Strategies for securing cache memories against software side-channel attacks

Amine Jaamoum

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : EEATS - Electronique, Electrotechnique, Automatique, Traitement du Signal (EEATS)
Spécialité : NANO ELECTRONIQUE ET NANO TECHNOLOGIES
Unité de recherche : CEA/LETI

Stratégies de sécurisation des mémoires cache contre les attaques par canaux auxiliaires logiciels

Strategies for securing cache memories against software side-channel attacks

Présentée par :

Amine JAAMOUM

Direction de thèse :

Giorgio DI NATALE Directeur de Recherche, Université Grenoble Alpes	Directeur de thèse
Thomas HISCOCK CEA	Co-directeur de thèse

Rapporteurs :

Guy Gogniat
PROFESSEUR DES UNIVERSITES, Université Bretagne Sud
Benoit Pascal
MAITRE DE CONFERENCES, Université Montpellier 2

Thèse soutenue publiquement le **12 décembre 2022**, devant le jury composé de :

Giorgio DI NATALE DIRECTEUR DE RECHERCHE, Université Grenoble Alpes	Directeur de thèse
Frédéric Pétrot PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Président
Clémentine Maurice CHARGE DE RECHERCHE, Université de Lille	Examinatrice
Guy Gogniat PROFESSEUR DES UNIVERSITES, Université Bretagne Sud	Rapporteur
Benoit Pascal MAITRE DE CONFERENCES, Université Montpellier 2	Rapporteur

Invités :

Thomas Hiscock
INGENIEUR DOCTEUR, Commissariat à l'énergie atomique et aux énergies alternatives (CEA)

ABSTRACT

Thanks to an active research and high support from industry, computer systems are now ubiquitous and have become an essential part of daily life. For many years, microprocessor designers have optimized architectures to maximize the number of instructions executed per clock cycle. One important challenge is the large gap between CPU and main memory speed, known as the "memory wall." Multiple cache levels are placed between the CPU and the main memory to fill this gap. A cache can be viewed as small temporary memory storage (from hundreds of kilobytes, for the smallest to a dozen megabytes, for the largest). The different cache levels are generally organized from the smallest and fastest to the largest and slowest. The last-level cache is usually shared between all CPU cores.

Process isolation is the most important security constraint enforced by operating systems in multitasking systems. One process isolation is memory isolation, to separates running processes from accessing each other. However, such isolation is not implemented at the hardware level. For example, all processes running on the same CPU eventually share the same L1 and L2 caches. Furthermore, all processes share the last level cache. A malicious program can then manipulate the cache state to derive information about other processes that share the same cache, thus breaking the isolation the operating system provides. However, despite all the isolation mechanisms described by the OS, hardware sharing can lead to information leaks and thus violate the isolation ensured by these mechanisms. Hence, processor architects must redesign the components of processors and address these vulnerabilities to mitigate them effectively in the new generation of processors.

This thesis first provides a comprehensive study of the most common cache-based side-channel attacks. These attacks occur when memory accesses depend on sensitive information. In particular, these attacks can retrieve the memory access sequence of the victim program. Although these attacks are easy to understand, their practical implementation is usually tricky and requires a deep understanding of poorly documented processor functions. Therefore, the lack of a set for microarchitectural side-channel attacks is an obstacle to analyzing the resilience of existing software/hardware countermeasures against microarchitectural side-channel attacks. For these reasons, we developed

the Micro-Architectural Analysis Toolkit (libMAAT) library to abstract the implementation of microarchitectural side-channel attacks on various CPU architectures (e.g., x86, ARMv7, ARMv8, and RISCV). We have used this library to prototype cache-based side-channel attacks and evaluate the resilience of our secure architectures against realistic attacks.

Additionally, on real systems, the attacker's observations using cache-based side-channel attacks can be disrupted because of the noise added by the optimizations, such as hardware prefetching, simultaneous multithreading (SMT), TLBs, buses, etc. This thesis offers a noise-free framework that abstracts the implementation details of inclusive memory hierarchies. We use this framework to analyze the side effects of cache attacks and evaluate them in randomized caches. In addition, it allows us to introduce ScrambleCache, a novel cache architecture that leverages randomized set placement to defeat cache side-channel analysis. A key property of the ScrambleCache is its low impact on performances and its small area overhead. We demonstrate that this countermeasure protects the system against known cache side-channel attacks while ensuring small overheads, making this solution suitable for both embedded and application systems. In addition, to eliminate conflict-based side-channel attacks, we propose to randomly evict a number of cache lines from the last-level cache each time a precomputed number of memory accesses are achieved. Our security analysis reveals that even in the strongest possible attacker model (noise-free), eviction set construction algorithms fail to find a set of congruent addresses.

Keywords: Cache-based side-channel attacks, Memory hierachy, Randomized caches.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

1

Context and Motivations

In this chapter, a general introduction of this thesis work. After, the introduction of modern systems and the associated threat model are presented. The next section presents contributions to this thesis work. In he last section, we explain the structure of this manuscript.

1.1 Introduction

Digital systems are ubiquitous and have became essential to the functionning of society. Over time, they have evolved from large, unwieldy machines that are difficult to operate and maintain to smaller, more user-friendly and increasingly interconnected machines. Digital systems are now used in a wide variety of applications, including administration, banking, communications, manufacturing, medicine, military, and many others.

Microprocessors are the core of every digital system, whether it is a cloud server, a personal computer, a tablet or a smartphone. Their performance and capabilities have improved rapidly over the years according to what is known as Moore's Law. This law was established in 1965 with the prediction by Gordon Moore [1] that the number of transistors on a chip would double approximately every two years. For the past fifty years, this law has been followed by the integrated circuit (IC) industry. This means that microprocessors have become smaller and more densely packed with new generations. This has allowed manufacturers to develop even more powerful processors without increasing the size or power consumption too much. For example, the first commercially available microprocessor was the Intel 4004, developed in the early 1970s. It had 2,300 transistors and could only perform about 60,000 operations per second. Today's microprocessors are much more powerful and have billions of transistors. They can perform billions of operations per second and are used in everything from personal computers to supercomputers. This has led to smaller and more powerful computers that can handle increasingly complex tasks. However, at the Intel Developer Forum in 2007, Gordon Moore predicted that his law would no longer be valid for ten to fifteen years. In fact, the industry is reaching the physical limits of microelectronics, where transistors will consist of only a few atoms that cannot yet be separated to make them more smaller. Industry will then have to look for completely new methods, which give birth to More than Moore's Law.

The advances of microprocessors have led to the development of increasingly complex operating systems. The operating system (OS) is a set of software that are able to take advantage of the increased speed and power of microprocessors to provide a better experience for users. Unlike older operating systems that could only run a single process at a time, today's modern operating systems are capable of running hundreds of processes simultaneously by efficiently sharing hardware resources. From the operating system's perspective, a process represents an instance of an application running on the

1. director of research and development at Fairchild Semiconductor

system. The operating system shares the resources of each process in two ways: temporally (time-sharing) when multiple applications use them at different time and spacially when multiple applications can use it simultaneously. For example, the memory system is shared spatially and temporally among all CPU cores because all cores can access the data simultaneously.

A typical system executes a wide variety of application at the same time. Each application is associated to a degree of trust that can be very different. To prevent a malicious process from manipulating or accessing the resources of other processes, operating systems has to provide some degree of isolation between the differents processes. To ensure the isolation of the resources of each running process, the operating system (OS) assigns each process a privilege level that corresponds to a trust level granted to that process. The higher the privilege level of a process, the more access it has to system resources. Modern processors such as ARM, Intel, AMD, and RISC-V support different privilege levels. For example, three privilege levels are currently defined in the RISC-V architecture: User level (U-mode), Supervisor mode (S-mode), and Machine mode (M-mode). The processor can only operate in one of these modes at a time. Machine mode is the highest privilege level, and processes running in M-mode are usually inherently trusted because they have low-level access to the system implementation. The M-mode can be used to manage secure execution environments on RISC-V. The U-mode and S-mode are respectively for conventional applications and operating systems.

1.2 Memory Isolation

Modern operating-system kernels allow computer resources to be shared by untrusted processes. In this context, modern operating systems have introduced memory isolation for security reasons to prevent accidental or malicious access by a process to an address outside its own address space.

1.2.1 Software Isolation Approaches

Memory isolation is the basis of modern operating system security. It consists of running each process in an execution environment isolated from other processes.There exists different implementations of memory isolation. Virtual memory management is the most widely used isolation mechanism in most modern processors. Each process is provided with a virtual address space that differs from the actual physical mem-

ory mapping, and that can be much larger than the accessible physical memory. The translation of virtual addresses into physical addresses is performed by the Memory Management Unit (MMU). The MMU is a hardware component that maps the virtual address space seen by the running process into the physical address space. The address translation is responsible of translating virtual addresses into physical addresses. It also returns the attributes access permissions of the associated memory region. On low power processors, when the hardware cost of MMU is not affordable, the Memory Protection Unit (MPU) is used as an alternative to support memory isolation.

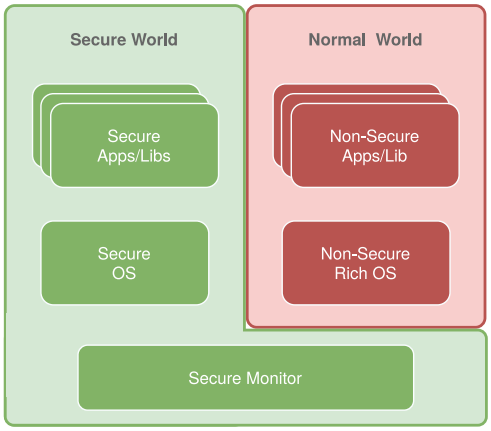
Virtualization is another isolation mechanism that allows the abstraction of hardware resources to make high-performance and isolated virtual machines (VMs). The virtualization was previously based on software techniques, but the massive use of virtual machines, especially for the Cloud, has led manufacturers to build hardware virtualization support mechanisms in their processors to boost performances, such as Intel's VT-x and VT-d or Virtualization Extensions in ARM processors. Virtualization can be used to run one operating system within another, or multiple operating systems in parallel (thanks to a hypervisor such as Xen [Bar+03]). In the case of multiple VMs, the hypervisor runs with the highest privileges. It is responsible for memory allocation, instruction execution and translation, and resource isolation of each virtual machine. The hypervisor should guarantee that a VM is not allowed to access resources allocated to another machine or to the hypervisor itself. One of the biggest advantages of virtualization is that no hardware changes are required - as long as there is an MMU in the processor.

1.2.2 Trusted Execution Environment

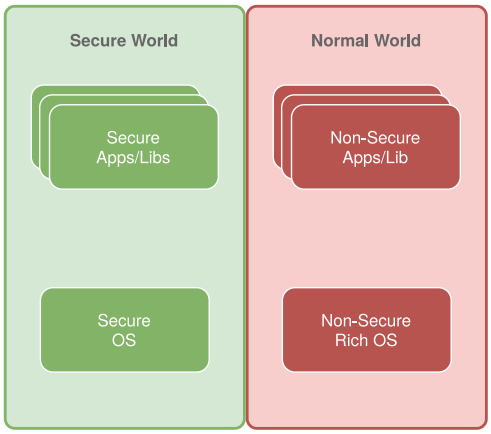
The ability to execute many tasks concurrently in the same system allow to run a large amount of software. The more software, the larger the attack surface and the higher the likelihood of vulnerabilities will be present. To circumvent this problem, the Trusted Execution Environment (TEE) was developed to isolate the execution of critical software, such as the ones that use sensitive information. The TEE is a secure region of the processor that ensures sensitive data is stored, processed, and protected in an isolated and trusted hardware environment. As such, it provides protection against software attacks generated in the Rich Operating System (Rich OS). In systems that support TEE, only trusted applications running on TEE have full access to the main processor, peripherals and memory resources. To prevent untrusted applications from accessing protected hardware resources, processor manufactures propose their own hardware implementa-

tions of TEE, e.g., ARM's TrustZone, Intel SGX, RISC-V's MultiZone, etc. These security extensions offer hardware mechanisms to process/store sensitive data in an isolated environment where even the most privileged software is compromised.

ARM TrustZone [Hol09] is a hardware security extension technology that aims to create a secure and isolated execution environment allowing the implementation of TEE. The TrustZone technology consists of dividing all hardware and software resources of the SoC into two execution worlds, namely the *normal world* and the *secure world*. ARM offers TrustZone technology as an optional hardware extension for its Cortex-A and Cortex-M processors, as shown in Figure 1.1. To communicate between the two worlds on Cortex-A processors, a privileged software known as the *Secure Monitor* (a lightweight OS running in secure world, such as OP-TEE) is called to ensure a secure context switching between the two worlds. ARM TrustZone offers memory isolation at the hardware level, by adding a *non-secure* bit to the physical addresses to indicate whether addresses are originating from a secure or insecure world. To make a tradeoff between performances and security, caches are shared between the two worlds, and the non-secure bit is added to the cache. For this reason, memory accesses from the normal world cannot access data in shared hardware if it is tagged as secure.



(a) TrustZone for Cortex-A processors.



(b) TrustZone for Cortex-M processors

Figure 1.1 – TrustZone Technology.

The ARM TrustZone is mainly used to protect sensitive data, such as banking applications that use the secure world to process passwords and card numbers; they are not accessible from the normal ARM world. Obviously, the execution of an entire application in the secure world is not recommended. The rationale behind the secure world is to isolate small, auditable and secure functionalities. However, running entire applications in the secure world increases the attack user face. An attacker who successfully exploits a vulnerability in an application could gain access to resources in the secure

world. This compromises the security of the whole system, making isolation useless. For example, a banking application on a smartphone might prompt the user to authenticate via face ID. The banking application runs most of its code in the normal world. When it wants to authenticate the user, it calls the secure world, which scans the user's face and compares it to the one stored in the secure memory. Even if an attacker were able to read the data from the banking application, he would not have access to either the scanned or the stored face ID, because both operations take place in the secure world. The only feedback from the secure world to the application is whether the user has been authenticated or not. For example, if we run the entire banking application in the secure world, an attacker can access the user's face ID data through a software vulnerability.

Intel Software Guard Extensions (Intel SGX) [CD16] are extension of the x86 architecture that allow an application to create isolated and encrypted enclaves. Enclaves are areas in the virtual address space that are protected from being accessed from any other processes, including the operating system itself and any virtual machine manager. Because of the ability of a user space application or operating system to define private, encrypted memory regions, any other user space process or privilege level process cannot read these private regions. Intel SGX provides similar guarantees to those offered by ARM Trustzone by including remote attestation, which is very convenient for execution in remote servers. The issue with this technology is that applications protected by SGX suffer from the same problems as with ARM's TrustZone. If the code itself contains vulnerabilities, Intel SGX cannot prevent their exploitation.

1.3 Hardware Sharing and Vulnerabilities

As processor architectures have evolved over the past decade, various strategies have been developed to improve the performances. At the hardware level, this relies on complex hierarchical caches, Table Lookaside Buffers (TLBs), instruction pipelining, simultaneous multi-threading (SMT), branch prediction, out-of-order execution, speculative execution, and also the implementation of some specific hardware accelerators to speed up the execution of critical operations. Researchers have shown many of those optimization techniques have the potential risk of information leakage due to hardware sharing.

This work focuses on *microarchitectural side-channel* attacks that aim to extract critical information on computer systems. These attacks target components that are shared data between multiple processes. These attacks that exploit hardware effects can be exploited remotely, in most cases, rather than requiring physical access. For example,

the RowHammer vulnerability [AAA17], can generate fault in a DRAM cell by repeatedly accessing memory at the same addresses to introduce bit flips. Another famous software-based microarchitectural side-channel attack is Spectre [Koc+19]. Spectre attack exploits hardware vulnerabilities linked to speculative execution. It temporarily overcomes software-defined security policies in order to leak secrets outside of the program's intended code/data flow, and it analyzes the microarchitectural side effects of the speculative execution to retrieve leaked informations.

In modern virtual memory systems, MMU also plays a crucial role in translating virtual addresses to physical addresses. For performance reasons, these translations are cached in the TLBs. On modern microarchitectures, they have a two-level hierarchy, conceptually similar to the cache architecture. The last level TLB is larger and shared for translations of both code and data addresses for different processes. Like other shared hardware components, TLBs are vulnerable to microarchitectural attacks as the mapping between the virtual address and the TLB set into which it is mapped is known for the attacker process. Some works have explored TLB-based attacks, including in Intel's SGX architecture [Wan+17].

1.4 Cache-based Side-Channel Attacks

By far, cache memories are among the greatest source of microarchitectural leakages. Modern processors use cache memories to store recently accessed memory. They are small buffers placed between CPU and DRAM to reduce the memory access latency. These components manage local copies of data following the principles of temporal and spatial locality of memory references in programs. Namely, the most recently accessed memory addresses and nearby addresses are often re-accessed many times. Caches are organized in multiple levels with different sizes and access speeds. For any cache, access latency to data present in the cache is relatively short (hit) compared to uncached data (miss). Cache-based side-channel attacks are a class of microarchitectural attacks that exploit the fact that the cache is shared among different processes and security domains. As the attacker shares the cache with the victim, he can observe the victim cache state and use it to make inferences about the victim. Alternatively, he can indirectly modify the cache state to affect the victim's execution time slightly. In all cases, cache-based side-channel attacks are attacks performed by exploiting time differences.

Cache-based side-channel attacks are particularly powerful because they are not limited to attacks on cryptosystems [IES15a; YK14; OST06; Ber05]. Several works have shown

that cache attacks are possible in all levels and all types of cache memory. For example, the Prime+Probe [OST06] cache attack was initially performed to retrieve the key used by AES on first-level data caches [Per05; NS06; OST06] and in the instruction caches [Aci07]. The LLC is a more interesting attack target because adversaries and victims do not need to share the same CPU. Improvements to the Prime+Probe [Zha+12; Liu+15] technique have allowed it to work on LLCs. Kayaalp et al. [Kay+16] further relaxed the assumptions of the attacks and achieved a better resolution. Various extensive survey studies have listed the threats at different cache levels [Lou+21; Mus+20]. Attacking the L1 cache, for example, has some advantages because fewer load instructions are required to fill or evict the L1 cache due to its small size. However, since the differences in access time between L1 and L2 caches in modern processors are only a few cycles, performing L1 cache attacks can be complicated due to noise in the timing measurements.

1.5 Problem Statement

Process isolation is the most important security constraint in multitasking systems. However, despite all the isolation mechanisms described above, hardware sharing can lead to information leaks and thus violate the isolation ensured by these mechanisms. In addition, sharing cache memories between different processes of different security domains leads to the loss of sensitive information, which makes the memory hierarchy vulnerable to cache-based side-channel attacks.

Therefore, to protect a memory hierarchy against cache-based side-channel attacks, the different cache levels should be protected. Extensive research has been conducted on countermeasures to mitigate CSCAs. However, despite efforts to develop these mitigation techniques, it remains a lot to explore. This is mainly due to the fact that CSCA mitigation techniques usually target only some specific cache vulnerabilities in a particular cache level and do not protect the entire memory hierarchy from cache attacks. The proposed countermeasures also completely eliminate or significantly reduce the performance benefits of resource sharing. In addition, new attacks that exploit new vulnerabilities continue to emerge, and the attack surface continues to grow. Recently, CSCAs have become more sophisticated and could overcome applied mitigation techniques. This raises the question of *how we can assess a secure memory hierarchy against CSCAs ?*

The goal of this research is to develop a feasible approach to mitigate cache-based side-channel attacks in different cache levels. To develop a useful solution, it must not only

provide *strong security* against cache attacks but also have a *low-performance overhead*. This point us to raise the question: *which countermeasures should be used at each cache level to ensure a tradeoff between security and performance ?* In line with the previously defined objective, a deep understanding of cache attacks and different classes of countermeasures is required. Addressing this problem will allow us to identify the limitations of these mitigation techniques and derive a set of quantitative criteria for a secure memory hierarchy.

