Information Systems 92 (2020) 101524

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA

Maria Mushtaq [a,*], Muhammad Asim Mukhtar [c], Vianney Lapotre [b], Muhammad Khurram Bhatti [c], Guy Gogniat [b]

[a] LIRMM, Univ Montpellier, CNRS, Montpellier, France
[b] Lab-STICC, University of South Brittany (UBS), Lorient, France
[c] ECLab, Information Technology University (ITU), Lahore, Pakistan

ARTICLE INFO

Article history:
Received 29 January 2019
Received in revised form 16 January 2020
Accepted 30 March 2020
Available online 6 April 2020
Recommended by D. Shasha

Keywords:
Security
Privacy
Cryptography
Side-channel attacks (SCAs)
Cache side-channel attacks
Countermeasures
RSA
Intel's x86 architecture
Multi-core architecture
Caches

ABSTRACT

Timing-based side-channels play an important role in exposing the state of a process execution on underlying hardware by revealing information about timing and access patterns. *Side-channel attacks (SCAs)* are powerful cryptanalysis techniques that focus on the underlying *implementation* of cryptographic ciphers during execution rather than attacking the structure of cryptographic functions. This paper reviews cache-based software side-channel attacks, mitigation and detection techniques that target various cryptosystems, particularly RSA, proposed over the last decade (2007–2018). It provides a detailed taxonomy of attacks on RSA cryptosystems and discusses their strengths and weaknesses while attacking different algorithmic implementations of RSA. A threat model is presented based on the cache features that are being leveraged for such attacks across cache hierarchy in computing architectures. The paper also provides a classification of these attacks based on the source of information leakage. It then undertakes a qualitative analysis of secret key retrieval efficiency, complexity, and the features being exploited on target cryptosystems in these attacks. The paper also discusses the mitigation and detection techniques proposed against such attacks and classifies them based on their effectiveness at various levels in caching hardware and leveraged features. Finally, the paper discusses recent trends in attacks, the challenges involved in their mitigation, and future research directions needed to deal with side-channel information leakage.

Contents

* Corresponding author.
E-mail address: maria.mushtaq@univ-ubs.fr (M. Mushtaq).

https://doi.org/10.1016/j.is.2020.101524
0306-4379/

1. Introduction

With the development of computing and storage infrastructure, information security has become one of the paramount concerns. In the past decade or so, there has been an explosion in the amount of digital data. For instance, according to IBM Big Data research, 2.5 quintillion bytes of data are created worldwide every day; so much that 90% of data in the world today was created in the last two years alone. The information buried in these data is valuable to society, be it commercial, economic, environmental, government statistics, or concern the health and privacy of individuals. Faced with this deluge of data, information processing infrastructure have evolved to increase their performance, energy efficiency, reliability, and safety. These platforms are now increasingly shifting from the end-user to centralized computing facilities (the cloud computing concept) in order to free end-user terminals from excessively high computational loads. Cloud computing is the delivery of on-demand computing resources including everything from applications to data centers over the Internet. The issue of *trust* between end-users and cloud computing platforms is, however, a major concern that is currently preventing universal acceptance of this new technological solution.

Modern-day cloud computing solutions offer Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) for both public and private cloud [1]. These services provide virtualized system resources to end-users that offer high utilization through resource sharing. Such systems usually co-host multiple virtual machines (VMs) on the same hardware platform, which is managed by a virtual machine monitor (VMM) to insulate VMs and system resources.

While virtualization is supposed to provide insulation and exclusive access to resources, in practice the VMs are designed to share the same physical resources thereby creating a loophole for potential interventions. The co-resident VMs that share physical resources are mutually distrusting. For instance, a malicious VM co-residing with a victim's VM can discover the information of other VM [2–4] through resource sharing and can cause serious damage by conducting side-channel attacks (SCA) on the victim's VMs [5,6], thus exposing the system to the conventional challenges of information security represented by the classical CIA (confidentiality, integrity, and availability) triad. Absolute system confidentiality, integrity, and availability cannot be achieved simultaneously. Therefore, all systems have design trade-offs resulting in inherent vulnerabilities and rendering the system vulnerable to attacks.

SCAs are powerful techniques used to retrieve sensitive information by observing the system behavior through side-channels including power consumption, timing variation, acoustic emanation. Although SCAs can be used in many contexts (user spying, data extraction, etc.), this paper focuses on SCAs of cryptosystem implementations and more specifically on RSA cryptosystems.

Rather than attacking the underlying structure of cryptographic functions, SCAs focus on the implementations of cryptographic ciphers [7]. Fig. 1 illustrates how useful information related to execution can leak through unintended side-channels during computation. SCAs use variations in physical parameters (e.g. power consumption [8], electromagnetic radiation [9], acoustic emanation [10], memory accesses or fault occurrence [5,11–18]) generated by the execution of specific implementation of a cipher to extract secret information. In general, SCAs can be classified in two types: hardware-based SCAs in which the attacker requires measurement equipment to get physical parameters and software-based SCAs in which the attacker uses software instead of measurement equipment to steal information such as memory access or fault occurrence that help retrieve cryptographic information [7,19].

Substantial research efforts have been made in the last decade to provide mitigation techniques through resource isolation. Both software and hardware-based cache partitioning strategies have been proposed as countermeasures against cache-based SCAs [20–24]. However, these strategies significantly reduce performance because of cache reservation. Moreover, hardware-based partitioning techniques require specialized features, like those proposed in [24] that use *cache allocation technology* (CAT) for partitioning. Software-based techniques like *page coloring* require system-level modifications, which may lead to incompatibility with architectural features [24]. Achieving strong isolation seems to be possible to some extent. Hardware developers are able to hide CPU's internal hierarchy but the internal timing leakage is still very visible, and can be exploited to observe cryptographic implementations as demonstrated in virtual machine set-ups [2,5,6,16,17,19,25]. Attacks such as Spectre [26], Meltdown [27] and some covert-channel attacks [28], [29] have recently been launched and are sophisticated and hard to detect and mitigate. These covert-channels are the vulnerabilities that have affected almost every processor, across virtually every operating system and architecture. They exploit system wide features such as speculative execution and out-of-order execution that are considered as optimized performance features. These attacks can exploit vulnerabilities in the computational part rather than in the storage part, i.e. caches. The attacks are powerful enough to exploit stored passwords in a password manager or browser, personal photos, emails, instant messages and business critical documents. The use of covert-channels has allowed attackers to retrieve the victim's information from kernel memory with no direct contact with the victim. Consequently, it is hard to gather system wide vulnerabilities to detect/mitigate such attacks. Some recent mechanisms tried to mitigate covert channels [30–38] but all are reported to slow down CPU performance [39]. Given the limitations of software mitigation techniques, it is essential to introduce real-time detection techniques for covert-channels. An all-weather protection mechanism against such attacks is required that does not slow down the performance of the CPU.

Three recent literature reviews [39–41] identified a wide range of cache-based side-channel attacks and countermeasures on contemporary hardware. A range of cache-based timing attacks and countermeasures on contemporary hardware is listed in [39,41]. [40] provides a hierarchy of hardware and software attacks and analyzes the performance degradation in most of the countermeasures proposed for AES cryptosystems. [39] and [40] provide a long list of diverse cryptographic algorithms used in different attacks and a mix of software and hardware countermeasures. [42] provides a systematic classification of side-channel attacks for mobile devices. [43] discusses only passive side-channels and their countermeasures for Nessie public key cryptography. [44] is a systematic study that only targets cache-based side-channels implemented on AES cryptosystems. To the best of our knowledge, no review of the literature today provides a complete leakage hierarchy, identifying different execution contexts at different levels of the Intel x86 cache hierarchy and a taxonomy of various attacks that exploit these leakage opportunities, both in hardware and software execution, in order to extract sensitive information from various implementations of RSA cryptosystem. In this paper, we provide an in-depth analysis of different types of cache-based timing side-channel attacks with respect to the leakage context being exploited. The contributions of this paper are the following:

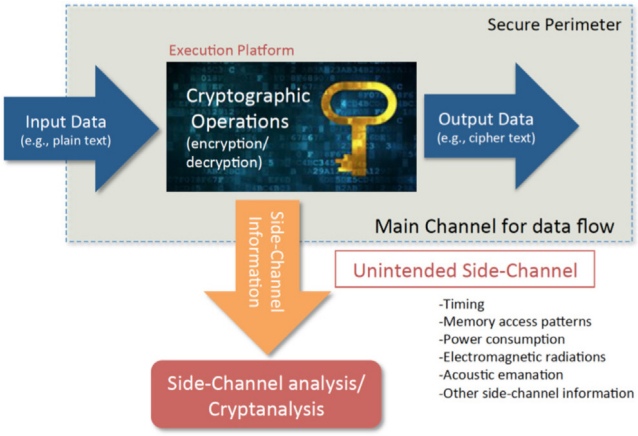


Fig. 1. Unintended side-channel information leakage.

- We propose a threat model to identify various leakage channels, both in software and hardware layers, to demonstrate possible threats and vulnerabilities. We identify leakages at different levels of the Intel x86 cache hierarchy that help narrow down major attack possibilities in caches.
- We provide a taxonomy of leakage channels, their classification, and the type of threat associated with each for different implementations of RSA cryptosystems in particular.
- We investigate the timing channels on various cryptographic implementations. In the last decade, different implementations of RSA cryptosystem have been attacked. We provide a detailed analysis of these attacks on different implementations of RSA and explain the problem of leakage and threat level involved in these implementations.
- We analyze and list software and hardware countermeasure techniques proposed so far against known attacks on RSA. We also identify significant mitigation techniques that are effective at various cache levels and that address different threat levels with respect to our proposed threat model. We analyze the efficiency of the proposed countermeasures and explain their efficacy against our proposed threat model.
- We identify existing auditing/detection techniques against cache-based side-channel attacks (CSCAs) using hardware performance monitoring counters (HPCs) to detect stealthy attacks. We underline the importance of detection mechanisms as a new angle of research to provide need-based mitigation toward CSCAs.
- We discuss various open threat areas in cache hierarchy that have not been properly addressed by the proposed mitigation techniques so far. We also discuss the challenges associated with hardware mitigation solutions and argue in favor of strong software countermeasures against threats in cache hierarchy in contemporary processors (Intel x86).

Table 1
List of acronyms.

Category	Name	Acronym
Channels	Covert-channel	CC
	Side-Channel	SC
	Timing-Channel	TC
	Persistent-State Channel	PS
	Transient-State Channel	TS
Leakage exploitation techniques	Prime + Probe	P + P
	Evict and time	E + T
	Flush + Reload	F + R
	Flush + Flush	F + F
	Prime + Abort	P + A
	Evict + Reload	E + R
	Attacker Address Space	AAS
	Victim Address Space	VAS
Cache-based side-channel attacks	Time-Driven Attack	Ti-DA
	Active-Time-Driven Attack	A-Ti-DA
	Passive-Time-Driven Attack	P-Ti-DA
	Trace-Driven Attack	Tr-DA
Leakage context	Context-Switch	CS
	Hyper-thread/Simultaneous Multi-threading	HT/SMT
	Multi-Core	MC
Cryptographic implementation	Square and Multiply Modular Exponentiation	S&ME
	Fixed Window Exponentiation	FWE
	Sliding Window Exponentiation	SWE
	Right to Left Sliding Window Exponentiation	R-L SWE
	Left to Right Sliding Window Exponentiation	L-R SWE
	Chinese Remainder Theorem	CRT
Counters	Hardware Performance Counter	HPC

Note that the scope of this paper is limited to an in-depth study and effectiveness analysis of cache-based SCAs and their respective mitigation techniques proposed in the period 2007 to 2018 for implementations of RSA public-key cryptosystems. Nevertheless, the paper provides a detailed threat model based on leakage channels in various levels in cache hierarchy of Intel x86 architecture, which is equally applicable to other cryptosystems and types of attacks.

The rest of this paper is organized as follows. Section 2 provides background knowledge on Intel x86 cache architecture and principles (Section 2.1), information leakage channels (Section 2.2). Cache-based timing side-channels are presented in Section 3. Classification of cache-based attacks relying on the source of leakage is explained in Section 3.1. Leakage contexts in Intel x86 architecture are explained in Section 4. Section 5 presents techniques to exploit leakage. Section 6 details the cryptographic operations of RSA and its working model. Section 7 provides a detailed review of known cache-based side-channel attacks on RSA implementations and their respective countermeasures proposed between 2007 and 2018. Section 8 categorizes known countermeasure techniques to resolve leakage from the levels proposed in the leakage contexts in Section 4. Section 8 discusses the applicability and practicality of different software countermeasures with respect to the leakage sources in modern processors, while Section 9 examines the trends of attacks and problems with protection mechanisms. Section 10 concludes the paper.

2. Background and concepts

This section provides the readers necessary background and concepts that are essential to follow the subsequent discussion in this paper. The section provides basic understanding of the Intel's x86 architecture and the information leakage channels that exist in this architecture. To facilitate the reading of the paper, the acronyms used in all the sections are listed in Table Table 1.

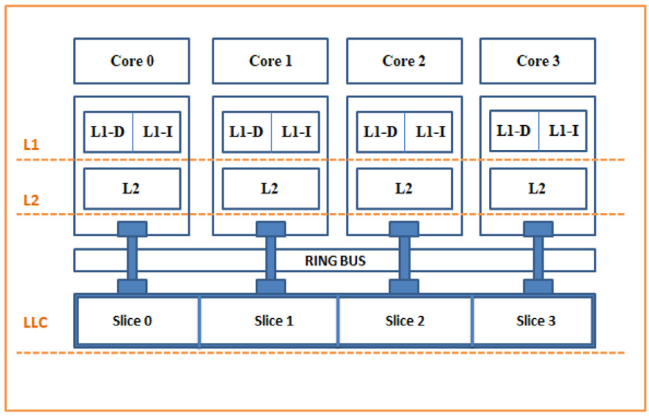


Fig. 2. Representative cache architecture of Intel Processors.

2.1. Intel x86 cache architecture and principles

Intel documentation states that DRAMs have a latency of up to approximately 200 times that of today's computing cores [45]. This gap is filled by stealth caches and its hierarchical design. Caches are smaller memories but one order of magnitude faster than DRAM. However, caches can impact the security of a software system in two ways. Firstly, Intel architecture depends on system software to arrange address-translation caches, which becomes a threat to security. Secondly, Intel architecture allows resource sharing of all software running in the processor. This raises the question of security in terms of cache timing attacks, which are a complete class of software attacks [5,14,25,39]. This section provides background knowledge on caching concepts and security issues that arise in Intel processors due to cache architecture. In theory, caches aim to resolve the problem of high locality in memory and hide the huge latency from main memory. By storing/caching recently accessed data, up to 90%–99% of the problem of main memory latency is satisfied [45]. Intel processors offer different levels of cache. The first level of cache, L1, consists of separating data (L1-D) and instruction (L1-I) caches. Fetching and decoding instructions are directly associated with L1-I cache, while operations that require read/write access from memory are directly associated with L1-D cache. The caches are all *inclusive* with private L1, L2 and shared L3 or LLC (last level cache) in all the cores. Fig. 2 is a general illustration of Intel cache architecture. For performance reasons, Intel architecture includes an arrangement that gives performance-sensitive applications some control over other applications. For instance, *prefetch* instruction prefetches a specific memory address to be used in future and *clflush* instruction evicts any cache line that has specific address from the entire cache hierarchy (L1, L2, and LLC). These instructions are available to software applications running at all privilege levels in order to provide high performance and optimization characteristics vis-a-vis caches. Cache properties, like inclusivity and flushing, have been exploited in many cache-based timing attacks including [5,14,25].

Cache architecture is composed of larger cache (i.e. LLC) in lower hierarchy and smaller caches (L1, L2) in the upper hierarchy for reasons of efficiency. Moreover, the farther the cache from the processing element, the greater the latency. Therefore, the size of each cache level is chosen with care to ensure the next level is faster. Based on the latest Intel processor documentation [47,48] and some studies [45,46] the approximate indicative parameters of Intel x86 architectures are listed in Table 2. It approximates the associativity, sharing, size of cache line, size of each level and access time of caches, however the memory

Table 2
Relevant indicative parameters of cache in Intel x86 architectures [45,46].

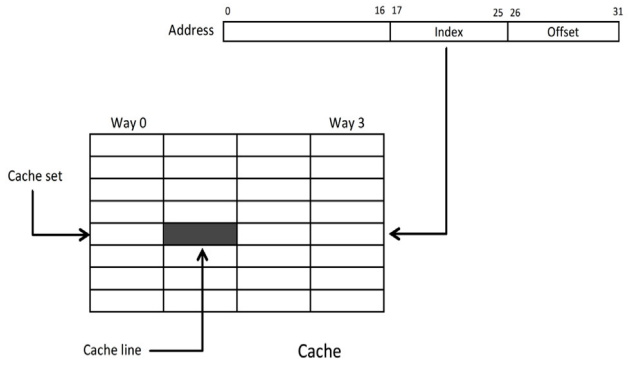
	L1-D	L1-I	L2	LLC
Associativity	8-Way	8-Way	4-Way	16-Way
Sharing	Private/Per-core	Private/Per-core	Private/Per-core	Shared/Inclusive
Size of cache line	64B	64B	64B	64B
Size of level	32 kB	32 kB	256 kB	8 MB
Access time	4 Cycles	4 Cycles	10 Cycles	40–75 Cycles

sizes and access time may vary in magnitude across different levels of the hierarchy. Table 2 explains that cache levels can be private and shared/inclusive. To explain that part, we provide details on the design of Intel architecture. Two processes running on the same core or across-core share the inclusive LLC by design, this being the core problem of sharing in contemporary architectures. Two processes that are not supposed to share their data, do share the data due to inclusive caches. Access to Intel x86 processors using privileged instructions like clflush and prefetch instructions, allow the attacker process to know the state of the victim process due to inclusivity. Inclusive caches exist in Intel x86 for many optimization and performance reasons, but cause critical sharing which may become a security problem. Cache properties, like inclusivity and flushing, have been exploited in many cache-based timing attacks including in [5,14,25].

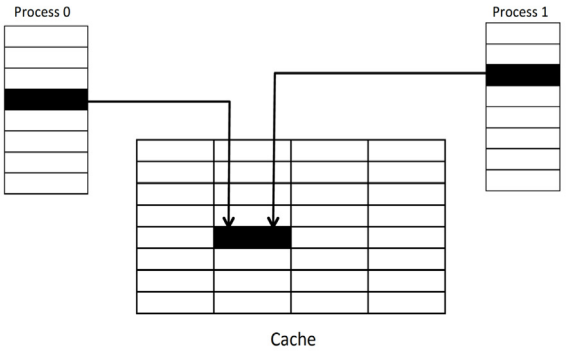
Intel's proposed cache hierarchy is designed to ensure better performance (by reducing latency). While it benefits performance, it does not solve the problem of security because resource sharing still exists in Intel processors [48,49]. For example, an attacker can flush specific memory lines from LLC causing a simultaneous flush of data from L1, L2, and LLC. In this case, the attacker can benefit at this point by simply measuring the time the victim takes to reload specific instructions. The attacker might get knowledge of the operations being performed via the process used by the victim. The retrieval of timing information from inclusive cache sharing has recently been proven to be a major security concern [5,16,19,50].

Another important aspect is mapping addresses toward different levels of cache in Intel x86. Different attacks and countermeasure techniques have different effects and applicability due to address mapping. L1 caches are often addressed virtually, while L2 and LLC are mapped physically. This is why some countermeasures do not provide a wide vector to prevent such attacks. L1 and L2 caches in Intel x86 architectures are multi-way set-associative with direct set indexing. Therefore, a W-way set associative cache contains its own memory that is divided into *sets*. Each set consists of W lines in which memory can be cached. However, LLC is divided into per core slices. Each slice is allocated to a separate core and can be used as a unified or separate cache [25,51] as shown in Fig. 2. Intel's documentation states that the hashing scheme maps physical addresses to LLC. It was designed to distribute memory circulation. The hashing scheme is not publicly available but has been reverse-engineered in past research work [52–54].

