16, GMF+16, GMM16], rowhammer attacks by Van der Veen et al. [vdVFL+16], rowhammer attacks designer Google Project Zero (which were modified to cover [QS16]), L1 cache attacks from Tampere university [ABG10], all versions of cache attacks for ARM designed by Lipp et al. [LGS+16] and DRAM access attacks designed by Abraham Fernandez, former Intel employee. Some of these codes were modified to cover several corner cases (e.g., thread created timers, eviction set creations, non-temporal accesses, doble-sided rowhammer, etc.). In total, we cover a total of 32 different microarchitectural attacks.

Our false positive analysis covers both x86-64 binaries compiled under Linux OSs and ARM targeted Android APKs. As the `x86-64` benign binary set that was included for our test framework, we chose around 750 binaries from the Ubuntu Software Center plus the Phoronix test-suite [pho]. The Phoronix test-suite choice was not random, but was intended to maximize the number of false positives that our tool can output, as it contains performance benchmarks that might use some of the attributes that our tool looks for. In the case of android APKs we analyzed all the applications offered by `www.androiddrawer.com`, which contain applications with a huge variety of functionalities, e.g., media streaming or communication. In total we have analyzed 1268 Android applications.

Table 7.4: Percentage of attacks correctly flagged by MASCAT (true positives).

Attack Type	Number of attacks	% of attacks correctly flagged
Cache attacks	25	100%
Rowhammer	5	80%
Bus Locking	2	100%
Total	**32**	**96.87%**

7.5 Results

This section presents the results obtained both from analyzing (a priori) benign and malicious applications. The idea behind this analysis is to obtain a notion for the false negative/positive rate that MASCAT presents in real world applications. We first present an analysis on the detection rate of all the microarchitectural attack codes that we were able to obtain. Next, in order to obtain a good estimation on the false positive rate that MASCAT presents with regular-purpose binaries, we analyze over 700 and a thousand Ubuntu Software Center and Android applications respectively.

7.5.1 Analysis of Microarchitectural Attacks

We evaluate the full range of microarchitectural attacks that we described in Section 7.4.4, i.e., a list of 32 different microarchitectural attacks. Note that these attacks include both `x86-64` binaries and Android APKs, compiled with Android NDK. The results, are presented in Table 7.4, which expresses the number of attacks that were correctly flagged detected by MASCAT.

We observe that the overall success rate of MASCAT was of 96.87% for the 32 malicious codes analyzed. In particular, MASCAT was able to correctly identify all cache attacks and bus locking covert channels, and all but one rowhammer attack codes. The only code that MASCAT was not able to identify was Drammer [vdVFL+16]. The reason behind it is that the authors utilize special DMA memory allocation that does not get loaded in the cache, and therefore no flushing, non-temporal loads nor eviction sets are necessary to apply it. However, as we stated in Section 7.4.3, Drammer is the *only attack that is identified by regular commercial antivirus tools*. Thus, we believe that MASCAT can be used in conjunction with regular malware antivirus tools to catch the entire attack spectrum.

7.5.2 Results for Benign Binaries

As for the `x86-64` benign binaries, the results are presented in Table 7.5, in which binaries are divided into groups as we downloaded them from the Ubuntu Software Center. We observed that even though binaries with at least one attribute of those

Table 7.5: Results for different groups of binaries from Ubuntu Software Center.

Group functionality	Number of binaries	% of binaries with at least one attribute	% of binaries considered threat
accessories	122	18	0
education	73	19.17	0
developer tools	96	23.9	0
games	116	42.2	0
graphics	84	36.9	1.2
internet	76	25.0	2.63
office	75	12	0
phoronix	100	56	6
total	**742**	**30.05**	**1.2**

we were looking for are rather common (30%), we only had 9 binaries in total that were categorized as threats. Particularly interesting is the case of including the Phoronix-test suite in our analysis, as 6 of the 9 binaries categorized as threats were part of it. This is not that surprising considering that Phoronix test-suite binaries are usually benchmarking applications. The overall false positive rate was 1.2%, but could have been lowered without considering the benchmarking applications.