1.6 Contributions

This thesis offers three major contributions:

1. **Creation of a new modular Micro-Architectural Analysis Toolkit Library.** Microarchitectural side-channel attacks exploit contention in shared processor components to leak information between processes. Although these attacks are easy to understand, their practical implementation is usually difficult and requires a deep understanding of poorly documented processor functions. Therefore, the lack of a set for microarchitectural side-channel attacks is an obstacle to analyze the resilience of existing software/hardware countermeasures against microarchitectural side-channel attacks. For this reason, we developed the Micro-Architectural Analysis Toolkit (libMAAT) library to abstract the implementation of microarchitectural side-channel attacks on various CPU architectures (e.g., x86, ARMv7, ARMv8, and RISCV). LibMAAT is a modular library that allows all available options to be changed at compile time. For example, the timing source can be specified on each execution without having to recompile libMAAT. We developed this library to quickly prototype microarchitectural attacks and evaluate the resilience of our secure architectures against realistic attacks.

2. **Free-noise analysis of randomized caches.** Modern microarchitectures incorporate many optimizations, such as hardware prefetching, simultaneous multithreading (SMT), TLBs, busses, etc. These optimizations allow programs to run faster and more effectively. However, they introduce noise to the attacker observations, making algorithms evicting addresses form cache memories less reliable. To make a favorable scenario for the attacker and the reproducibility of attacks, we developed a noise-free model where there is no source of noise. This cache framework abstracts the implementation details of the hardware and simulates only the memory hierarchy behavior. In this study, we also attempted to analyze

whether secure randomized caches are worth considering from a security perspective. We show that randomizing L1 caches improves security, but is not sufficient to mitigate all known cache-based side-channel attacks. Nonetheless, we demonstrate that L1 randomization can be combined with a lightweight random eviction method in higher-level caches (LLC) to mitigate known conflict-based cache attacks.

3. **Scramble Cache Architecture.** While working on analysing dynamic randomization on first-level cache, we observe that miss rate increase extremely at each time we change how the cache memory is addressed. There are many solutions and countermeasures in the literature that protect L1 caches. However, most of them do not cover all cache-based side-channel attacks. In this work, we present a novel L1 cache architecture that uses dynamic randomized set placement to prevent cache side-channel analysis. Unlike partitioning techniques, this approach allows full cache sharing, which benefits performance. The ScrambleCache uses a lightweight permutation function to randomize the address-to-cache mapping. The permutation can be renewed at any time to not allow cache side-channel analysis. Ideally, the ScrambleCache would change the permutation as frequently as possible to spread memory accesses across the entire cache. A key property of this architecture is its low impact on performance and its small footprint. We show that this countermeasure can protect the system from known cache-based side-channel attacks while guaranteeing low overhead area and performance.

1.7 Thesis Outline

Besides the introduction chapter, this thesis manuscript is composed of five other chapters:

Chapter 2 - Background and state-of-the-art This chapter starts by providing a brief background on modern micro-architecture features and cache hierarchy designs, specially for x86 architectures. We then etablish a detailed state-of-the-art of cache-based side-channel attacks. Finally, we review related work on countermeasures against cache attacks.

Chapter 3 - LibMAAT In this chapter, we present LibMAAT, a toolkit for experimenting with micro-architectural side-channel attacks. At the beginning, we describe the

libMaat overview and the need of a modulare toolkit. At this time LibMaat includes Flush+Reload, Flush+Flush and Evict+Reload techniques. In another hand, we use this library to implement the first-round AES attack on various platforms.

Chapter 4 - Noise-free analysis This chapter presents a noise-free security analysis of random dynamic caches. We begin by describing our framework for the inclusive memory hierarchy that we used to study the various algorithms for constructing a conflict-based cache attack. These algorithms are used to construct a set of addresses that target a particular cache line and therefore observe the behavior of another process. A first study targets cache memories without countermeasures to understand in detail how these algorithms work. In a second step, we study the impact of dynamic randomization in different levels of caches on the success of the algorithms.

Chapter 5 - ScrambleCache Architecture This chapter presents the first robust hardware defense architecture against cache-based side-channel attacks for L1 caches. It has been shown that hardware sharing provides a significant attack surface for micro-architectural side-channel attacks. Unfortunately, defending against cache-based attacks on the L1 cache is difficult and presents several challenges. We begin with a description of the dynamic randomization supported by the ScrambleCache architecture. We then present the various modules that have been added to address the coherence and performance issues. We conclude with a performance and security analysis using the gem5 simulator.

Chapter 6 - Conclusion This chapter summarizes the results of the thesis and discusses future works and perspectives.

2

Background and State-of-the-art

In this chapter, we present the elements composing modern processors which are necessary to understand the rest of the manuscript. The objective is to recall the generic principles as well as the main hardware mechanisms used to optimize the execution of programs. We then provide more details on the cache-based side-channel attacks that exploit interferences in cache memories and their implementations. Many works have shown that shared resources can be exploited to break isolation offered by the OS. We then focus on software and hardware countermeasures against these attacks and present their limitations. Finally, we compare and discuss these state-of-the-art solutions.

2.1 Modern Micro-architectures

The single core micro-architecture is the main responsible for computation. It is a critical part of CPU design since it is responsible for executing all the instructions of a program running on the machine. Different mechanisms were introduced to optimize its execution time. This section briefly describes the most common optimization techniques present in modern micro-architectures needed to understand the rest of this thesis.

Pipelinning

Pipelining is a mechanism for increasing the instruction execution speed in a microprocessor. In a microprocessor without a pipeline, instructions are executed one after another. A new instruction is not started until the previous instruction has been fully executed. In a pipelined microprocessor, the microprocessor starts a new instruction before it finishes the previous one. Therefore, multiple instructions are executed simultaneously in the CPU core. This does not reduce the execution time of a single instruction. On the other hand, the throughput of the microprocessor, i.e., the number of instructions executed per clock cycle, is increased. Standard pipeline support five stages: Fetch, Decode, Execute, Memory, and Writeback.

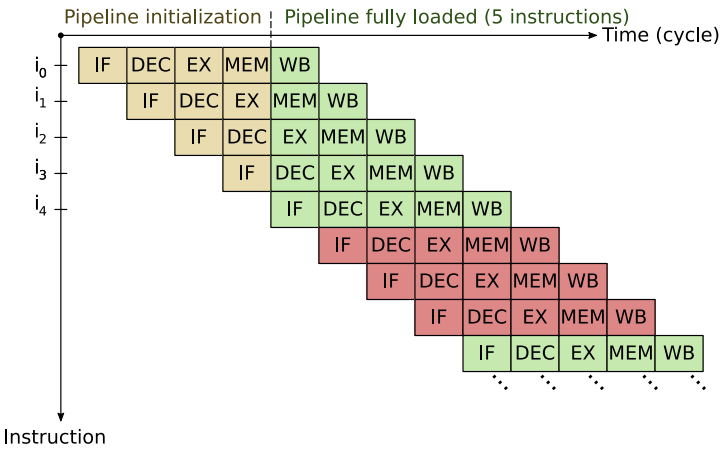


Figure 2.1 – Instruction execution in a processor with a 5-stage pipeline. It takes nine cycles to execute five instructions. At t = 5, all stages of the pipeline are called, and all five operations take place at the same time.

Superscalar

A superscalar processor seeks to exploit parallelism between instructions to speed up program execution. This approach avoids modifying programs to exploit parallelism: the processor itself detects which instructions can be executed in parallel. However, this approach also increases the complexity and power consumption of the hardware, which limits current processors to a few instructions per cycle. To further exploit the available parallelism, vector instructions and multithreaded or multicore processors are used.

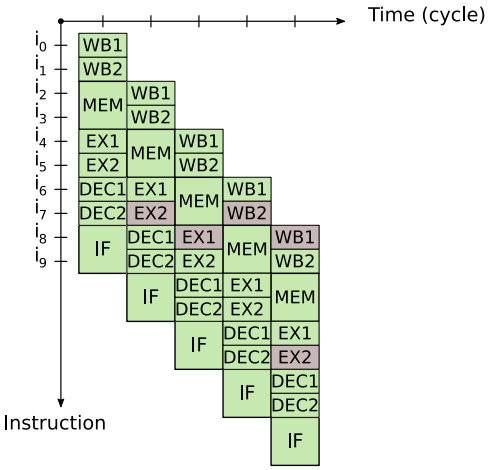


Figure 2.2 – Instruction execution in a processor with a 5-stage superscalar pipeline capable to execute maximum two instructions per pipeline stage.

Out-of-Order Execution

The pipelining mechanism is used to increase the throughput of executed instructions. However, the execution throughput is quickly limited by the dependencies between the instructions. For example, suppose there is a data dependency between two instructions. In that case, the pipeline is blocked until the dependencies are resolved, which slows down the execution of the pipeline. *Out-of-order execution* is an optimization technique used in modern processor architectures to maximize the use of computing units. Instead of executing instructions sequentially, the processor executes them according to the availability of resources, taking into account the various dependencies between instructions by using some specific registers, called renaming registers. As a result, multiple instructions can be executed in parallel and in an order that may differ from that specified in the assembly code.

Branch Prediction

All micro-architectures provide branch instructions in their Instruction Set Architecture (ISA), where the execution flow can change to execute a different address, instead of the one stored at the next address. A branch instruction can be either an *unconditional branch*, which always results in a branch or a *conditional branch*, which may or may not trigger a branch depending on some condition (e.g., comparing values). It also depends on how the address of the new instruction sequence (the target address) is specified. They are generally classified as *direct* if the instruction contains the target address (e.g., a register or memory location) and is known when the instruction is decoded, or *indirect* if the target address must be computed in the execution phase. In either case, the branch instruction always jumps to executing a different address. These instructions break the linearity of the execution flow by jumping to a new address. This type of event can significantly affect the efficiency of the pipeline if not properly scheduled.

The processor would have to wait until the conditional branch instruction passed through the execution phase before the next instruction could reach the fetch phase in the pipeline. The presence of *branch predictor* attempts to avoid this time loss by guessing whether or not the conditional jump has a high probability of being executed. The branch assumed to be most likely is then extracted and executed speculatively. If it later turns out that the guess was wrong, the speculatively executed or partially executed instructions are discarded, and the pipeline starts over with the correct branch, causing a delay. It exists different hardware components used in modern micro-architectures that are used to predict whether a branch instruction is taken and also specify the target address speculatively.

Prefetcher

Another phenomenon that affects pipeline performance is the occurrence of a cache miss during memory access to data or instructions. The system must wait for the target address to be cached before it can be used. This process may require access to one or higher levels of the memory hierarchy, and in this way, many cycles will be lost. Therefore, using hardware prefetchers allows the prediction of future memory accesses. It aims to cache data and instructions that are likely to be used. As with branching prediction, most of these mechanisms maintain a history of previous accesses to infer the following executed instruction or data.

2.2 Memory Paging and Virtual Memory

In the past, processors executed only a single program that had access to the entire physical memory of the system. Physical memory refers to the main memory where programs and data are stored as they are used. Very quickly, the need arose to run multiple tasks on the same processor. The first multitasking systems worked directly with physical memory, allocating an area in physical memory to each process. The operating system was responsible for dividing the processor execution time among the different processes (see Figure 2.3). However, direct use of physical memory introduces some performance and security issues.

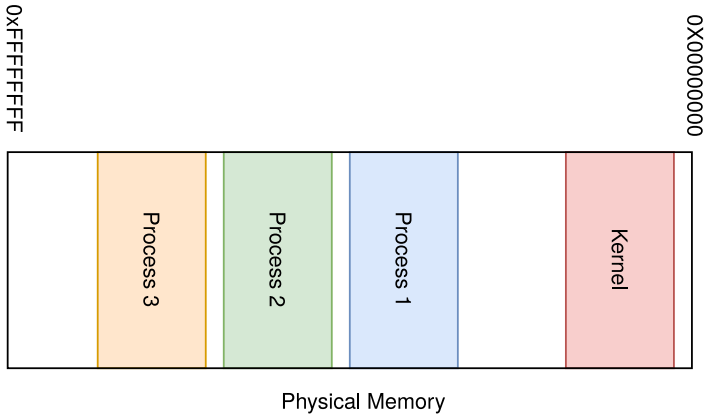


Figure 2.3 – Muti-tasks physical memory organisation

Excepted small micro-controllers, recent processors all use virtual memory, a technique that provides an abstraction of physical memory. This can be beneficial for a number of reasons:

1. It allows a process to address more memory than is physically present in the system. This can be important for processes that need to access large amounts of data, such as databases or scientific applications.
2. It can be used to map physical memory to different address spaces. This can be useful for sharing data between processes.
3. It can be used to protect a process from accessing unauthorized memory. If a process tries to access a memory area that is not allocated to its address space, an error occurs and the process is prevented from continuing. This can help prevent security vulnerabilities such as buffer overflows [Fos+05].
4. It can be used to improve the performance of a system by allowing processes to run in parallel without accessing the same areas of physical memory.

2.2.1 Paging System

Segmentation is a memory management scheme. It is used to divide a process's virtual address space into variable size segments. Thus, the virtual address space is the collection of segments of variable size. Paging is another technique of virtual memory that is the most used in modern processors. The principle is to divide the address space of each process into fixed-size pages. The size of these pages is 4KiB in most modern systems and in some cases can vary from 4KB to 64KB or even serval megabytes or gigabytes for so-called huge pages. The operating system, in cooperation with the hardware, must maintain a record of the address of each page in physical memory.

Huge Page

Huge pages are large buffers of virtual memory to be mapped into contiguous physical chunks of 2MB (or 1GB), instead that of regular 4KB. On one hand, huge pages save page walks when traversing arrays of more than 4KiB, improving performance. On the other hand, they increase the risk of memory fragmentation, what might lead to wasting resources.

For a given virtual address, to find out where the associated data is located in physical memory, the virtual address is divided into two parts. The most significant bits indicate the *virtual page number*, while the least significant bits represent the *page offset* (as shown in Figure 2.4). The memory management unit (MMU) is a hardware unit responsible for translating virtual addresses to physical addresses. When the MMU receives a virtual address, it extracts the most significant bits and then looks for the corresponding *physical page number*. If this page translation is not found, a *page fault* occurs and is reported to the operating system. The operating system is then responsible for updating the translation of the missing page in the memory. If the page is found, the permissions are checked, and if they are correct, the physical address is sent to the memory bus. Otherwise, a page fault is raised (e.g., a write attempt to a read-only page), and the processor enters a privileged mode (e.g., kernel mode) and sets some registers indicating the nature of the fault.

2.2.2 Page Table Lookup

The address translation mechanism is based on the use of a page table that defines the correspondence between virtual pages and physical pages. The page table is stored

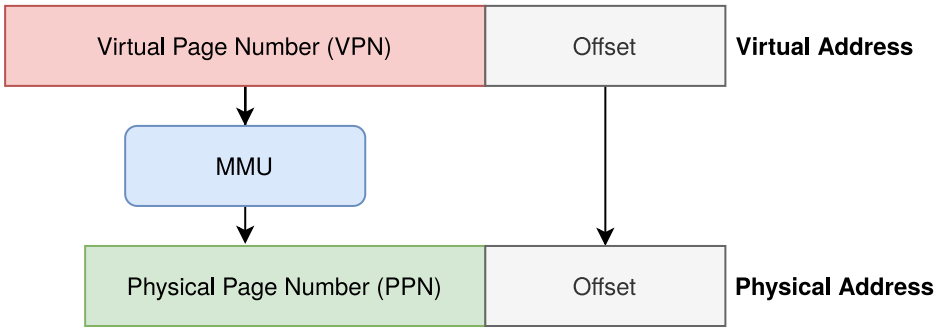


Figure 2.4 – Translation of a virtual address to a physical address

in main memory. For each process, the operating system uses a special register that references the page table base address, called the *page table base register* (PTBR). This register value is updated when a context switch occurs, and the operating system is responsible for this task. This is the basis for process isolation, where each process has its own virtual address mappings and can only access its own address space.

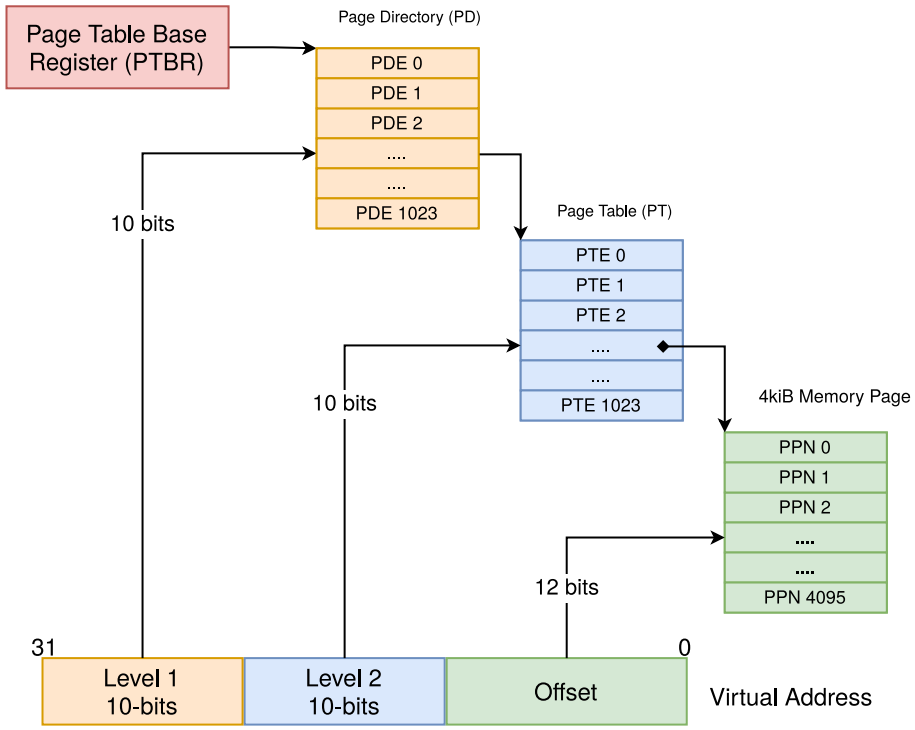


Figure 2.5 – Address translation for 4KB pages using a two-level page table. Starting from the PTBR that returns the specified process page table, the processor determines the physical address by gradually using parts of the virtual address.

However, the physical memory occupied by this page table is a major drawback. For example, for a 32-bit address space with 4 KiB pages, a single-block page table would have 2^{20} entries. Since modern micro-architectures run several processes in parallel, and the page table of each process occupies 4 MiB ($= 4{*}2^{20}$), thus a large part of DRAM memory

would be occupied by the page tables, which degrades the performance of the system. To solve this problem, the use of multi-level page tables was introduced. The goal is to store only the highest level of the page table in memory.Also, all page tables need to be in main memory only when they are needed. This avoids unnecessary memory consumption. For example, to translate a virtual address into a physical address using a two-level page table, the 32-bits virtual address space is divided into 1024 memory regions, each with 4 MiB page directory (PD) indexed by the most significant bits of the virtual address. Each PD entry (PDE) is associated with a page table (PT) of 1024 entries, each controlling a 4 KiB page, which is the default page size for most use cases (as shown in Figure 2.5). This mechanism of translation is called *page table walker*.