To help understand the cache address, index, associativity, cache set and cache line in Intel x86 in Table 2, Fig. 3 demonstrates attacks using the cache architecture, illustrating how a potential CSCA could exploit the features of cache organization that are introduced to improve performance in processors. Typically, as illustrated in Fig. 3(a), data/instructions are loaded into a specific *set* depending on the addressing index and a cache line is loaded in a specific *way* within the selected set in a set-associative cache depending on the cache replacement policy. For a CSCA exploiting the sharing feature of caches, the attacker process maps itself with the shared library (shared memory addresses in the cache) used by the victim's process, which is usually a cryptosystem. Such mapping allows the attacker to examine the address space of victim process as illustrated in Fig. 3(b). Once



(a) Typical addressing of a cache line in 4-ways set-associative shared cache.



(b) Shared address space between any two processes due to shared libraries and data/instruction de-duplication.

Fig. 3. Exploitation of cache memory organization and sharing by CSCAs in Intel x86 architecture.

mapped, the attacker process can use the legitimate operations of the cache replacement policy to evict selected data/instructions from a particular location and to measure the difference in timing between a cache *hit* and a cache *miss* with respect to the victim's process. Such measurements not only reveal the difference in timing but also expose the memory access pattern of the victim's process. A CSCA can use this information to extract or extrapolate the secret key of the target cryptosystem. Interestingly, the attacker's process monitors which lines are accessed by the victim, and not their content. Furthermore, CSCAs have also proved to be effective without sharing the address space with the victim. In Section 5, we will also explain the techniques cache architectures use to attack without sharing the same address space as the victim's process. CSCAs demonstrate that the organization of the memory, which is used to better address modes, exposes the structured address space of processes and makes them vulnerable to attacks.

2.2. Information leakage channels

Various behavioral features of caches, and to some extent the problems associated with such features, are explained in the

preceding section. These features create a state that initiates a level of distrust between co-running processes and causes serious issues of confidentiality and integrity. These issues are based on preceding computation of operations or sometimes by caching useless data. In practice, the process of caching data/instructions is transparent and has no impact as such on the outcomes of the operations (run by either the victim or the attacker). However, it causes two issues; (1) co-running processes lose confidentiality through various channels that are being exploited by the attackers and (2) the overall performance is compromised. In most cases, the timing and/or access pattern information is revealed during program execution. This section briefly elaborates major information leakage channels being reported in the state-of-the-art. Some of these information leakage channels can be prevented at the design time if they are known to exist, e.g., through proper isolation of processes or by partitioning caches, but many go unnoticed until the system is deployed, thus creating a potential attack surface.

2.2.1. Covert-channels

A covert-channel is a communication channel that was not intended or designed to transfer information between a sender (process) and a receiver (process) [55]. Covert-channels typically leverage unusual methods for communication of information, never intended by the system's designers. These channels can include use of timing, power, thermal emanations, electromagnetic radiation, acoustic emanations, and possibly others. With the exception of timing channels, most channels require some physical proximity and sensors to detect the transmitted information, *e.g.*, use of EM probe to sense EM emanations. Meanwhile, many timing-based covert-channels are very powerful as they do not require physical access, only that sender and receiver run some code on the same system. Covert-channels are important when considering intentional information exfiltration where one program manipulates the state of the system according to some protocol and another observes the changes to read that "information" that are sent to it through changes. Covert-channels are a concern because even when there is explicit isolation, *e.g.*, each program runs in its own address space and cannot directly read and write another program's memory, a covert-channel may allow the isolation mechanisms to be bypassed.

Covert-channel attacks target a system's confidentiality. For instance, a program inside an Intel SGX Enclave (*i.e.*, the sender) may modify processor cache state based on some secret information it is processing, and then another program outside the Intel SGX Enclave (*i.e.*, the receiver) may be able to observe the changed cache behavior and deduce what the sensitive information was, *e.g.*, bits of an encryption key. The execution and implementation of a covert-channel strictly depends on the microarchitecture on which it is processed to abuse the information. Implementations of covert-channels have also been evaluated in recent studies [26–29,56]. Fine-grain classification of covert-channels is also explained in Section 2.2.2 (Transient-state channel) below.

2.2.2. Side-channel

A side channel is similar to a covert-channel, except that the sender does not intend to communicate information to the receiver, rather the sending (*i.e.*, leaking) of information is a side effect of the implementation and the way the computer hardware or software is used [57]. Side channels can use same means as covert-channels, *e.g.*, timing, to transmit information. Typically, covert-channel attacks are analyzed as both sender and receiver are under control of the potential attacker and it is easier to create a covert-channel. However, side channels are usually more difficult to create since the victim (i.e., sender) is not under control of the attacker. The goal of a side-channel attack is to extract some information from the victim. Meanwhile, the victim does not observe any execution behavior change nor is aware that they are leaking information. This way, confidentiality can be violated as data, such as secret encryption keys, is leaked out. Interestingly, side-channel attacks can work in *reverse* as well. A side channel can also exist from attacker to victim. In a reversed attack, the attacker's behavior can "send" some information to the victim that causes processor's state change. Such information affects how the victim executes, without the victim knowing there is a change. For example, the attacker can fill processor cache with data, causing the victim to run more slowly. Or, the attacker can affect behavior of the branch predictor, causing the victim to execute extra instructions before processor is able to detect that these instructions should not be executed and nullifies their ISA-visible change.

Side-channels can work in two different ways, attacks based on the physical parameters of hardware architecture (like power consumption [8], electromagnetic radiation [9], acoustic emanation [10], memory access or fault occurrence [5,11–18]) and software attacks that work specifically on cache behaviors, timing, execution, etc. All side-channels that rely on the behavior of caches and timing are so-called cache-based timing side-channels that constitute a rather bigger class of software attacks and therefore focus of this study. Cache-based side-channel attacks (CSCAs) exploit the vulnerability of cache in terms of minute variations in timing to detect competition for space in cache with different processes or within a process. When all the resources are located on different cores in modern Intel processors, there is a risk that different resources may leak information at the time of execution. Attackers exploit the precise timing of these cache behaviors to perform cache-based timing side-channel attacks. We detail the classification of side-channels as cache-based timing channels separately with their types in Section 3. Side-channels can be further classified in two categories: Transient-state channels and Persistent-state channels, that are detailed in the following. This classification was inspired by the research work presented in [7], which explains that significant attacks are due to contention for microarchitectural hardware resources. Other research work like [58] also distinguished between the microarchitectural side-channel attacks based on whether they exploit a persistent-state channel and a transient-state channel. Persistent-state channels create the vulnerability around restricted storage space of the targeted microarchitectural resource and transient-state channels create vulnerability around the restricted bandwidth of the target resource.

Persistent-State Channels: Persistent-state channels are created by the restricted storage space and exploited by the attacks while remaining in the targeted microarchitecture. The restricted storage of cache sets is exploited in this type of attacks to classify the sets used by the victim. For instance, persistent state channels are best-suited for Prime + Probe attacks [16,59] in which the attacker fills its data in the cache and lets the victim execute. When the victim accesses the cache, its data are loaded into the cache by replacing some of the attacker's data. Thus, the attacker is able to clearly establish relationship between the victim's accessed data and attacker's own evicted data from the cache. Persistent-state channel attacks have demonstrated their applicability on L1-data cache [16,19,59,60], L1-instruction cache [5,6,61,62], LLC [25,63–65], branch prediction buffers [66,67] and DRAM open rows [68].

Transient-State Channels: The field of transient-state execution attacks has emerged suddenly and proliferated, leading to a situation where people are not aware of all its variants and their implications. The concept (and potential vulnerabilities) of transient-state execution attacks stems from the fact that modern CPU pipelines are massively parallelized allowing hardware

logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, pipelines may stall when operations have a dependency on a previous instruction, which has not been executed (and retired) yet. Hence, to keep the pipeline full at all times, it is essential to predict the control flow, data dependencies, and possibly even the actual data. Crucially, however, as these predictions may turn out to be wrong, pipeline flushes may be necessary, and instruction results should always be committed according to the intended in-order instruction stream. The pipeline flush discards any architectural effects of pending instructions, ensuring functional correctness. Hence, the instructions are executed *transiently*, i.e., first they are, and then they vanish. While the architectural effects and results of transient instructions are discarded, microarchitectural side effects still remain beyond the transient execution, thus creating the transient-state channels. These channels are the foundation of Spectre [69], Meltdown [70] and Foreshadow [71] attacks. These attacks exploit transient execution to encode secrets through microarchitectural side effects (e.g., cache state) that can later be recovered by an attacker at the architectural level [72].

3. Cache-based timing side-channels

Observing and revealing minute timing variations is an important aspect of timing channels, as these can exploit a lot of secret operations including in context switching, preemptive scheduling, hyper-threading, simultaneous multi-threading and threats in multi-cores [11,19,73–75] (explained in Section 4). When the state of cache is shared between two different programs for execution, there may be a risk of timing channel as the victim and attacker could be sharing the same resources and attacker can observe the victim's execution with minute timing variations because the timing of one program depends on the execution of the other program [76,77]. Cache-based side-channels are also possible even though efforts are made to ensure some strict partitions (i.e. attacker and victim run on separate cores).

Normal flushing of cache usually also reveals the timing information by simply observing the timing of victim's program to fetch the addresses of interest. Exploiting timing channels requires methods that really count the time needed to perform the operations. These properties can be achieved by using some quick and efficient counters like a pair of clocks [78] that show that if there is a difference in timing between two clocks, there is a timing channel between the two. Some preventive measures were taken in the past to avoid cache-based timing side-channels. One way to avoid timing channels is not sufficient to avoid the contention between different processes. One solution is introducing noise, which reduces the efficiency of timing channels. Noise actually makes it difficult for the spy program to observe the timing of victim's program activity but these solutions did not prove to be very efficient because the victim's actual signal was still there. But, depending on the program behavior, it is very important to produce an intelligent form of noise in the program to effectively remove the traces of victim's timing and execution. To date, smart noise creation mechanisms are very restrictive and difficult to integrate in real world interactions [16, 39]. Recent studies are still trying to find new ways to avoid cache-based timing side-channels to prevent different processes leaking information, which is the topic of this paper.

3.1. Cache-based side-channels attacks

In this section, we describe the classification of cache-based side-channel attacks. It is important to understand the type of information that is leaked to distinguish and classify the attack accordingly. [79] distinguishes between time-driven and trace-driven attacks. Time-driven attacks have been further classified in two sub-types known as active time-driven and passive time-driven attacks [20].

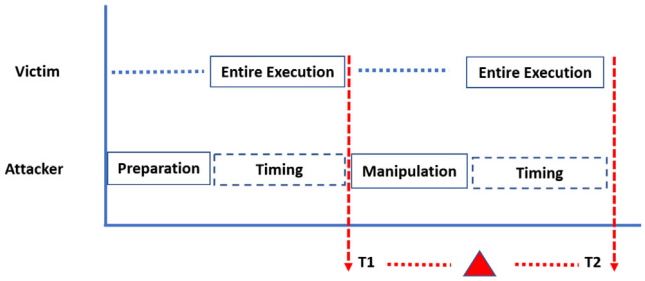


Fig. 4. Principle behind time-driven attacks.

3.1.1. Time-driven attacks

Time-driven attacks, also known as timing attacks, result in quantifiable execution information related to timing. This type of timing information can relay secret cryptographic operations. It can be comprehended as the number of cache hits and misses relaying timing execution information throughout the encryption process. This type of timing difference enables an attacker to extract the complete key. Fig. 4 is a representation of time driven attacks in which the attacker observes the victim's timing information before and after execution. Hence, these types of attacks are efficient in measuring the entire execution time by which secret information belonging to the victim can be recovered. Depending on the location of the attacker, time-driven attacks fall into two categories: active time-driven cache attacks and passive time-driven cache attacks. The passive attacker has no access to the victim's machine and is thus not able to influence the victim's machine directly or indirectly [19,80]. Consequently, the attacker cannot probe the timing information on the victim's machine. On the other hand, an active attacker can influence the victim by running code on the same machine. The attacker is then well informed about the victim's timing information and can manipulate the victim's machine [19,81,82].

3.1.2. Trace-driven attacks

As the name suggests, this type of attack tries to manipulate the trace of victim's accesses [20,39,79,83,84]. This type of attack aims to access the cache line that has been used by the victim by analyzing and reviewing the cache state repeatedly as shown in Fig. 5. Trace-driven attacks are also called access-driven attacks. It can be concluded that active time-driven cache attacks and trace-driven cache attacks need to operate on the same machine as the victim. If a good trace of information related to the victim has been obtained, trace-driven attacks are more effective, resourceful and refined than time-driven attacks.

Trace-driven attacks are destructive in the case of simultaneous multi-threading (SMT) or hyper-threading that enable the hardware to execute several threads simultaneously. This can be hazardous as the threads use the same processor resources. [59] describes this kind of potential attack on RSA where the attacker process observing L1 activity of RSA encryption can easily get the information on multiplication and squaring operations based on Chinese remainder theorem (CRT). CRT is a standard used in different RSA implementations to address modular operations on secret RSA keys. Furthermore, trace-driven attacks [85] proved to be more rigorous and can be performed without multi-threading technologies. The attack was performed on single threaded processor while attacking the Advanced Encryption Standard (AES).

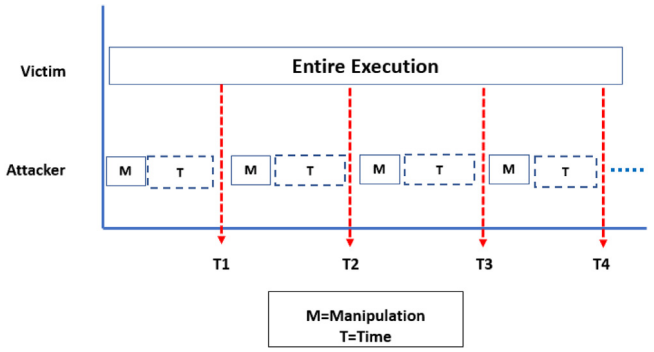


Fig. 5. Principle behind trace-driven attacks.

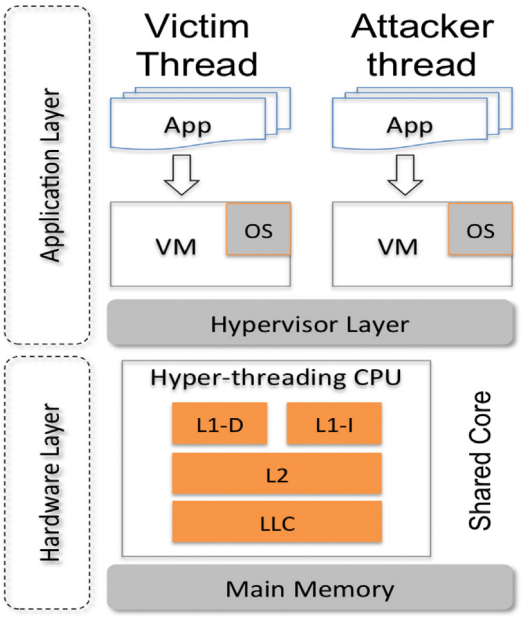
Later, [11] also undertook a similar attack to achieve full key recovery in AES encryption and this attack proved to be a fully functional asynchronous attack in a real life scenario. More quantifiable research on trace-driven attacks on caches were performed by [12,16], a breakthrough work that analyzed the two rounds of AES. Access-driven attacks also come under the umbrella of trace-driven attacks. Access-driven attacks are considered fine grained because they provide specific information related to the victim's access to addresses of interest.

4. Leakage contexts in intel x-86 architecture

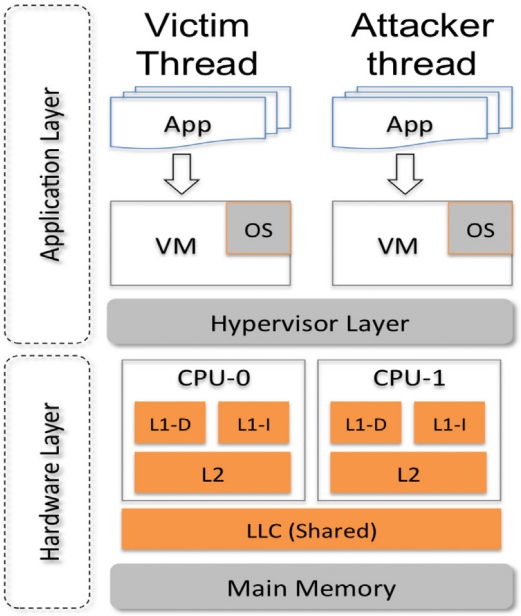
In this section, we focus on information leakage from all levels of cache in Intel processors, which can happen both in a virtualized environment and in standalone applications. Indeed, when executing processes on modern single and multi-core architectures, we identified three possible leakage contexts depending on resource sharing at the hardware-level, as shown in Fig. 6. First, these leakages can be due to multi-threading CPUs that run both victim and attacker threads on the same physical core despite having separate VMs. Second, the leakage can happen due to context-switching between various running processes, particularly when the attacker process is waiting for the context switch in the victim's process. Third, leakage can also happen in a multi-core context due to shared last level cache (LLC), which is being used to access both the attacker's and victim's data/instruction.

As illustrated in Fig. 6(a), the first situation concerns processes sharing the same core (threat to L1, L2 and LLC) through hyper-threading in Intel processors. Hyper-threading [7] lets the threads run concurrently on a shared core. Such threads share a number of resources and all levels of cache, which is considered a potential and significant threat to side-channels. For hyper-threading, it is usually suggested to disable the hyper-threading and for simultaneous execution, it is suggested to enforce gang scheduling [20] but no software solution has yet been devised for such threats other than simply disabling them. The second situation also concerns processes sharing the same core (threat to L1, L2 and LLC) and is executed through context switching [20]. Preemptive scheduling allows two different processes to run on the same core and to share all levels of cache. The system can thus use the scheduling to switch the processor between processes from different VMs. During each context switch, the attacker process can observe all the victim's activity.

The third situation concerns processes executed in a multi-core context as shown in Fig. 6(b). The processes are running on different cores and are assumed to be safer since resource sharing is limited. However, LLC is still shared between the cores. Because of the inclusive property of Intel architectures, sharing LLC is a major vulnerability and there is a significant risk of information leakage because an attacker process on one core is still able



(a) Hardware threading in single core architecture



(b) Multi-core architecture

Fig. 6. Classification of leakage contexts in Intel x86 architecture.

to manipulate caches of a victim's process on a second core. Inclusivity in Intel architecture ensures data coherence, which leads to better performance in multicore architectures. In order to maintain data coherence, however, inclusive caches allow privileged instructions to remove cache lines. Such instructions can be used by malicious processes in a shared memory architecture to remove data/instructions of the victim's processes from shared caches, for instance LLC. In non-inclusive caches, this privilege does not exist. Therefore, non-inclusive caches are less vulnerable as they do not possess privileged instructions.

At the software layer level, an attacker is able to build different attack techniques to exploit these execution contexts. These techniques are described in the following section.

5. Leakage exploitation techniques

This section presents the state-of-the-art techniques that are used to demonstrate cache-based attacks using leakage channels in various cache levels as discussed in Section 2.2.

5.1. Prime + probe technique

LLC-based cross-core attacks are usually Prime + Probe attacks [52] that come under the classification of trace-driven attacks, in which the attacker process gets to know which cache sets have been used by the victim's process. The attacker initiates a spy program to observe the cache contention of the victim's process, as shown in Fig. 7. In the prime step, the attacker process fills different cache sets with its own code (Fig. 7(a)). The attacker then goes into idle state in which it lets the victim's program run and execute its code (Fig. 7(b)). In the probe phase, the attacker program observes its own filled cache and continues to execute normally. Meanwhile, the attacker observes the time to load each set of its data that it already placed in the cache (primed). Some of the cache sets will be evicted by the victim from the cache and will take a long time to fetch, which will be observed by attacker program using latency to fetch the data. In this way, the attacker program obtains information on addresses that are sensitive for the victim, described in Fig. 7(c).

Prime + Probe attacks are actually harder to perform in LLC than in the L1 level of cache due to perceptibility of processor-memory activity at LLC [25], difficult to perform prime and probe steps for all LLC [2–4,6,59,60,62], classifying cache sets related to the victim's security critical program and probing resolution. Using the Prime + Probe technique [16,59] to perform attacks is a common way of exploiting a contemporary set of associative cache. This technique has been used to exploit different levels of cache including L1-data (L1-D) cache [16,59], L1-instruction (L1-I) cache [86] and last level cache (LLC) [87]. Many other attacks are performed in this way [5,14,16,25,51,74].