Table 7.7 shows the reason why those 9 binaries flagged as threats. Nqueens, multichase, mencoder and fs_mark benchmarks flagged due to the usage of non-temporal instructions. Aseprite, Claws mail, Fldigi, Fio and ffmpeg binaries flagged because of the usage of timers, memory barriers and pointer chasing/set congruent jump instructions within the same location. Lastly fs_mark and again ffmpeg flagged because of the usage of timers and atomic instructions, indicating a potential memory bus locking covert channel exploitation.

In the case of Android applications, a total of 1268 APKs were analyzed, for which table 7.6 presents per group threat detection statistics, including the percentage of APKs within the group that presented at least one of the attributes we look for, and the percentage of APKs that were considered a threat. Our overall results indicate that the number of APKs that might have at least one of the attributes that we look for is rather high, almost 1/3 of the APKs. However, our results also indicate that the number of APKs that we found to be flagged as threat is very low, i.e., only 4 of the total of 1268 APKs were considered threats. These applications are google allo, azar, eve period tracker and spendless. As before, Table 7.7 shows that the 4 APKs flagged because of the same reason: a combination of timers, fences and pointer chasing loop eviction mechanisms that can be an indicator of both cache/-DRAM attacks. Thus, we observe that the combination of attributes necessary to develop an attack is rather difficult to observe in benign applications, giving a very low number of false positives.

In summary covering the 4 microarchitectural attack range only lead us to 97% true positive rate in benign benchmarking code and an average of 0.75% when we analyzed a total of around two thousand regular purpose applications.

Table 7.6: Results for different groups of APKs.

Group functionality	**Number of APKs**	**% of APKs with at least one attribute**	**% of APKs considered threat**
books	28	42	0
cars	3	0	0
comic	10	30	0
communication	130	32.3	1.53
education	49	16.3	0
entertainment	48	31.25	0
finance	35	22.8	2.86
food	1	100	0
games	130	32.3	0
health	65	24.61	1.53
libraries-demo	2	0	0
life style	30	26.6	0
media video	25	64	0
music audio	39	48.7	0
magazines	52	19.2	0
personalization	59	20.33	0
photography	64	50	0
productivity	144	22.2	0
shopping	24	29.1	0
social	59	32.2	0
sports	16	20	0
tools	175	23.4	0
transport	20	20	0
travel	38	34.2	0
video transport	1	100	0
weather	21	19	0
total	**1268**	**28.3**	**0.31**

Table 7.7: Explanation for benign binaries classified as threats.

binary flagged	**Reason**
Ubuntu Software Center:	
aseprite	timers, memory barriers and pointer chasing
claws mail	timers, memory barriers and pointer chasing
ffmpeg	timers, memory barriers pointer chasing and lock
fio	timers, memory barriers and pointer chasing
fldigi	timers, memory barriers and pointer chasing
fs_mark	timers, lock and non-temporal load/store
mencoder	non-temporal load/store
multichase	non-temporal load/store
nqueens	non-temporal load/store functions
Android APKs:	
google allo	timers, memory barriers, pointer chasing
azar	timers, memory barriers, pointer chasing
eve period	timers, memory barriers, pointer chasing
spendless	timers, memory barriers, pointer chasing

7.5.3 Limitations

As with any other static analysis approach, we need to take into account the possible obfuscation techniques that an attacker can implement to bypass MASCAT:

- **CISC instruction offset obfuscation:** An attacker can take advantage of the variable instruction length in CISC ISAs by inserting arbitrary amount of junk bytes that would later be unreachable during run time. In this way, static disassemblers mistakenly interpret a large part of the binary. As our approach largely depends on a good interpretation of the instructions utilized, this approach would indeed bypass our current version of MASCAT. However, multiple de-obfuscation techniques have been proposed to cope with this issue, both for other disassemblers and for IDA Pro [KRVV04, Rol16]. Thus, our tool can be utilized in conjunction with one of these approaches to avoid the successful usage of such obfuscation techniques.