2.2.3 The Table Lookaside Buffer (TLB)

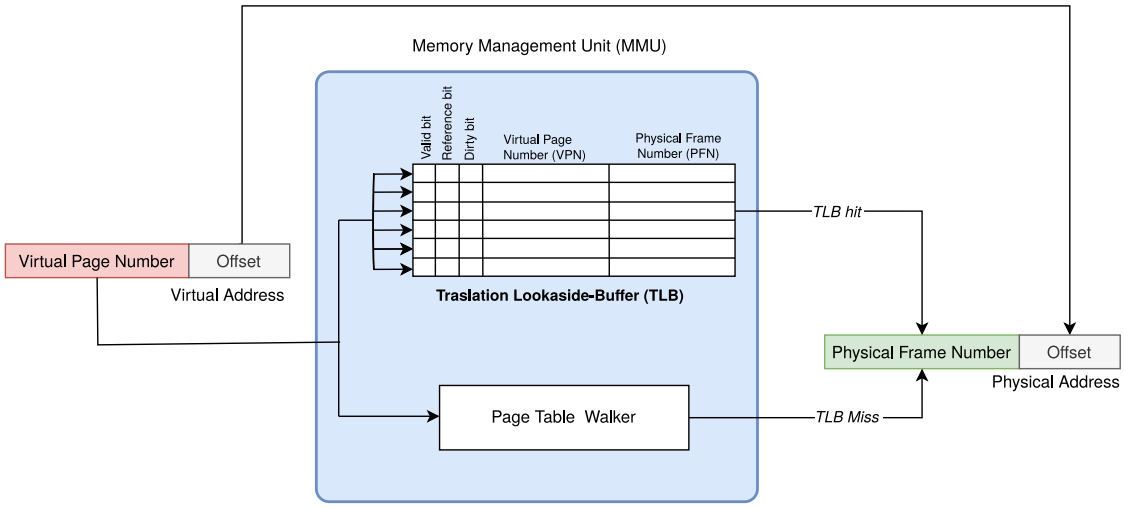


Figure 2.6 – Address translation using a TLB. On a hit, the corresponding physical page address can be returned immediately. On a TLB miss, the page tables are accessed.

Virtual address translation is a complex operation because it involves traversing the various levels of the page table to find the corresponding physical address, which slows down memory access time. To speed up virtual page translation, modern processors use a cache of recent address translations, called the translation lookaside buffer (TLB). The TLB (as shown in Figure 2.6) is a fully associative memory that stores the most recently translated virtual pages and their physical pages, as well as the permissions associated with each page. To translate a virtual address, the system first looks for the physical page number in the TLB instead of translating it directly using the page table. In case of a TLB hit, the physical address associated with the correct TLB entry is used. Otherwise, a TLB miss is reported and the processor uses the page table walker to translate the virtual page number. If the appropriate physical address is in main memory, it is used

for translation and loaded into the TLB. If not, a page fault is reported to the operating system.

2.3 Memory Hierarchy

One of the most important bottleneck in modern processors is the access latency to main memory (DRAM). This is due to the consequences of the frequency gap between processor and memory, which has been growing steadily over the past decades, this phenomena is called *memory wall*. And this implies that the latency and bandwidth of memory becomes insufficient to provide processors with enough instructions and data to continue computing, processors will effectively always be stuck waiting for memory.

Today, processors use a memory hierarchy that take advantage spatial and temporal locality of memory accesses and trade-offs in the cost-performance of memory technologies. The typical memory hierarchy of a computer system is shown in Figure 2.7.

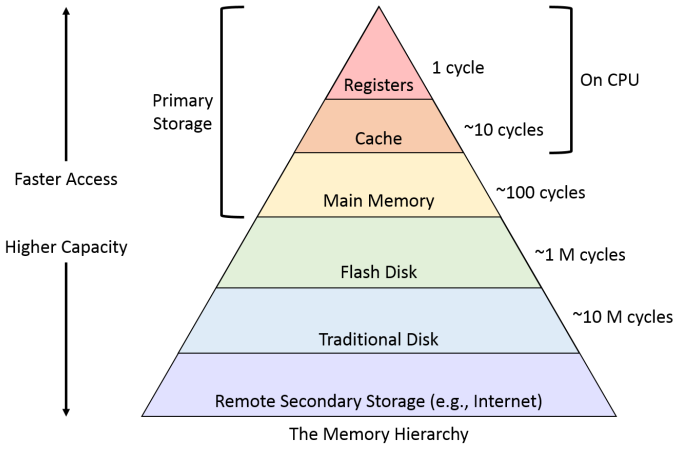


Figure 2.7 – Typical memory hierarchy of a computer system, with indicator values for the access latency and capacity of each memory level.

Because fast memories are expensive, a memory hierarchy is organized into multiple levels, each smaller, faster and more expensive thant the higher levels, which is farther from the processor. As illustrated in Figure 2.7, a memory hierarchy is divided into: *primary* and *secondary* storages. Secondary storage is persistent storage for non-critical data that doesn't need to be accessed as frequently as data in primary storage or that doesn't have the same performance or availability requirements. Primary storage typically requires expensive, high-performance storage systems, while secondary storage systems can effectively operate on lower performance devices that are better suited for

long-term storage (e.g., disks storage). Primary storage consists of three main storage components:

Registers The CPU has a few registers which can be directly accessed. The processor registers are the fastest memory in the system and contain variables and temporary data which is required for the execution of each program. The data in the registers can be accessed in a single clock cycle. This is due to the fact that the registers are part of the CPU.

Cache memory The cache is also a part of the CPU which are much larger than the registers, but are still relatively fast with access times of a few clock cycles. Cache memories are based on Static Random Access Memory (SRAM) technology. The processor's caches are the next level of the memory hierarchy and are used to store data that the processor is likely to use in the near future (see section 2.4).

Main memory If a data is not stored in the registers or in the cache, the main memory is accessed. the main memory is based on Dynamic Random Access Memory (DRAM) technology. The main memory is considererd as the more larger and slower memory in the primary storage (as shown in Figure 2.7).

2.4 Cache Memory

The cache memory is one of the most complex and critical subsystem in modern processors and is characterized by a number of features. In this section, we present the most important ones. In particular, we present the concept and the interest of the cache memory in a system, as well as their functioning. We also detail data redundancy between the different cache levels, writing policy and also the replacement policy.

2.4.1 Locality of Reference

The purpose of cache memories is to provide the processor with instructions and data as quickly as possible and to have a large capacity to store all available information. However, the larger the memory is, the slower it is to access it. Cache memories transparently stores data that the processor has used in the near past in hopes of using it again in the near future. When a computer program accesses a particular memory location, there is a high probability that it will also need to access nearby memory locations - this phenomenon is called *locality of reference*. There are two types of locality of reference: temporal and spatial locality.

- **Temporal locality.** Temporal locality refers to the tendency of a program to repeatedly access the same instructions or data. When a data item is referenced, it is likely to be referenced again in the near future. To take advantage of this, CPUs typically use a cache memory to store recently accessed data. This way, if the same data is needed again, it can be retrieved directly from the cache memory much faster than from the main memory.
- **Spatial locality.** Spatial locality refers to the tendency of a program to access instructions or data that are near the data it is currently using. In this case, it makes sense to cache not only the data that the processor needs, but also the neighboring data so that it is already available when it is needed.

2.4.2 Cache Organizations

A cache can be represented as a vector of memory lines, also called *cache blocks*, where each line can contain a copy of a data block in main memory. A *cache line* is the smallest amount of data used to communicate with the higher level of the memory hierarchy, so that for each data transfer from the cache to main memory, the amount of data transferred is equal to one cache line. Most modern processors have a *cache line size* of 64 bytes. A cache line is divided into 32- or 64-bit *memory words*, depending on the CPU micro-architecture. A memory word represents the smallest amount of data that can be transferred between the processor and the cache memory (as shown on Figure 2.8). The decision to store multiple words in a single cache line is based on the principle of spatial and temporal locality. Namely, if CPU executes instruction in the address $@_{PC}$, there is a high probability that CPU will execute the same instruction in $@_{PC}$ or instruction in $@_{PC+1}$.

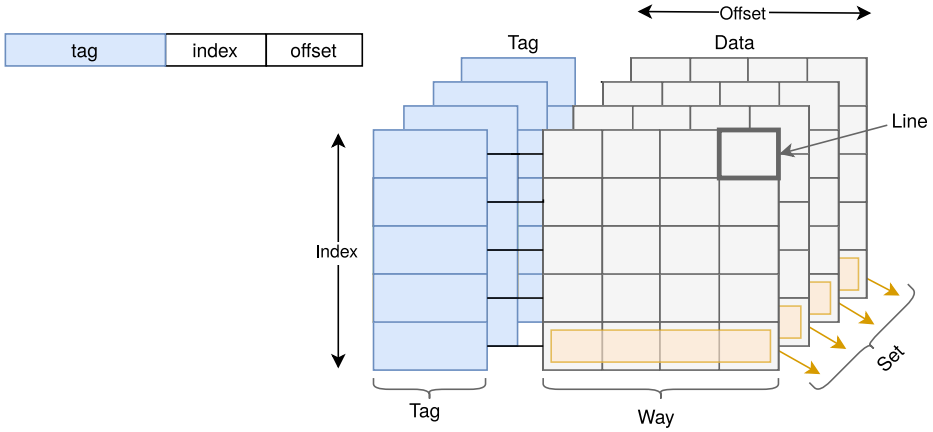


Figure 2.8 – Cache terminology of a 4-way set-associative cache.

Each cache line is associated with a set of metadata, which includes a *tag*. A tag is computed from the highest bits of the memory address and is used to identify the block of memory contained in a cache line. The metadata also include some other bits that indicate the state of each cache line, such the *valid bit* this is used to indicate whether the cache line is valid or not.

To access a particular address in the cache, depending on the cache organization, the cache index is determited and the cache line that contains the data being addressed is returned. The tag value associated with that cache line is compared to the tag value of the address. If they are equal and the valid bit of this cache line is set, a *cache hit* is triggered and the requested data is sent to the CPU. In this case, the data is returned to the CPU within a few clock cycles. Otherwise, a *cache miss* is triggered and the request is sent to the next cache level until the data is found in one of the cache levels or in main memory. A cache miss therefore results in a longer access time, which can be determined with accurate timing measurements (see chapter 3 for more details).

Since cache memory cannot contain all data present in the main memory, an addressing method must be defined to determine in which cache line the fetched data should be stored. That is, when information is transferred from main memory to the cache, a decision must be made about where to place that information. This method is called *address-to-cache mapping*. There are three main cache organizations that manage the mapping between cache lines and main memory blocks in a different way.

Direct-mapped Caches

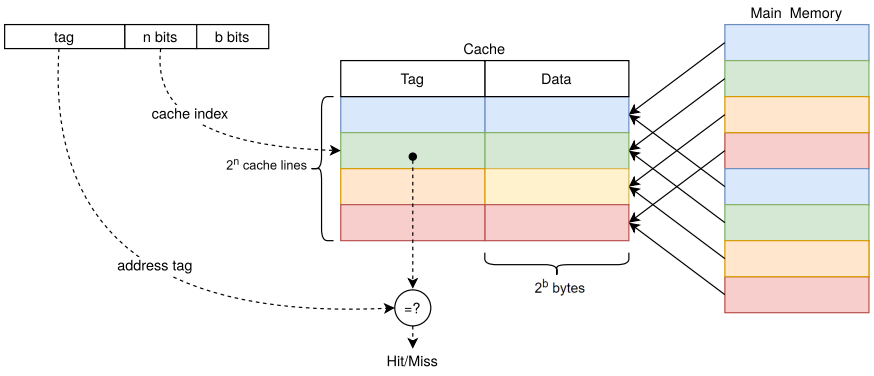


Figure 2.9 – Organization of a direct-mapped cache.

The simplest one is a *direct-mapped* cache, as shown in Figure 2.9. With a direct-mapped cache, each block fetched from main memory is allocated to only one cache line. As shown in Figure 2.9, the direct mapped cache consists of 2^n cache lines. The lowest b

bits of the address are used as an *offset* within the cache line data. To access a particular address in the cache, the middle n bits of the address are used as the *cache index*. Consequently, the access time to this cache line is very fast since no cache lookup needs to be performed. However, since the main memory is much larger than the cache memory, this means that very likely a large number of data blocks that have the same cache index will also have to use the same cache entry (called *congruent addresses*), which increase the number of conflict addresses and thus generally have a higher miss rate.

Full-associative Caches

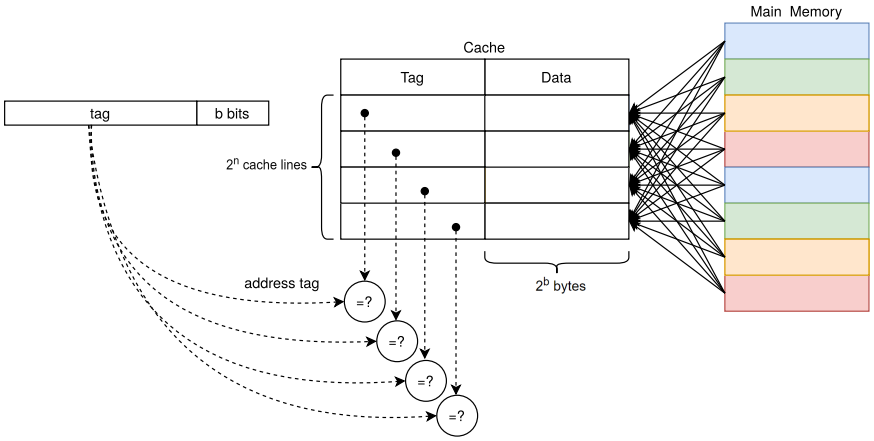


Figure 2.10 – Organization of a full-associative cache.

Unlike direct-mapped caches, *full-associative* caches map memory addresses to any cache line, as shown in Figure 2.10. The tag is used to compare it to all cache lines at once and return the valid cache line. This method requires a lot of logic to allow simultaneous access to all cache lines, which can result in higher power consumption because the entire cache must be searched at each memory access. The main disadvantage of this cache organization is that the cache lookup process is slower and impractical for large caches because data can be anywhere in the cache. Therefore, fully associative caches are only suitable for small cache memories such as TLBs.

Set-associative Caches

A *set-associative* cache is a compromise between the directly mapped cache and the fully associative cache in terms of power consumption and performance. This type of organization is used to reduce the congruency problem by storing multiple cache lines in the same cache index. The cache is then divided into 2^n *cache sets*, and a memory address

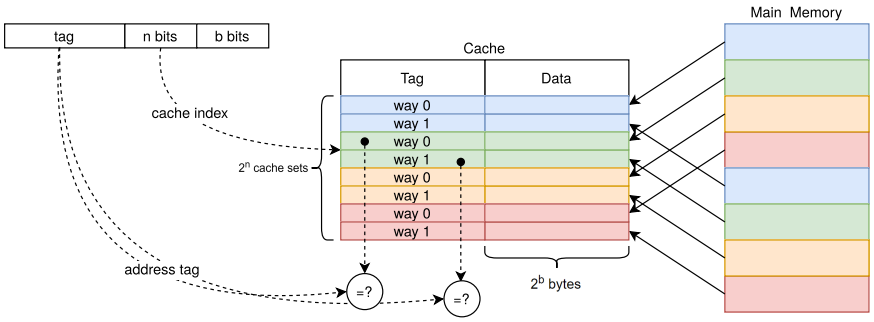


Figure 2.11 – Organization of a set-associative cache.

is now associated with a set of *cache ways* (e.g., cache lines) instead of a single cache line, as shown in Figure 2.11. These caches are commonly used in modern processors for data and instruction caches, but also for the translation lookaside buffer. They are usually referred to as w-ways set-associative caches, where w is the number of cache ways per cache set.

2.4.3 Virtual and physical tags and indexes

A cache can be indexed either *virtually*, where the index is provided directly by the virtual address, or *physically*, where the cache memory is accessed via physical addresses. Virtual addressing reduces cache access time because no address translation mechanism needs to be traversed before indexing the cache. The use of virtual addresses can lead to coherence problems between different processes. This in turn reduces performance. To uniquely identify the actual address cached in a particular line, the address tag is used. This tag can also be based on the virtual or physical address. The advantages and disadvantages of the different combinations are listed below:

VIVT - virtually indexed, virtually tagged. The virtual address is used for both the index and the tag, which improves access time because the CPU address does not need to be translated for each memory access. However, the cache memory should be invalidated for each context switch to avoid coherence problems due to the similar address spaces of different processes. Also, shared memory between different processes is cached more than once.

PIPT - physically indexed, physically tagged. The physical address is used for both the index and the tag, which increases access time because the address CPU must be translated for each memory access. With this addressing technique, the physical tags are unique. Therefore, the cache is not invalidated with each process context switch.

PIVT - physically indexed, virtually tagged. The physical address is used for the index and the virtual address for the tag. This combination has no advantages because the address must be translated, the virtual tag is not unique, and shared memory can still be cached more than once.

VIPT - virtually indexed, physically tagged. The virtual address is used for the index and the physical address for the tag, which means that the cache set can be read in parallel with the translation lookup in the TLB. This means that lookup is usually much faster than with a physically indexed cache, where the cache cannot be looked up until after translation. This combination is most commonly used in modern processors.

2.4.4 Cache Replacement Policy

For set-associative caches, the number of ways is not sufficient to store all congruent addresses mapped to the same cache set. There are a number of techniques that can be used to improve the hit rate. One of these is to increase the size of the cache, but this can also increase the access latency. Another technique is to increase the number of ways, which increases the number of possible cache locations for congruent addresses. However, increasing the number of ways leads to higher hardware complexity and power consumption. Another technique is to use a replacement policy that select efficiently which cache blocks should be replaced when a miss occurs.

The least recently used (LRU) policy [Mat+70] is the most used policy in Intel processors. On a cache eviction, the LRU policy simply evicts the oldest line among a set of given cache lines. To find the oldest line, the LRU policy conceptually maintains a stack to keep the history of accessed cache lines. LRU performs well when there is temporal locality of reference, that is, when data that was used recently is likely to be reused in the near future. However, it performs poorly for two types of access patterns. First, it can be a bottleneck when the application's working set size exceeds the cache size. Second, LRU performs poorly in the presence of scans, because it caches more recently used scans at the expense of older lines that are more likely to be reused.

In modern micro-architectures, the replacement policies are often undocumented. However, the replacement policy for some micro-architectures has been reverse-engineered. For example, Sandy Bridge and Ivy Bridge use a pseudo-LRU replacement policy [Won13]. Moreover, Ivy Bridge, Skylake, and Haswell use adaptive cache replacement policies, which only work as pseudo-LRU in some situations [Jal+10b]. Cache replacement policies are regularly updated across processor generations.

2.4.5 Multi-core Caches

Cache Levels

In the case of a cache miss, the purpose of using only a single cache will be rendered useless and the CPU will have to fetch required data from the main memory. However, with a multi-level cache, if the required data is missing in the closest cache to the processor namely first-level cache (or L1 cache), it will then search the higher cache levels and only access the main memory if all cache levels don't contain the required data. The general tendency is to keep the L1 cache small with a latency of some clock cycles ($\approx$ 2-5 cycles) from the processor. And using higher cache levels with more capacity than L1 cache in order to decrease its miss rate.

Most modern micro-architectures use multiple cache levels (as shown in Figure 2.12 for intel CPUs). The data contained in the caches can be accessed in a few cycles, depending on the cache level. In x86 micro-architectures, the last-level cache (LLC) is often shared among all cores to improve performances and simplify cache coherency. The first-level caches (L1) are dirctely connected to the CPU core and they are usually private to each core. In a Harvard architecture, each core typically has two separate L1 caches, an instruction cache (L1I) and a data cache (L1D).