5.2. Evict and time technique

As the name suggests, this technique aims to evict lines along with time measurements of complete execution (as shown in Fig. 8), and is consequently classified as a time-driven attack. In the first step, shown in Fig. 8(a), the attacker fills the cache with its own data. This step is referred to as the prime step. In the second step, shown in Fig. 8(b), the attacker allows the victim's program to execute its code and access its instructions/data in the caches. The victim's access to memory causes the eviction of certain cache lines that were initially primed by the attacker as shown in Fig. 8(c). In the third step, the attacker observes the variation in the victim's execution time, which clearly reveals which lines were accessed (Fig. 8(d)) due to the measured time difference in load. There are many implementations for this type of technique with potential attack mechanisms including in [12, 16,88].

5.3. Evict and reload technique

Evict and Reload is a variant of the Evict and Time technique suggested in [15]. Evict and Reload affects the two major steps, eviction and reloading, described in Fig. 9 and classified as trace-driven attacks. During the initialization step, it evicts cache lines, as shown in Fig. 9(a). When the cache is evicted, the attacker lets the victim execute and remains in the wait state, as shown in Fig. 9(b). Next, the attacker performs a reload step to measure the time of cache lines to observe cache hits and misses (Fig. 9(c)). This variation in time helps the attacker obtain knowledge of the victim's access to specific addresses.

5.4. Flush + reload technique

Flush + Reload [5] is a different mechanism from Prime-Probe and Evict and Time, as shown in Fig. 10 and falls under the classification of trace-driven attacks because it relies on the presence of page sharing, as shown in Fig. 10(a). Due to inclusive caches, Intel x86 architecture provides privileged instructions, e.g. the *clflush* instruction, on how to flush the memory lines from all cache levels, including the last level cache (LLC), which proves to be a major threat and core advantage for attacks using the Flush + Reload technique. In the first phase, the attacker flushes (evicts a shared cache line) using clflush instruction (Fig. 10(b)). After flushing the cache line, the attacker remains in the idle state and allows the victim to execute and access its data/instructions from cache as shown in Fig. 10(c). In the next phase, called reload, it observes the timing information by reloading the shared cache line as shown in Fig. 10(d). The timing information reveals the interest of the victim's program. Stealth reload indicates that this cache line was affected by the victim and slow reload shows that it was not. Contemporary Intel x86 architectures can use the Flush + Reload mechanism to measure the time of clflush instruction. The advantage of this technique is that the attacker is able to aim at a precise cache line [5,11,15,17,18,50,65,73,89–91] instead of whole cache set as described in Prime + Probe technique.

5.5. Flush + flush technique

Flush + Flush [14], is a new Flush + Reload technique and is classified as a trace-driven attack. It measures deviation in the execution timing of the clflush instruction in Intel x86 architecture, as shown in Fig. 11. In this technique, the victim and the attacker share the same address space (Fig. 11(a)). Flush + Flush relies only on the execution time of flushing to check if the data are cached or not. In the first step, the attacker flushes the victim's cache line as shown in Fig. 11(b)). After flushing, the attacker allows the victim's process to execute and access its data/instructions as shown in Fig. 11(c). In the next step, the attacker flushes the same cache address again and measures the timing variation due to the presence or absence of the victim's data/instructions as shown in Fig. 11(d). Unlike the reload phase in the Flush + Reload attack, Flush + Flush does not generate excessive memory read operations in the second step, making it a stealth type of technique. It is considered stealthy because it does not access any memory and the rate of cache misses is reduced due to constant flushing. The attacker process in this technique is undetectable based on the rate of cache hits. Flush + Flush runs at a higher frequency which is why it is considered faster than any existing attack techniques. This technique is considered noisier than Flush + Reload and Prime + Probe attacks. It has a little higher error rate than Flush + Reload and Prime + Probe attacks [5,14,39].

5.6. Prime + Abort technique

Prime + Abort [46] is a stronger mechanism than all the above techniques in accuracy and efficiency and is classified as a trace-driven attack. Existing attack techniques involve three steps: initialization, waiting and measurement. Unlike Prime + Probe, Flush + Reload, Flush + Flush, Evict and Time, Evict and Reload techniques, this technique does not rely on setting a threshold for timing and determining an eviction set to precisely measure the timing information as shown in Fig. 12. Prime + Abort takes advantage of Intel TSX hardware which is widely used in server and consumer processors. In this technique, the attacker primes the cache with its own working set and waits for the victim to operate with no initial threshold calculation and determining

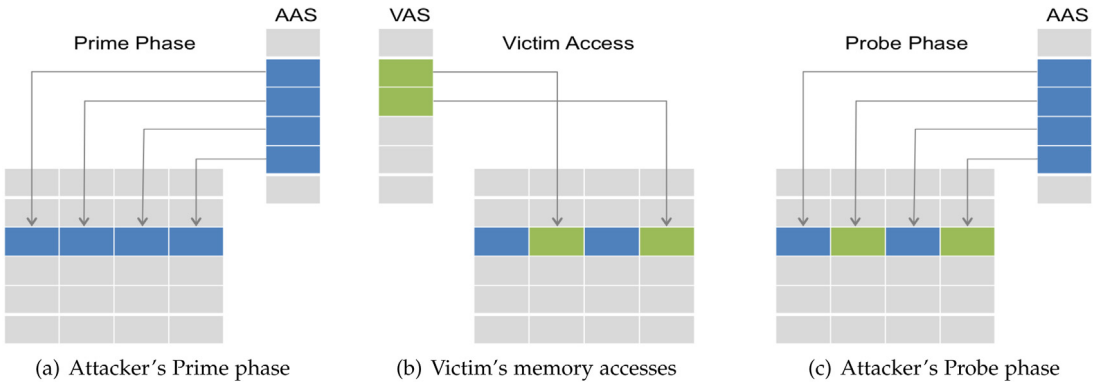


Fig. 7. Working principal of Prime + Probe.

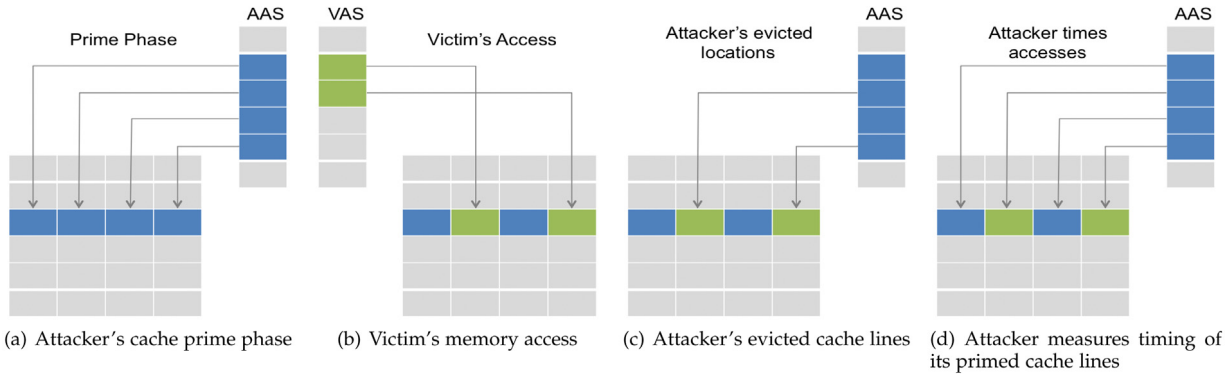


Fig. 8. Working principal of Evict and Time.

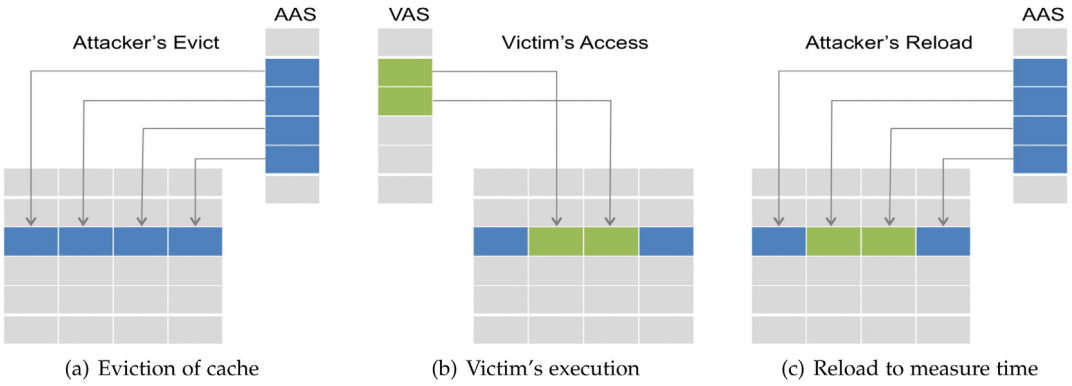


Fig. 9. Working principal of Evict and Reload.

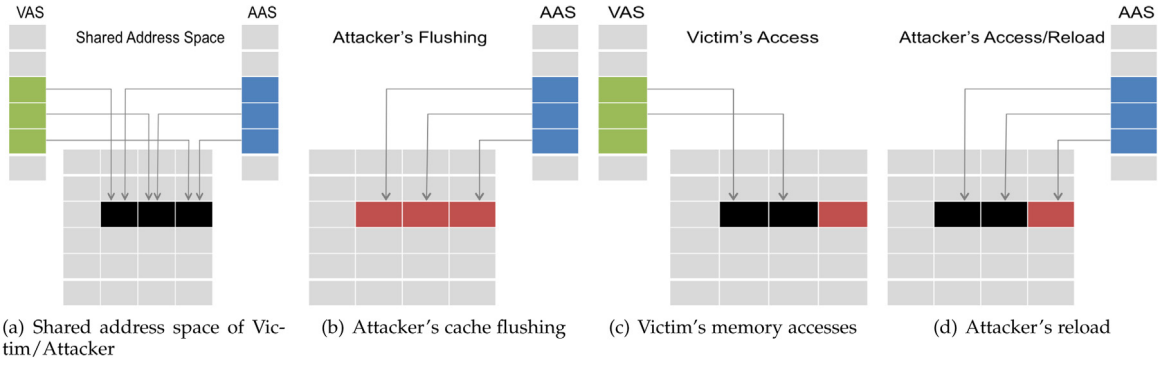
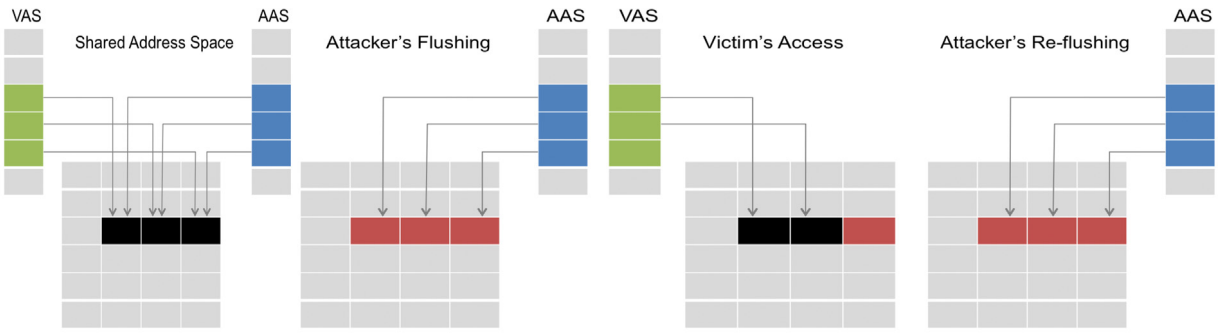
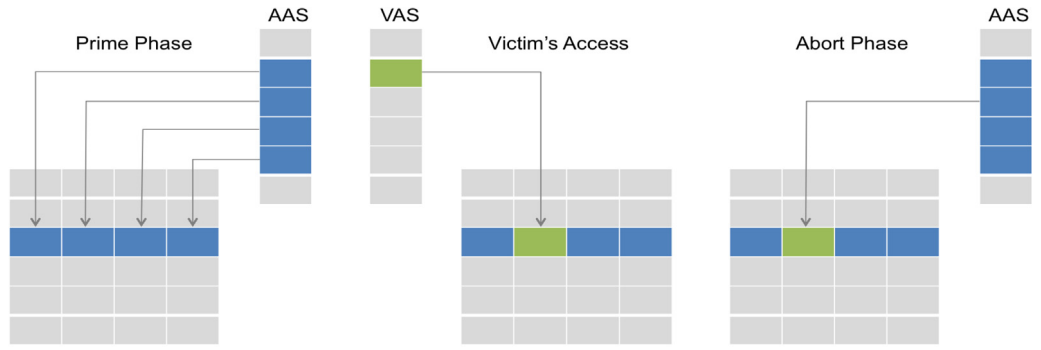


Fig. 10. Working principal of Flush + Reload.



(a) Shared address space of Victim/Attacker (b) Attacker's cache flushing (c) Victim's memory accesses (d) Attacker's re-flushing

Fig. 11. Working principal of Flush + Flush.



(a) Attacker's cache Prime phase (b) Victim's memory access (c) Attacker's abort phase on victim's detection

Fig. 12. Working principal of Prime + Abort.

eviction set (which is a prior step in other techniques) as shown in Fig. 12(a). Whenever the victim executes, it evicts an attacker's cache line that was primed (Fig. 12(b)). This flag leads the system to activate the abort mechanism. In this way, the attacker does not need the timing information. The attacker will always be able to know whenever victim has accessed the cache by the abort mechanism (Fig. 12(c)). To date, this technique has been applied as a proof-of-concept for Prime + Probe and Flush + Reload techniques and is considered to be effective in its variants of Evict and Time, Evict and Reload and Flush + Flush because they all monitor timing. This technique takes advantage of the hardware properties of Intel x86-64 processors. Note that Prime + Abort does not fall into the category of timing-attacks because it does not depend on the timing measurements. For this reason, this attack is not directly comparable with other techniques based on timing.

5.7. Discussion

After describing attack exploitation channels, classification of attacks, and attack techniques, we conclude this section with a review of all the attacks published in the literature with respect to their algorithmic implementations, level of cache (which is exploited) and technique in which they fall as described in Table 3. This table identifies recent attacks and provides an exhaustive taxonomy of techniques published up to 2017.

It is worth noting that the six techniques described in this section can be used in the three leakage contexts described in Section 4. However, their efficiency is affected by the model in which they are actually implemented. For instance, Evict and Time only targets L1 caches because it is sensitive to noise. Flush + Reload and Flush + Flush, which rely on the clflush instruction, are particularly efficient in a multi-core model when the attacker is able to share memory with the victim's process. That is why these attacks have been used for cross-VM attacks through the LLC sharing between virtual machines. Finally, Prime + Probe and Evict and Reload are the most generic techniques that can be used efficiently in the three leakage contexts described in Section 4.

Table 3 lists the implementation vulnerabilities of the cryptographic systems used. For instance, different versions of RSA have been developed for secure cryptographic implementations, but they have all been exploited in the past. In this paper, we focus on implementations of RSA in past decade (2007–2018) and discuss why they were vulnerable when deployed on hardware. Some countermeasures have been proposed against these attacks. We analyze the efficiency of these countermeasures at their respective threat level in these implementations. We discuss the weakness of different sets of operations that make it easier for the attackers to exploit these features. The aim of the state-of-the art in Table 3 is thus to stress one direction of trend in type of cryptographic implementation and to discuss it in detail. In later sections, we identify vulnerabilities that have been exploited in RSA implementations and analyze whether the published countermeasures are able to completely stop such leakages.

6. RSA public key cryptosystem implementations

The scope of this survey is limited to state-of-the-art attacks, detection and mitigation techniques that target RSA cryptosystems. This section reviews the fundamental concepts of RSA to facilitate the reading of the rest of the paper. RSA (Rivest, Shamir and Adleman) is an extensively used public key cryptosystem that

Table 3
Summary of state-of-the-art trace-driven cache-based attacks.

No.	Leakage exploitation techniques	Exploitation features of the attack	Target level of cache	Cryptographic implementation	Year of publication
1	Flush + Reload	Inclusive Cache, Page Sharing [11,18,25]	LLC	RSA, AES	2014, 2015, 2011
		Shared Libraries [65,89,90]	LLC	ECDSA	2015, 2016, 2014
		Shared libraries [73]	LLC	DSA	2016
		Memory mapping [11]	LLC	AES	2011
		Page de-duplication [18]	LLC	AES	2012
2	Flush + Flush	Stealth flushing, inclusive cache [14]	LLC	AES	2016
3	Prime + Probe	Hyper threading, Cache bank conflicts [7]	L1-D	RSA	2017
		Symmetric Multi-threading [12,16,59]	L1-D	RSA, AES	2010, 2006, 2005
		Addresses in Look-up Tables [12,16]	L1-D	AES	2010, 2006
		Preemption of RSA in Minute Intervals [61]	L1-I	RSA	2007
		Spy Process Entry in RSA as a routine [62]	L1-I	RSA	2010
		Symmetric Multi-threading [60]	L1-D	ECDSA	2009
		Symmetric Multi-threading [62]	L1-I	DSA	2010
		Interprocessor Interrupts [6]	L1-I	ElGamal	2012
		Huge Page [25,91]	LLC	RSA, ElGamal	2015, 2016
		Huge Page [64]	LLC	AES	2015
		Inclusive Cache, Large page Mappings [25]	LLC	RSA, ElGamal	2015
		Collision Entropy [92], Right-to-Left Sliding Window Exponentiation [93]	LLC	RSA	2017
		Branch Prediction, Exploiting Average time to read Instruction from Memory [61,86]	L1-I	AES	2007, 2008
		Preemption in Short Intervals [85]	L1-D	AES	2006
4	Prime + Abort	No timing dependency needed to attack, Intel TSX Hardware [46]	LLC	AES	2017
5	Evict + Time	Virtual and Physical Addresses of lookup tables [12,16]	L1-D	AES	2010, 2006
		Addresses in Look-up Tables [12,16]	L1-D	AES	2010, 2006
6	Evict + Reload	Shared Libraries, Key-stroking [15]	LLC	AES	2015

ensures encryption and digital signature [94]. RSA proposes a specific mechanism for the encryption and decryption of messages. The use of RSA as public key encryption is described below.

- Choose two large prime numbers p and q of roughly the same bit length
  Compute $n = pq$ and Compute $\varphi = (p-1)(q-1)$
- Select a random component as exponent:
  $e = 65,537$, *such as* $1 < e < \varphi$, $\gcd(e, \varphi) = 1$, $ed \equiv 1 (\mathrm{mod} \varphi)$
- Estimate the private exponent:
  $d \equiv e^{-1} \pmod{(p-1)(q-1)}$
- Public key: (n, e)
- Private key: (p, q, d)

The components e and d are called encryption exponent and decryption exponent, respectively, n is the modulus. To encrypt the message m, the sender undertakes the following steps:

- Displays the message m in interval $[0, n-1]$
- Encryption Function: $E(m) = m^e \bmod n$

To retrieve the message m in plain text or to decipher the message c, the receiver of message computes:

- Decryption Function: $D(c) = c^d \bmod n$

RSA-CRT (Chinese Remainder Theorem) [7], is the fastest optimized decryption mechanism proposed by RSA. It actually splits the private key into two parts to perform an optimized and fast decryption. So the secret key d is split into parts $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, two sections of the message can be computed, message m can be computed as, $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. The message m can be computed from m_p and m_q using Garner's formula, which states:

$h = (m_p - m_q)(q^{-1} \bmod p) \bmod p$
$m = m_q + hq$

For some attacks, it is possible to achieve more than 50% of bits, and other bits are recovered just by chinese remainder theorem coefficients, d_p and d_q. To have the full knowledge of key and recover it, Heninger and Sacham algorithm [95] and [63] was used in some attacks, which provides us with following modular multipliers, $K_p K_q N \bmod e$. While K_p and K_q can be used to derive the values of d_p and d_q of RSA-CRT. The important operation achieved during RSA deciphering is the modular exponentiation which calculates $a^b \bmod k$, for some private or secret exponent b and several implementations have been proposed in the literature to achieve it. As an encryption standard, RSA uses different exponentiation algorithms such as squaring and multiplication exponentiation [96], sliding window exponentiation [59], fixed window exponentiation [7], left to right and right to left sliding window exponentiation [93] in combination with GnuPG and diverse versions of Libcrypt libraries. All these algorithms offer different degrees of confidentiality in implementation. In Section 4, we explained the leakage context that helps exploit different algorithmic implementations of RSA with the latest libraries. The following sections detail the implementation of each algorithm and the exploitation of leakage for a better understanding of attacks (Section 7) and countermeasures that enhance confidentiality (Section 8).