- **RISC instruction offset obfuscation:** An attacker can also introduce junk padding bytes on RISC arichtectures, aiming at distracting our static disassembler. However, since the ARM architecture features a fixed length ISA, our defense mechanism would only have to be run with four different offsets (0, 1, 2 and 3 byte offset) to be able to cover all possibilities, thereby detecting the obfuscation.

- **Encrypted executable code:** Another popular approach to bypass static analysis is to have a binary that contains, in an encrypted form, the code containing a microarchitectural attack. When the binary is executed, the microarchitectural attack code is decrypted and saved into another binary file, which is subsequently executed. Such an approach would bypass our system, but fortunately, it is very unlikely that it bypasses other commercial antivirus software. In order to cope with this issue we suggest utilizing MASCAT *together with* other existing antivirus tools: not only to detect this particular malicious case, but to detect whether the binary contains further non-microarchitectural malware.

- **Alternative mechanisms/covert channels to execute microarchitectural attacks:** An attacker, knowing the parameters we are looking for, can try to bypass our approach by discovering and implementing alternative methodologies to execute an attack. One of the ways to defend against this is to hide the approach that MASCAT performs, so that the attacker does not have a motivation to change his attack patterns. An alternative approach is to keep adding features to MASCAT as new attack methodologies arise. Indeed ,we encourage the research community to vary their mechanisms so that we can incorporate them to MASCAT and target a broader set of characteristics.

7.5.4 Outcomes

We presented the first static code analysis tool, MASCAT capable of scanning for common microarchitectural attacks. Microarchitectural attacks are particularly damaging as they are hard to detect since they use standard resources, e.g. memory accesses, made available to applications at the lowest privilege level. Given the rise of malicious code in app stores and online repositories it becomes essential to scan applications for such stealthy attacks. The proposed tool, MASCAT, is ideally suited to fill this need. MASCAT can be used by app store service providers to perform large scale fully automated analysis of applications, and prevent a malicious user from posting a microarchitectural attack in the guise of an innocent looking application to an official app store. The initial MASCAT suite includes attack vectors to cover popular cache/DRAM access attacks, rowhammer and memory bus covert channels. Nevertheless, our tool is easily extensible to include newer attack vectors as they arise.

Chapter 8

Conclusion

In the last years we witnessed the large adoption of hardware co-resident multi-tenant/application solutions. The confidence that customers place on these solutions is mainly based on the fact that their applications are executed in sandboxed environments that should, en theory, ensure proper customer isolation. Examples of these techniques are implemented, among others, in commercial IaaS clouds or smartphones. However, such mechanisms mostly focus on achieving good software isolation, while they disregard the potential threats due to concurrent hardware usage. Our work responds to such a concern; first by showing that microarchitectural attacks bypass sandboxing techniques and are able to recover fine grain information, second, by proposing mechanisms to avoid such exploitation from succeeding.

Our work demonstrates that exploiting hardware leakages in realistic widely adopted scenarios like cloud computing infrastructures can become a major threat for hardware co-resident users. In particular, we utilized the shared last level cache to infer information, which offers both high resolution and applicability at least across cores. We introduced three main attack designs, *Flush and Reload*, *Invalidate and Transfer* and *Prime and Probe*, each of which has different characteristics and can be applied in different scenarios. The first two are low noise attacks that can be implemented in systems where memory deduplication mechanisms are implemented, e.g., default configurations of VMware or the KVM hypervisor. The latter does not require special assumptions to succeed, and thus, is applicable in all major commercial clouds, as demonstrated by our successful attack in Amazon EC2. Further, the proposed attack demonstrated to recover fine grain information like AES and RSA keys and TLS session messages across VMs. Any of these key retrievals happening in non-academic environments would have severe security implications for the victim, as the attacker would be able to, e.g., read encrypted communications or impersonate him.