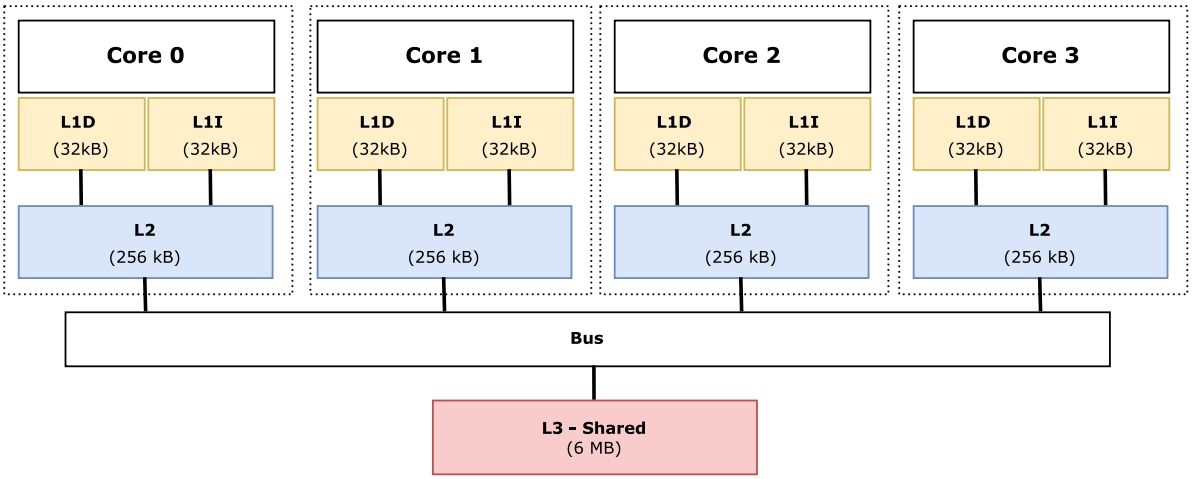


Figure 2.12 – Modern memory hierarchy for Intel CPUs. Each CPU core has a private L1D and L1I caches, and also has private unified L2 cache. The L3 cache is the LLC cache and is shared among all CPU cores.

Inclusion Policy

The inclusion policy is one of the key design decisions in cache hierarchies. The goal of the inclusion policy is to reduce data traffic between different cache levels. There are

three main types of inclusion policies: inclusive, exclusive and non-inclusive.

Inclusive. In an inclusive policy, data is redundantly stored at all levels of the cache hierarchy. This means that if a block is present in one level, it is also guaranteed to be present in all higher-level caches. One advantage of having data redundancy between different cache levels is that it can improve performance by reducing the number of cache misses.

Exclusive. In contrast, with an exclusive cache policy, data present at a given cache level is not guaranteed to be present at any other cache levels. This means that the data is not duplicated between the different levels of cache. The advantage of an exclusive cache is that reduces data redundancy and offers more cache space than an inclusive cache.

Non-inclusive. The non-inclusive policy [ZAF07] means that a given block of data can be present at multiple cache levels, but not all levels. So, if a block is present in the L1 cache, it might also be present in the L2 cache, but not in the L3 cache. This policy provides a compromise between inclusive and exclusive policies, allowing for some data redundancy. With this policy, the amount of cache actually available is close to an exclusive policy [Jal+10a]. However, implementation of a non-inclusive policy it is more difficult, as the system needs to keep track of what data is present in each cache level.

Cache Coherence

With the advent of multiprocessor architectures, the use of caches brings additional problems because data can then be shared between several cores. This led to data replication within the different caches and therefore the possibility of having different copies of the same data in memory. Data consistency then became an important problem.

Data coherency in multiprocessor micro-architectures can be managed into different levels: Software or hardware level. In software level, the programmer is responsible for data coherency, and therefore must invalidate/forward the cache lines shared by multiple cores when necessary, such as after a write by one core. Hardware coherency is completely transparent to the programmer because the hardware system invalidates the cache lines. Hardware coherency has its price in terms of area and complexity, but it provides very good performance and simplifies the programmer's work, which is why this technique is generally used.

2.4.6 Caches on Intel x86 CPUs

In recent Intel processors, the last-level cache is divided into slices that are connected to the cores through a ring interconnect (see Figure 2.13). Moreover, the last-level cache is inclusive, which means that it is a superset of the L1 and L2, i.e., it contains all data present in L1 and L2. As well as the set and way indices, a slice index must also be determined in order to place a line into this sort of partitioned cache. Although microprocessor designers do not disclose the mapping that is used between memory blocks and cache slices, different works have been reverse-engineered the undocumented mapping functions on x86 architecture [Mau+15b].

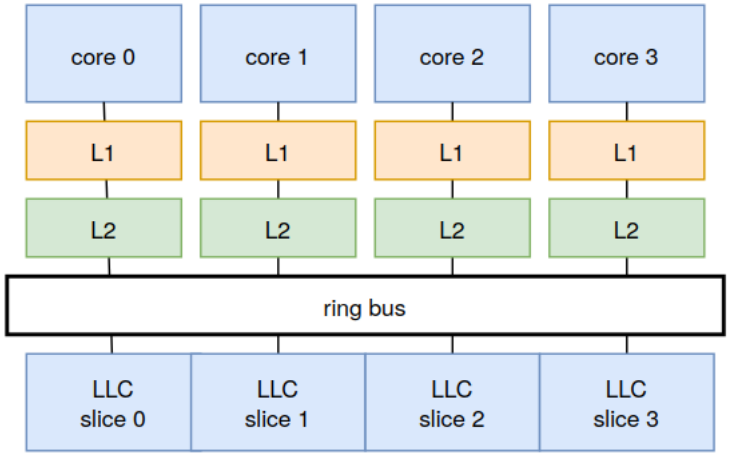


Figure 2.13 – Cache architecture of a quad-core Intel processor (sicen Sandy Bridge microarchitecture). The LLC is divided into slices, and interconnected with each core by a ring bus.

2.5 Cache-based Side-Channel Attacks

2.5.1 Information Leakage Channels in Cache memories

Information leakage can occur when an attacker can gain access to information that they are unsupposed to have access to. Information leaks can significantly compromise the security of systems because they can lead to the disclosure of secrets or the circumvention of security mechanisms. Side-channel attacks are a type of attack that takes advantage of the physical characteristics of a system to extract information that would otherwise be hidden. These attacks exploit the fact that all devices leak information about their internal state through side effects. Multiple attack vectors have been proposed, exploiting machine features such as power consumption [KJJ99], electromag-

netic field [QS01], acoustic emanation [GST14], and execution time variation [TOS10; GBK11; YK14]. Depending on the side-channel information to exploit, the attacker may or may not require physical access to the system.

In contrast to physical side-channel attacks, which need physical access to the target device, they are software-based micro-architectural attack. In a software-based micro-architectural, malicious applications exploit the timing and behavior differences that are caused through micro-architectural optimizations. These kind of attacks exploit the fact that the micro-architectural optimizations are shared among different applications. The most common example of shared hardware are the CPU, cache memories, the branche predictor, and the Translation Lookaside Buffer (TLB). This work focuses on timing side-channel attacks that target cache memories. Other software optimization, such as, shared libraries, and deduplication techniques offer better memory footprint for the running processes, but they also allow interference between the running processes.

Caches are by far the largest source of micro-architectural leaks. They are inherently shared between different processes and security domains running on the system, either sequentially (i.e., over time) or concurrently. This sharing opens the door for a large class of attacks, known as cache-based side-channel attacks (CSCAs). The data from these processes are mapped into different cache lines based on their addresses. Thus the different processes are in constant conflict with the cache space; the data of other processes are evicted to make space for other data. CSCAs aim to abuse these cache interactions to observe other processes' activities, and therefor extract sensitive information by exploiting the access time variation. The time variation between a cache hit and a cache miss can result in information loss. This means that the memory access time measurement can directly indicate whether the data was fetched from the cache or the main memory. In a multi-level hierarchy, each cache level has different penalties for a cache miss, so the level in which the data is found can be measured with some accuracy. Thus, the total execution time correlates with the number of cache hits and cache misses. CSCAs are the most prominent class of software-based micro-architectural attacks, in which, a malicious process retrieve sensitive information of a victim process by observing its cache activities. They allow to a malicious process to have a partial view of memory addresses that are being accessed by the other processes.

This ability can be turned into various security exploitations; for example, leaking kernel memory as part of original Spectre [Koc+19] and Meltdown [Lip+18] attacks and creating keyloggers [GSM15] or covert channels [Mau+15a; Lip+16]. Such attacks have also been successfully employed to retrieve secret keys from various cryptographic al-

gorithms, such as AES [Ber05; OST06], RSA [Aci07; AS08] and ElGamal [Zha+12].

2.5.2 Classification of Cache-based Side-Channels Attacks

Different classification of cache side-channel attacks based on kind of the leaked information can be found in the literature [Pag02; HL17; Acı+09]. A common classification, proposed byAcıçmez et. al [Acı+09], depending on the type of the used information. They identifie three categories of CSCAs: *Timing-driven*, *Trace-driven*, and *Trace-driven* attacks.

Time-driven attacks. Time-driven attacks [Tsu+03; Ber05; BM06], also known as cache timing attacks, exploit the number of cache hits and cache misses, typically through an indirect measurement of the overall execution time of the victim process. This information is useful when the victim's memory access depends on security-critical information. Time-driven attacks are referred to as *passive* attacks (such as the Bernstein attack [Ber05]) when no changes to the victim's cache are required and only address conflicts of the victim algorithm are exploited; or *active* attacks when the attacker influences the execution of the victim process by forcing the eviction of certain cache lines, as in the Evict+Time attack of Osvik et. al [OST06].

Access-driven attacks. In access-driven attacks [NS06; GBK11], the attacker observes the cache conflict with the victim process to learn its memory access pattern. In this type of attack, the attacker puts the cache memory in a known state (e.g., fills the cache memory with its own data) and then performs difference analysis after the victim has executed. The attacker observes a cache conflict in the victim process when the cache state changes. Access-driven attacks include Prime+Probe attacks [OST06; Per05] and its variants, and cache template attacks [GSM15].

Trace-driven attacks. In trace-driven attacks [AK06; GKT10; Ge+18], the attacker observes the state of some specific cache lines during the victim's execution. Thus, the attacker obtains the entire sequence of cache hits and cache misses of the victim process in real time (e.g.,Flush+Reload [YK14] or Flush+Flush technique). Trace-driven attacks are very powerful compared to time-driven attacks, where the sequence of hits and misses by the victim is generally hiden throughout the execution time rather than the exact sequence. Trace-driven attacks are destructive in the case of simultaneous multithreading (SMT) or hyperthreading, which allow hardware to execute multiple threads simultaneously. This can be dangerous as the threads use the same processor resources.

Another classification of CSCAs, according to which they are categorized into two types: *Sharing-based* and *Conflict-based* attacks. CSCAs work differently if the target addresses are shared or not.

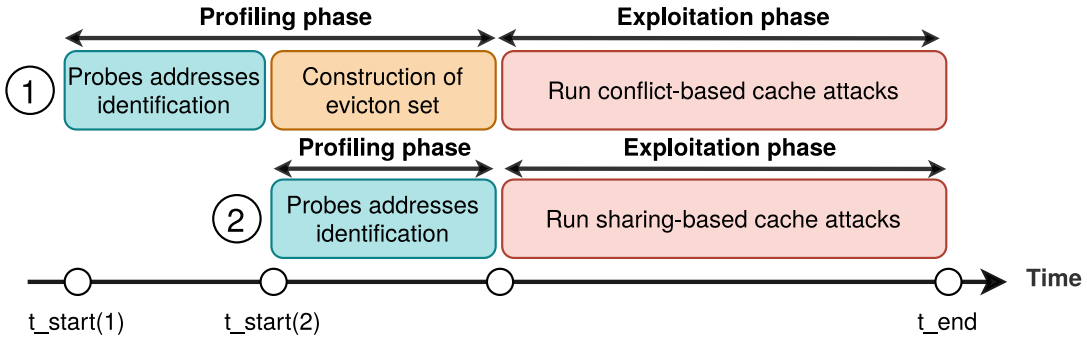


Figure 2.14 – Cache-based side-channel attack procedure.

Sharing-Based Attacks. Sharing-based attacks rely on unexpected memory sharing between the attacker and the victim. Such sharing is, for example, introduced by the operating system (OS) with shared libraries or memory deduplication. Therefore, the attacker can remove a shared address from any cache level. This can be achieved with a cache maintenance instruction (e.g., `clflush` instruction in x86 processors), as in the Flush+Reload [YK14] attack. `clflush` guarantees that the addresses are written back to memory and invalidated from the cache. Another way to remove a shared address from the memory hierarchy is to access a set of addresses mapped to the same cache set, which forces the replacement policy to evict the target address. An example of such an attack is Evict+Reload [Lip+16]. In both of these attacks, the attacker reloads the target address. The access latency of the load reveals whether the victim has accessed that address or not.

Conflict-Based Attacks. Unlike sharing-based attacks, conflict-based attacks do not require address sharing, which make them more powerful. The idea is that when the victim's addresses and the attacker are mapped to the same cache set, they may evict each other from the cache (this phenomenon is called "cache contention"). The attacker can use such eviction stategy to make the target cache set in a know state. This is done by construction a number of addresses that are mapped to the same cache set (called an *eviction set*). Complex addressing functions and replacement policy of modern systems make construction of eviction sets more difficult. It exists some algorithms in the literature to construct an efficient eviction set. Thus, the attacker exploit conflicts in these addresses to know whether the victim has accessed the targeted cache set.

2.5.3 Cache-based Side-Channel Attacks - General Steps

A formal way to describe cache attacks is modeled as in two phases, a *profiling phase* and an *exploitation phase* (see Figure 2.14). In the profiling phase, the attacker identifies *probe addresses*, whose access patterns may leak information about the victim's access patterns. For example, those probe addresses can be associated with cryptographic keys [OST06] or keyboard input events [Lip+16]. In the exploitation phase, CSCAs follow a general pattern consisting of three steps [Dis+17]: an *initialization* step, a *waiting* step, and a *measurement* step.

In the initialization step, the attacker places the cache memory into a defined state so that the victim's execution can cause detectable side effects. A common approach is to remove the target cache lines from the targeted cache level. There are two strategies to evict a cache line. These strategies work differently depending on whether the probe addresses are shared (case ② on Figure 2.14) or not (case ① on Figure 2.14). The main difference is that if the addresses are not shared, the attacker must construct an eviction set to perform his attack, otherwise he can use a cache maintenance instruction if it is available from the user-level.

In the wait step, the attacker waits a period of time to allow the victim to execute, which may access some sensitive data and changing the cache state. To increase the accuracy of the cache attacks, the attacker should carefully configure the *wait interval* [YK14]. The optimal wait interval is the key to a good tradeoff between accuracy and attack resolution. If the wait interval is too short, more accurate information about the victim's accesses can be collected during this period, but the initialization and measurement phases are performed more frequently, and all the victim's accesses are lost while the attacker is busy with these steps. If the attacker's waiting step is too long, the number of accesses by the victim during this time interval could be too large to be accurately identified [YK14].

In the measurement step, the attacker takes some measurements and infers whether the cache state was changed by the victim's execution during the wait step.

2.5.4 Leakage Exploitation Techniques

Bernstein's attack.

In [Ber05], Bernstein proposes a CSCA that exploits access time variations to recover the secret key of an AES T-table implementation. The AES T-tables are preprocessed

computations used to improve the performance of the AES. Therefore, the AES algorithm can be implemented as a sequence of T-table lookups, and a few computations between those. Each round of AES accesses some elements of the T-tables, which are then fetched into cache memory. By observing the execution time, the attacker can deduce which T-table entry was accessed and thus retrieve part of the AES key. However, Bernstein provides no analysis of his methodology and no explanation of why the attack is successful. In [NSW06], Neve et al. fill this gap by presenting a complete analysis of Bernstein's attack technique and explaining the correlation model. Later, Aciiçmez et al. [ASK07] extended the Bernstein attack to also use the information from the second round of AES encryption to obtain the full AES key. In [WHS12], Weiss et al. bconsider a virtualization-based system where the trusted environment runs an AES server. Under this assumption, the authors show that a man-in-the-middle attack using a customised version of the Bernstein attack can significantly reduce the AES key space, making brute force attacks feasible.

Evict+Time.

Evict+Time was described by Osvik et al. [OST06] as a generic cache-based timing attack technique in which the attacker triggers multiple computations of the victim process and measures the execution time, and is consequently classified as a time-driven attack. In the Evict+Time technique, the attacker measures the impact of a particular cache set on execution time. As is illustrated in algorithm 1, the attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time. Then, the attacker attempts to manipulate the cache state by evicting the content of a particular cache set. The victim's access to memory may cause the eviction of certain cache lines that were fetched by the the attacker process. In the last phase, the attacker observes the time difference in the victim's execution time, which reveals to the attacker if the targeted cache set was accessed or not.

Algorithm 1: Evict+Time technique

Input: N=Number of samples

```
1 T_baseline ← Measure the baseline execution time of victim program.
2 for i ← 0 to N by 1 do
3 |   Evict a specific cache set.
4 |   T ← Measure execution time of victim program again.
5 |   if T > T_baseline then
6 |   |   The evicted cache set was accessed.
7 end
```

The Evict+Time technique provides information on cache set granularity. An advantage of this technique is that it doesn't require shared memory paging between the attacker and victim. As Evict+Time technique is sensitive to noise, it target only L1 caches. Evict+Time was demonstrated in the L1 data cache to extract AES key from `OpenSSL` cryptographic library [OST06]. However, the authors demonstrated that the Evict+Time technique requires multiple computations of the victim process to extract sensitive information.

Prime+Probe

The second technique proposed by Osvik et al. [OST06] is more powerful. The idea of the Prime+Probe technique is to fill a specific cache set with its own data and then let the victim program run for a while. If the victim's accesses fall into the attacker's cache set during execution, the cache controller must evict the attacker's data and replace it with the victim's data. In the second step, the attacker accesses the data that was previously filled into the targeted cache set and measures how long this takes. A higher time means that at least one cache line has been replaced. Otherwise, when a lower time is measured, the attacker concludes that no cache line has been replaced in the target cache set. In contrast to the Evict+Time technique, the Prime+Probe method offers more accuracy because it measures the cache access times directly, whereas Evict+Time measures them indirectly via the total execution time.

Algorithm 2: Prime+Probe technique

Input: N=Number of samples

```
1 T_threshhold ← Measure the threshhold between cache misses and cache hits.
2 for i ← 0 to N by 1 do
3 |   Occupy a specific cache set.
4 |   Schedule the victim's program.
5 |   T ← Measure the access time to the occupied cache set.
6 |   if T > T_threshhold then
7 |   |   The cache set is likely occupied by the victim program.
8 end
```

The Prime+Probe technique, unlike most other CSCAs, does not rely on shared memory. The original form of the Prime+Probe technique was demonstrated in the L1 data cache by Osvik et al. [OST06] to retrieve the AES key in `OpenSSL 0.9.8`. In other works, Aciicmez et al. [Aci07; AS08] exploit cache contention in the L1 instruction cache to knows the victim's controle flow. Thus, they retrieve the RSA [Aci07] and the digital signature algorithm (DSA) keys used by `OpenSSL 0.9.8` library [AS08]. Zhang et

al. [Zha+12] demonstrated that key-recovery attacks using the L1 instruction cache are possible between virtual machines. They demonstrated that the Prime+Probe technique is capable of recovering ElGamal keys [ElG85].

Flush+Reload

The Flush+Reload technique is often considered the strongest cache exploitation technique because operates at the cache line granularity, which increase the attack resolution. Unlike the Evict+Time and Prime+Probe techniques, which use a cache set granularity, this property can tell the attacker whether specific cache line has been cached or not (as illustrated in algorithm 3). The first generic Flush+Reload technique was presented by Yoram et al. [YK14]. This technique relies on the existence of shared virtual memory (such as shared libraries or page deduplication) between the attacker and the victim program. Hence, in scenarios where shared memory is not available, Flush+Reload cannot be applied and an attacker has to resort to Prime+Probe technique. It also exploits the presence of the cache maintenance instruction `clflush`, which is available from the user-level in x86 micro-architectures. This instruction allows a process to write back and invalidate a specific virtual address in the memory hierarchy. Because the cache operates on physical addresses, shared memory exists only once in the LLC. The Flush+Reload technique does not require knowledge of virtual-to-physical address mapping, since `clflush` directly uses virtual addresses. Therefore, if the shared virtual address is flushed in one process, it is flushed for all processes. This weakness exposes the x86 micro-architectures since the LLC it is inclusive and shared by all cores.