7. Cache-based SCAs on RSA

This section provides a detailed overview of cache-based side-channel attacks that target various algorithmic implementations

Table 4
Timing channel attacks in RSA due to cache contention: detailed review.

Attributes	List of implementations in attacks				
	Square & Multiply modular exponentiation	Fixed window exponentiation	Sliding window exponentiation		Left-to-right sliding window exponentiation
	[5]	[7]	[25]	[61]	[93]
Year	2014	2017	2015	2007	2017
Channel	Persistent-channel	Transient-channel	Persistent-channel	Persistent-channel	Persistent-channel
Target Level of Cache	LLC	L1-D	LLC	L1-I	LLC
Target cryptosystem	RSA (GnuPG 1.4.13)	RSA (OpenSSL 1.0.2f) with CRT Heninger-Shacham Algorithm	RSA, ElGamal (GnuPG 1.4.13, 1.4.18)	RSA-1024 (OpenSSL-0.9.8d)	RSA1024, 2048 (Libcrypt 1.7.6) with SWE
Exploitation Features by Attack	Inclusive Cache, Page Sharing, Page De-duplication	Cache Bank Conflicts, Scatter Gather Technique, Hyper-threading	Inclusive Cache, Large Page Mappings	Microarchitectural Attacks, Branch Prediction, Exploiting Average time to read Instruction from Memory	Collision Entropy [92], Right-to-Left Sliding Window Exponentiation
Implementation Platform	Intel core i5-3470 Ivy Bridge, Intel Xeon E5-2430, VMware, KVM	Intel Xeon E5-2430, Sandy Bridge Processor	MC, Dell Server Platform-R720, HP Elite 8300 (Desktop)	HP Elite 8300 with Intel i5–3470	MC, HP Elite 8300 with Intel i5-3470
Complexity (No. of encryptions required for successful attack)	Single decryption	After observing 16,000 decryptions: 1st Part- 60% bits extraction of 4096-bit secret key, 2nd Part- Remaining 40% extraction of 4096-bits secret key	1-decryption for S & ME at RSA, Elgamal (GnuPG 1.4.13, 2-step decryption for S & ME at ElGamal (GnuPG 1.4.18)	single decryption: scattered 200 bits of per 512-bits key	single encryption for both key of 1024 and 2048-bits: less than 40% for $w = 4$ and less than 33% for $w = 5$
Bit Retrieval	96.7% of 1024-bit key	Full 4096-bit Key	Full 3072-bit Key	39% of 1024-bit Key	Full 1024-bit Key, 13% of 2048-bit Key
Leakage Exploitation Technique	Flush + Reload	Prime + Probe	Prime + Probe	Prime + Probe	Flush + Reload
Leakage Context	Multi-core	HW-Threading	Multi-core	HW-Threading	Multi-core

of RSA as well as the underlying caching hardware. First, Table 4, provides a systematic categorization and we describe the attacks in Section 7.1, which is followed by a description of various RSA implementations and attacks that target such implementations in Sections 7.3–7.6. We also discuss the effectiveness of specific attacks on specific implementations.

7.1. Systematic categorization

This section presents known cache-based side-channel attacks on different implementations of RSA. Our systematic categorization of these SCAs is based on multiple parameters. Table 3 provides a first categorization based on the way the attacks target underlying hardware. The motivation behind this classification is that difference SCAs target different cache levels and therefore use different exploitation features of caching hardware to mount an attack. Our extensive study of cache SCAs (Section 2) revealed that, to date, no single cache SCA is capable of exploiting the entire cache hierarchy for information leakage. Table 3 thus provides an essential categorization based on the type of CSCA, leakage exploitation technique, target cache level and the features of caching hardware being exploited by these attacks. The same table lists cryptosystems currently targeted by the attacks along with their publication date.

The classification provided in Table 4, categorizes attacks specific to the algorithmic implementations of RSA. This categorization divides attacks into five major categories with attributes such as the type of information leakage channel, target level of cache, the cache feature(s) exploited, implementation platform, complexity of attack in terms of the encryption required to mount a successful attack, bit retrieval rate and the underlying leakage exploitation techniques used against each algorithmic implementation of RSA. Table 4 also details the leakage context to demonstrate the threat to the caching hardware used. The categorization in Table 4 provides the reader with specific knowledge of potential attack surface against the selected RSA implementation, this may help select the appropriate mitigation techniques. When discussing mitigation techniques in Section 8, we highlight the techniques that are effective against the attacks included here.

7.2. Attacks specific to algorithmic implementations of RSA

In this section, we present individual RSA algorithmic implementations, their vulnerabilities and the attacks that exploit these specific vulnerabilities. We analyze the strengths and weaknesses of the attacks with respect to specific implementations and the effectiveness of countermeasures designed by authors to counter the attacks as a quick patch. Details on countermeasures are given in Section 8.

7.3. Square and multiply modular exponentiation

Algorithm 1 shows the implementation of RSA's public key square and multiply algorithm for modular exponentiation. The algorithm illustrates the different operations performed by RSA to calculate modular exponentiation while encrypting a (victim's) process. When RSA is applied to the victim's process, it performs two major *Square* and *Multiply* operations of, each followed by a *Modulus/Reduction* in this implementation.

Algorithm 1 Square and Multiply Exponentiation [5]

```
1: Function exponent(b, d, m)
2: x ← 1
3: for i ← |d|−1 downto 0 do
4:    x ← x^2
5:    x = x mod m
6:    if (d_i = 1) then
7:       x ← xb
8:       x ← x mod m
9:    end if
10:   return x
11: end for
```

There are two possible sequences of operations. The first sequence is when a square is trailed by a modulus/reduction that is again followed by a multiply trailed by modulus/reduction. In this case, the computation corresponds to a bit "1" (lines 3 − 9). The second operation sequence is when a square is trailed by a modulus/reduction that is again followed by a square trailed by a modulus/reduction. In this case the computation corresponds to a bit "0" (lines 6 − 9). The extraction of a pattern of exponents also reveals the pattern of secret key/operations.

Based on this understanding, we present an attack on the RSA square and multiply algorithm for modular exponentiation implementation. Flush + Reload is one of the most popular attacks on the last level cache with a high resolution as demonstrated in [5]. It focuses on cross-core VMs, where one VM acts as an attacker and spies on the information of another victim VM. The attacker relies on page sharing to trace the instructions performed by the victim. Indeed, the attacker process flushes (or reloads) specific memory lines corresponding to instructions executed for square, multiply and modulus/reduction operations. For each monitored instruction, if a cache hit is detected during the reload step, the attacker considers that the instruction has been executed by the victim. Based on the observed sequence of instructions, the attacker can retrieve most of the exponent bits. There are two characteristics of this attack that make it significantly more effective than previous microarchitectural attacks. First, it can identify access to specific memory lines instead of larger cache sets as it has a higher resolution. Second, it focuses on last level cache, which is farthest of all cores and is shared by multiple cores (all levels). Attacking LLC is common in many other attacks [2,4], but no attack has this level of resolution, frequency, and the fine granularity required for cryptanalysis.

Flush + Reload is an attack that depends to a great extent on memory sharing to achieve high resolution. Virtualization vendors do not suggest sharing between different VMs and nor does any IaaS provider ignore this fact either [13]. It is strongly recommended to perform constant-time implementations for such attacks. Memory locations and sequences of operations that are detectable in such attacks cannot be further detected by constant-time implementations due to having no branching operations. In [5], an attack was demonstrated on the RSA implementation in GnuPG version 1.4.13. The authors demonstrated a one-round decryption attack, which was sufficiently precise and strong to retrieve an average of almost 96.7% of the bits of the secret key.

7.4. Fixed window exponentiation

CacheBleed [7], is an attack on *scatter-gather* technique that prevents cache-based timing attacks. After the exploitation of square and multiply exponentiation [5], to accomplish the modular exponentiation required for execution of RSA secret key operations, OpenSSL 0.9.7c used sliding window exponentiation but it was also attacked [25], as described in Section 7.5. Implementing sliding window exponentiation clearly showed that each multiplier deals with a different set of cache lines and it is easy to identify the retrieved multipliers and to recover the secret key [16,19]. Fixed window exponentiation was proposed to provide a patch for sliding window exponentiation [59,95], it depends on *scatter-gather* implementation [97]. In *scatter-gather*, Intel proposed a countermeasure that changed the memory layout of precomputed multipliers that scatters the multipliers in memory to make sure that similar cache lines will be accessed regardless of the multiplier used [98] and the gather process accesses all cache lines with the use of bit mask pattern to obtain those that contain the fragment of the required multiplier, as explained in Algorithm 2. The main operation performed by OpenSSL during encryption or decryption (using RSA) is modular exponentiation. To calculate this exponentiation, OpenSSL frequently does five squaring operations followed by one multiplication, with reduce operations between each. There are thus 32 possibilities in the multiplications for a multiplier candidate. Knowledge of the multiplier in the multiplication in fact discloses the private exponent and the key. In the past, OpenSSL and GnuPG traced the multipliers by observing the cache lines in which multipliers are located. To protect the multipliers against such attacks, several multipliers are kept in each cache line by making sure that all cache lines are used in each multiplication. Hence, we know that multipliers are intelligently placed in the cache banks.

This technique describes the fact that *scatter-gather* technique is not time constant and it is exploited in CacheBleed [7]. CacheBleed exploits the cache bank conflicts on Sandy Bridge microarchitecture. Cache bank conflicts happen when many requests are made in parallel to the same cache bank and in the case of conflict, some of the clashed requests are postponed. The attacker recognizes the timing information at which the victim accesses its data in observed cache bank by determining the delays caused by clashes on the cache bank. In the case of attack, it is considered that victim and attacker are running in parallel hyper-threads of the same core of processor. The victim and the attacker share the same L1-Data cache. The L1-Data cache is divided into multiple banks and banks cannot handle multiple requests at the same time. So the attacker takes advantage of this situation and launches a large number of load accesses to the cache bank. Meanwhile, the attacker observes the time needed to accomplish these accesses. If during this process, the victim also accesses the same cache bank, there will be a clash between the victim's and attacker's accesses to the same cache bank which causes the access to be suspended. Therefore, it means that if victim accesses the same cache bank, it will take longer for the attack to process.

Past studies [12,16,19,99], mentioned that cache bank conflicts cause timing variations and already warned that such timing-based channels may leak low address bit information. The fragments of all multipliers are stored in prefixed offsets of each cache line. All the *scatter-gather* implementations contain memory accesses that rely on used multiplier and the secret key. Only the last three bits of the multiplier are encoded as addresses, while the other two bits are considered as the index of cache line within the group of cache lines containing the fragment. It was considered that secret-dependent accesses are at a finer level than cache line granularity, so the *scatter-gather* approach was considered secure toward SCAs. But cache bandwidth is also a bottleneck in Intel processors, due to which, Intel introduced multiple banks [100]. Delays due to cache bank conflicts have also been documented for many processor versions [101]. CacheBleed takes advantage of causing many accesses to the same cache bank. The memory layout is designed in such a way that fair scheduling is applied to respond to each process request. Thus,

the attacker takes advantage of causing these conflicts at the cache bank to observe timing variations. Attacks have demonstrated that the memory layout of Intel Sandy Bridge processors was designed to scatter multipliers intelligently but by causing cache bank conflicts, it is still possible to leak information of multipliers. This type of vulnerability works on Sandy Bridge, Nehalem and Core 2 processors but does not work on Intel Haswell processors, where, apparently cache bank conflicts are no longer an issue.

Scatter-gather has no secret-dependent accesses to cache lines, but it does have secret-dependent access to the cache bank. So, knowledge of lower bits makes it easy to observe the number of accesses to the cache bank and to visualize the corresponding cache banks with bins in the memory allocation. For example, having this knowledge of lower bits, we can understand that lower bits 000 correspond to multipliers 0, 8, 16, 24 with bin 0 that span cache banks 0 and 1. 001 correspond to multipliers 1, 9, 17, 25 and bin 1 that span cache banks 2 and 3. This was the first round of attack that explored the 3 least significant bits of attack by extracting almost 60% of the key. This information explained that three least significant bits are enough to explore 100% of the multipliers and out of 32 multipliers, we have reduced the knowledge of multipliers to four. By having the knowledge of three least significant bits of every window of five bits for Chinese remainder theorem coefficient d_p and d_q, we used Heninger and Shacham [63,95] algorithm. It extracts all information of RSA secret key which will give us $k_p k_q N$ mod e mode and we can deduce the information on d_p and d_q. Later, using the branch and prune algorithm [95], we extract exact multiplier and the complete key (the second round of complete key extraction). CacheBleed proposes some mitigation techniques for this attack e.g. disabling hyper-threading [48,49] (Table 5, Section 8). Another proposed solution is increasing the bandwidth of caches to reduce the contention in cache banks (Table 5, Section 8).

Algorithm 2 Fixed Window Exponentiation [7]

input : window size w, base a, modulus k, n-bit exponent $b = \sum_{i=0}^{\lceil n/w \rceil} b_i \cdot 2^{wi}$
output : a^b mod k

```
1: //Precomputation
2: a_0 ← 1
3: for j = 1, ..., 2^w − 1 do
4:     a_j = a_{j−1} . a mod k
5: end for
6:
7: //Exponentiation
8: r ← 1
9: for i = ⌈n/w⌉ − 1, ..., 0 do
10:    for j = 1, ..., w do
11:        r ← r^2 mod k
12:    end for
13:    r ← r · a_{b_i} mod k
14: end for
15: return r
```

7.5. Sliding window exponentiation

Sliding window exponentiation is a countermeasure patch provided by Intel in response to Flush + Reload attack on square and multiply algorithm for modular reduction. As shown in Algorithm 3, sliding window exponentiation precomputes the set of multipliers and utilizes the multiply pattern to measure squares. Multiplication is used to point the multiply routines that include square and real multiply operation in itself. So, we conclude that sliding window exponentiation performs the necessary number of multiplications and precomputes multipliers for better performance and stores them in tables that can be indexed at any time. The algorithmic implementation images the exponent from the most significant bit to the least significant bit and for each digit, intermediate results are squared. Whenever, it reaches the least significant bit of a non-zero window, a multiplication occurs. Therefore, sliding window exponentiation still leaks the location of non-zero multipliers that can be noted between square and multiply operations. Furthermore, the number of squarings between constant multipliers can leak the value of some zero bits [59]. The attacker can distinguish the location and value of the non-zero multiplier used needed to explore the exponent.

The authors in [25] present the first *Prime + Probe* type of attack on LLC that targets both the square and multiply modular exponentiation and sliding window exponentiation on different versions of GnuPG (1.4.13, 1.4.18). The implementation of sliding window exponentiation with other cryptographic implementations such as Elgamal are also tackled in this paper with version1.4.18 of GnuPG [25]. Two basic techniques have been demonstrated in this attack; (1) probing one specific cache set with no knowledge of virtual address mapping because it is hard for attackers to probe all LLC as this requires many more probe cycles, and (2) use of temporal access patterns instead of traditional spatial access patterns to classify the victim's critical accesses.

There are certain challenges to exploiting the attack on SWE (GnuPG 1.4.18), these include some precomputed multipliers that are called dynamically and we do not know to which cache set they are allocated because they are sparse and irregular, making it difficult to distinguish squares from multiplications. To do so, we need to find the multiplication operation in the same way as the square operations were found in the previous discussion (exploiting the square and multiply algorithm for modular exponentiation, Section 7.3). In the trace execution, we will have all the multipliers, but they do not filter the exact location of every multiplication.

One property of multipliers is that the multiplier access pattern repeats itself across multiplications. The means that if a specific multiplier is used in the third multiplication, it will always be used in the third multiplication. Thus, we know that there is a temporal access pattern we need to discover. This can be achieved by identifying enough patterns that can cluster similar properties (repeating patterns). By collecting a long trace with clustering algorithm, it is possible to understand the pattern of the multiplication line and arbitrary cache sets, i.e., for the multiplication set to know when multiplication is taking place. This method makes it possible to recover all the multipliers and with all the multipliers we have knowledge of exactly eight precomputed multipliers. This attack has two rounds, in the first round it collects the traces and in second round it computes the multipliers by clustering.

Certain mitigation techniques are proposed for this type of attack like fixing GnuPG and writing a sensitive code for exponent blinding. Writing a constant-time code without branching conditions and secret memory dependent accesses has also been suggested. This paper proposes fixed window exponentiation as a countermeasure against such attacks but it has also been tested and some leakage of information occurred proving that it is not a constant-time implementation (detailed in Section 8.5).

Authors in [61], have introduced the Instruction-cache (or I-cache) as another source of side-channel attack. They have experimentally verified leakage in the execution flow through I-cache by targeting the Sliding Window Exponentiation implementation of RSA-1024 as implemented in OpenSSL (version 0.9.8d). This I-cache attack is based on the concept of executing a spy process, which keeps track of the changes in the state of interested I-cache lines of level-1 during the execution of a cipher process.

Table 5
State-of-the-art on hardware/software countermeasure techniques w.r.t. cache hierarchy.

Cache Level	Countermeasure	Description	Year	Type
L1	Disable Hardware Threading [59,102]	A way to reduce the cache flushing	2005, 2010	Hardware Threading
	Newcache [103]	Dynamic randomized memory-to-cache mapping	2016	
	Auditing [104]	Detecting malicious behaviors	2014	
	Increasing bandwidth of Cache [48,49]	Reducing Contentions	2014, 2012	
	Constant-Time Techniques [7,105,106]	Fixed time instructions	2017, 2015, 2016	Time Slicing
	Cache Flushing [107–109]	No privilege to flush specific lines	2014, 2013, 2013	
L2	Hardware Cache Partition [21,22,110,111]	Partitioning cache for security sensitive applications	2012, 2007, 2015, 2005	Hardware Threading
	RP Cache [110,112]	Random Indexing of lines	2009, 2007	
	Disable Hardware Threading [59,102]	A way to reduce the cache flushing	2005, 2010	
	Minimum Timeslice [107]	Preventing attacker to observe cache state in preemptions	2014	
	Cache Flushing [107–109]	No privilege to flush specific lines	2014, 2013, 2013	
	Retired Instruction Count [113]	Scheduling based on retired instruction counts	2013	
LLC	Hardware Cache Partition [39]	Isolation of cache for sensitive applications	2016	Multi-core
	Cache Coloring [114–116]	Allocating colored pages for sensitive application	2014, 2014, 2011	
	STEALTHMEM [20]	Allocating colored pages for sensitive applications	2012	
	Quasi-Partitioning [117]	Allocating a budget per cache to set security domain	2016	
	RP Cache [39]	Random Indexing of lines	2016	
	Noise Fuzzy Time Reducing Resolution of Clock Time Warp [19,65,76,99,118,119]	Adding external processes to confuse attacker process	2005, 1992, 2015, 2013, 1994, 2012	
	Disable Page Sharing [16,117]	Prevention, copy-on access scheme	2006, 2016	
	Disabling Cache Sharing [102]	Logical isolation with in a physical cache	2010	
	Scheduling-based Obfuscation [120]	Scheduled noise induction	2014	
	Leakage Feedback [121]	Quantify leakage to use as an input to mitigate the attacks	2017	

Algorithm 3 Sliding Window Exponentiation [25]

input : window size S, base b, modulo m, n-bit exponent e represented as n windows w_i of length $L(w_i)$
output : $b^e \bmod m$

```
1: //Precomputation
2: g[0] ← b mod m
3: s ← MULT(g[0], g[0]) mod m
4: for j from 1 to 2^(s−1) do
5:    g[j] ← MULT(g[j − 1], s) mod m
6: end for
7:
8: //Exponentiation
9: r ← 1
10: for i from n downto 1 do
11:    for j from 1 to L(w_i) do
12:       r ← MULT(r, r) mod m
13:    end for
14:    if w_i ≠ 0 then
15:       r ← MULT(r, g[(w_i − 1)/2]) mod m
16:    end if
17: end for
18: return r
```

Attacker observes the changes in state of I-cache by passing through three phases. First, attacker initializes the predetermined state of particular lines in I-cache with dummy instructions by prefetching. Then, the Attack lets the cipher program to execute for a short time, which may replace the dummy instruction from I-cache depending on the execution flow of cipher. Lastly, the Attacker observes the existence of the same dummy instructions in previously loaded I-cache lines by measuring their execution time upon access. The re-execution of dummy instructions would naturally take more time if they were were evicted by the cipher during execution, which indicated the cache line were modified by the cipher. Otherwise, their re-execution would take shorter time if the cipher did not modify concerned cache lines. This I-cache access information directly indicates the execution flow of RSA that relates with the key bits (secret information). Authors remained unsuccessful in retrieving full key because of the encoding in SWE and reported the extraction of 200 "scattered" key bits. Moreover, authors did not relate the extracted execution flow with the key bits. This attack presented experimental results in the form of captured execution flow of RSA while executing a single decryption round of RSA.