Despite the severity of the threat that microarchitectural attacks posed in existing commercial software and the popularity they acquired in the last years in academic environments, we have not observed existing countermeasures being deployed in the systems that we tried to exploit. Examples of potential countermea-

sures include page coloring, performance hardware event monitorization or cache line locking. However, all of them introduce big performance overheads that seems that commercial system designers are not willing to pay. In response to this concern, we propose two countermeasures that can help cache attacks from being widely adopted while not suffer from continuous performance costs. The first verifies the correctness of cryptographic software to ensure no cache leakage is present. We believe that one of the safest mechanisms to avoid sensitive information from being stolen is ensure the proper design of the security solution. For that purpose, our tool finds key dependent data that is verified for leakage. Although we found alarming numbers (50% of the implementations leaked information), we worked closely with the designers to address them. The second countermeasure we propose is MASCAT, a tool that detects microarchitectural attacks embedded in binaries to prevent their distribution in official app repositories, emulating a "microarchitectural attack antivirus"'. Currently online application distributors lack of proper mechanisms to verify that the binaries being offered are microarchitectural attack free. MASCAT is the perfect tool to be adopted for such a purpose.

In short, our work demonstrated that microarchitectural attacks have to be considered one of many threats that systems relying on sandboxing techniques have to take into account. In this sense, our final goal was to help building stronger security solutions that ensure the privacy of those using them, specially, as the number of scenarios in which hardware co-resident applications are concurrently executed increases. We believe that the adoption of the proposed (and other) countermeasures is highly necessary to defeat the threat, and thus, encourage security solution designers to make it happen. Their goal and ours is indeed the same: guarantee the protection of our data.

Outcomes of Our Work

The microarchitectural attacks that we exploited are based on both specific characteristics of the systems in which they were carried out and exploitation of vulnerabilities on widely used open source cryptographic libraries. Thus, our work required several vulnerability analysis skills. We believe it is important to verify and strengthen the design security solutions against all possible attack vectors, including the microarchitectural attacks we exploited. We understand we have a responsibility to work with the designers of those products that we exploited. Therefore, we notified all of them about our findings, including but not limited to Amazon EC2, VMware, Intel, AMD and several cryptographic library designers. To help them fix the vulnerabilities, we offered our help to reproduce the attack and proposed possible solutions. In fact, many of our results resulted in a security update of the exploited product.

In Section 4, we developed our attack mainly utilizing VMware as our default hypervisor. Thus, we decided to alert them about the threat that such memory sharing mechanisms could add to virtualized environments. We also aid them re-

producing the attack described in Section 4.4 in their systems. After successfully reproducing our results, VMware decided to disable Transparent Page Sharing by default (CVE-2014-3566), and leave it as a add-on that end-users can activate understanding the risk that it poses. A better description explaining the reasons to change the feature settings can be found in [VMwa].

In Section 4.5 we further exploited vulnerabilities in the design of the TLS code, mainly involving secret dependent non-constant execution flow. All the leakage cases observed in the section were notified accordingly to the appropriate cryptographic library designers. However, only PolarSSL (now MbedTLS) decided to notify us about the changes made to avoid the leakage, and asked us to verify the correctness of the new implementation, although without issuing a CVE number.

In Section 7.3 we performed an analysis of the correctness of the design of AES, RSA and ECC cryptographic algorithms in 8 main cryptographic libraries. We found that 50% of them leaked information that could lead to secret key retrieval. We disclosed all our findings to the appropriate cryptographic library designers. Although some of the vulnerabilities are still being studied and fixed, our work already resulted in many of these vulnerabilities fixes. WolfSSL updated the AES, RSA and ECC design in response to our leakage analysis, as described in CVE numbers 2016-7438, 2016-7439 and 2016-7440 respectively. Bouncy Castle updated the AES design to avoid the leakage exploited in this work, as described in CVE number 2016-10003323. Lastly, the RSA implementation of Intel's closed source cryptographic library was updated after we found that a vulnerability could lead to the full secret key extraction, and received the CVE-2016-8100 number.

Bibliography

[ABB+16] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2010.

[ABL97] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[Acı07] Onur Acıiçmez. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, 2007.

[AE13] Hassan Aly and Mohammed ElGayyar. Attacking AES Using Bernstein's Attack on Modern Processors. In *AFRICACRYPT*, pages 127–139, 2013.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by Using KSM. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[AKKS07] 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. 2007.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[Alp09] Seeking Alpha. Looking At Nvidia's Competition, April 2009.

[AMD] AMD. HyperTransport Technology white paper. `http://www.hypertransport.org/docs/wp/ht_system_design.pdf`.