Algorithm 3: Flush+Reload technique

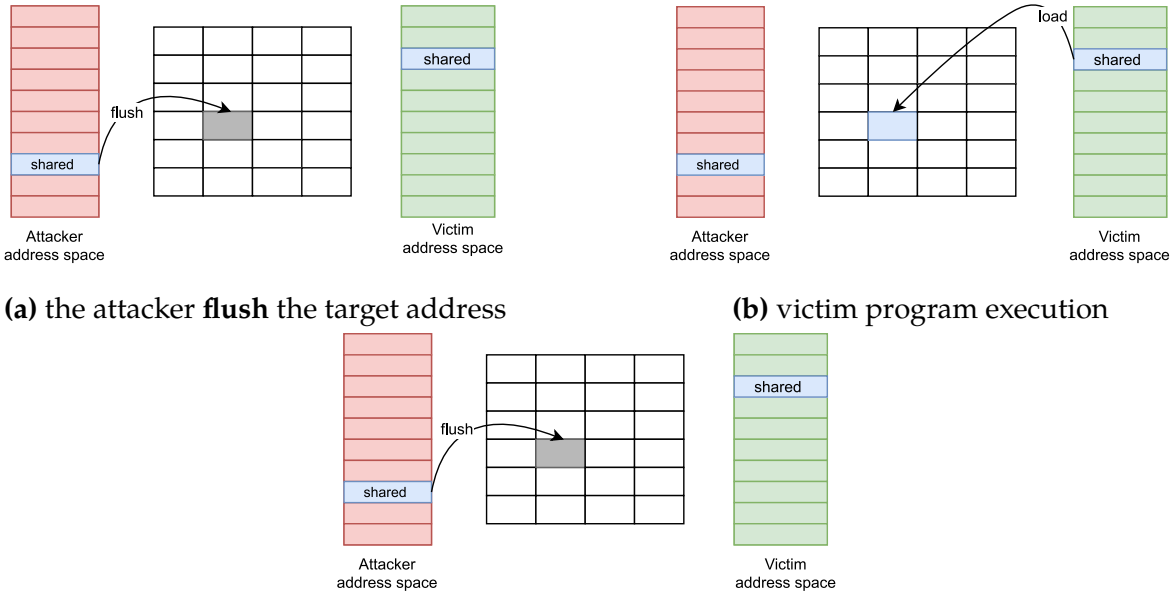
Input: N=Number of samples

```
1 T_threshhold ← Measure the threshhold between cache misses and cache hits.
2 Map binary (e.g., shared object) into address space.
3 for i ← 0 to N by 1 do
4 |   Flush a specific cache line.
5 |   Schedule the victim's program.
6 |   T ← Measure the access time to the flushed cache line.
7 |   if T < T_threshhold then
8 |   |   The victim program accessed the flushed cache line.
9 end
```

Flush+Reload technique enables more fine-grained attacks that have already been demonstrated against different cryptographic algorithms. Yarom and Falkner [YK14] demonstrated a Flush+Reload technique on the last-level cache for attacking square and multiplies RSA implementation in a multi-core platform. Irazoqui et al. [Ira+14] ex-

tended this technique to extract the full AES key from another virtual machine. Yarom and Benger [YB14], Benger et al. [Ben+14] and Van de Pol et al. [PSY15] used similar technique to recover `OpenSSL` Elliptic Curve Digital Signature Algorithm (ECDSA) nonces and thus the secret key. Gulmezoglu et al. [Gul+15] exploit a shared resource optimization technique called memory deduplication to mount a powerful known ciphertext cross virtual machine attack on an `OpenSSL` implementation of AES.

Flush+Flush



(a) the attacker **flush** the target address

(b) victim program execution

(c) the attacker **flush** again the target address

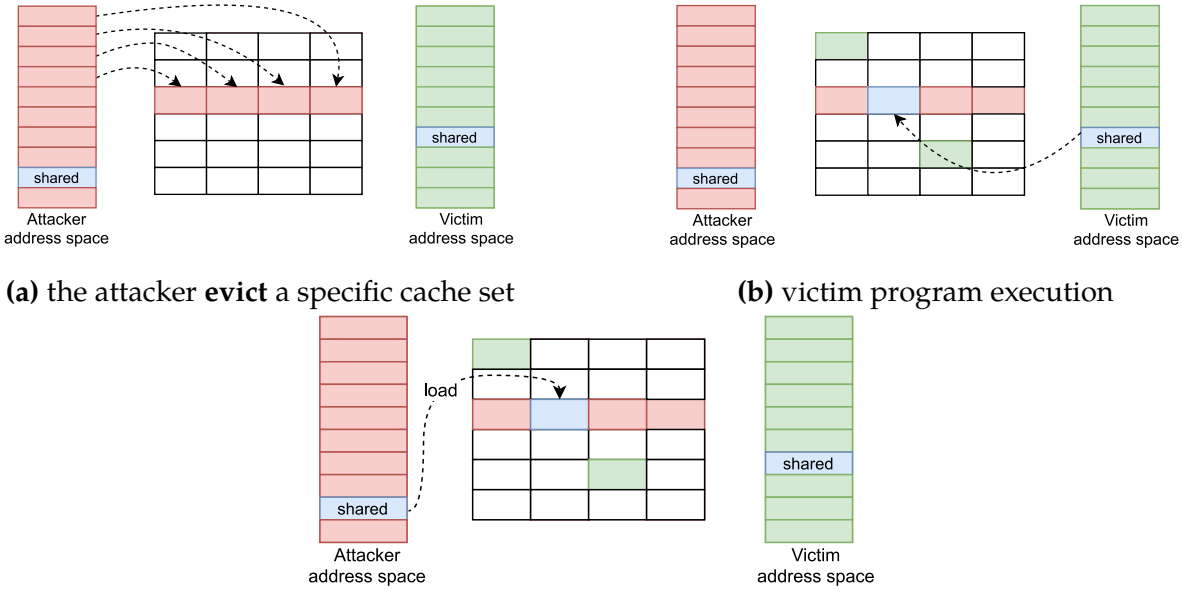
Figure 2.15 – Illustration of Flush+Realod technique in a 4-way set-associative cache (columns) and 6 cache sets (lines). In Figure 2.15a, the attacker flushes the shared address, and schedule to the victim program, as shown in Figure 2.15b. In Figure 2.15c, the adversary measures the flushing time to reveal whether the victim has accessed the shared address.

Gruss et al. [Gru+16] developed Flush+Flush technique a new variant of Flush+Reload that only depends on the execution time of the x86 `clflush` instruction. In the first step of this technique, the attacker flush the target address. Then waits the victim's process execute. However, instead of the reloading step where the target address is accessed, it is flushed again causing no cache activity. If the execution time that the flush instruction takes is fast, the attacker learns that the target address is not present in the memory hierachy and likely no other process accessed it during the wait interval. Otherwise, the victim process accessed this address, and the flush instruction needs more time to

invalidate it in the memory hierachy. Figure 2.15 illustrates the Flush+Flush steps, in case when the victim process access the target address during it execution.

The advantages of Flush+Flush technique, over Flush+Reload technique are that it runs at a higher frequency which makes it faster that any other techniques. It also allows for a higher resolution, and that it generate less LLC activity which can be used for detecting CSCAs. lush+Flush technique, is considered noisier than Flush+Reload technique [DM21], resulting in a slightly higher error rate [Ge+18]. Gruss et al. [Gru+16] demonstrated that this technique can be used to implement a powerful covert channel that achieved a bandwidth of 496 KiB/s, almost seven times faster than any previously published covert channels.

Evict+Reload



(a) the attacker **evict** a specific cache set

(b) victim program execution

(c) the attacker **reload** the target address.

Figure 2.16 – Illustration of Evict+Reload technique in a 4-way set-associative cache (columns) and 6 cache sets (lines). In the evict step, the attacker load a specific cache set, and schedule to the victim program, as shown on Figure 2.16b. In the reload step, the adversary measures the access time to reveal if the victim has accessed the target address.

While Flush+Reload and Flush+Flush techniques are only feasible in x86 CPUs, which due to the presence of the unprivileged `clflush` instruction. Other processors that do not provide an unprivileged such instruction (e.g., ARM micro-architecture), these techniques are not practical. Gruss et al. [Lip+16] introduced the Evict+Reload technique as anothor variant of Flush+Reload technique that uses eviction instead of the flush in-

struction. In Evict+Reload technique, the victim and the attacker share the same memory pages. To evict a shared cache line from a specific cache level, the cache set should be filled with as many congruent addresses such as the replacement policy decides to evict the targeted cache line (see Figure 2.16).

2.6 Eviction Set Construction

Many cache-based side-channel attacks require control of a specific cache line, making it in a known state. When there is no maintenance instruction such as `clflush` on x86 micro-architecture, the only way to control the activity of a cache line is to remove it from the cache set. This is done by filling the same cache set with enough data to force the replacement policy to evict the targeted cache line. Hence, finding small eviction sets is a fundamental step in many cache-based side-channel attacks. In this section we describe the existing techniques for constructing a set of addresses mapped to the same cache set, called an *eviction set*. To perform a conflict-based cache attack, the size of the eviction set must be as small as possible. Accessing a large eviction set has more chances to evict the target access. However, accessing more addresses is slower and can generate noise. This noise may be due to the number of addresses accessed that are not mapped to the same cache set and do not participate in the eviction of the target cache set. In this section, we describe also the different techniques to optimize the eviction set size.

2.6.1 Defining Eviction Sets

We said that two virtual addresses x and y are *congruent* when they both fall into the same cache set and the same cache slice. For example, this is the case when the cache indexes and the slice bits of their respective addresses $x_{physical}$ and $y_{physical}$ are the same.

Eviction Set

A set of virtual addresses E is an *eviction set* for a target address x when $x \notin E$ and at least $a \geq W$ addresses in E are congruent with x, where W is the number of cache ways, and all of them should be filled to ensure eviction of cache lines in the target set.

The goal of an eviction set is the following: if x is initially cached, accessing elements

of E by using a specific access strategy can systematically evict x from the cache set. Depending on the supported replacement policy, accessing the eviction set can evict the target address with a certain probability δ.

Eviction Probability

We denote E_δ as an eviction set E that succeeds in evicting the target x with probability δ. When the number of congruent addresses $a \leq W$, the probability of evicting the target address converges to $E_\delta = 0$. This is because not enough addresses are accessed; therefore, not all cache lines are accessed to force the replacement policy to evict the address x.

For example, in systems that support permutation-based replacement policies such as LRU, FIFO, and PLRU, sequentially accessing W addresses in E ensures that x will not be maintained in the cache later. However, caches are adopting other replacement policies, such as pseudo-random policy, requiring a set of addresses that contain more than W addresses and a specific accessing order to increase the eviction probability. Several works [Lip+16; VKM19] have proposed different techniques to increase eviction probability by changing the access pattern to elements in E. Different works identified *dynamic* and *static* approaches for finding efficient eviction sets.

2.6.2 Static Approach

Since the last-level cache (LLC) is physically indexed and tagged, constructing an eviction set requires knowledge of mapping virtual to physical addresses. For CPUs without cache slicing (e.g., ARM), eviction sets can be constructed directly using the page mapping in `/proc/self/pagemap`. Lipp et al. [Lip+16] demonstrate cross-core `Prime+Probe`, `Evict+Reload`, and other cache attacks on ARM-based Android smartphones without any privileges. Fortunately, Google patched Android in March 2016 [1], and now root privileges are required to expose physical addresses, making eviction sets harder to find.

To complete the conflict-based cache attack from user-space, Irazoqui et al. [IES15a] used large (2MiB) pages to probe cache sets of the LLC without having to resolve the mapping from virtual to physical addresses. However, it isn't easy to fill all LLC cache sets, and it takes a long time to fill them. Gruss et al. [GMM16] have shown that it

1. Patch CVE-2016-0823: https://source.android.com/docs/security/bulletin/2016-03-01

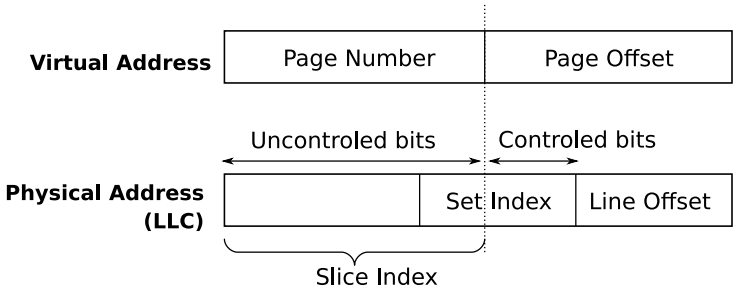


Figure 2.17 – The decomposition of a virtual address into a physical cache location.

possible to construct eviction sets using 2MiB *transparent huge pages* [Org] to perform `RowHammer` attack from on ARM micro-architecture.

Finally, modern CPUs with LLC slicing (e.g., x86) use an undocumented hash function, complicating the static approach to constructing efficient eviction sets. Several works have attempted to reverse engineer the supported hash function by 1) allocating and identifying groups of conflicting addresses [Mau+15b; Sea15] and 2) reconstructing the slice function using Hamming distance [HWH13] or by solving the system equations [IES15b]. For example, Maurice et al. [Mau+17] propose reverse engineering of the LLC slice addressing function of various Intel microarchitectures and use it to speed up the eviction set construction with large pages when the mapping from virtual to physical addresses is unknown.

2.6.3 Dynamic Approach

The attacker has no knowledge about the LLC slicing hash function or physical addresses; whereas the static method uses the reverse engineered hash and (partial) information about the physical addresses to compute eviction sets. Indeed, as we show in Figure 2.17, the upper address bits are used to determine the set and the slice are contolled by the operating system through its page allocation mechanism, and thus they can be knwon only to priveleged processes. On Linux, it has been shown that the buddy allocator can be tricked into allocating continuous chunks of memory [Van+16], but page allocation can usually be viewed as a random oracle. In the dynamic approach, the attacker starts by collecting an initial set of virtual addresses, which is called *candidate set*.

Candidate Set

A candidate set is a set of n randomlly collected virtual addresses that has a good chance to contain enough congruent addresses to evict the target addresse x.

Suppose that a process has no control over virtual addresses (e.g., the address mapping is randomized); in that case, addresses have to be selected randomly. For a candidate set to be used as an eviction set, the collection of n virtual addresses must satisfy the eviction set definition. Let suppose X be a random variable representing the number of congruent addresses found in a set of n virtual addresses. According to [VKM19], with $p = \frac{1}{S}$ (where S is the number of sets), the probability of having at least a congruent addresses in a candidate set of size n is given by:

$$\Pr(X \geq a) = 1 - \sum_{i=0}^{a-1} \left(\begin{array}{c} n \\ i \end{array} \right) p^i (1-p)^{n-i} \tag{2.1}$$

To estimate the average latency of finding a candidate set, we assume that adversary collect a fixed size of for the candidate set. The adversary repeatedly collects a set of addresses randomly and verifies it until a candidate set is found. In the verification step, the adversary accesses these addresses after accessing the target address, and then verifies whether the target address was evicted. The authors in [SL19] estimate the average latency for finding a candidate set with n addresses by using the number of memory accesses as a time measure, which can be computed as $T_c(n)$:

$$T_c(n) = \frac{n}{\Pr(X \geq a)} \tag{2.2}$$

2.7 Eviction Set Optimization Algorithms

Algorithms for constructing eviction sets aim to find the congruent addresses in a candidate set and discard all other addresses that do not affect the eviction of the target address x (non-congruent addresses). In this section, we describe the different state-of-the-art eviction set construction algorithms.

2.7.1 Single Address Elimination Algorithm

The *single address elimination* algorithm (SAE) was introduced by Yarom et al. in [Liu+15] to perform Prime+Probe attacks on last-level caches (see Algorithm 4). The algorithm takes a virtual address x and a candidate set C as input. For each element $c \in C$, the algorithm tests whether the candidate set is still an eviction set without c (lines 3-4). When the test succeeds, the address c is removed from C (line 6). When the test does not succeed, the address c is considered necessary for E to be an eviction set. Then, the address is added to E and the algorithm keeps iterating. The algorith stops when enough (at least W) congruent addresses are founded (line 2).

Algorithm 4: The single address elimination algorithm

Input: C=candidate set, x=target address

Output: E=minimal eviction set

```
1 E ← { } ;
2 while |E| < W do
3 |   c ← pick one address from C;
4 |   if E ∪ C\{c} does not evict x then
5 |   |   E ← E ∪ {c};
6 |   end
7 |   C ← C\{c};
8 end
9 return E;
```

The single address elimination method construct an eviction set in $\mathcal{O}(n^2)$ memory accesses for a candidate set size of n elements. This means that the number of memory accesses grows quadratically with the size of the candidate set.

2.7.2 Group Testing Algorithm

The *group testing* (GT) algorithm [VKM19] is an optimization of the single address elimination method that reduce the time complexity to line a (see algorithm 5). In the group testing algorithm, the initial candidate set is split into l groups (line 2). If $l > W$, it is guaranteed that $l-W$ groups can be removed in each while iteration. vila et al. [VKM19] set l to $W + 1$ to maximize the group size. The algorithm iterates over all groups and tests whether E without the group still evicts x (lines 3-4). The group is removed from E (line 5) when this is true. In the other case, the algorithm tests other groups until it finds one that satisfies this requirement. Then, E is divided into $W + 1$ groups again, and the same process starts over. This is repeated until there are $W + 1$ elements left in

E (line 1).

Algorithm 5: The group testing algorithm

Input: E=candidate set, x=target address

Output: E=minimal eviction set

```
1  while |E| > W do
2      {T_0 ... T_{W+1}} ← split E into W + 1 groups;
3      for i ← 1 to W + 1 do
4          if E\{T_i} evict x then
5              E ← E\{T_i};
6              break;
7          end
8      end
9  end
10 return E;
```

By using the GT algorithm, an entire group of addresses can be eliminated per iteration instead of just one element, as in the single address elimination algorithm. The group testing method requires $\mathcal{O}(W^2 n)$ cache accesses as described in [VKM19].

2.7.3 Prime–Prune–Probe Algorithm

The *Prime–Prune–Probe* (PPP) [Pur+21] algorithm has shown that faster eviction set construction is possible. As shown in Algorithm 5, the PPP technique begins with the *Prime* step, where the attacker generates a set of random addresses and accesses them to fill in either a subset or the entire cache, i.e., the candidate set. To eliminate false positives, the attacker accesses the candidate set again in the *Prune* step and removes any addresses that result in a high access latency. This process is repeated many times to remove self-evicted addresses until all remaining addresses in the Prime set are concurrently cached. Regarding the number of cache accesses, authors in [Pur+21] demonstrate that the pruning process is generally completed in fewer than two rounds. After pruning, the PPP returns a set of n' addresses, where n' is less than or equal to the initial size n. The pruning step aims to absorb noise so that congruent addresses can be more easily identified in the Probe step. In the *Probe* step, n' addresses are reaccessed, and then we check if an eviction has occurred. These three steps are repeated until enough

congruent addresses are found and added to E.

Algorithm 6: The Prime–Prune–Probe algorithm

Input: n=candidate set size , x=target address

Output: E=minimal eviction set

```
1  E ← { } ;
2  while True do
3      C ← generate n random addresses;
4      /* Prime step                                          */
5      Access(C ∪ E);
6      /* Prune process                                       */
7      while They are no self-eviction in C ∪ E do
8          for each c ∈ C do
9              if get access timing of c > threshold then
10                 C ← C\{c} ;
11             end
12         end
13     end
14     /* Probe step                                          */
15     for each c ∈ C do
16         if get access timing of c > threshold then
17             E ← E ∪ {c} ;
18         end
19     end
20     /* Test eviction                                       */
21     if E evict x then
22         break ;
23 end
24 return E;
```

This algorithm can be used to find eviction sets in LLCs using different replacement policies. The number of cache accesses is estimated as $\mathcal{O}(SW)$ when the LLC uses an LRU as the replacement policy, i.e., the smallest of the three fast algorithms. However, when using a random replacement policy, the number of cache accesses increases to $\mathcal{O}(W \times n)$. This increase in complexity is due to the candidate set size after the pruning step being much smaller than the cache size SW. Using a small set of addresses reduces the chance of finding a congruent address in each PPP iteration.