7.6. Right-to-left and left-to-right sliding window exponentiation

Sliding window exponentiation (SWE) is an efficient implementation of RSA shown in Algorithm 4. Algorithm 4 consists

Algorithm 4 Left-to-Right Sliding Window Modular Exponentiation [93]

input : Three integers b, d and p where $d_n.....d_1$ is the binary representation of d.
output : $a \equiv b^d (\mathrm{mod} p)$

1: $b_1 \leftarrow b, b_2 \leftarrow b, a \leftarrow 1, z \leftarrow 0$
2: *//Precomputation*
3: **for** $i \leftarrow 1$ *to* 2^{w-1} **do**
4: $\quad b_{2i+1} \leftarrow b_{2i-1} \cdot b_2 \bmod p$
5: **end for**
6: $i \leftarrow n$
7: *//Exponentiation*
8: **while** i *from* n **downto** 1 **do**
9: $\quad z \leftarrow z +$ COUNT_LEADING_ZEROS$(d_i....d_1)$
10: $\quad i \leftarrow i - z$
11: $\quad l \leftarrow \min(i, w)$
12: $\quad u \leftarrow d_i....d_{i-l+1}$
13: $\quad t \leftarrow$ COUNT_TRAILING_ZEROS(u)
14: $\quad u \leftarrow$ SHIFT_RIGHT(u, t)
15: $\quad$ **for** $j \leftarrow 1$ *to* $z + l - 1$ **do**
16: $\quad\quad a \leftarrow a \cdot a \bmod p$
17: $\quad$ **end for**
18: $\quad a \leftarrow b_u \cdot a \bmod p$
19: $\quad i \leftarrow i - l$
20: $\quad z \leftarrow t$
21: **end while**
22: **return** a

of a precomputation step (lines 3–5) and an exponentiation step (lines 8–21). Odd multiples of base are computed once in the computation step, and are then used repeatedly in the exponentiation step. In the exponentiation step, first key is divided into zero and non-zero groups (lines 9–14) and the exponent is then computed by squaring (line 16) and multiplication (line 18) based on zero and non-zero groups respectively. The non-zero groups are created such that each group can be represented in odd digits, so that the odd multiples computed in the precomputation step can be used. RSA using sliding window exponentiation encodes the key to compute the exponent efficiently, and it also limits the leakage of key bits from the cache. The encoded key, which is a sequence of zero and non-zero digits, defines the squaring and multiplication rather than the binary key, as discussed in Section 7.3. Hence, the sequence of square-and-multiply access patterns extracted using side-channel attacks represents the zero and non-zero digits without the value of non-zero digits, which hide much of the information about the location and number of bits "1" in the key. Because of encoding, the attack presented in [61] failed to retrieve the full key and reported the extraction of only 200 "scattered" key bits per 512 bits of key.

There are two approaches to encode keys based on the direction of scanning key bits: (1) Left-to-right encoding approach and (2) right-to-left encoding approach. RSA using SWE typically encodes the key by scanning starting from the most significant bit is called the *left-to-right* approach. This approach represents the key in a sequence of odd decimal digits ranging from 0 to 2^w, where w is the window size. To generate odd decimal digits, the left-to-right approach creates groups that start and end with a bit "1" of at most window size. It first tries to create a group of maximum size w but if it cannot find bit "1" that is w bits apart from the starting bit, it reduces the size of group so it finds bit "1" at the end. For example, let key $=$ *key* $=$ 10100110 and window size $=$ 3, then the left-to-right approach creates a group of keys such as 101 00 11 0 and yields decimal digit representation of key as 005 00 03 0. Algorithm 4 is a left-to-right sliding window exponentiation that creates a group in each iteration of the loop by removing the leading zeros to find a starting bit "1" and trailing zeros to find an ending bit "1". The algorithm then executes a squaring operation on each zero digit and both square and multiply operations on non-zero digits.

In cache-based side-channel attacks, the square and multiply pattern extracted by the attack indicates sequences of zero and non-zero digits in the encoded key without the non-zero digits. Because the value of non-zero digits is not known, the attacker has no information about the number of bits "1" or the exact location of bits "1" in each group. *Right-to-left* is another encoding approach used in the sliding window exponentiation method. This approach starts scanning the binary representation of the key beginning from the least significant bit and groups the key bits of fixed window size that satisfy the two following conditions: (1) each group contains at least one bit "1" and (2) each group starts (right most bit) with bit "1". The right-to-left approach then converts each group of binary bits into its decimal representation. For example, let *key* $=$ 10100110 and window size $=$ 3, then the right-to-left approach creates a key group such as 101 0 011 0 and yields a decimal digit representation of key as 005 0 003 0. Lastly, to compute the exponent, it again scans the encoded key and executes *square* and *multiply* operations based on zero and non-zero digits: only *square* operation on each zero digit; both *square* and *multiply* operations on each non-zero digit. After understanding the working principle of left-to-right and right-to-left SWE, the weakness of both algorithms is exploited in [93] which states that both implementations help mount an attack and that both implementations are vulnerable, but left-to-right SWE is more vulnerable than right-to-left SWE; the paper thus provides a larger window to compute maximum exponent bits of a secret key. The author in [93] demonstrated vulnerability of the left-to-right sliding window exponentiation using probabilistic theory and showed that the left-to-right approach leaks about 18% more key bits than the right-to-left approach.

Trailing Zeros: The left-to-right encoding algorithm tries to group the key bits such that odd non-zero digits are created. To do so, the algorithm creates a group that starts and ends with bit "1" of the key. In addition, the group cannot be bigger than the size of the window.

Leading One: The case of shorter window size in which left-to-right encoding creates a group with no trailing zeros. This yields two immediately consecutive multiplications in the sequence of squaring-and-multiplication pattern extracted using cache-based side-channel attacks.

Leading Zeros: The encoded key bits between two consecutive groups are vulnerable because they are always"0" bits. In the square and multiply pattern, the attacker reveals "0" bits if it finds two consecutive multiplications that are more than the window size apart. This type of scenario exists in both left-to-right and right-to-left encoding of key bits.

The attack in [93] comprises the following three steps: first, the attack extracts the square and multiply access sequence from I-cache using a *Flush + Reload* attack. The attack then applies recovery rules to reveal trailing zeros, leading one and leading zeros in the key. This results in the recovery of key bits that are less than or equal to the partial key. Lastly, if previous phases yield more than or equal to 50% of key bits, the attack applies the modified-Heninger–Shacham algorithm to recover the entire key. Some solutions are proposed to counter the attack, such as restricting flushing property (Section 8.3.5) or reducing clock and noise resolutions (Section 8.2).

7.7. Discussion

After being subjected to many attacks, RSA is in constant danger and a solution in GnuPG is highly desirable because it does not address the issue of maintaining isolation and preventing information leakages. For the moment, hardware developers' efforts have targeted efficient utilization, maintaining coherency and optimized cache architecture [39,114]. Consequently, providing further hardware solutions is not possible for the moment. Instead software-based countermeasures are urgently required that are quick and efficient solutions to the problem of information leakage, which compromises the confidentiality and integrity of systems which is also a concern in this paper.

8. Countermeasure techniques

There has been extensive research work on countermeasure techniques to mitigate cache-based side-channel attacks. These countermeasure techniques broadly fit three categories; mitigation techniques based on new hardware design [21,112,122,123], application-specific (software) mitigation techniques [12,16,124], and compiler-based mitigation techniques [125]. Table 5 provides a detailed overview of software and hardware mitigation techniques that have been proposed so far with respect to cache hierarchy. The table also includes architectural and application-specific features of these techniques. The countermeasures in Table 5 are categorized according to different types and levels of cache along with their description. Unfortunately, there are not many general hardware-based mitigation techniques for classical systems that can be adopted for mainstream processors. These mitigation techniques have a huge performance overhead rendering their practical adaptation nearly impossible [20,58].

In this section, we discuss several software mitigation techniques proposed over the last decade or so. Since these mitigation techniques often exploit architecture-specific or application-specific features, as discussed in Section 7, we cannot suggest one recipe for all types of implementations. Different mitigation techniques deal with different levels of threat at application and architecture levels as explained in Section 4. Recent countermeasures for hardware and software are listed in Table 5. They were identified in major classes including hardware threading (core-shared state at L1–L2 levels of cache due to hyper-threading/simultaneous multi-threading), time slicing (core-shared state on L1–L2 levels of cache due to timing variation and self-contention) and multi-core (package-shared state on LLC creating side-channels and covert-channels) [39].

We present some practical techniques along with their advantages and disadvantages. Table 6 provides an exhaustive list the software-based countermeasures published to date. These countermeasures are divided into sub-categories to make it easy to distinguish the class of related software countermeasures. Table 7 lists all software countermeasures w.r.t the cache hierarchy for which the mitigation can be used, the threat level (hardware-threading, context switch, multi-core) it addresses, and leakage context as discussed in Section 4. This categorization was inspired by a work presented in [13].

These countermeasures can be used against the specific attack category of RSA referred to in Section 7, and can also be used to mitigate exploitation in other cryptographic algorithms. Software countermeasures are not restricted to solving one type of problem in one type of cryptographic algorithm. Rather these countermeasures are used in a generic sense to mitigate cache-based side-channel attacks. Modern architectures are complex in nature, and mitigation techniques proposed for a specific leakage may consequently not fully protect the system. As discussed in Section 4, hardware and software developers need to consider the entire threat model that could be exploited by the malicious applications. While discussing existing mitigation techniques, we also carefully review the architectural features that are exploited and their effects on these mitigation techniques within the scope of proposed threat model. We also consider security critical parameters in both the application layer and the architectural layer that can be used in mitigation without changing the underlying architectural features. At the end of this discussion, we will be able to analyze the practicality and efficacy of the countermeasures on our threat model explained in Section 4.

Table 6
Categorization of state-of-the-art software countermeasures.

Category	Countermeasures
Logical/Physical Isolation-based Countermeasure Techniques (Section 8.1)	• Cache Coloring [114–116,118] • CloudRadar [126] • Migration of VMs [127] • STEALTHMEM [20] • CacheBar [117]
Noise-based Countermeasure Techniques (Section 8.2)	• Fuzzy Time [76] • Eliminating Fine Grained Timers [87] • Bystander Workloads [128] • Anti-correlated Noise [114]
Scheduler-based Countermeasure Techniques (Section 8.3)	• Scheduling-based Obfuscation [6,120,129] • Leakage Feedback [121,130,131] • Server Side Defenses (cache Flushing) [109] • Retired Instruction [113] • Minimum Timeslice [107] • Cache Flushing [6,108,109]
Partitioning-Time Countermeasure Techniques (Section 8.4)	• Kernel Address Space Isolation [30,74]
Constant Time Countermeasure Techniques (Section 8.5)	• CacheAudit [132] • FlowTracker [133,134] • Valgrind [135] • Catalyzr [136]
Auditing & Detection Techniques (Section 8.6)	• NIGHTs-WATCH [137] • SCADET [138] • HExPADS [139] • Deja-Vu [140] • CacheShield [141] • SpyDetector [142] • CloudRadar [126] • Misc. [104,143–151]

8.1. Logical/physical isolation-based countermeasure techniques

Disabling resource sharing and executing applications in complete physical and/or logical isolation to protect against adversaries has been a popular mitigation technique, even though it is still conceptually trivial. In this section, we present software mitigation techniques based on partitioning/isolation to counter recent cache-based side-channel attacks.

8.1.1. Cache coloring

Cache coloring is a mechanism to partition the cache with the help of software. Cache coloring is intended to generally enhance cache performance in real time and to reduce cache contention [118,152,153]. Cache coloring has proved to be an important mitigation technique against cache-based timing SCAs. Cache coloring segregates the memory into colored pools and assigns memory from distinct pools to be transformed into a security restricted domain. Physical frames whose addresses differ from the colored bits are never mapped to the similar cache set. There are many implementations for cache coloring including static and dynamic cache coloring [114–116,118].

Table 7
State-of-the-art software countermeasures for different levels of cache and threat model within Intel x86.

Cache level	Context switching	HW threading	Multi-core
L1/D-I	Constant Time Implementation [132–135], Minimum Timeslice [107], Düppel [108], Server Side Defenses [109], Kernel Address Space Isolation [30,74], Migration of VMs [127], CloudRadar [126]	Cache Flushing [6,108,109], Auditing [6,108], Retired Instruction [113]	Minimum Timeslice [107], Cache Flushing [6,108,109], Constant Time Implementation [132–135], Fuzzy Time [76]
L2	Eliminating Fine Grained Timers [87], Bystanders Workloads [128], Anti-correlated Noise [114], Düppel [108], Auditing [6,108], Retired Instruction [113]	Eliminating Fine Grained Timers [87], Bystanders Workloads [128]	Minimum Time Slicing [107], Cache Flushing [6,108,109]
LLC/L3	STEALTHMEM [20], Auditing [6,108]	Gang Scheduling [20]	STEALTHMEM [20], Cache Flushing [108,109], Cache Coloring [115,116,118], Injecting Noise [76,87,114,128], CacheBar [117], Quasi-Partitioning [117], Auditing [6,108], CloudRadar [126], Quasi-Partitioning [117], CacheBar [117], Scheduling-based Obfuscation [120], Leakage Feedback [121]

In static coloring, some static colors are allocated for security critical applications. As the number of security-demanding applications is increasing, static coloring is unable to respond to all the requests dynamically. That is why an approach for dynamic coloring was introduced, which considers a dynamic number of secure colored pages to the security critical applications at run time. One such approach in [116] proposes a non-intrusive and low overhead technique of page coloring named Chameleon. The Chameleon technique provides secure color to the secure process so that a strict isolation in virtualized environment can be maintained. Before a process goes to a security critical section, hypervisor is notified and during that section, the secure color is only available for security critical operations and cannot be used by any other VMs co-located on the same hardware platform. This technique provides both *full* mode and a *selective* mode protection mechanism, but does not compare the results with other dynamic coloring approaches to evaluate the performance parameters, and the impact of this approach on stopping any kind of cache-based side-channel attacks is not documented in paper [115].

In static coloring, some static colors are allocated for security critical applications. If the number of security demanding applications increases, static coloring is unable to respond all the requests dynamically. That is why, approach for dynamic coloring was introduced, which represents dynamic number of secure colored pages to the security critical applications at run time. One such approach is discussed in [116], which proposes non-intrusive and low overhead technique of page coloring named as *Chameleon*. The Chameleon technique provides secure color to the secure process so that a strict isolation in virtualized environment could be maintained. Before a process goes to a security critical section, hypervisor is notified and during that section, the secure color is only available for security critical operation and cannot be used by any other co-located VMs of the same hardware platform. This technique provides both *full* mode and *selective* mode protection mechanism, but it did not compare the results with other dynamic coloring approaches to evaluate the performance parameters. Furthermore, the impact of this approach to stop any kind of cache-based side-channel attacks has not been documented in the paper [116].

Another form of cache coloring is XEN's memory management tool in [115]. This technique demonstrates a complete closure of side-channel between different virtual machines with the help of cache coloring. The authors also managed to analyze the performance cost that is 50% for Apache-2013 benchmark and there was far less penalty with small working sets. One problem seen with cache coloring is that technique is unable to use large pages, whereas many processors are able to use large pages up to nearly 1 GB in *x*86 architectures. One benefit of having large pages would be the reduction of overlapping pages and requirement for colored pages will be reduced to a very small number.

The effectiveness of using cache coloring to reduce the impact of cache-based covert-channels is described in [114]. The mechanism has been proved to be more efficient on cores with simpler structures than in cores with complex structures because of TLB contention that can be solved by flushing TLBs on a context switch of VMs. A rather new challenge moving from directly-mapped cache to cache sets. While we know that LLC is divided among cores connected with a ring bus as illustrated in Fig. 2, locating the physical address of the cache line depends on addressing a cache block and a set within that block. The newer Intel microarchitectures contain a hash function to locate these blocks. Without prior knowledge of hash functions, the available colors are confined within a cache block [54]. Several authors have reverse-engineered the hash function of multiple processor models that support the use of multiple colors [52–54,64,154] but this might not be possible for future CPUs [39].

8.1.2. Stealthmem

STEALTHMEM [20] is a software mitigation approach that uses a limited principle of cache coloring to mitigate cache-based side-channel attacks with three different perspectives; it checks the impact of its proposed stealth pages in the case of context switch, hyper-thread and sharing the LLC and analyzes its performance with dynamic cache coloring. It provides a small amount of colored memory, which was targeted to avoid contention and flushing in the LLC. The aim of this approach is to provide stealth pages for secure critical data that are encrypted. This specific approach reserves stealth pages for each core on which each VM is located. Using the same stealth page for two different cores is made impossible in this approach and a regular check system is maintained called a PTA (Page table alert) scheme. The PTA scheme ensures that the cache implements K-LRU mechanism, in which a cache miss is declared not to flush any of the K lines from recently accessed lines. This mitigation technique ensures use of small number of stealth pages and locks them for each core in the LLC. Therefore, an attempt to access any other page that is reserved triggers a page fault in the form of STEALTHMEM. Pre-arranging cache colors minimizes the number of cache sets utilized. This mechanism was analyzed against the probability of a context switch and sharing cores but for hyper-threading only disabling hyper-threading has been suggested as a straightforward solution. The performance of STEALTHMEM was analyzed and showed relatively small overhead for SPEC-2006 benchmark, around 5.9% for STEALTHMEM and 7.2% for PTA due to having extra faults. The overall performance degradation of using this mitigation is around $2-5\%$ for three encryption algorithms, DES, AES and Blowfish.

8.1.3. Migration of VMs

Information leakage in co-residing VMs has become a major threat to cloud environments. To mitigate such channels, *Nomad* [127] implemented a software-based solution to mediate the migration of VM workloads. Migration-as-a-Service cloud computation model believes in placement algorithm of VMs. Past and current VM assignments are saved in epochs as input and the next placement of VMs is decided based on this information. It identifies provider-assisted VM migration as a novel defense strategy for information leakage that happens due to side-channels. The system is analyzed on a scalable online VM migration where it has shown that this heuristic is able to handle massive data center workloads. To minimize the effect of services running on each VM, *Nomad* provides client API, which allows clients to monitor non-relocatable VMs. This mitigation technique provides performance overhead for traditional cloud applications such as web services and Hadoop MapReduce.

8.1.4. Quasi-partitioning

Manipulation of resources helps the attacker to obtain information on the victim and to conduct an effective access-driven side-channel attack. *CacheBar* [117] is a mitigation technique against access-driven side-channel attacks that target last level caches (LLC) shared across cores in processors. The property of sharing makes it possible to leak information between security domains such as clouds and tenants in a big picture. *CacheBar* arranges physical memory pages in a dynamic way to prevent sharing of LLC lines and to prevent side-channels occurring due to Flush + Reload techniques in LLC. It also creates a cacheability mechanism of memory pages to work against Prime + Probe attacks happening in LLC. CacheBar is a memory management subsystem within Linux kernel to effectively work on such side-channels. It allocates a budget in cache for the security sensitive applications to execute.