[ARM09] ARM. ARM Security Technology - Building a Secure System using TrustZone Technology, April 2009.

[AS07] Onur Aciicmez and Jean-Pierre Seifert. Cheap Hardware Parallelism Implies Cheap Security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[AS08] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and its Demonstration on OpenSSL. In *Topics in Cryptology–CT-RSA 2008*, pages 256–273. Springer, 2008.

[BB03] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[BECN+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[BH09] Billy Bob Brumley and Risto M Hakala. Cache-Timing Template Attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 667–684. Springer, 2009.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-Collision Timing Attacks against AES. In *CHES 2006*, volume 4249 of *Springer LNCS*, pages 201–215, 2006.

[BM15] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms. In *Cryptographic Hardware and Embedded Systems – CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 248–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BM16] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In Benedikt Gierlichs and Y. Axel Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 602–624, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BMD+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *ArXiv e-prints*, February 2017.

[Bri] Intel Sandy Bridge. Intels Sandy Bridge Microarchitecture. `http://www.realworldtech.com/sandy-bridge/`.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14,2011. Proceedings*, pages 355–371, 2011.

[BvdPSY14] 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.

[CCC+13] Wei Chen, Szu-Liang Chen, Siufu Chiu, R. Ganesan, V. Lukka, W.W. Mar, and S. Rusu. A 22nm 2.5MB Slice on-Die L3 Cache for the Next Generation Xeon Processor. In *VLSI Circuits (VLSIC), 2013 Symposium on*, pages C132–C133, June 2013.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CFS] CFS. CFS Scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[CJ06] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, 2006.

[CKD+10] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[CRR03] Suresh Chari, JosyulaR Rao, and Pankaj Rohatgi. Template Attacks. pages 13–28, 2003.

[CS04] Matthew J. Campagna and Amit Sethi. Key recovery method for CRT implementation of rsa. Cryptology ePrint Archive, Report 2004/147, 2004. `http://eprint.iacr.org/`.

[CSY15] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. Technical report, Cryptology ePrint Archive, Report 2015/1034, 2015.

[CVBS09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 45–60, Washington, DC, USA, 2009. IEEE Computer Society.

[CWR09] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and Limits of Remote Timing Attacks. *ACM Trans. Inf. Syst. Secur.*, 12(3):17:1–17:29, January 2009.

[D'A15] Sophia D'Antoine. Exploiting Out of Order Execution For Covert Cross VM Communication. In *Blackhat 2015*, Las Vegas, Aug 2015.

[DMWS12] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 106–117, Washington, DC, USA, 2012. IEEE Computer Society.

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.

[DSKM+13] Fabrizio De Santis, Michael Kasper, Stefan Mangard, Georg Sigl, Oliver Stein, and Marc Stöttinger. On the Relationship between Correlation Power Analysis and the Stochastic Approach: An ASIC Designer Perspective. In Goutam Paul and Serge Vaudenay, editors, *Progress*

in Cryptology INDOCRYPT 2013, volume 8250 of *Lecture Notes in Computer Science*, pages 215–226. Springer International Publishing, 2013.

[Dwo04] Morris J. Dworkin. Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical report, Gaithersburg, MD, United States, 2004.

[EP16] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 843–857, New York, NY, USA, 2016. ACM.

[Fan15] Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level Cache Side Channel Attacks are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 605–622, Washington, DC, USA, 2015. IEEE Computer Society.

[FP13] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, SP '13, pages 526–540, May 2013.

[GBK11] 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.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2016*, pages 426–442. Springer, 2008.

[GMF+16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 300–321, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, pages 279–299, Cham, 2016. Springer International Publishing.

[GPPT15] 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.

[GSM15] 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.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *CRYPTO 2014*, pages 444–461, 2014.

[Ham09] Mike Hamburg. Accelerating AES with Vector Permute Instructions. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009: 11th International Workshop Lausanne, Switzerland, September 6-9, 2009 Proceedings*, 2009.

[Ham13] Mike Hamburg. Bit level Error Correction Algorithm for RSA Keys. Personal Communication, Cryptography Research, Inc., 2013.