2.8 Defenses against Cache-based Side-Channel Attacks

To defeat cache attacks, extensive research on techniques to identify and mitigate information leaks through memory access patterns and timing side channels have been proposed. These techniques can be applied at the different levels: the *application level*, the *system level* (e.g., operating system or hypervisor), and the *hardware level*. However, many solutions only protect against a subset of attacks and impose performance overhead. The ideal countermeasure should provide robust security with minimal performance degradation while being scalable and applicable to existing infrastructures. This section includes an overview of the fundamental approaches, to protect sensitive information against CSCAs.

2.8.1 Application level

Disabled Cache

Osvik et al. [OST06] is proposed to disable caching to prevent information leakage through cache memories. This solution requires modifying the code of sensitive applications to avoid the attacker process to monitor memory accesses that rely on sensitive information. This eliminates the timing variation as long as data is loaded from main memory. Crane et al. [Cra+15] verified the feasibility of this approach by measuring the performance of AES in `libcrypto` by marking the lookup tables as uncachables. They observed that disabling caching a single lookup table, the AES performance degrades significantly ($\approx$ x75 slower than expected).

Constant Time.

Bernstein [Ber05] proposed manually changing sensitive programs in such a way that they run in constant time. For example, in a cryptographic context, this countermeasure consists at ensuring that the memory cache accesses and the execution time of cryptographic algorithm should be independent of program secrets. Barthe et al. [Bar+14] prove that constant-time execution do not leak sensitive information through cache accesses. Agosta et al. [Ago+07] claimed that timing side channels could be eliminated if there are no memory accesses or branches that create side effects through timing channels. However, developping programs in constant-time is often infeasible and hard to verify. To that effect, various works [Alm+16; Doy+15; KMO12] proposed differents tools that verify the timing side effect of a given implementations, which guide

constant-time programming. Although constant-time approach eliminates information leakage through timing side-effets, the technique must be applied to each application and each hardware platform. Moreover, constant-time applications usually result on less efficient implementations [BK07].

2.8.2 Operation system or hypervisor level

Cache Flushing

Another solution is to partially or completely flush the cache contents at each context switch [OST06; Bar+14; ABG10]. This technique degrades performance, but it can eliminate timing side channels. Hence, the attacker will not be able to detect temporal variations since all data is retrieved from main memory and has a long access time. In addition, the scheduling period between two context switches plays an important role in the performance of this technique. If the scheduling period is long, the application can fill the cache and take advantage of it. However, if the scheduling period is short, the application needs to access the data from the main memory frequently because the caches are constantly flushed, which increase the performance overhead. Moreover, in [VRS14], the authors studied the impact of flushing the L1 cache on a 6-core Intel Xeon E5645 processor. They measured 8.4 μs of direct cost, resulting in an overall 17% increase in latency in the ping benchmark, which issues the ping command in 1 ms intervals. However, for larger higher-level caches (e.g., L2 and LLC), flushing likely results in significant performance degradation.

Disable Page Sharing

Because sharing-based CSCAs depend on shared memory pages, preventing page sharing can mitigate these attacks. VMware Inc. [Bas14] recommends disabling the transparent page sharing feature [Wal02] to protect caches from cross-VM sharing-based CSCAs. Zhou et al. [ZRZ16] proposed CacheBar as a software solution to prevent concurrent access to physical shared pages by automatically detecting such access and creating multiple copies; this technique is called copy-on-access. They showed that allocating a copy of the shared page for each security domain prevents the Flush+Reload technique and its variants.

Noise Injection

A prerequisite for a successful CSCAs, is that the attacker be able to make fine-grained measurements to observe time variation. If the timing measurements becomes noisier, for example, due to the introduction of dummy instruction or random delays, the attacker observation become more difficult. In [MDS12], the authors explore this technique by mask the value returned by a clock counter register so that only the upper n bits are returned. They demonstrated that adding noise to the timing measurement can defeat CSCAs. On the other hand, noise can also be generated at an access level. For example, additional code that performs random accesses to memory or randomly evicting some cache lines can make an attack more difficult [Pag03]. For example, Brickell et al.[Bri+06] suggested using compressed and randomized lookup tables for the AES algorithm. In [Muk+20], the authors proposed a generic solution called Flush+Prefetch technique, which obfuscates the memory access behavior of a secure application using independent threads that randomly access the secure application's memory. In this case, the attacker cannot distinguish between the access due to the victim process or to the Flush+Prefetch technique.

Process Isoaltion

A common system-level countermeasure is isolating the various processes, so they do not share the resources that cause information leakage. In this strategy, the operating system or hypervisor, allocate different parts of hardware resources to isolate sensitive applications. For example, page coloring [KH92] is a purely software-based solution. In this technique, the operating system maps virtual page addresses to physical pages such that they are mapped to different sets in the LLC. This technique was initially proposed to improve the overall system performance [Ber+94], by reducing cache conflicts between different virtual machines. Shi et al. [Shi+11] proposed a cache coloring solution for protecting sensitive applications from interference with others in the LLC and thus protect applications against LLC CSCAs. Other works [ZRZ16; GZ14; KPM12] implemented a similar protection mechanism in different platforms; they benchmarked only a small performance impact. However, the different work demonstrate that memory overhead is significant. This is due to the use of colors in physical addresses, and thus large portions of physical memory have to be assigned to the same virtual machine or process to provide strict cache coloring without other virtual machines or processes working in the same cache set.

Detection

A detection mechanism can also be used to detect cache-based side-channel attacks at the system level. They are used to detect malicious behaviour in the system and notify it to take security action. The detection mechanisms are usually designed to observe the side effects in the hardware layer during the attack. For example, Hardware Performance Counters (HPC) are commonly used to detect the side effects for each cache layer, such as the number of cache misses/hits, accesses, etc. For example, Chiappetta et al. [CSY16] implement a software detection mechanism that uses HPC counters to measure LLC and L2 side effects. The authors in [Pol+21] propose a machine learning-based detection mechanism strategy that targets Instruction Per Cycle (IPC) and some other side effects of the memory hierarchy. Both mechanisms are runtime detection mechanisms. MASCAT [IES16] is an offline detection solution that statically analyses binary Elf files. In these files, MASCAT searches for features resulting from micro-architectural attacks. MASCAT can detect cache-based side-channel attacks.

2.8.3 Hardware level

Instruction set

Intel recently introduced a new hardware instruction to mitigate CSCAs on cryptographic implementations. They extended their ISA with a new hardware instruction, AES-NI [HC12], that can perform encryption and decryption without using cache memories. Thus, the AES algorithm performs in constant time. However, this extension is implemented only to support the implementation of AES. Therefore, we still need defenses to protect other cryptographic algorithms. In addition, the use of some instructions makes it easier for the attacker to leak sensitive information, such as the use of the `clflush` instruction at the user level. Unlike x86 processors, the flush instruction is reserved for more privileged software on ARM processors. Therefore, the Flush+Reload technique is not feasible. This solution is inefficient because the attacker can still use the Evict+Reload technique to leak the victim's sensitive information.

Partitionning-based Caches

Intuitively, cache partitioning [DFS20; Kir+18; Gru+17; Wan+16; KPM12; Dom+12; SK11; Pag05] used to be an effective countermeasure against CSCAs. It aims to avoid

sharing critical resources with other processes and prevent them from monitoring critical resources. This is done by dividing the cache memory into different regions. This mechanism prevents internal timing, but external timing attacks are still possible since measuring the execution time of the protected program can reveal some patterns about the victim's memory accesses. Also, other applications cannot use the reserved cache blocks, which reduces the cache size, and thus increase the performance overhead.

The simplest way to achieve cache partitioning is to divide the cache memory statically into multiple partitions based on the cache ways and assign these partitions to different processes, as in Statically-Partitioned Cache [HL17] and SecVerilog Cache [Zha+15; ZAM12]. However, static partitioning significantly reduces the effective cache capacity for each running process, resulting in a huge performance degradation. Therefore, a more promising direction is dynamic partitioning.

To address the drawbacks of static partitioning, Wang et al.[WL07] proposed a Partitioned-Locked (PL) Cache that uses a fine-grained dynamic cache partition. This approach aims to lock specific cache lines to avoid interference between different processes, allowing sensitive data such as AES lookup tables to selectively and temporarily locked into the cache. Once a particular cache line is locked, it cannot be removed by another process. To implement this approach, the authors add a protection bit to each cache line that indicates whether it needs to be locked. In addition, the authors propose two solutions to lock a cache line: either by adding user-level lock and unlock instructions to the ISA or by using more privileged software, such as the operating system. ARM processors also provide a similar mechanism, called AutoLock [Gre+17], to prevent cache lines from being removed. Unlike the PL cache, when the AutoLock mechanism is enabled, the lines in the LLC are protected only if they are in a core-private cache level. The protection is immediately broken as soon as these lines are removed from the core-private caches.

DAWG [Kir+18], SecDCP [Wan+16], and Non-Monopolizable (NoMo) Cache [Dom+12] are dynamic way-partitioning techniques. In non-monopolizable cache, authors modify the replacement policy to reserve a number of cache lines in each set of an associative cache for each running thread. As a result, the victim data cannot be removed from the protected cache lines by the co-executing threads. DAWG [Kir+18] introduces some modifications to fully isolated side channels via cache line states, coherence states, and replacement information variations. However, when the number of cache ways is less than the number of processes or security domains, DAWG is forced to reallocate the different cache lines from one partition to another. This reallocation can leak information about the victim's memory accesse. Other dynamic partitioning techniques, such

as SecDCP, and NoMo caches have similar security issues.

CATalyst [Liu+16a] is another partitioning technique that uses Intel Cache Allocation Technology (Intel CAT) [Int21] to partition LLC cache between different users. Intel CAT is a hardware support that implements the cache partitioning mechanism. Intel CAT was originally intended to improve performances, not to mitigate CSCAs. CATalyst takes advantage of intel CAT and splits the LLC cache into a secure partition and a non-secure shared partition. The secure partition is reserved for addresses that could reveal sensitive information. Cache interference within the secure partition is blocked through page coloring. The non-secure shared region shares normal resources with other running applications. To secure sensitive data, a user-level program allocates secure pages and preloads the data into the secure cache partition. Lui et al. [Liu+16a] demonstrate that the CATalyst is effective in protecting square-and-multiply algorithm present in `GnuPG 1.4.13`.

Randomization-based Caches

Randomization-based techniques add non-determinism and noise to the behavior of the cache to obfuscate the observation of side channels and make sensitive information extraction more difficult for an attacker. Address randomization is a family of randomization techniques that randomize the the address-to-cache mapping. The idea of this approach is prevent an attacker to construct a set of congruent addresses to attack a particular cache set. Therefore, randomization-based architectures can more efficiently mitigate conflict-based CSCAs. Unlike partitioning-based approaches, the randomization approach also enables cache sharing among different processes, which helps ensure that performance is not degraded. In a *static randomization* scheme [Liu+16b; WL07], the address mapping is fixed, while in a *dynamic randomization* [Wer+19; Qur18], the mapping can change over time. Existing address permutation functions rely on lookup tables [Liu+16b; WL07], hashing schemes [Wer+19], or block ciphers [Qur18].

Wang and Lee [WL07] proposed an RP cache to staticlly randomize cache mapping by using an indirection table. This table stores the correspondence of the cache sets. The address set is first used to index the indirection table for each cache access. Then, the returned cache set is used to access the cache memory. Such a design is linearly scalable in terms of cache sets and the number of concurrent applications. Therefore, it is not suitable for larger caches. Moreover, the effectiveness of table-based randomization schemes depends on the OS to assign different hardware process identifiers and classify applications into protected and unprotected applications.

Recent studies have proved that static randomization does not defeat conflict-based cache attacks due to advanced eviction set construction algorithms [VKM19]. For this reason, dynamic randomization [Qur18; Wer+19] has been introduced in last-level caches. In the rest of this work, we use the term *remapping* to refer to a change in the address mapping. The interval in which the mapping is changed determines the time window that is available to an attacker to perform the entire attack. However, there is an overhead associated with changing the mapping; in addition to the misses generated, multiple write-backs and data moves in the cache may be required. To minimize the impact on performance, it is important to choose the highest possible remapping interval that ensures safety.

CEASER [Qur18] was proposed to secure the last-level cache with a keyed indexing encryption function. It also requires dynamic remapping to its encryption function to change the mapping and limit the time interval in which an attacker can construct a efficient eviction set. It has been shown that CEASER can defend against attacks that use a eviction set construction algorithm with a complexity of $\mathcal{O}(n^2)$ when the remapping rate is 1% (on average, one line remapped per 100 accesses). However, for eviction set construction algorithms with a complexity of $\mathcal{O}(n)$ [Qur19] (e.g., the group testing algorithm), the remapping rate must increase to 35–100%, which results in high-performance overheads. To mitigate these algorithms, the same authors have proposed CEASER-S [Qur19], which uses a skew-associative cache where each cache way uses a separate function to compute the cache set in this way. In CEASER-S, the cache ways are divided into multiple partitions and each uses a different encryption key. Thus, a cache line for each partition is assigned to a different cache set, making the construction of an efficient eviction set more complicated.

ScatterCache [Wer+19] was also proposed to achieve randomization mapping using a key-dependent cryptographic function. Moreover, its mapping function depends on the security domain, where the indexing of the cache set is different and random for each domain. Therefore, a new key may be required at certain intervals to prevent the attacker from creating and using an eviction set to collide with the victim's access. However, ScatterCache, which claims to tolerate years of attacks, has been broken by more advanced eviction set construction algorithms [PV19]. Mirage [SQ21] attempts to overcome the weaknesses of ScatterCache by preventing faster eviction set construction algorithms. Mirage is a fully associative cache that uses pointer-based indirection to associate tags with data blocks and vice versa (inspired by V-way Cache [QTP05]). The eviction candidates are randomly selected from all of the lines in the cache that are to be protected against set conflicts.

PhantomCache [Tan+20] relies on address randomization by using an efficient hardware hash function and XOR operations to map an incoming cache block to one of eight randomly selected cache sets, increasing the associativity to $8 * W$. Unfortunately, this requires accessing $8 * W$ memory locations for each cache access to check whether an address is stored in the cache, resulting in a high-power overhead of 67%. The authors show that PhantomCache can safely defeat the group testing algorithm; however, its effectiveness over the fastest eviction set construction algorithm needs further investigation.

Randomization-based defenses, such as CEASER/-S [Qur18; Qur19] and ScatterCache [Wer+19], claim to thwart conflict-based cache attacks. ScatterCache [Wer+19] even argues that dynamic remapping is unnecessary due to the complexity of using a skew-associative cache. Song et al. [Son+20] identified the flawed hypothesis in the implementations of ScatterCache and CEASER-S [Qur19]. By exploiting these flaws, the authors succeeded in constructing eviction sets. Both caches are also vulnerable to cryptanalysis, which can be used to construct eviction sets. For example, Bodduna et al. [Bod+20] identified invariant bits in the encrypted address that was computed in CEASER/-S [Qur19]. The authors show that these bits can be managed to construct a valid eviction set, even when the mapping changes.

Bao and Srivastava [BS15] exploited the lower latency of 3D integration technology to implement the random eviction strategy to mitigate against cache-based side-channel attacks. The authors show that such a technique provides inherent security benefits within a 3D cache integration. They investigated a random eviction component that evicts one cache line every five cycles. On average, their experimental results show that such techniques reduce the performance overheads from 10.73% (in a 2D configuration) to around 0.24% (in a 3D integration). The performance overhead is more negligible in 3D integration since the penalty for a cache miss is smaller.

Hardware vulnerability identification

Several approaches have been developed to assess the security of cache architectures. One study [ZL14] uses mutual information to measure potential side-channel leaks in the cache memory. He et al. [DXS19] proposed an approach to quantitatively evaluate the resilience of caches against CSCAs. They construct a probabilistic information flow graph to model cache interference and calculate the probability of success for different attack's steps, and also the probability of success of the entire attack. However, both approaches [DXS19; ZL14] only evaluate the security of caches against a limited

set of cache leakage techniques, including Bernstein's attack, Evict+Time, Prime+Probe, Flush+Reload, and Cache Collision attack. In a separate work, Demme et al.[Dem+12; Dem+13] proposed the Side-channel Vulnerability Factor (SVF), a metric to quantitatively measure side-channel leakage by measuring the Pearson correlation between the attacker's observations and the victim's execution trace. Meanwhile, CSV [Zha+13] improved the SVF metric and uses a direct correlation between the attacker and victim traces. Further analysis of secure cache architectures was conducted using computational tree logic [DXS18], the three-stage model [DXS19], flow graphs [Wan+20], and neural networks [ZZL18].

Prunal et al. [Pur+21] also attempt to quantitatively analyze the security of randomized caches. In another concurrent work [Bou+20], Bourgeat et al. analyze the end-to-end security of secure caches. The timing-channel toolkit Mastik [Yar16] was presented to experiment micro-architectural side-channel attacks for x86 processors, including Prime+Probe, Flush+Reload, and Flush+Flush techniques. Lipp et al. [Lip] developed the Libflush framework, which implements the nedded primitives to facilitate the development of CSCAs on ARM and x86 processors.

2.9 Summary and Conclusion

The micro-architectures of modern processors are the result of several decades of work. They are complex systems that use many optimization mechanisms. Thus, the execution of a single instruction leads to several hardware operations. The main goal of most of these mechanisms is to improve performance, increase the throughput of executed instructions, and thus increase the clock frequency. For example, cache memories are used to significantly reduce memory access time. While these optimization mechanisms focus on the execution of a single program, others such as multithreading or multicore organizations enable the optimization of cases where multiple independent programs are to be executed. The micro-architecture is then designed to allow the simultaneous execution of different programs.

These various optimization mechanisms were developed without consideration of safety requirements. We have shown in this chapter that basic security mechanisms such as privileges are no longer sufficient. Since they can be bypassed by software attacks that exploit the micro-architecture through side channels, mainly attacks that target cache memory, these attacks exploit timing variations during program execution. Exploiting these variations can lead to a malicious process of obtaining information about the execution of other processes. This is possible because the victims and the

attacking processes share resources. The shared hardware resources thus pose an isolation problem between different users.

This chapter also discusses the strategies that can be used to counter problems with timing variations and conflicts within caches. The first category of countermeasures is dedicated to the management of shared resources. The proposed work focused mainly on recovering the isolation specification at the hardware by submitting new mechanisms for managing shared resources or partitioning hardware resources. While this strategy is effective, it only partially solves the isolation problem. This is because a malicious process sharing the same SoC with the victim can still exploit the total execution time or the number of generated misses. In addition, partitioning resources leads to performance degradation, as each user has less cache memory available to execute their program.

The second category of countermeasures aims to limit the leakage caused by the caches by inserting noise to make it harder to exploit the leaks. The idea behind this solution is to disrupt the attacker's observations while ensuring resource sharing. The proposed changes are based on two axes: changing the time perception to disrupt the attacker's measurements to collect timing information that is no longer reliable or changing the hardware architecture of the caches to achieve non-deterministic behavior. Unlike partitioning, randomization preserves the idea of sharing resources among processes to not degrade system performance. However, the security evaluation of this class of countermeasures has not been addressed in depth in the literature. In addition, randomization in LLC has mainly been used, but its implementation at lower levels of the memory hierarchy has not been evaluated. In this sense, the work in this manuscript should be understood.