8.2. Noise-based countermeasure techniques

Except Prime and Abort, all the attacks we analyzed in Section 7 refer to the accuracy of measurement of minute timing variations by the attacker, whether encryption itself or access to the attacker's memory. A suggestion to counter timing attacks is to introduce noise in the observed timings by executing random delays in the operations being performed. This slows down the attacker's performance and the attacker will then have to average many executions and measurements. Theoretically, it was suggested to prevent the exploitation of timing channels through an increase in contention, thereby ensuring that the attacker's measurements have to deal with a lot of noise that in practice, it is worthless for the attacker to monitor. Noise-based countermeasure techniques are one of the most popular approaches to mitigate side-channel attacks. The main goal of such approaches is to reduce the attacker's measurement precision in such a way that the gathered data cannot be exploited to retrieve the targeted secret information. It is worth noting that this approach does not remove the source of leakage and hence, does not provide full protection. Consequently, the attacks presented in Section 7 can still be performed in a noisy environment. However, the attacker has to invest much more effort to collect and analyze the measurements. Depending on the execution context of the victim, this could lead to a situation in which the attacker can only retrieve partial information that is not sufficiently relevant to finalize the attack.

8.2.1. Fuzzy time approach

Fuzzy time approaches [76] introduce noise into all the events that are visible in a process aimed at mitigating attacks. Modification of XEN hypervisor to inject noise to eliminate fine grained timers is explained in [87], where noise is injected into high resolution timing measurements in VMs by modifying the results of RDTSC instruction. This mitigation technique addresses some potential research questions of other sources of fine-grained timers. A bystander VM for injecting noise into the cross-VM L2-cache covert-channel is described with a configurable workload in [128]. This technique uses a time Markov process to check the effect of bystanders on a cross-VM covert-channel. The effect is analyzed in two terms: scheduling of the virtualization platform and the workload (bystanders). In this study, influential factors that affect covert-channels in Prime + Probe attacks are analyzed by scheduling on XEN (to evaluate the error rate of bystander VM). In checking this, the authors were able to see that as long as bystander VMs tune the consumption time of CPU, they are unable to affect cross-VM covert-channel. It was demonstrated in this attack that when injecting noise into Prime + Probe channels, bystander VMs need to modulate their working sets and memory access patterns. The efficiency of said mechanism is evaluated through trace-driven simulations in which VMs are supplied with provisioning strategies.

8.2.2. FLUSH + PREFETCH technique

A noise-based mitigation known as FLUSH + PREFETCH [155] technique has been proposed to obfuscate the memory access behavior of a secure application using independent threads that randomly access the memory belonging to secure applications. As a proof of concept, FLUSH + PREFETCH defends the secret key of RSA cryptosystem against a high-resolution cache-side channel attack known as Flush + Reload. The countermeasure benefits from two attacker shortcomings; the attacker is unable to identify the source that generated a specific cache access and attacker cannot detect multiple operations happening simultaneously on a cache line. The two limitations have been exploited by injecting noise into cache access pattern using concurrent threads that contain prefetch and clflush instructions. Generating noise in such a way makes it difficult for the attacker to extract the encryption/decryption key from cache access information.

8.2.3. Anti-correlated noise

Anti-correlated noise is suggested in [114], which can in theory completely close the channel. The rate of (uncorrelated) noise increases as the channel capacity decreases dramatically. But the use of such mechanisms has a significant performance overhead and it is considered unrealistic to reduce the bandwidth of channels in such magnitudes [39].

Approaches for eliminating hardware timing channels require new hardware design architecture to minimize the risk of sharing or loosely-coupled architectures to minimize the availability of shared resources. System designers are trying to design highly secure systems and such approaches may be a drawback in terms of performance degradation [76]. Abandoning contemporary processors means abandoning the installed application layer as well as OS. The existence of hardware timing channels is thus a major threat. However, introducing noise to obtain highly secure systems has been shown to be inefficient [114]. Previous techniques cannot deal with them because closing the signal for the channel is a difficult task and those that can be closed undergo a dramatic reduction in performance [39,114].

8.3. Scheduler-based countermeasure techniques

Scheduling is another effective technique to mitigate timing channel attacks. Such attacks are passive, so dealing with them is not a simple task. The hypervisor scheduler cannot differentiate between malicious and victim VMs. But it is possible to limit information leakage using novel scheduling schemes to minimize the attacking VMs intervention in the victim's memory accesses. One way of scheduling is to minimize the overlap time of VMs but it comes with a major performance cost due to excessive context switching. The time overlap can be limited by the hypervisor introducing some noise before the timeout for each VM can interrupt the transmission of data to an attacker VM through a timing side-channel. Attacks with simultaneous or consecutive access that share the same hardware resources can be mitigated in two ways: either by providing exclusive time sliced accesses or by carefully managing the transition between each time-slice.

8.3.1. Scheduling-based obfuscation

A hypervisor scheduler can call obfuscation functions in order to inject noise into the potential side-channel. In [120], the authors modified the XEN scheduler and proposed a new scheme that uses two parameters: *overlap_cap* and *noise_function*. *overlap_cap* is the ceiling value for overlapping time of execution of two VMs, and *noise_function* is injected noise for different side-channels. For example, in order to cater to memory bus contention based side-channel attacks [156], the administrator can induce the noise function as some atomic memory access. Hence the attacker will not be able to distinguish whether the signal is coming from the victim or is caused by hypervisors' noise. These parameters could be used to achieve the appropriate security/performance trade off depending on the administrator's preference. The authors in [129] propose a scheduling based technique called as *Shuffler* that efficiently limits the vulnerable probability of attacks in VMs. The solution claims to distribute CPU time to vCPUs with equal probability which would reduce the overall vulnerable probability of the system. *Shuffler* scheduler, hence, shows minimum information leakage to mitigate cross-VM SCAs with negligible performance penalty while preserving high resource utilization.

8.3.2. Leakage feedback

Schedulers are not aware of any security related task that may leak the information. However, if schedulers are designed in a way that they are conscious of the sensitivity of a process, information leakage can be minimized. Some approaches like those in [130] and [131] use the flushing memory at the end of every sensitive operation to remove the footprints of traces. But such frequent flushing operations jeopardize schedulability and can be expensive especially for real-time tasks in meeting their deadlines. Schedulers can be designed such that the information leakage can be quantified to be used as feedback to suppress it. The authors in [121] follow a workflow model used in real-time systems in which jobs are periodically produced to be scheduled and to be completed before assigned deadlines. The tasks are divided into steps that individually consist of three parameters: execution time, leakage value and security level. The steps consist of atomic operations independent of scheduler preemption and help assess the behavior of the tasks. The authors propose a heuristic approach to flushing to achieve zero leakage while still achieving acceptable scheduling.

8.3.3. Retired instructions

Another way of secure scheduling can be based on retired instructions (RI) count. RI is a parameter available in hardware performance counters (HPC) in modern CPUs. In [113], the authors suggest an instruction based scheduler that does not impact timing in hardware in terms of cache, TLB and CPU buses. The authors claim that the impact of their implementation on performance is minimal compared to time based scheduling. However, their solution, needs to be tested for multi-core architectures.

8.3.4. Minimum time slicing

This mitigation technique investigates the principle of soft-isolation to minimize the risk of sharing by providing a sophisticated scheduling mechanism that confines the occurrence of preemptions for VMs and can effectively prevent existing Prime + Probe cache-based side-channel attacks [107]. Determining a minimum time slice for exploitable component prevents the attacker from scrutinizing the state in the middle of any sensitive operation at the cost of increased latency. Attacks containing the Prime + Probe approach [6], are dependent on the ability to inspect the state of victim by frequently targeting preemptions. The soft-isolation approach increases the latency to such a mediation point that the preemption interval increases and the attacker cannot inspect the state of the victim. This defense mechanism is specific to one approach (Prime + Probe) but can probably be exploited by more sophisticated attacks such as Flush + Flush, Flush + Reload, etc. [39].

8.3.5. Cache flushing

The obvious problem in context switching is that the attacker VM can observe the state of the victim VM during switching. The obvious solution to this problem is flushing the victim's data VM before every switch. This mechanism makes it hard for the attacker VM to observe the state of the victim's VM. Flushing during switching has been proposed in a technique named Düppel in [108]. This defense system includes mitigation for time-shared caches such as L1 and L2, TLB and BTB. In this mechanism, a tenant can construct its VM to introduce additional noise into the timings that the attacker might observe from the cache. This timing information is very important for the attacker because it enables him to infer the victim's sensitive information, and injecting noise makes his job difficult. Düppel modifies the guest OS Kernel and does not need to change hypervisor or cloud providers. Unlike noise producing techniques, Düppel repeatedly cleans the L1 cache along with the execution of tenant workload. But this mitigation, i.e. flushing the local state of cache, has a performance overhead.

The scheduling policy and its implementation can help to mitigate the cache-based side-channel attacks presented in Section 7 by removing or reducing the time period when an attacker and a victim share cache memories or by applying a strict security policy that cleans the microarchitectural states between context switches. The solutions proposed above effectively mitigate the attacks on time shared caches by flushing but both have a cost in terms of performance overhead; and the effect of flushing L1 cache is analyzed in [107]. A 17% increase in latency has been reported when these types of mitigation techniques are applied. Flushing the upper levels of cache in a VM switch is appropriate if it reduces performance degradation. The size of the L1 level cache is relatively small (32 KB in Intel x86 architectures) [48] and the typical expected context switch rate is also low. The normal switching rate of schedulers in XEN to make scheduling decisions is 30 ms [6]. There is thus limited probability of a newly scheduled VM finding any data or instruction in the cache, meaning that the indirect cost of flushing the L1 caches on switching the VMs is insignificant [51]. But for the lower level of caches that are larger, flushing involves significant performance degradation.

Some of the server-side defenses suggest flushing all levels of cache during the context switch of VMs in cloud computing. This is a server-side approach to improve security without inconveniencing the cloud [109]. This research motivated two perspectives; (1) cloud architecture is particularly susceptible to cache-based side-channel attacks and (2) attacks in clouds cannot be solved without interfering in the cloud model. The technique proposed is a server (hypervisor) based solution in an entire cloud system with no interference in the cloud's mode of operation (requires no changes to client or underlying hardware).

8.4. Partitioning time countermeasure techniques

With the introduction to prefetch side-channel [74] and Meltdown [27] attacks, it has been demonstrated that by-passing memory addresses of kernel space is possible which raises the question of more potential attacks on modern computing systems. One of the countermeasures based on time partitioning was proposed to isolate the kernel from the user space and is explained below.

8.4.1. Kernel address space isolation

Prefetch side-channel attacks have been proposed as a new class to exploit potential weaknesses in prefetch instructions [74] that allow unauthorized attackers to obtain addresses to compromise the whole system. Prefetch instructions can fetch unreachable confidential memory in caches in Intel x86. Meltdown attacks [27] also target the memory addresses in kernel address space and one phase of the attack uses the CSCA technique (Flush + Reload) to retrieve information concerning the victim's addresses. As a mitigation technique, some strong kernel address space isolations at OS level are also proposed in [74], to reduce the impact of a prefetch instruction that exploits the victim's secret information. For this reason, distinct kernel threads do not run in the same address space as user threads. This mitigation requires some modifications to OS kernels. This type of mitigation is useful for attacks on time shared caches that follow prefetch instructions. The performance cost of this mitigation technique appears to be 0.06 to 5.09%. Kernel isolation is also proposed in the Meltdown attack named KAISER [30], which completely isolates the kernel from user address space so that no exception can lead to memory addresses in the kernel space.

8.5. Constant-time countermeasure techniques

A well-known approach for mitigating information leakage focuses on cryptographic operations using constant-time techniques. They are mathematically sound but when we implemented them on certain hardware, they tended to leak information in different ways, so some changes are required to these cryptographic algorithms such as: use of fixed time instructions that depend on secret data, there should be no conditional branches that lead to secret data and there should be no memory access patterns that lead to secret data. Considerably difficulty and complexity is involved in changing cryptographic operations for remote attacks and contention-based attacks [19]. Implementations that provide secret-dependent accesses at coarser grain than cache line granularity can leak secret information. In [16] Osvik warns that processors can still leak information on address bits, and proof of this claim is provided in *CacheBleed* [7]. The fact these problems can evolve in Intel processors as described in [99] has been consistently red flagged. If we consider the fact that secret memory accesses should not depend on secret information that might be leaked, then mitigating such leaks is still not sufficient. Many possible leaks have been demonstrated such as instructions that are data dependent, timing of execution and memory dependent data of cryptographic operations [125]. Many tools and frameworks have been developed to provide mitigation by constant-time techniques [134,135]. The authors in [135] presented a formal framework to design a constant-time code able to detect the flow of secret information. [134], described an upper bound of information that can leak from the implementation of a cryptographic algorithm. Some approaches like CacheAudit [132] and FlowTracker [133] have helped provide security at better level of abstraction and modified existing compilers to keep track of the flow of information for detection of channels. Another tool, Catalyzr [136] functions based on finding the relation between vulnerabilities and exploitations. It propose a method to characterize the leakage induced by non-constant-time constructs of the source code. Catalyzr recovers known attacks and suggests possible unknown attacks.

Constant-Time Countermeasure Techniques are really efficient in mitigating side-channel attacks. However, Section 7 has shown that being able to propose a leakage-free implementation is very challenging. Indeed, the main disadvantage of constant-time implementations is that they work on one hardware deployment constantly but not on other hardware platforms. For example, [114] is a constant-time mitigation of Lucky 13 attack [157], but it is not applicable on ARM platforms (AM3358).

Similarly, some attacks do not work on different processors such as CacheBleed [7] which works only on Sandy Bridge processors but cannot work on other processors that do not include multi-threading and Flush + Reload [5], which does not work on ARM architectures because ARM processors do not have inclusive caches. The same goes for constant-time techniques, that are specific to a certain hardware platforms, and we thus need to develop different parameters of constant-time implementations for different hardware platforms.

8.6. Detection techniques

SCA detection techniques can be divided into two main categories: signature-based detection and anomaly-based detection. [149] presents a method to check the robustness of NIST post-quantum standardization for timing-based cache attacks. The proposed vulnerability tool explains that 80% of the analyzed implementations contain at least one significant flaw. It provides a comprehensive study to identify flaws in present implementations and how to fix them. Detection approaches propose a first line of defense to counter an attack as soon as it is detected. There are detection mechanisms that inherit the properties of both anomaly and signature-based detection mechanisms such as [126,148]. The majority of CSCA detection approaches use signature-based techniques [137–140,143–145,150,151,158–161] or a combination of signature and anomaly-based detection techniques [126,147,148].

8.6.1. Signature-based detection techniques

One of the CSCA detection techniques able to detect malwares/CSCAs based on their signatures is presented in [143]. The authors in [143] used HPCs to detect malwares and CSCAs. Another signature-based detection technique in [144] inspects Prime + Probe and Flush + Reload CSCAs while using machine learning (ML) models and HPCs and running the AES cryptosystem. [150] debates the use of ML in security and explains selection metrics for ML models to perform run-time detection in real-time. Three recent research works [137,145,151] perform signature-based detection. This research targets a larger set of attacks including Flush + Reload, Flush + Flush and Prime + Probe running on RSA and AES cryptosystems. The authors in [137] propose a run-time detection mechanism called the NIGHTs-WATCH. NIGHTs-WATCH considers HPCs as input features under

attack/no-attack scenarios. NIGHTs-WATCH claims to have low performance overhead thanks to embedding detection inside the cryptosystem. In a similar work, the authors in [145] use both linear and non-linear ML classifiers to detect variants of Prime + Probe attack running under AES cryptosystems. [151] uses machine learning and HPCs to perform run-time detection of Flush + Reload and Flush + Flush attack and their variants on RSA and AES cryptosystems. [151] claims the proposed detection mechanism works with high detection accuracy under realistic system load conditions. A novel approach named *SCADET* [138] is a signature-based detection tool that detects Prime + Probe attacks using high-level semantics and invariant patterns of attacks. One of the *HexPADS* [139] uses the values of cache miss rates and page faults to detect an attack. *HexPADS* is able to detect CSCAs (Flush + Reload), cache template attacks [15] and CAIN (C5) based on Prime + Probe. A detection approach based on threshold determination, named *Deja-Vu*, was introduced to detect attacks on programs guarded by SGX [162].

8.6.2. Anomaly-based detection techniques

Some novel detection techniques rely on anomaly-based detection of CSCAs [141,142,146]. The detection mechanism in [146] is based on *Intel Cache Monitoring Technology (CMT)* and HPCs using Gaussian Anomaly detection for inspecting CSCAs at the level of VMs in IaaS Cloud platforms. *CacheShield* [141] is an Anomaly-based detection mechanism for CSCAs on legacy software (victim applications) which involves monitoring of HPCs while using an unsupervised anomaly detection algorithm. Another anomaly-based detection mechanism, *SpyDetector* [142], is proposed to detect CSCAs using HPCs and has been validated on Flush + Reload, Flush + Flush and Prime + Probe attacks running on RSA, AES and ECDSA cryptosystems.

8.6.3. Signature + Anomaly-based detection techniques

Some detection techniques use a combination of signature and anomaly-based detection techniques [126,147,148]. [147] proposes a machine learning based detection mechanism for Flush + Reload attack on AES and ECDSA cryptosystem. Another technique *CloudRadar* [126], correlates cryptographic execution of applications on virtual machines and the anomalous behavior of caches to detect CSCAs in cloud systems. A three-step detection mechanism on cache and branch predictor based CSCAs is proposed in [148]. This method includes HPCs and machine learning models and comprises three stages: detecting the anomaly, finding the class of anomaly and correlating the malicious process with the victim.

9. Trends, challenges, and future directions

This section summarizes some of the trends in modern attack approaches and their defenses both at software and hardware level.

9.1. Future directions in attacks

The extensive literature review we conducted revealed that attacks have been practically demonstrated across the entire computation stack exploiting resource sharing. Cache-based attacks have targeted all levels of the cache hierarchy. For instance, attacks on L1 and L2 caches of platforms on which hardware threading is enabled are presented in [7,59,61,62,86]. Attacks targeting last level cache are presented in [5,25,51,54,93]. In this survey, we provide a (non-exhaustive) list of important cache-based attacks and their classification (Table 3) along with their time line, threat level, leakage metric, and the implementation platforms they target. There are many attacks that do not exploit multi-threading, yet they are successful in attacking L1 cache simply by generating cache-contentions between various processes and interactive VMs such as [6,7,61,86]. The trend in these attacks is to exploit system-wide shared resources because defenses against these attacks incur heavy performance penalties. Primarily, the proposed solutions are based on cache partitioning such as RPcache, random fill cache, minimum time slicing techniques, and cache coloring [107,114–116,118]. Other than caches, attacks have also been demonstrated on cores. These attacks are based on cores shared for computation, which is relatively easy to mitigate by disallowing shared computation between multiple processes. While caches reveal a lot of sensitive information about the content and order of execution, they can still be kept isolated even if the cores are shared. Another vulnerability already highlighted in earlier research [5] is the privilege level of certain instructions, such as *clflush* in Intel x86 architecture. Intel x86 architecture supports non-selective flushing of L1 cache, which is already being successfully exploited by attackers. More recent cache-based attacks have moved to the last level cache (LLC), which is shared by multiple processes. LLC attacks have proven to be very effective as have high resolution cross-core attacks, as described in [5,14,15,54,65,90,93]. There are high resolution attacks that do not even need memory sharing to attack a system as described in [25,91,163].

Moving to the system level, as a future direction, side-channel attacks on the interconnect network are possible, although we have not yet come across any attack on the interconnect network, but SCAs through snooping could be an interesting future direction. Future trends in attacks will move towards stealthier approaches as defenses are getting stronger and detection mechanisms using hardware performance monitoring have made early-stage attack detection possible.

9.2. Future directions in mitigation techniques

A general and somewhat obvious trend in mitigation techniques over the last decade is preparing defenses against known attacks. However, this trend is now shifting towards more *Secure-by-Design* approaches in both hardware and software. Hardware, architecture platforms such as Intel's SGX [164] and HARP [165] or ARM's TrustZone [166], are serious attempts in this direction. Software-based countermeasure approaches are also moving towards secure-by-design systems. For instance, operating-system-based countermeasure techniques that use run-time monitoring of system performance using PMUs is one such approach that is being put into practice. These approaches offer detection as well as mitigation of attacks on-the-fly, which helps reduce performance overheads. OS-based countermeasure techniques obfuscate the execution order of processes to prevent leakage of useful timing and/or access information at cache level.