[HDWH12] 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.

[hen] Intel Ivy Bridge Cache Replacement Policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`.

[HFFA10] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Near-Optimal Cache Block Placement with Reactive Nonuniform Cache Architectures. *IEEE Micro's Top Picks*, 30(1):29, 2010.

[HGLS07] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA, 2007. ACM.

[HMV03] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th edition, 2011.

[HWH13] 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, 2013.

[i65] Intel i650. Intel Core i5-650 Processor . `http://ark.intel.com/es/products/43546/Intel-Core-i5-650-Processor-4M-Cache-3_20-GHz`.

[i74] Intel i7402M. Intel i7-4702M Processor . `http://ark.intel.com/products/75119/Intel-Core-i7-4702MQ-Processor-6M-Cache-up-to-3_20-GHz`.

[IES15a] 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 *36th IEEE Symposium on Security and Privacy (S&P 2015)*, pages 591–604, 2015.

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Proceedings of the 2015 Euromicro Conference on Digital System Design*, DSD '15, pages 629–636, Washington, DC, USA, 2015. IEEE Computer Society.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 353–364, New York, NY, USA, 2016. ACM.

[IGES16] Mehmet Sinan İncİ, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Co-Location Detection on the Cloud. In *Constructive Side-Channel Analysis and Secure Design: 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, pages 19–34, Cham, 2016. Springer International Publishing.

[İGI+16a] Mehmet Sinan İnci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016:*

18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, 2016.

[IGI+16b] Mehmet Sinan İncİ, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813, page 368. Springer, 2016.

[IIES14a] Gorka Irazoqui, Mehmet Sinan Inci, 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, 2014.

[IIES14b] Gorka Irazoqui, Mehmet Sinan İncİ, Thomas Eisenbarth, and Berk Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 299–319, Cham, 2014. Springer International Publishing.

[IIES15] Gorka Irazoqui, Mehmet Sinan İncİ, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 85–96, 2015.

[IIES16] Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Efficient Adversarial Network Discovery Using Logical Channels on Microsoft Azure. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 436–447, New York, NY, USA, 2016. ACM.

[iM] Intel i5 4200M. Intel Core i5-4200M Processor. `http://ark.intel.com/es/products/75459/Intel-Core-i5-4200U-Processor-3M-Cache-up-to-2_60-GHz`.

[inta] Intel Core i5-650 Processor . `http://ark.intel.com/es/products/43546/Intel-Core-i5-650-Processor-4M-Cache-3_20-GHz`.

[intb] Intel Core Xeon E5-2609 Processor . `http://ark.intel.com/products/64588/Intel-Xeon-Processor-E5-2609-10M-Cache-2_40-GHz-6_40-GTs-Intel-QPI`.

[intc] Intel Core Xeon E5-2640 Processor . `http://ark.intel.com/es/products/83359/Intel-Xeon-Processor-E5-2640-v3-20M-Cache-2_60-GHz`.

[Intd] Intel. An Introduction to the QuickPath Interconnect. http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[Inte] Intel. How to Use Huge Pages to Improve Application Performance on Intel Xeon Phi Coprocessor . https://software.intel.com/sites/default/files/Large_pages_mic_0.pdf.

[IQP] IQP. Intel QuickPath Architecture. http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf.

[JC07] Lei Jin and Sangyeun Cho. Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring. In *Proc. Intl Workshop Chip Multiprocessor Memory Systems and Interconnects*, CMP-MSI '07, 2007.

[Jon10] 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/, 2010.

[JY03] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *Cryptographic Hardware and Embedded Systems - CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers*, pages 291–302, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[KASZ09] J. Kong, O. Aciicmez, J.-P. Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009*, HPCA '09, pages 393–404, Feb 2009.

[KDK+14a] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[KDK14b] G. Kurian, S. Devadas, and O. Khan. Locality-Aware Data Replication in the Last-Level Cache. In *IEEE 20th International Symposium on High Performance Computer Architecture 2014*, HPCA '14, pages 1–12, Feb 2014.