In the remainder of this work, we focus on the issue of cache randomization. It affects almost all of the cache-based attacks presented above, whether they are attacks that exploit shared memory or attacks that exploit shared resources based on cache contention. We also aim to develop a memory hierarchy with security mechanisms against cache attacks. The work done in this thesis is organized on two axes.

The first axis provides a deep understanding of cache attacks. The goal is to define the limits of secure architectures based on the randomization technique. A choice will then be made to determine the limitations of using randomization at different cache levels. The second focus of work is to propose countermeasures at each cache level based on the studies conducted in the first focus. Finally, the performance of our countermeasure will be evaluated to make it scalable on different types of processors.

3

Practical analysis of cache attacks

In this chapter, we describe the Micro-Architectural Analysis Toolkit (libMAAT), developed as part of this Ph.D. that contains different primitives needed to implement micro-architectural side-channel attacks on x86 and ARMv7, ARMv8 as well as RISCV architectureS. It contains different modules that can be used to build micro-architectural attacks. We developed this library to quickly prototype micro-architectural attacks and evaluate our secure architectures' resistance against realistic attacks. As proof of concept, we implemented two rounds of AES attacks using libMAAT. We target the AES T-Table implementation available in OpenSSL using the Evict+Reload technique.

3.1 Motivations

Timing attacks on micro-architecture via side channels exploit interference between internal processor components to leak sensitive information through timing variation. While such attacks are simple in theory, practical implementations are often tricky and require a deep understanding of poorly documented processor functions.

Two popular libraries implement cache attacks: libflush [Lip] and Mastik [Yar16]. However, those frameworks do not offer the flexibility required for our experiments. Mastik, for example, can only be used on Intel x86 micro-architectures. On the other hand, Libflush is a concurrent library that supports primitives for cache-based side-channel attacks on ARMv7 and ARMv8 micro-architectures. However, Libflush requires privileged access to construct cache attacks, which is more challenging and does not reflect the attacker model.

For this reason, we have developed Micro-Architectural Analysis Toolkit (libMAAT), a library that contains several primitives needed to create micro-architecture side-channel attacks on the x86, ARMv7, ARMv8, and RISCV architectures. This library is intellectual property that belongs to the CEA company Therefore, the sources of libMAAT will not be released on open source platforms. We developed this library to quickly prototype attacks and evaluate the resilience of our secure architectures against realistic attacks. In doing so, we have found that the characteristics of modern processors must be considered in order to perform successful cache-based side-channel attacks. We therefore address the following questions:

- *How should we approach the complexity of modern processors and the undocumented components used?*
- *Can we map attack and victim process on the same core in a multicore environment?*
- *How can we measure precise timing from user space when out-of-order execution is supported?*
- *What are the possible sources of noise?*

LibMAAT contains several modules that can be used to create attacks on micro-architectures. LibMAAT is a modular library that allows all available options (such as the eviction strategy, time sources, etc.) to be changed at execution time.

3.2 LibMAAT: Micro-Architectural Analysis Toolkit Library

In this section, we describe the design philosophy and modular architecture of libMAAT. In a nutshell, the framework contains a collection of building blocks for micro-architectural attacks. In our opinion, this gives security researchers more freedom in designing the skeleton of the attack models. The organization of libMAAT modules is show on Figure 3.1.

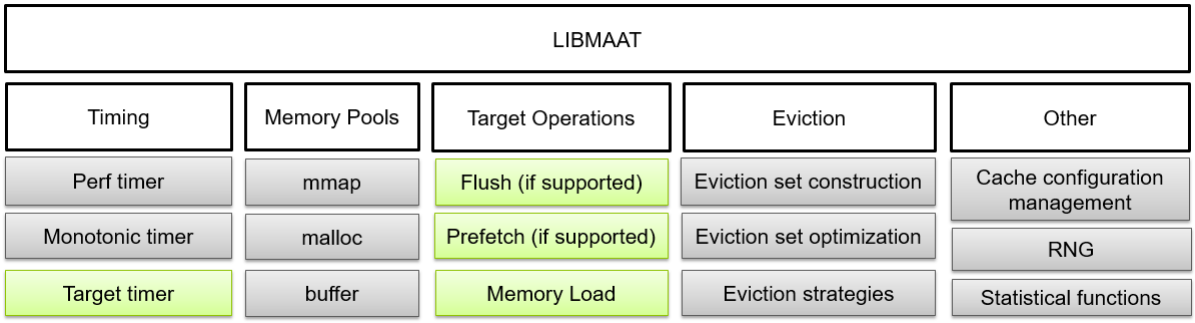


Figure 3.1 – Overall architecture of libMAAT

- **Timing measurements module:** This module contains different techniques to measure the execution time from privileged and unprivileged execution levels.
- **Memory pools module:** This module offers different techniques to allocate the memory area needed, for example, to find congruent addresses.
- **Cache eviction module:** In this module, we implement the different strategies in the state-of-the-art to construct an efficient eviction set for both root and user processes.
- **Cache configuration module:** This module is responsible for finding the memory hierarchy configuration dynamically.
- **Statistical functions module:** This module offers some statistical functions that may be needed to analyze the side effects observations. In addition, it also contains a specific module to generate pseudo-random numbers.

3.2.1 Targeted Micro-Architectures

Micro-architectural attacks are strongly dependent on the instruction set architecture (ISA). Security researchers must adapt their code to each architecture. However, the high-level description of the attack is generally the same across all architecture. For example, cache-based side-channel attacks can be expressed in terms of memory loads, flush operations, instruction barriers, and timing measurements. LibMAAT aims to provide a framework that can be used to write portable micro-architectural timing side-channel attacks. Thus, the libMAAT framework contains the abstraction layers for several ISAs, including x86, ARMv7, ARMv8, and RISC-V.

3.2.2 Timing Measurements

For observing small variations in memory load latency, high-resolution timers are required. LibMAAT provides different time sources available, either in privileged or unprivileged execution levels. In this section, we describe the different implemented time sources. The choice of the time source can be made at the execution time, which does not require recompiling the codes. This is done by calling the libMAAT function `uint64_t maat_timer_sample(maat_timer_t* timer)`, where the parameter `timer` is the pointer to the timer source. They are three specific time source implemented in libMAAT: *cpu_counters*, *perf*, and *monotonic*.

CPU hardware registers

Processors generally offer high-precision cycle counters that hold the number of CPU cycles since reset. This is, in most cases, the highest resolution timer available. On x86, the `rdtsc` and `rdtscp` instructions provide access to those cycle counters. Unlike `rdtsc`, `rdtscp` is a serialization call that can be used to prevent the CPU from reordering it. However, `rdtscp` call is only avaible in moden CPUs. RISC-V micro-architectures support also an equivalent user-space instruction `rdcycle` to retrieve the number of cycles from the CSR register.

On modern processors, timing measurement using CPU counters is challenging because of the optimizations that modern processors provide. One of the optimizations is the presence of superscalar and out-of-order execution, which can be used to optimize the penalties due to the different instruction latencies. Unfortunately, this feature does not guarantee that the temporal sequence of the single compiled C instructions

will respect the sequence of the instruction themselves as written in the source C file. When we call the `rdtsc` instruction, we pretend that that instruction will be executed exactly at the beginning and at the end of the code being measured (e.g., we don't want to measure compiled code executed outside of the `rdtsc` calls). According to the Intel specification [Pao10], the best way to call serializing instruction befor calling `rdtsc`. A serializing instruction is an instruction that forces the CPU to complete every preceding instruction of the C code before continuing the program execution. By doing so, we guarantee that only the code that is under measurement will be executed in between the `rdtsc` calls and that no part of that code will be executed outside the calls. This is done in the manner summarized in algorithm 7. Accordingly, the natural choice to avoid out-of-order execution would be to call the `cpuid` instruction just before both `rdtsc` calls. As shown in algorithm 7, the `cpuid` instruction is executed first to ensure that all previous instructions have been completed before retrieving the current timestamp. Otherwise, instruction overlaps may slow down the measured code in a non-deterministic manner, as more or fewer additional instructions will be measured depending on the execution state before the measurement begins. Once the code is measured, we use `rdtscp` instead of `rdtsc`, since all measured operations must complete before the timestamp is read a second time instead of being executed in parallel. The `rdtscp` instruction waits until all previous instructions have finished before retrieving the value of the timestamp counter. In the last line, we again use `cpuid` to prevent subsequent instructions from starting their execution while the timestamp is being read, thus slowing down `rdtscp`. Even though `cpuid` has a high variance, this does not affect our timing since we use it before and after sampling the timestamp.

Algorithm 7: Timing measurement code following Intel benchmarking white paper [Pao10]

```
1 cpuid
2 start ← rdtsc
3 /* Measured function                                    */
4 end ← rdtscp
5 cpuid
6 return end - start
```

In ARM processors, there are no similar unprivileged instructions for timing on either the ARMv7 or ARMv8 architectures. However, they provide a performance monitoring register called Performance Monitor Cycle Count Register (`PMCCNTR`) that counts processor cycles since reset. For example, Figure 3.2 illustrates the access time to an address in the cache or main memory of a dual-core Cortex-A9 measured by the `PMCCNTR` register. As a result, cache hits and cache misses are easy to distinguish. Although these

measurements are fast and accurate, access to these performance counters is restricted to kernel space by default. However, the User Enable Register (`PMUSERENR`), which is writable only in privileged modes, can be configured to allow access to the `PMCCNTR` in user space. Therefore, a kernel module and root privileges are required, making it difficult to access this timing source. The ARMv8-A architecture provides similar registers.

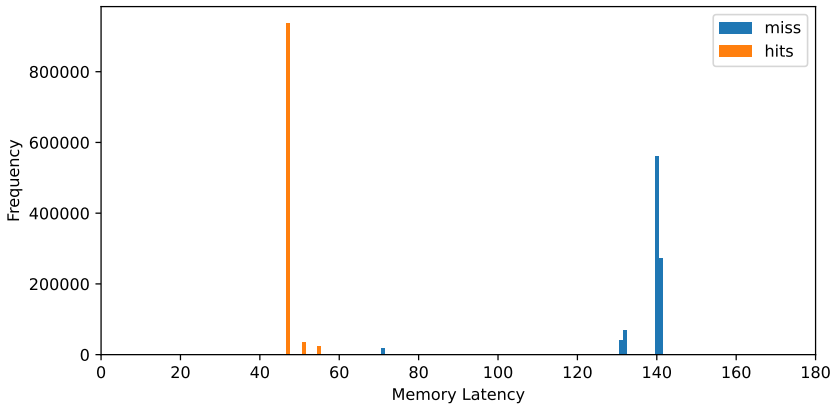


Figure 3.2 – Histogram of cache hits and cache misses measured cross-CPU on the dual-core Cortex-A9 using CPU counter to measure timing.

Perf tool timing source

Modern Linux distributions introduce the `perf` tool in the kernel part. It provides a powerful interface to instrument CPU performance counters and tracepoints regardless of the hardware used. It is also capable of sampling lightweight sampling. It is also included in the Linux kernel and is frequently updated and extended. In addition, the perf event system call can access such information from user space and provides accurate timing information like the privileged instructions described earlier (e.g., ARM).

POSIX timing sources

The `perf` interface described in the previous paragraph is enabled by default on most devices. However, perf is rarely enabled in production for security reasons. Therefore, we implement the POSIX function `clock_gettime()` to retrieve the time as an equivalent time source. It allows the measurement of time with a resolution from microseconds to nanoseconds. Figure 3.3 illustrates the histogram access timing, using the clock `CLOCK MONOTONIC` as the timing source. However, we note that small timing differences are no longer detectable and that cache hits and misses can still be distinguished.

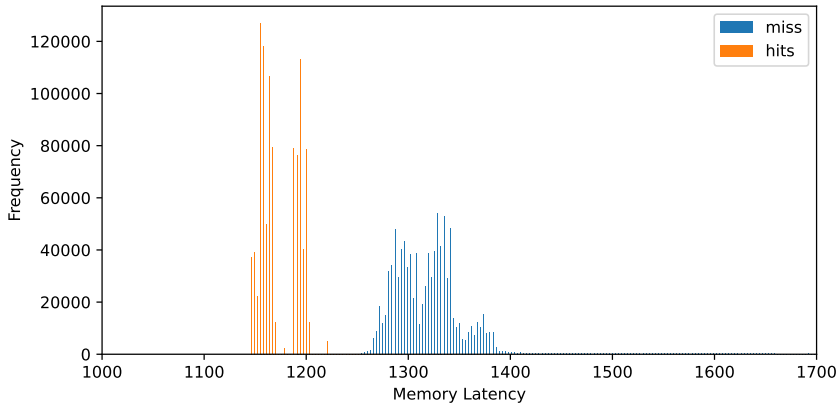


Figure 3.3 – Histogram of cache hits and cache misses measured cross-CPU on the dual-core Cortex-A9 using the `clock_gettime()` function to measure timing.

3.2.3 Cache Eviction

As discussed above, in some processors (such as ARMv7 and RISC-V), users have no access permission to cache flush operations, as these operations are privileged. In such a situation, an attacker can exploit cache eviction in order to exclude a specific cache line from the memory hierarchy. LibMAAT provides a three-stage eviction strategy for conducting a conflict-based attack: construction, optimization, and eviction.

Construction Step

The construction step provides different ways to create an eviction set, depending on the attacker's privileges. LibMAAT support three techniques to collect addresses: using a pool of ① virtual addresses, ② physical addresses, and ③ virtual addresses with huge pages.

LibMAAT collects a set of virtual addresses for an attack from user space. LibMAAT ensures that all collected virtual addresses have similar bits in the page offset field, such as bits used in the last-level cache to index the set. Thus, even though part of the set index is unknown from user space, libMAAT can collect many addresses to increase the probability of finding enough congruent addresses, thus evicting the target address.

If the attacker has root privileges, libMAAT provides the ability to use the physical addresses directly. LibMAAt consults */prog/pid/pagemap* file to get the informations about the translation of virtual addresses into physical addresses. LibMAAT offers the ability to use huge pages and transparent huge pages depending on the attacker's privileges.

The ability to use larger pages is to control more bits in physical addresses, thus facilitating the construction of eviction.

Eviction Set Optimization

For building an eviction set, the first step is to generate multiple addresses to increase the probability of finding enough congruent addresses in the candidate set. The set is then optimized by removing all addresses that do not contribute to the eviction of the target address. LibMAAT includes different algorithms for optimizing the size of the eviction set: the baseline [Liu+15] algorithm and the group-testing [VKM19] algorithms (see section 2.6).

Eviction Strategy

After creating an eviction set and optimizing its size, the final step is to access its addresses in a specific order to evict the target address. A linear implementation where the eviction set is accessed linearly leads to poor results due to undocumented used replacement policies. Although libMAAT supports several methods: ① *pointer-hunting*, ② *pointer-chasing*, and ③ *sliding window* eviction strategies.

- **Pointer-hunting strategy.** In this eviction strategy, the addresses of the eviction set are placed into a linked list (optionally, randomly permuted); the targeted cache set is later evicted by traversing this list.

- **Pointer-chasing strategy.** LibMAAT supports a pointer-chasing eviction strategy to reduce the impact of hardware prefetching. Tromer et al. [OST06] is the first to acknowledge the impairment of Prime+Probe attacks by CPU prefetching, where the prefetcher loads some addresses that can add noises to the attacker's observations. The authors tried to set up a linked-list structure with random ordering and use the pointer-chasing technique when accessing the eviction set memories to prevent the stream prefetcher from aggressively loading many cache sets due to the unpredictable access pattern. This technique is used in almost all known Prime+Probe implementations. Therefore, the eviction set is now represented as a linked list with two pointers to access their addresses. In the algorithm 8, two addresses are accessed at each iteration, with an offset O distance between both addresses.

Algorithm 8: Pointer-chasing eviction strategy

Input: C = eviction set

Input: O = offset

```
for i ← 0 to |C| do
    access(C [i]);
    access(C [(O + i) % |C|]);
end
```

•**Sliding window strategy.** LibMAAT provides the generalized *sliding window* eviction strategy proposed by Gruss et al. [GMM16], and Lipp et al. [Lip+16]. This eviction strategy yields a sequence of accesses in the form of a window that slides over all congruent addresses. It accesses the eviction set on a specific order and repetition, which can increase the probability of eviction under some unknown replacement strategies. The authors demonstrate that this strategy can successfully be applied to ARM processors. This strategy is illustrated in algorithm 9.

Algorithm 9: Sliding window eviction strategy

Input: C = eviction set

```
/* number of windows                                          */
for i ← 0 to N − 1 by L do
    /* number of repetitions per window                       */
    for j ← 0 to A − 1 do
        /* number of addresses per window                     */
        for k ← 0 to D − 1 do
            access(C[i+k]);
        end
    end
end
```

In the algorithm, N denotes the total number of generated windows, A defines the repetitions per window, and D denotes the number of addresses per window. The parameter L denotes the offset between two windows. We define the window sliding eviction strategy as the triple $\zeta - N - A - D - L$. This strategy requires an eviction set size of $N + D - 1$ congruent addresses. For example, the strategy $\zeta - 3 - 2 - 2 - 1$ uses an eviction set of 4 congruent addresses and accesses them in a specific order as illustrated in Figure 3.4.

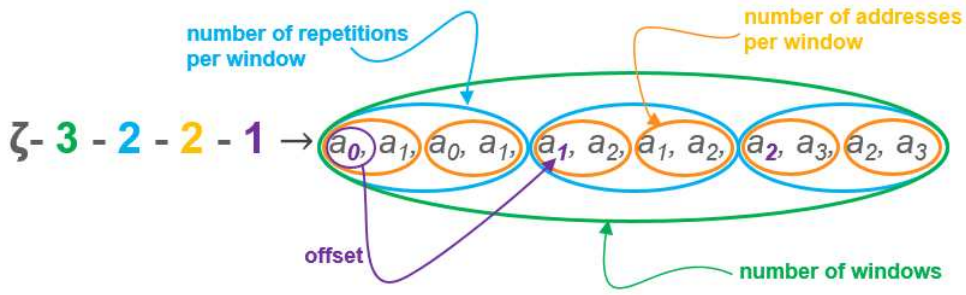


Figure 3.4 – Eviction set accessing sequence using window sliding eviction strategy $\zeta - 3 - 2 - 2 - 1$.

3.2.4 Memory pool allocation

The memory pool module manages allocated memory to create eviction sets. To collect addresses from the address space, libMAAT provides three supported allocation strategies: `static`, `mmap`, and `malloc` functions.

In the `static` approach, libMAAT allocate memory pool using static buffers. Otherwise, in the dynamic approach, LibMAAT provides two approaches `mmap` and `malloc` functions. The mmap system call requests the kernel to find an unused and contiguous area and is usually called to allocate a large set of addresses. Malloc, on the other hand, can be used to allocate a small set of addresses.

3.3 Evaluation of Eviction Set Construction

3.3.1 Target Platforms

In the following sections, we propose a practical evaluation of eviction set construction using the libMAAT. For simplicity, all evaluations will be performed on a single core. However, we use core 0 to run our tests for a system with several cores, avoiding noise due to multi-threading or multicore execution. Table 3.1 contains the parameters for the processors considered in this work.