Some recent research [13] argues strongly in favor of resource isolation alone as a defense against SCAs. Such countermeasures, although very effective from the point of view of security, are simply not viable for certain application domains, such as cloud computing, where security represents a trade-off with the fundamental economic model, which in this case, is based on resource sharing. Existing solutions based on resource isolation suggest physically and temporally isolating process execution. Techniques that provide physical isolation with the help of software implementations include cache coloring [114–116, 118], STEALTHMEM [20], Migration of VMs [127], Quasi Partitioning, CacheBar [117], Auditing/Detection [6,14,108,147] and CloudRadar [126]. There is still room to analyze these approaches in terms of complete protection of cache hierarchy and performance penalty. On the other hand, techniques that provide temporal isolation at software level include Minimum time-slicing

[107], Cache Flushing, Düppel [108], Kernel Address Space Isolation [30,74]. However, all these techniques have proven to be expensive as they involve significant performance overheads, particularly for application domains where resource sharing is vital for financial and performance benefits. Another issue with these techniques is that they do not protect the entire computation stack. For instance, cache coloring and STEALTHMEM provide protection mechanisms against LLC but creating colored pages for relatively smaller L1, which is virtually indexed, is not supported. LLC is physically indexed and larger in size, which is why these solutions worked for other LLC. Therefore, future mitigation techniques must take a holistic approach and provide solutions that are not necessarily based entirely on resource isolation.

A very important aspect in mitigation against attacks that requires the research community's focus is detection. Detection based solutions [137,145,147,150] determine whether a system is in safe mode or under attack by monitoring system features such as timing, access behavior, and traces of execution of instructions with the help of hardware performance monitoring. Detection techniques are a good first line of defense to mitigate side-channel attacks. Indeed, when an attack is known, an efficient early detection mechanism can be provided. Moreover, these approaches are a solid base on which to build detection mechanisms that enable detection of new attacks or variants of known attacks by monitoring microarchitectural behaviors at run-time. In this context, the capability of a detection system is closely linked to the sensors available to perform the monitoring task. An *all-weather* protection against attacks is computationally expensive and implies a performance cost. However, if the mitigation is coupled with high-resolution detection, it can be activated whenever an attack is detected, thereby reducing performance cost. A word of caution for detection-based protection is needed as these solutions may generate false positives if detection is imprecise and low resolution while the system is under attack. In this survey, we argue in favor of *need-based* protection for future mitigation to maintain the performance benefits of highly optimized architecture platforms.

10. Conclusion

This paper provides an overview of cache-based side-channel attacks, along with the microarchitectural details, auditing and mitigation techniques that have been proposed over the last decade (2007–2018). Our particular focus has been on the identification of vulnerabilities in different RSA cryptographic implementations, which leak information when deployed on underlying hardware platforms. The paper provides a threat model based on the features in caches that are being leveraged for such attacks across cache hierarchy in underlying platforms such as Intel x86. It also classifies these attacks based on the source of information leakage. The main focus of this paper has been on the qualitative analysis of existing attacks in terms of secret key retrieval efficiency, complexity, and the features being exploited on target cryptosystems. We also conducted an extensive review of the mitigation and auditing techniques currently proposed against such attacks in a similar fashion and classified them based on their effectiveness at various levels in the cache hierarchy and leveraged features.

Finally, we looked at future research trends, challenges, and directions for cache-based side-channel attacks as well as for mitigation techniques. We argue in favor of a holistic approach to counter SCAs through a secure-by-design approach to hardware and need-based protection approach to software. We conclude that future trends in SCAs are moving towards stealthier approaches as defenses are getting stronger. What is more, in the future, mitigation based on resource isolation will not be viable from an economic and performance point of view, as resource sharing is tending to increase in modern computing infrastructure to sustain performance benefits. We also underline the importance of high-resolution detection techniques using hardware performance monitoring to detect sophisticated and stealthier attacks in future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the Pak-France joint research project e-health. SECURE under PHC PERIDOT Program (ID: 3-6/HEC/R&D/PERIDOT/2017) and the National Center for Cyber Security (NCCS), Pakistan.

References

[1] B.P. Rimal, E. Choi, I. Lumb, A taxonomy and survey of cloud computing systems, in: 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 44–51, http://dx.doi.org/10.1109/NCM.2009.218.

[2] T. Ristenpart, E. Tromer, H. Shacham, S. 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, in: CCS '09, ACM, New York, NY, USA, 2009, pp. 199–212, [Online]. Available http://doi.acm.org/10.1145/1653662.1653687.

[3] Z. Wu, Z. Xu, H. Wang, Whispers in the hyper-space: High-speed covert-channel attacks in the cloud, in: Proceedings of the 21st USENIX Conference on Security Symposium, in: Security'12, USENIX Association, Berkeley, CA, USA, 2012, p. 9, [Online]. Available http://dl.acm.org/citation.cfm?id=2362793.2362802.

[4] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, R. Schlichting, An exploration of l2 Cache covert-channels in virtualized environments, in: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, in: CCSW '11, ACM, New York, NY, USA, 2011, pp. 29–40, [Online]. Available http://doi.acm.org/10.1145/2046660.2046670.

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

[6] Y. Zhang, A. Juels, M.K. Reiter, T. Ristenpart, Cross-VM side channels and their use to extract private keys, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, in: CCS '12, ACM, New York, NY, USA, 2012, pp. 305–316, [Online]. Available http://doi.acm.org/10.1145/2382196.2382230.

[7] Y. Yarom, D. Genkin, N. Heninger, Cachebleed: A timing attack on openssl constant time RSA, in: B. Gierlichs, A.Y. Poschmann (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 346–367, [Online]. Available http://dx.doi.org/10.1007/978-3-662-53140-2_17.

[8] P.C. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, in: CRYPTO '99, Springer-Verlag, London, UK, UK, 1999, pp. 388–397, [Online]. Available http://dl.acm.org/citation.cfm?id=646764.703989.

[9] J.-J. Quisquater, D. Samyde, Electromagnetic analysis (EMA): Measures and counter-measures for smart cards, in: I. Attali, T. Jensen (Eds.), Smart Card Programming and Security: International Conference on Research in Smart Cards, E-Smart 2001 Cannes, France, September 19–21, 2001 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 200–210.

[10] D. Genkin, A. Shamir, E. Tromer, Rsa key extraction via low-bandwidth acoustic cryptanalysis, in: J.A. Garay, R. Gennaro (Eds.), Advances in Cryptology – CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 444–461.

[11] D. Gullasch, E. Bangerter, S. Krenn, Cache games – bringing access-based cache attacks on AES to practice, in: Proceedings of the 2011 IEEE Symposium on Security and Privacy, in: SP '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 490–505, http://dx.doi.org/10.1109/SP.2011.22.

[12] E. Tromer, D.A. Osvik, A. Shamir, Efficient cache attacks on AES, and countermeasures, J. Cryptol. 23 (1) (2010) 37–71, http://dx.doi.org/10.1007/s00145-009-9049-y.

[13] Q. Ge, Y. Yarom, D. Cock, G. Heiser, A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, J. Cryptogr. Eng. (2016) 1–27, http://dx.doi.org/10.1007/s13389-016-0141-6.

[14] D. Gruss, C. Maurice, K. Wagner, S. Mangard, Flush + flush: A fast and stealthy cache attack, in: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721, in: DIMVA 2016, Springer-Verlag New York, Inc., NY, USA, 2016, pp. 279–299.

[15] D. Gruss, R. Spreitzer, S. Mangard, Cache template attacks: Automating attacks on inclusive last-level caches, in: Proceedings of the 24th USENIX Conference on Security Symposium, in: SEC'15, Berkeley, CA, USA, 2015, pp. 897–912, [Online]. Available http://dl.acm.org/citation.cfm?id=2831143.2831200.

[16] D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: The case of AES, in: D. Pointcheval (Ed.), Topics in Cryptology – CT-RSA 2006: The Cryptographers' Track At the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 1–20, http://dx.doi.org/10.1007/11605805_1.

[17] Y. Zhang, A. Juels, M.K. Reiter, T. Ristenpart, Cross-tenant side-channel attacks in paas clouds, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, in: CCS '14, ACM, New York, NY, USA, 2014, pp. 990–1003, [Online]. Available http://doi.acm.org/10.1145/2660267.2660356.

[18] G. Irazoqui, M.S. Inci, T. Eisenbarth, B. Sunar, Wait a minute! a fast, cross-VM attack on AES, in: A. Stavrou, H. Bos, G. Portokalidis (Eds.), Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings, Springer International Publishing, Cham, 2014, pp. 299–319, [Online]. Available http://dx.doi.org/10.1007/978-3-319-11379-1_15.

[19] D.J. Bernstein, Cache-timing attacks on AES, Technical Report, 2005.

[20] T. Kim, M. Peinado, G. Mainar-Ruiz, Stealthmem: System-level protection against cache-based side channel attacks in the cloud, in: Proceedings of the 21st USENIX Conference on Security Symposium, in: Security'12, USENIX Association, Berkeley, CA, USA, 2012, p. 11, [Online]. Available http://dl.acm.org/citation.cfm?id=2362793.2362804.

[21] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, D. Ponomarev, Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks, ACM Trans. Archit. Code Optim. 8 (4) (2012) 35:1–35:21, [Online]. Available http://doi.acm.org/10.1145/2086696.2086714.

[22] D. Page, Partitioned cache architecture as a side-channel defence mechanism, 2005, Cryptology ePrint Archive, Report 2005/280, [Online]. Available https://eprint.iacr.org/2005/280.

[23] Z. Wang, R.B. Lee, New cache designs for thwarting software cache-based side channel attacks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, in: ISCA '07, ACM, New York, NY, USA, 2007, pp. 494–505, [Online]. Available http://doi.acm.org/10.1145/1250662.1250723.

[24] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, R.B. Lee, CATalyst: Defeating last-level cache side channel attacks in cloud computing, in: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 406–418, http://dx.doi.org/10.1109/HPCA.2016.7446082.

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

[26] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, 2018, CoRR, abs/1801.01203, arXiv:1801.01203.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, 2018, pp. 973–990, [Online]. Available https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[28] D. Evtyushkin, D. 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, in: CCS '16, ACM, New York, NY, USA, 2016, pp. 843–857, [Online]. Available http://doi.acm.org/10.1145/2976749.2978374.

[29] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C.A. Boano, S. Mangard, K. Römer, Hello from the other side: SSH over robust cache covert-channels in the cloud, NDSS, CA, US, San Diego, 2017.

[30] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, S. Mangard, KASLR is dead: Long live KASLR, in: International Symposium on Engineering Secure Software and Systems, Springer, 2017, pp. 161–176.

[31] K.N. Khasawneh, E.M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, N. Abu-Ghazaleh, SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation, 2018, arXiv preprint arXiv:1806.05179.

[32] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, A. Roychoudhury, Oo7: Low-overhead defense against spectre attacks via program analysis, 2018, arXiv preprint arXiv:1807.05843.

[33] K. Krüger, M. Volp, G. Fohler, Vulnerability analysis and mitigation of directed timing inference based attacks on time-triggered systems, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 106, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, p. 22.

[34] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, J. Torrellas, Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum), in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2018, pp. 428–441.

[35] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, J. Emer, DAWG: A defense against cache timing attacks in speculative execution processors, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2018, pp. 974–987.

[36] Retpoline: A Branch Target Injection Mitigation, [Online]. Available: https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org.

[37] Site Isolation, [Online]. Available: https://www.chromium.org/Home/chromium-security/site-isolation.

[38] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, C. Fetzer, You shall not bypass: Employing data dependencies to prevent bounds check bypass, 2018, arXiv preprint arXiv:1805.08506.

[39] Q. Ge, Y. Yarom, D. Cock, G. Heiser, A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, IACR Cryptol. ePr. Arch. 2016 (2016) 613.

[40] S. Anwar, Z. Inayat, M.F. Zolkipli, J.M. Zain, A. Gani, N.B. Anuar, M.K. Khan, V. Chang, Cross-VM cache-based side channel attacks and proposed prevention mechanisms: A survey, J. Netw. Comput. Appl. 93 (Supplement C) (2017) 259–279.

[41] Y. Lyu, P. Mishra, A survey of side-channel attacks on Caches and countermeasures, J. Hardw. Syst. Secur. 2 (1) (2018) 33–50.

[42] R. Spreitzer, V. Moonsamy, T. Korak, S. Mangard, Systematic classification of side-channel attacks: A case study for mobile devices, IEEE Commun. Surv. Tutor. 20 (1) (2018) 465–488.

[43] E. Oswald, B. Preneel, A survey on passive side-channel attacks and their countermeasures for the nessie public-key cryptosystems, NESSIE public reports (2003) [Online]. Available https://www.cosic.esat.kuleuven.ac.be/nessie/reports.

[44] H. Mantel, A. Weber, B. Köpf, A systematic study of cache side channels across aes implementations, in: International Symposium on Engineering Secure Software and Systems, Springer, 2017, pp. 213–230.

[45] V. Costan, S. Devadas, SGX explained, IACR Cryptol. ePr. Arch. 2016 (2016) 86.

[46] D. Craig, K. David, P. Leo, T. Dean, Prime + abort: A timer-free high-precision l3 cache attack using intel TSX, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX, Vancouver, BC, 2017, pp. 51–67.

[47] D. Levinthal, Performance analysis guide for intel core i7 processor and intel xeon 5500 processors, Intel. Perform. Anal. 30 (2009) 18.

[48] P. Guide, Intel® 64 and IA-32 architectures software developer's manual, in: Volume 3B: System programming Guide, Part, Vol. 2, 2011.

[49] Intel, Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation, 2014.

[50] Y. Yarom, N. Benger, Recovering openSSL ECDSA nonces using the flUSH+RELOAD cache side-channel attack, IACR Cryptol. EPr. Arch. 2014 (2014) 140.

[51] G. Qian, Y. Yuval, L. Frank, H. Gernot, Contemporary processors are leaky - and there's nothing you can do about it, 2016, pp. 29–35, [Online]. Available arXiv:1612.04474.

[52] M.S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, B. Sunar, Cache Attacks Enable Bulk Key Recovery on the Cloud, Vol. 9813, CHES, Santa Barbara, CA, USA, 2016, pp. 368–388.

[53] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, A. Francillon, Reverse engineering last-level cache complex addressing using performance counters, in: Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404, in: RAID 2015, Springer-Verlag New York, Inc., New York, NY, USA, 2015, pp. 48–65, [Online]. Available http://dx.doi.org/10.1007/978-3-319-26362-5_3.

[54] Y. Yarom, Q. Ge, F. Liu, R.B. Lee, G. Heiser, Mapping the last-level cache, 2015, [Online]. Available https://Eprint.Iacr.Org.

[55] J. Zferer, Principles of Secure Processor Architecture Design, in: Synthesis Lectures on Computer Architecture, vol. 9048, Morgan & Claypool Publishers, 2018, pp. 1–175, http://dx.doi.org/10.2200/S00864ED1V01Y201807CAC045.

[56] D. Evtyushkin, D. Ponomarev, N. Abu-Ghazaleh, Jump over ASLR: Attacking branch predictors to bypass ASLR, in: Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on, IEEE, 2016, pp. 1–13.

[57] R.B. Lee, Security basics for computer architects, in: Synthesis Lectures on Computer Architecture, vol. 8, Morgan & Claypool Publishers, 2013, pp. 1–111, http://dx.doi.org/10.2200/s00512ed1v01y201305cac025.

[58] O. Aciicmez, J.-P. Seifert, Cheap hardware parallelism implies cheap security, in: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography, in: FDTC '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 80–91, [Online]. Available http://dx.doi.org/10.1109/FDTC.2007.4.

[59] C. Percival, Cache missing for fun and profit, in: Proc. of BSDCan, 2005.

[60] B.B. Brumley, R.M. Hakala, Cache-timing template attacks, in: Advances in Cryptology – ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 667–684.

[61] O. Aciicmez, Yet another microarchitectural attack:: Exploiting i-Cache, in: Proceedings of the 2007 ACM Workshop on Computer Security Architecture, in: CSAW '07, ACM, New York, NY, USA, 2007, pp. 11–18, [Online]. Available http://doi.acm.org/10.1145/1314466.1314469.

[62] O. Aciicmez, B.B. Brumley, P. Grabher, New results on instruction Cache attacks, in: Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems, in: CHES'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 110–124, [Online]. Available http://dl.acm.org/citation.cfm?id=1881511.1881522.

[63] M.S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, B. Sunar, Seriously, get off my cloud! cross-VM RSA key recovery in a public cloud, 2015, Cryptology ePrint Archive, Report 2015/898 http://eprint.iacr.org/2015/898.

[64] G. Irazoqui, T. Eisenbarth, B. Sunar, S$A: A shared cache attack that works across cores and defies vm sandboxing – and its application to AES, in: Proceedings of the 2015 IEEE Symposium on Security and Privacy, in: SP '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 591–604, [Online]. Available http://dx.doi.org/10.1109/SP.2015.42.

[65] J. van de Pol, N. Smart, Y. Yarom, Just a little bit more, in: Topics in Cryptology - CT-RSA 2015, in: Lecture Notes in Computer Science, vol. 9048, Springer International Publishing, 2015, pp. 3–21.

[66] O. Acmez, S. Gueron, J.-P. Seifert, New branch prediction vulnerabilities in openSSL and necessary software countermeasures, in: In 11th IMA International Conference on Cryptography and Coding, 2007, pp. 185–203.

[67] O. Aciicmez, C.K. Koc, J.-P. Seifert, Predicting secret keys via branch prediction, in: Topics in Cryptology – CT-RSA 2007: The Cryptographers, Track at the RSA Conference 2007, Proceedings, San Francisco, CA, USA, 2007, pp. 225–242.

[68] P. Pessl, D. Gruss, C. Maurice, S. Mangard, Reverse engineering DRAM addressing and exploitation, 2015, Published in ArXiv.

[69] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, in: 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[70] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium (USENIX Security 18), 2018.

[71] J.V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T.F. Wenisch, Y. Yarom, R. Strackx, Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution, in: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018., 2018, pp. 991–1008, [Online]. Available https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[72] C. Canella, J.V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, D. Gruss, A systematic evaluation of transient execution attacks and defenses, 2019, arXiv preprint arXiv:1811.05441.

[73] C. Pereida García, B.B. Brumley, Y. Yarom, Make sure DSA signing exponentiations really are constant-time, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, in: CCS '16, ACM, New York, NY, USA, 2016, pp. 1639–1650, [Online]. Available http://doi.acm.org/10.1145/2976749.2978420.

[74] D. Gruss, C. Maurice, A. Fogh, M. Lipp, S. Mangard, Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, in: CCS '16, ACM, NY, USA, 2016, pp. 368–379, [Online]. Available http://doi.acm.org/10.1145/2976749.2978356.

[75] M. Schaefer, B. Gold, R. Linde, J. Scheid, Program confinement in KVM/370, in: Proceedings of the 1977 Annual Conference, in: ACM '77, ACM, New York, NY, USA, 1977, pp. 404–410, [Online]. Available http://doi.acm.org/10.1145/800179.1124633.

[76] W.-M. Hu, Reducing timing channels with fuzzy time, J. Comput. Secur. 1 (3–4) (1992) 233–254, [Online]. Available http://dl.acm.org/citation.cfm?id=2699806.2699810.

[77] Y. Tsunoo, E. Tsujihara, K. Minematsu, H. Miyauchi, Cryptanalysis of block ciphers implemented on computers with cache, 2002.

[78] J.C. Wray, An analysis of covert timing channels, J. Comput. Secur. 1 (3–4) (1992) 219–232, [Online]. Available http://dl.acm.org/citation.cfm?id=2699806.2699809.

[79] D. Page, Theoretical use of cache memory as a cryptanalytic side-channel, 2002, Technical Report, available in IACR Cryptology ePrint Archives, https://eprint.iacr.org/2002/169.pdf.

[80] O. Aciicmez, W. Schindler, C.K. Koc, Cache based remote timing attack on the AES, in: Proceedings of the 7th Cryptographers' Track At the RSA Conference on Topics in Cryptology, in: CT-RSA'07, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 271–286, [Online]. Available http://dx.doi.org/10.1007/11967668_18.