[KJC+10] Joonho Kong, Johnsy K John, Eui-Young Chung, Sung Woo Chung, and Jie Hu. On the thermal Attack in Instruction Caches. *IEEE Transactions on Dependable and Secure Computing*, 7(2):217–223, 2010.

[KJJ99] 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. 1999.

[KJJR11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[Koc94] C. K. Koc. High-Speed RSA Implementation. `ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf`, 1994.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, 1996.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

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

[KSS10] M. Kasper, W. Schindler, and M. Stottinger. A stochastic method for security evaluation of cryptographic FPGA implementations. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 146–153. IEEE, December 2010.

[KVM] KVM. Huge Page Configuration in KVM. `http://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaattunconfighp.htm?lang=en`.

[LD00] Julio Lpez and Ricardo Dahab. An overview of elliptic curve cryptography. Technical report, 2000.

[LGS+16] 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, Austin, TX, USA, August 10-12, 2016.*, pages 549–564, 2016.

[LGY+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE Symposium on High-Performance Computer Architecture*, HPCA '16, pages 406–418, Barcelona, Spain, mar 2016.

[Lib] Best 2016 antivirus. `http://www.pcmag.com/article2/0,2817,2388652,00.asp`.

[Lin] Linux. Cross-Referenced Linux and Device Driver Code.

[LL14] Fangfei Liu and Ruby B Lee. Random Fill Cache Architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Micro '14, pages 203–215. IEEE, 2014.

[LSG+16] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. *arXiv preprint arXiv:1603.06952*, abs/1611.06952, 2016.

[MHSM09] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261–270, Sept 2009.

[Mic] Microsemi. Side Channel Analysis. `https://www.microsemi.com/products/fpga-soc/security/side-channel-analysis`.

[Mil86] Victor S Miller. Use of elliptic curves in cryptography. In *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[ML09] A. Miri and P. Longa. Accelerating scalar multiplication on elliptic curve cryptosystems over prime fields, March 14 2009. CA Patent App. CA 2,602,766.

[MRR+15] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 865–880, 2015.

[MSN+15] Clmentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurlien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters . In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 48–65, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[MWX+15] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, Washington, D.C., 2015. USENIX Association.

[NS07] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 147–162. 2007.

[OKSK15] 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.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Paa15] C. Paar. High-Speed RSA Implementation. `https://www.emsec.rub.de/media/attachments/files/2015/09/IKV-1_2015-04-28.pdf`, 2015.

[Pag05] Daniel Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. IACR Cryptology ePrint Archive, 2005. `https://eprint.iacr.org/2005/280.pdf`.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[PGM+16] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[pho] Phoronix Test Suite Tests. https://openbenchmarking.org/tests/pts.

[PR09] Emmanuel Prouff and Matthieu Rivain. Theoretical and practical aspects of mutual information based side channel analysis. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, 2009.

[Puk94] Friedrich Pukelsheim. The Three Sigma Rule. *The American Statistician*, 48(2):88–91, 1994.

[QS16] Rui Qiao and Mark Seaborn. A New Approach for Rowhammer Attacks. *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–166, 2016.

[RDNG16] Nitin Rathi, Asmit De, Helia Naeimi, and Swaroop Ghosh. Cache Bypassing and Checkpointing to Circumvent Data Security Attacks on STTRAM. *arXiv preprint arXiv:1603.06227*, 2016.

[rdt] How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[Riv11] Matthieu Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. Cryptology ePrint Archive, Report 2011/338, 2011. http://eprint.iacr.org/2011/338.

[RM15] Chester Rebeiro and Debdeep Mukhopadhyay. A Formal Analysis of Prefetching in Profiled Cache-Timing Attacks on Block Ciphers. Cryptology ePrint Archive, Report 2015/1191, 2015. http://eprint.iacr.org/2015/1191.

[RNG16] Nitin Rathi, Helia Naeimi, and Swaroop Ghosh. Side Channel Attacks on STTRAM and Low-Overhead Countermeasures. *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2016.

[RNR+15] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, pages 62–67. IEEE, 2015.

[Rol16] Rolf Rolles. Transparent deobfuscation with ida processor module extensions, 2016.