[81] T. Tsunoo, T. Yukiyasuand Saito, T. Suzaki, M. Shigeri, H. Miyauchi, Cryptanalysis of DES implemented on computers with Cache, in: C.D. Walter, Ç.K. Koç, C. Paar (Eds.), Cryptographic Hardware and Embedded Systems - CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 62–76, [Online]. Available http://dx.doi.org/10.1007/978-3-540-45238-6_6.

[82] J. Bonneau, I. Mironov, CaChe-collision timing attacks against AES, in: Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems, in: CHES'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 201–215, [Online]. Available http://dx.doi.org/10.1007/11894063_16.

[83] O. Aciicmez, C.K. Koc, Trace-driven Cache attacks on AES (short paper), in: Proceedings of the 8th International Conference on Information and Communications Security, in: ICICS'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 112–121, [Online]. Available http://dx.doi.org/10.1007/11935308_9.

[84] J.-F. Gallais, I. Kizhvatov, M. Tunstall, Improved trace-driven Cache-collision attacks against embedded aes implementations, in: Proceedings of the 11th International Conference on Information Security Applications, in: WISA'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 243–257, [Online]. Available http://dl.acm.org/citation.cfm?id=1949945.1949967.

[85] M. Neve, J.-P. Seifert, Advances on access-driven Cache attacks on AES, in: E. Biham, A.M. Youssef (Eds.), Selected Areas in Cryptography: 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 147–162, [Online]. Available http://dx.doi.org/10.1007/978-3-540-74462-7_11.

[86] O. Aciicmez, W. Schindler, A vulnerability in RSA implementations due to instruction Cache analysis and its demonstration on openssl, in: Proceedings of the 2008 the Cryptopgraphers' Track At the RSA Conference on Topics in Cryptology, in: CT-RSA'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 256–273, [Online]. Available http://dl.acm.org/citation.cfm?id=1791688.1791711.

[87] B.C. Vattikonda, S. Das, H. Shacham, Eliminating fine grained timers in xen, in: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, in: CCSW '11, ACM, New York, NY, USA, 2011, pp. 41–46, [Online]. Available http://doi.acm.org/10.1145/2046660.2046671.

[88] R. Hund, C. Willems, T. Holz, Practical timing side channel attacks against kernel space ASLR, in: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 191–205.

[89] T. Allan, B.B. Brumley, K. Falkner, J. van de Pol, Y. Yarom, Amplifying side channels through performance degradation, in: Proceedings of the 32Nd Annual Conference on Computer Security Applications, in: ACSAC '16, ACM, New York, NY, USA, 2016, pp. 422–435, [Online]. Available http://doi.acm.org/10.1145/2991079.2991084.

[90] N. Benger, J. Pol, N.P. Smart, Y. Yarom, Ooh aah... just a little bit: A small amount of side channel Can go a long way, in: Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731, Springer-Verlag New York, Inc., New York, NY, USA, 2014, pp. 75–92, http://dx.doi.org/10.1007/978-3-662-44709-3_5.

[91] L. Moritz, G. Daniel, S. Raphael, M. Clementine, M. Stefan, ARMageddon: Cache Attacks on Mobile Devices, in: Proceedings of the 25th USENIX Security Symposium, Austin, TX, 2016.

[92] A. Rényi, et al., On measures of entropy and information, in: Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics, The Regents of the University of California, 1961.

[93] D.J. Bernstein, J. Breitner, D. Genkin, L. Groot Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, Y. Yarom, Sliding right into disaster: Left-to-right sliding windows leak, in: W. Fischer, N. Homma (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2017, Springer International Publishing, Cham, 2017, pp. 555–576.

[94] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Commun. ACM 21 (2) (1978) 120–126, [Online]. Available http://doi.acm.org/10.1145/359340.359342.

[95] N. Heninger, H. Shacham, Reconstructing RSA private keys from random key bits, in: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology, in: CRYPTO '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 1–17, [Online]. Available http://dx.doi.org/10.1007/978-3-642-03356-8_1.

[96] D.M. Gordon, A survey of fast exponentiation methods, J. Algorithms 27 (1) (1998) 129–146, [Online]. Available http://dx.doi.org/10.1006/jagm.1997.0913.

[97] E. Brickell, G. Graunke, M. Neve, J.-P. Seifert, Software mitigations to hedge AES against cache-based software side channel vulnerabilities, IACR Cryptol. EPr. Arch. 2006 (2006) 52.

[98] E. Brickell, G. Graunke, J.-P. Seifert, Mitigating Cache/timing based side-channels in AES and RSA software implementations, in: RSA Conference 2006 Session DEV-203, 2006.

[99] D.J. Bernstein, P. Schwabe, A word of warning, in: Conference on Cryptographic Hardware and Embedded Systems, in: CHES'13 (Rump Session), Santa Barbara, USA, 2013, pp. 18–22.

[100] D.B. Alpert, M.R. Choudhury, J.D. Mills, Interleaved Cache for Multiple Accesses per Clock Cycle in a Microprocessor, Google Patents, 1996, US Patent ID: 5, 559, 986.

[101] A. Fog, The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, Copenhagen University College of Engineering, 2012, pp. 02–29.

[102] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, V. Bertocci, Security best practices for developing windows azure applications, Microsoft Corp. (2010) 1.

[103] F. Liu, H. Wu, K. Mai, R.B. Lee, Newcache: Secure cache architecture thwarting cache side-channel attacks, IEEE Micro. 36 (5) (2016) 8–16.

[104] Y. Tan, J. Wei, W. Guo, The micro-architectural support countermeasures against the branch prediction analysis attack, in: TrustCom, 2014 IEEE 13th International Conference on, IEEE, 2014, pp. 276–283.

[105] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, H. Shacham, On subnormal floating point and abnormal timing, in: Proceedings of the 2015 IEEE Symposium on Security and Privacy, in: SP '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 623–639, [Online]. Available http://dx.doi.org/10.1109/SP.2015.44.

[106] A. Rane, C. Lin, M. Tiwari, Secure, precise, and fast floating-point operations on x86 processors, in: 25th USENIX Security Symposium (USENIX Security 16), USENIX Association, Austin, TX, 2016, pp. 71–86, [Online]. Available https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/rane.

[107] V. Varadarajan, T. Ristenpart, M. Swift, Scheduler-based defenses against cross-VM side-channels, in: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA, 2014, pp. 687–702, [Online]. Available https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/varadarajan.

[108] Y. Zhang, M.K. Reiter, DüPpel: Retrofitting commodity operating systems to mitigate Cache side channels in the cloud, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, in: CCS '13, ACM, New York, NY, USA, 2013, pp. 827–838, http://dx.doi.org/10.1145/2508859.2516741, [Online]. Available http://doi.acm.org/10.1145/2508859.2516741.

[109] M. Godfrey, M. Zulkernine, A server-side solution to Cache-based side-channel attacks in the cloud, in: Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing, in: CLOUD '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 163–170, [Online]. Available http://dx.doi.org/10.1109/CLOUD.2013.21.

[110] Z. Wang, R.B. Lee, New Cache designs for thwarting software Cache-based side channel attacks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, in: ISCA '07, NY, USA, 2007, pp. 494–505, [Online]. Available http://doi.acm.org/10.1145/1250662.1250723.

[111] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, A. Wolman, Protecting data on smartphones and tablets from memory attacks, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS '15, ACM, New York, NY, USA, 2015, pp. 177–189, [Online]. Available http://doi.acm.org/10.1145/2694344.2694380.

[112] J. Kong, O. Aciicmez, J.P. Seifert, H. Zhou, Hardware-software integrated approaches to defend against software cache-based side channel attacks, in: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, 2009, pp. 393–404.

[113] D. Stefan, P. Buiras, E.Z. Yang, A. Levy, D. Terei, A. Russo, D. Mazières, Eliminating cache-based timing attacks with instruction-based scheduling, in: European Symposium on Research in Computer Security, Springer, 2013, pp. 718–735.

[114] D. Cock, Q. Ge, T. Murray, G. Heiser, The last mile: An empirical study of timing channels on seL4, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, in: CCS '14, ACM, New York, NY, USA, 2014, pp. 570–581, [Online]. Available http://doi.acm.org/10.1145/2660267.2660294.

[115] M.. Godfrey, M. Zulkernine, Preventing cache-based side-channel attacks in a cloud environment, IEEE Trans. Cloud Comput. 2 (4) (2014) 395–408.

[116] J. Shi, X. Song, H. Chen, B. Zang, Limiting Cache-based side-channel in multi-tenant cloud using dynamic page coloring, in: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops, in: DSNW '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 194–199, [Online]. Available http://dx.doi.org/10.1109/DSNW.2011.5958812.

[117] Z. Zhou, M.K. Reiter, Y. 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, in: CCS '16, ACM, New York, NY, USA, 2016, pp. 871–882, [Online]. Available http://doi.acm.org/10.1145/2976749.2978324.

[118] B.N. Bershad, D. Lee, T.H. Romer, J.B. Chen, Avoiding conflict misses dynamically in large direct-mapped caches, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS VI, ACM, New York, NY, USA, 1994, pp. 158–170, [Online]. Available http://doi.acm.org/10.1145/195473.195527.

[119] D.J. Bernstein, T. Lange, P. Schwabe, The security impact of a new cryptographic library, in: International Conference on Cryptology and Information Security in Latin America, Springer, 2012, pp. 159–176.

[120] F. Liu, L. Ren, H. Bai, Mitigating cross-VM side channel attack on multiple tenants cloud platform, J. Comput. 9 (2014).

[121] F. Biondi, M. Chadli, T. Given-Wilson, A. Legay, Information leakage as a scheduling resource, in: Critical Systems: Formal Methods and Automated Verification, Springer, 2017, pp. 83–99.

[122] J. Kong, O. Aciicmez, J.-P. Seifert, H. Zhou, Deconstructing new cache designs for thwarting software cache-based side channel attacks, in: Proceedings of the 2Nd ACM Workshop on Computer Security Architectures, in: CSAW '08, ACM, New York, NY, USA, 2008, pp. 25–34, [Online]. Available http://doi.acm.org/10.1145/1456508.1456514.

[123] Z. Wang, R.B. Lee, Covert and side channels due to processor architecture, in: 22Nd Annual Computer Security Applications Conference, in: ACSAC'06, IEEE, USA, 2006, pp. 473–482.

[124] T. Müller, A. Dewald, F.C. Freiling, AESSE: A cold-boot resistant implementation of AES, in: Proceedings of the Third European Workshop on System Security, in: EUROSEC '10, ACM, New York, NY, USA, 2010, pp. 42–47, [Online]. Available http://doi.acm.org/10.1145/1752046.1752053.

[125] B. Coppens, I. Verbauwhede, K.D. Bosschere, B.D. Sutter, Practical mitigations for timing-based side-channel attacks on modern x86 processors, in: 2009 30th IEEE Symposium on Security and Privacy, 2009, pp. 45–60, http://dx.doi.org/10.1109/SP.2009.19.

[126] T. Zhang, Y. Zhang, R.B. Lee, Cloudradar: A real-time side-channel attack detection system in clouds, in: Int'L Symp on Research in Attacks, Intrusions, and Defenses, 2016, pp. 118–140.

[127] S.-J. Moon, V. Sekar, M.K. Reiter, Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, in: CCS '15, ACM, New York, NY, USA, 2015, pp. 1595–1606, [Online]. Available http://doi.acm.org/10.1145/2810103.2813706.

[128] R. Zhang, X. Su, J. Wang, C. Wang, W. Liu, R.W.H. Lau, On mitigating the risk of cross-VM covert-channels in a public cloud, IEEE Trans. Parallel Distrib. Syst. 26 (8) (2015) 2327–2339.

[129] L. Liu, A. Wang, W. Zang, M. Yu, M. Xiao, S. Chen, Shuffler: Mitigate cross-VM side-channel attacks via hypervisor scheduling, in: International Conference on Security and Privacy in Communication Systems, Springer, 2018, pp. 491–511.

[130] S. Mohan, M.K. Yoon, R. Pellizzoni, R. Bobba, Real-time systems security through scheduler constraints, in: 26th Euromicro 26th Conference on Real-Time Systems, ECRTS'14, IEEE, 2014, pp. 129–140.

[131] R. Pellizzoni, N. Paryab, M.-K. Yoon, S. Bak, S. Mohan, R.B. Bobba, A generalized model for preventing information leakage in hard real-time systems, in: RTAS'15, IEEE, 2015, pp. 271–282.

[132] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, J. Reineke, CacheAudit: A tool for the static analysis of cache side channels, in: Presented As Part of the 22nd USENIX Security Symposium (USENIX Security 13), USENIX, Washington, D.C., 2013, pp. 431–446, [Online]. Available https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

[133] B.R. Silva, D. Aranha, F.M. Pereira, Uma Técnica de Análise Estática para Detecç ao de Canais Laterais Baseados em Tempo, Technical Report, 2015.

[134] B. Köpf, L. Mauborgne, M. Ochoa, Automatic quantification of Cache side-channels, in: Proceedings of the 24th International Conference on Computer Aided Verification, in: CAV'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 564–580, [Online]. Available http://dx.doi.org/10.1007/978-3-642-31424-7_40.

[135] A. Langley, Valgrind, 2010, [Online]. Available: https://github.com/agl/ctgrind.

[136] S. Carré, A. Facon, S. Guilley, S. Takarabt, A. Schaub, Y. Souissi, Cache-timing attack detection and prevention, in: International Workshop on Constructive Side-Channel Analysis and Secure Design, Springer, 2019, pp. 13–21.

[137] M. Mushtaq, A. Akram, M.K. Bhatti, M. Chaudhry, V. Lapotre, G. Gogniat, NIGHTs-WATCH: A Cache-Based Side-Channel Intrusion Detector using Hardware Performance Counters, in: Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, 2018.

[138] M. Sabbagh, Y. Fei, T. Wahl, A. Ding, SCADET: a side-channel attack detection tool for tracking prime-probe, in: Proceedings of the International Conference on Computer-Aided Design, ACM, 2018, p. 107.

[139] M. Payer, HexPADS: a platform to detect "stealth" attacks, in: International Symposium on Engineering Secure Software and Systems, Springer, 2016, pp. 138–154.

[140] A. Raj, J. Dharanipragada, Keep the pokerface on! thwarting cache side channel attacks by memory bus monitoring and cache obfuscation, J. Cloud Comput. 6 (1) (2017) 28.

[141] S. Briongos, G. Irazoqui, P. Malagón, T. Eisenbarth, Cacheshield: Detecting cache attacks through self-observation, in: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, ACM, 2018, pp. 224–235.

[142] Y. Kulah, B. Dincer, C. Yilmaz, E. Savas, Spydetector: An approach for detecting side-channel attacks at runtime, Int. J. Inf. Secur. (2018) [Online]. Available http://dx.doi.org/10.1007/s10207-018-0411-7.

[143] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, S.J. Stolfo, On the feasibility of online malware detection with performance counters, in: ISCA, 2013.

[144] Z. Allaf, M. Adda, A. Gegov, A comparison study on flush+reload and prime+probe attacks on AES using machine learning approaches, in: UK Workshop on Computational Intelligence, Springer, 2017, pp. 203—213.

[145] M. Mushtaq, A. Akram, M. Bhatti, N.B.R. Rao, V. Lapotre, G. Gogniat, Run-time detection of prime+probe side-channel attack on AES encryption algorithm, in: Global Information Infrastructure and Networking Symposium (GIIS), 2018.

[146] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, J.-M. Menaud, Cache-based side-channel attacks detection through intel Cache monitoring technology and hardware performance counters, in: Fog and Mobile Edge Computing (FMEC), 2018 Third International Conference on, IEEE, 2018, pp. 7–12.

[147] M. Chiappetta, E. Savas, C. Yilmaz, Real time detection of cache-based side-channel attacks using hardware performance counters, Appl. Soft Comput. 49 (C) (2016) 1162–1174, [Online]. Available http://dx.doi.org/10.1016/j.asoc.2016.09.014.

[148] M. Alam, S. Bhattacharya, D. Mukhopadhyay, S. Bhattacharya, Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks, 2017, [Online]. Available https://eprint.iacr.org/2017/564.

[149] A. Facon, S. Guilley, M. Lec'Hvien, A. Schaub, Y. Souissi, Detecting cache-timing vulnerabilities in post-quantum cryptography algorithms, in: 2018 IEEE 3rd International Verification and Security Workshop (IVSW), IEEE, 2018, pp. 7–12.

[150] M. Mushtaq, A. Akram, M.K. Bhatti, M. Chaudhry, M. Yousaf, U. Farooq, V. Lapotre, G. Gogniat, Machine learning for security: The case of side-channel attack detection at run-time, in: ICECS-2018, Bordeaux, France, 2018, [Online]. Available https://hal.archives-ouvertes.fr/hal-01876792.

[151] M. Mushtaq, A. Akram, M.K. Bhatti, U. Ali, V. Lapotre, G. Gogniat, Sherlock holmes of cache side-channel attacks in intel's x86 architecture, in: IEEE-Communications and Network Security, Washington DC, United States, [Online]. Available https://hal.archives-ouvertes.fr/hal-02151838.

[152] R.E. Kessler, M.D. Hill, Page placement algorithms for large real-indexed Caches, ACM Trans. Comput. Syst. 10 (4) (1992) 338–359, [Online]. Available http://doi.acm.org/10.1145/138873.138876.

[153] J. Liedtke, H. Hartig, M. Hohmuth, OS-Controlled Cache Predictability for Real-Time Systems, in: Proceedings Third IEEE RTAS'97, 1997, pp. 213–224.

[154] W.-M. Hu, Lattice scheduling and covert-channels, in: Proceedings of the 1992 IEEE Symposium on Security and Privacy, in: SP '92, IEEE Computer Society, Washington, DC, USA, 1992, [Online]. Available http://dl.acm.org/citation.cfm?id=882488.884165.

[155] M.A. Mukhtar, M. Mushtaq, M.K. Bhatti, V. Lapotre, G. Gogniat, Flush + prefetch: A countermeasure against access-driven cache-based side-channel attacks, J. Syst. Archit. (2019) 101698, [Online]. Available http://www.sciencedirect.com/science/article/pii/S1383762119305053.

[156] Z. Wu, Z. Xu, H. Wang, Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud, IEEE/ACM Trans. Netw. 23 (2) (2015) 603–614.

[157] N.J. Al Fardan, K.G. Paterson, Lucky thirteen: Breaking the TLS and DTLS record protocols, in: 2013 IEEE Symposium on Security and Privacy (SP), IEEE, 2013, pp. 526–540.

[158] Z. Allaf, M. Adda, A. Gegov, ConfMVM: A hardware-assisted model to confine malicious VMs, in: UKSim-AMSS 20th International Conference on Modelling & Simulation, 2018.

[159] S.h. Peng, Q.f. Zhou, J.l. Zhao, Detection of Cache-based side channel attack based on performance counters, DEStech Trans. Comput. Sci. Eng. (2017).

[160] S. Briongos, P. Malagón, J.L. Risco-Martín, J.M. Moya, Modeling side-channel cache attacks on AES, in: Proceedings of the Summer Computer Simulation Conference, Society for Computer Simulation International, 2016, p. 37.

[161] M. Chouhan, H. Hasbullah, Adaptive detection technique for Cache-based side channel attack using bloom filter for secure cloud, in: 3rd International Conference on Computer and Information Sciences (ICCOINS), 2016, pp. 293–297.

[162] S. Chen, X. Zhang, M.K. Reiter, Y. Zhang, Detecting privileged side-channel attacks in shielded execution with Déjá vu, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM, 2017, pp. 7–18.

[163] Y. Oren, V. Kemerlis, S. Sethumadhavan, A. Keromytis, The Spy in the Sandbox – Practical Cache Attacks in Javascript, ACM, New York, NY, USA, 2015, pp. 1406–1418.

[164] Intel, intel's SGX architecture, 2018, [Online]. Available https://software.intel.com/en-us/sgx.

[165] Intel's hardware accelerator research program, 2018, [Online]. Available https://software.intel.com/en-us/hardware-accelerator-research-program/.

[166] Arm, 2018, [Online]. Available https://www.arm.com/products/security-on-arm/trustzone.