[RTSS09] 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, 2009.

[Sch16] Matthias Schunter. Intel Software Guard Extensions: Introduction and Open Research Challenges. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 1–1, New York, NY, USA, 2016. ACM.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. ACM.

[SIYA12] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Effects of Memory Randomization, Sanitization and Page Cache on Memory Deduplication. 2012.

[SKH11] Orathai Sukwong, Hyong S Kim, and James C Hoe. Commercial Antivirus Software Effectiveness: an Empirical Study. *Computer*, 44(3):63–70, March 2011.

[SKZ+11] Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut T. Kandemir, Mary Jane Irwin, and Yuan Xie. MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy. In *17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA*, pages 231–242, 2011.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In JosyulaR Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer Berlin Heidelberg, 2005.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications*

of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings, pages 443–461, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SWG+17] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *ArXiv e-prints*, February 2017.

[TASS07] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing Shared L2 Caches on Multicore Systems in Software. In *In Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[TNM11] Tatsuya Takehisa, Hiroki Nogawa, and Masakatu Morii. AES flow interception: Key snooping method on virtual machine - exception handling attack for AES-NI -. *IACR Cryptology ePrint Archive*, 2011:428, 2011.

[UGV08] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting Hardware Performance Counters. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pages 59–67. IEEE, 2008.

[Vau02] 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, 2002.

[vdVFL+16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1675–1689, New York, NY, USA, 2016. ACM.

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

[VMWb] VMWare. Understanding Memory Resource Management in VMware vSphere 5.0. `http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf`.

[VMwc] VMware. VMware Large Page performance . `http://www.vmware.com/files/pdf/large_pg_performance.pdf`.

[VZRS15] 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., August 2015. USENIX Association.

[WL07] 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.

[WOM11] Carolyn Whitnall, Elisabeth Oswald, and Luke Mather. An Exploration of the Kolmogorov-Smirnov Test as a Competitor to Mutual Information Analysis. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 234–251. Springer Berlin Heidelberg, 2011.

[WTM13] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 215–224. IEEE, 2013.

[WW09] P. Weisberg and Y. Wiseman. Using 4KB Page Size for Virtual Memory is Obsolete. In *Proceedings of the 10th IEEE International Conference on Information Reuse & Integration*, IRI'09, pages 262–265, 2009.

[xena] Xen 4.1 Release Notes. `http://wiki.xen.org/wiki/Xen_4.1_Release_Notes`.

[Xenb] Xen. X-XEN : Huge Page Support in Xen. `https://www.kernel.org/doc/ols/2011/ols2011-gadre.pdf`.

[XWW15] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 929–944, Washington, D.C., August 2015. USENIX Association.

[XZZT16] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 19–35, Austin, TX, August 2016. USENIX Association.

[Yar16] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016.

[YF14] 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.

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 346–367, 2016.

[YGL+15] 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.

[YS06] Hai Yan and Zhijie Jerry Shi. Studying software implementations of elliptic curve cryptography. In *Third International Conference on Information Technology: New Generations (ITNG'06)*, pages 78–83, April 2006.

[YWCL14] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 381–392, New York, NY, USA, 2014. ACM.

[ZHS16] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated Detection of Instruction Cache Leaks in RSA Software Implementations. In *Smart Card Research and Advanced Applications: 15th International Conference, CARDIS 2016, Cannes, France, November 7–9, 2016, Revised Selected Papers*, pages 228–244, Cham, 2016. Springer International Publishing.

[ZIUN08] Li Zhao, Ravi Iyer, Mike Upton, and Don Newell. Towards Hybrid Last Level Caches for Chip-multiprocessors. *SIGARCH Comput. Archit. News*, 36(2):56–63, May 2008.

[ZJOR11] 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.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In

Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM.

[ZJRR14] 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.

[ZJS+11] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.

[ZL14] Tianwei Zhang and Ruby B. Lee. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 96–105, New York, NY, USA, 2014. ACM.

[ZRZ16] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 871–882, New York, NY, USA, 2016. ACM.

[ZZL16] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, 2016.

[ZZL17] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation. *AsiaCCS 2017*, 2017.