Platform		**Raspberry Pi3B+**	**Raspberry Pi4**	**FPGA Zybo-7z20**
CPU		quad-core ARM Cortex-A53	quad-core ARM Cortex-A72	dual-core ARM Cortex-A9
Frequency		1.4GHz	1.5GHz	667MHz
L2 cache	Size	512KiB	1MB	256KiB
	Associativity	16	8	8
	Replacement policy	pseudo-random	pseudo-random	pseudo-random
Cache line size		64B	64B	32B

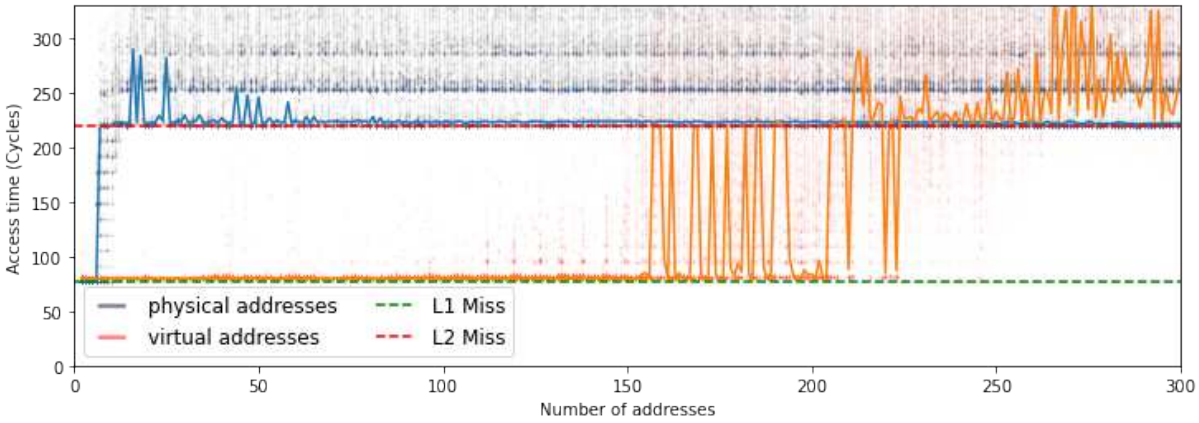
Table 3.1 – List of targeted platforms

3.3.2 Candidate Set Size Evaluation

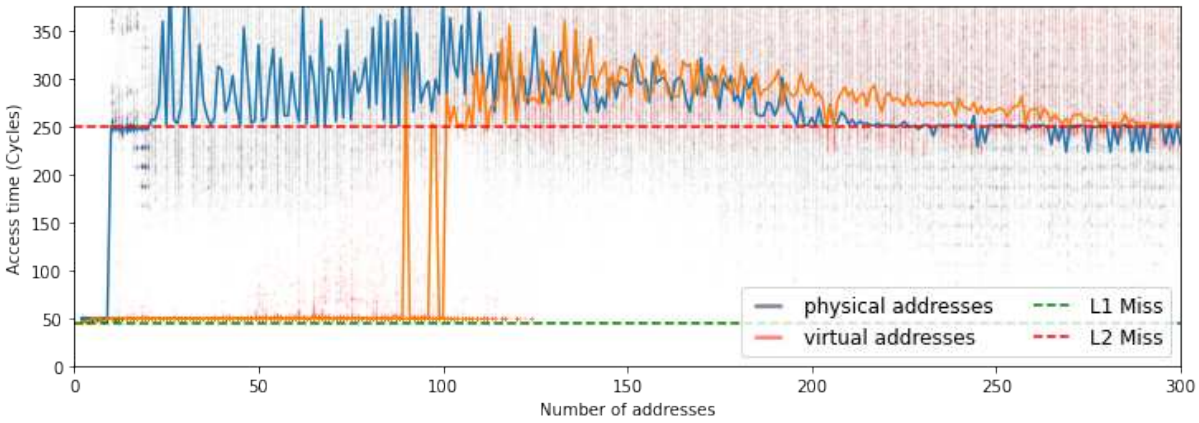
We evaluate the construction of the candidate set on different platforms described in Table 3.1. The idea is to evaluate the number of virtual addresses needed to find a candidate set that contains enough congruent addresses to evict a given cache line from each cache level. For each platform, the number of virtual addresses required to evict a target address T from the last level cache is determined as follows. First, we create a set of n virtual addresses, then access the target address to cache it. Then, we access the n virtual addresses and reaccess the target address by measuring the execution time. We repeat this test 1,000 times for different candidate set sizes. We also use physical addresses to create a candidate by accessing the pagemap information.

Figure 3.5 illustrates the results of the candidate set size evaluation for each platform. Each figure shows the reload time of each test and its average for each candidate set size. In contrast to creating candidate sets with virtual addresses, physical addresses are more efficient because the candidate set contains only congruent addresses. Thus, just above the cache associativity, a small number of congruent addresses need to be collected. On the other hand, it is noted that evicting a cache line using virtual addresses requires the collection of hundreds of addresses. This is due to the presence of non-congruent addresses that are not participating in the eviction of the target address. We found that eviction of a cache line on the Raspberry Pi3B and FPGA Zybo-7z20 platforms requires about 100 virtual addresses to create a candidate set. The Raspberry Pi4 platform, on the other hand, requires about 220 virtual addresses, twice as many as the other platforms. This increase in the number of virtual addresses is due to the associativity and the size of the last-level cache. By following the Equation 2.1, we conclude that increasing cache size increases the complexity of finding enough congruent addresses using virtual addresses.

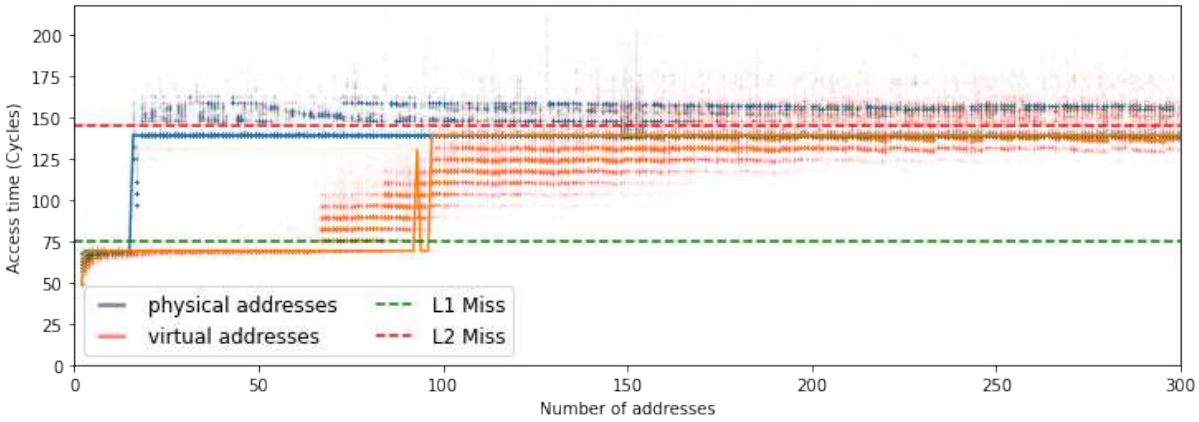
Figure 3.6 illustrates the eviction rate for creating a set of candidates in different platforms. To calculate the eviction rate, we first determine a threshold for each plat-



(a) quad-core Cortex-A72



(b) quad-core Cortex-A53



(c) dual-core Cortex-A9

Figure 3.5 – Evaluation of the candidate set size using virtual and physical addresses for different platforms.

form.Based on the L1 and L2 miss latencies presented in Figure 3.5, we compute a threshold for each device based on the latency for an L1 miss and L2 miss for each platform. The thresholds are 150, 150, and 115 for Raspberry Pi3B, Raspberry Pi4, and the FPGA board. Based on these thresholds, we count the number of successful evic-

tions to compute the eviction rate for each candidate set size. We observe that to evict an address from the LLC cache of raspberry Pi4, the candidate set requires an initial candidate set that contains at least 350 addresses to find enough congruent addresses. Other platforms require only 260 virtual addresses to evict the target address from the LLC.

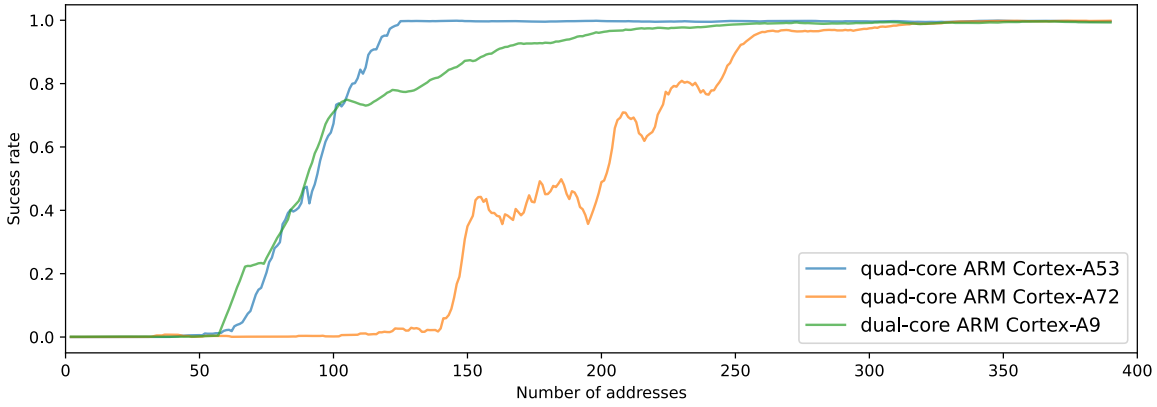


Figure 3.6 – Sucess rate of creating a candidate set using virtual addresses

3.3.3 Evaluation of Cache Eviction Strategies

In order to evict a cache line from the LLC, we evaluate the different strategies supported by libMAAT. This section aim to evaluate the sliding window [Lip+16; GMM16] and pointer-chasing [OST06] eviction strategies in different plaftorms. We assume that a target address $@_T$ is cached on the line L in the cache set S. In order to evict the line L from S, we first should construct a set of addresses congruent to the target address $@_T$. We construct a minimal eviction set using physical addresses and then focus on how these addresses should be accessed to force the replacement policy to evict the cache line L. The translation of virtual addresses into physical addresses is retrieved by consulting */proc/pid/pagemap* file.

Sliding Window Eviction Strategy

The optimal strategy $\zeta - N - A - D - L$ is specific to each processor and replacement policy. Thus the parameters $N - A - D - L$ must be determined once for each processor. This can be done by creating an eviction set with enough congruent addresses and exhaustively iterating over multiple choices of the parameters N, A, D, and L. To find the best strategy, we calculate each configuration's success rate of evicting the target cache line L. Each success rate is an average of 1,000 eviction results. For each strategy, we also measure the eviction time, assuming that the eviction time of each configuration $\zeta - N - A - D - L$ is the median of all samples.

We test and evaluate thousands of different eviction strategies in ARMv7 and ARMv8 micro-architectures. Table 3.2 shows in order the fastest ten strategies that are able to evict the cache line L with a success rate of more than 90% in different micro-architectures. By continuously checking the success of the eviction, the strategy with the least number of memory accesses (eviction time) that still provides reliable eviction can be determined. On the ARMv7 platform (with a dual-core cortex-A9), we find that evicting a cache line from the entire memory hierarchy requires about 2805 cycles for an eviction set of 28 addresses with a success rate of 90.6%. On raspberry Pi3B with ARMv7 micro-architecture, evicting the target address requires 3928 cycles with an eviction rate of 91.4% for an eviction set of 16 congruent addresses. On the ARMv8 platform (with a quad-core Cortex-A72), the eviction succeeds in about 3063 cycles with a success rate of 92.6%, for an eviction set of 25 virtual addresses, compared to the privileged instruction, which requires between 150 and 250 cycles depending on whether the target address is cached. In ARMv8, the flush instruction can be used without any privileges.

N	A	D	L	Eviction time (Cycles)	Eviction rate (%)
18	1	8	2	3063	92.6
13	1	11	2	3475	90.9
20	1	8	2	3482	97.9
19	1	8	2	3497	98.7
18	1	11	3	3522	91.7
17	1	11	3	3525	94.7
16	1	10	2	3650	93.3
15	1	10	2	3692	95.3
18	1	9	2	3959	95.9
19	1	10	3	3981	92.4
N	A	D	L	Eviction time (Cycles)	Eviction rate (%)
14	2	3	8	3928	91.4
13	2	8	3	3948	95.3
15	2	8	3	4036	93.8
12	2	11	3	4301	96.6
16	1	8	3	4682	96.4
17	1	8	3	4759	96.5
18	1	8	3	4772	97.0
16	1	8	2	4909	96.5
15	1	8	2	4931	96.8
13	1	10	2	4989	96.2
N	A	D	L	Eviction time (Cycles)	Eviction rate (%)
16	1	13	3	2805	90.6
25	1	9	3	2863	90.5
29	1	8	3	2909	93.2
28	1	8	3	2914	92.5
16	1	14	3	3112	92.5
26	1	10	3	3114	92.1
27	1	10	3	3114	91.0
25	1	10	3	3123	91.3
17	1	14	3	3124	92.3
18	1	14	3	3125	92.2

Table 3.2 – Classification of the best ten configurations using sliding window eviction strategy for raspberry Pi4, raspberry Pi3B+, and FPGA Zybo7z20 platforms, respectively.

Pointer-Chasing Eviction Strategy

Size	Offset	Eviction time (Cycles)	Eviction rate (%)
36	19	221	90.4
35	19	221	90.5
31	20	243	91.4
32	20	243	91.7
27	10	243	92.5
Size	**Offset**	**Eviction time (Cycles)**	**Eviction rate (%)**
31	2	218	100.0
61	10	218	100.0
61	9	218	100.0
61	8	218	100.0
61	7	218	100.0
Size	**Offset**	**Eviction time (Cycles)**	**Eviction rate (%)**
23	5	137	100.0
23	5	137	100.0
56	43	137	100.0
56	44	137	100.0
56	45	137	100.0

Table 3.3 – Classifiaction of the best five configurations using pointer-chasing eviction strategy for raspberry Pi4, raspberry Pi3B+, and FPGA Zybo–7z20 platforms, respectively.

To evaluate the pointer-chasing eviction strategy, we test different parameters, including the eviction set size n and the offset O (see algorithm 8). For each couple (n, O), the evaluation is repeated 1,000 times. Table 3.3 shows the fastest five strategies that are able to evict the cache line L with a success rate of more than 90% in different micro-architectures, using the pointer-chasing eviction strategy. On the raspberry Pi3B platform, we find that evicting a cache line from the entire memory hierarchy requires an eviction set of 31 addresses and an offset of 2, evicts the target cache line in about 218 cycles with an eviction rate of 100%. On FPGA Zybo-7z20 that supports an ARMv7 CPU, evicting the target address requires 137 cycles with an eviction rate of 100% for an eviction set of 23 congruent addresses. On the ARMv8 platform (with a quad-core Cortex-A72), the eviction succeeds in about 221 cycles with a success rate of 90.4% for an eviction set of 36 virtual addresses, which is close on performance to the flush instruction. Comparing the sliding window and pointer-chasing techniques, we observe that the pointer-chasing has the best performance in evicting a cache line from all cache levels (which is around ten times faster than the sliding window eviction strategy).

3.4 Attack on T-Table based AES implementation

In this section, we use the libMAAT library to perform a cache-based side-channel attack to retrieve the AES T-table implementation key. First, we demonstrate the possibility of using an efficient eviction set from user space to observe cache activity in ARM processors using the Evict+Reload technique. Then, we perform a cross-core attack on a native environment where the attacker and the victim process share the LLC and are in different cores. Finally, we have limited our analysis to using a cipher key of 128 bits and

a 128-bits plaintext, even though the algorithm also supports different key and block sizes. It is expected that analysis using other keys and block sizes would give similar results.

3.4.1 AES implementation

We describe the outline of the Rijndael [DR99] symmetric encryption algorithm used as the Advanced Encryption Standard (AES). It was adopted by NIST (National Institute of Standards and Technology) in 2001 and was used to replace DES. Instead, AES allows cipher key sizes of 128, 192, and 256 bits and plaintext of 128-bits blocks. In this section, we describe only the 128-bit AES implementation.

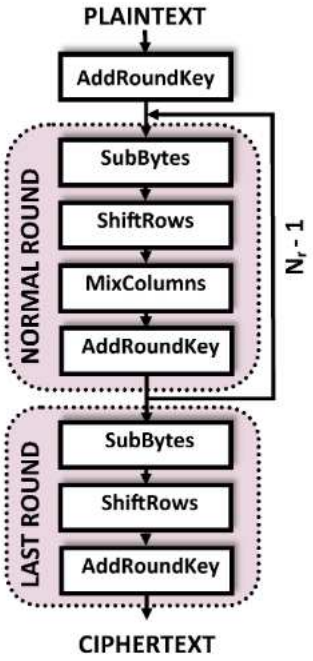


Figure 3.7 – A global overview of the AES algorithm [Bri+19].

The AES algorithm performs computation on a 4x4 byte matrix, which presents the basic data structure of the algorithm. The algorithm comprises several rounds N_r depending on the secret key size. For example, when the used key is 128-bits, the AES performs ten rounds of computations. The first $N_r - 1$ rounds are composed of four steps: `SubBytes`, `ShiftRows`, `MixColumn`, and `AddRoundKey`. The last round ignores the `MixColumn` step (see Figure 3.7). A separate key scheduling function generates all round cipher keys, which are also represented as 4x4 byte-matrices, from the initial key. Decryption is the reverse operation of encryption, and the transformations are performed in the opposite direction. More details of the algorithm can be found in [DR99] and [DR20].

In a typical AES encryption algorithm, each round r takes two inputs, a 16-bytes intermediate state matrix $s^r = \{s^r_0, s^r_1, \ldots, s^r_{15}\}$, and an expanded cipher key $k^r = \{k^r_0, k^r_1, \ldots, k^r_{15}\}$. The initial state s^0 is computed by $s^0_i = p_i \oplus k_i (i = 0, \ldots, 15)$, where p_i are bytes of given plaintext $p = \{p_0, p_1, \ldots, p_{15}\}$, and k_i present bytes of the initial cipher key $k = \{k_0, k_1, \ldots, k_{15}\}$. Then, the intermediate states at each round except the last one are computed following Rijndael equations [DR99].

An optimized software implementation, called the T-Table implementation is described

in [DR20]. Since the elementary operations in AES are done a Galois Field, they are more expensive than regular arithmetic computations. The T-Tables are precomputed results of `SubBytes`, `ShiftRows`, and `MixColumn`. For instance, 4 lookup tables T_0, T_1, T_2, and T_3 can implement the first $N_r - 1$ rounds. Since `MixColumn` operation is ignored in the last round, another lookup table T_4 is constructed to tackle this last round. Each T-table contains 256 4-bytes words (1 KB per table).

3.4.2 Attack Description

As explained, the T-tables are precomputed, allowing encryption and decryption by simple memory accesses and XOR operations. Instead, the initial intermediate state bytes are computed by $s_i^0 = p_i \oplus k_i$ in the first round. According to the byte index i and s_i^0, the T-table element $T_{(i \mod 4)}[s_i^0]$ is accessed to retrieve precomputed element. Using a shared-based cache attack, it is possible to spy the access patterns of the targeted cache line $T_{(i \mod 4)}[s_i^0]$, and thus, possible to retrieve the upper n bits of each byte $k_i = p_i \oplus s_i^0$ in case the plaintext p_i is known.

Algorithm 10: Known-plaintext AES attack using Flush+Reload technique.

Input: N = Number of encryptions
Input: T = Base addresses of each AES T-table

```
for i ← 0 to 4 do
    for j ← 0 to N do
        plaintext ← generate random plaintext
        target ← T[i]
        Flush(target)
        callback_encryption_process(plaintext)
        latency ← load_and_time(target)
    end
end
```

Let's assume the adversary monitors the memory line corresponding to the first positions of each T-table. In addition to the timing information t whether the targeted T-table element was accessed, the adversary needs to know the corresponding plaintext p. That is, we assume the adversary is able to perform several encryptions of random plaintexts and return the tuple $< p, t >$. A generic pseudo code of the presented AES cache attack is illustrated in algorithm 10.

However, it is only possible to derive the upper n bits of each byte k_i through first-round cache-based side-channel attacks due to the cache line size that contains multiple T-table elements. For a cache line of 2^λ-byte, the attacker can retrieve $\lambda - 2$-bit per

key-byte, thus for a $64B = 2^6B$ cache line size, the attacker is able to recover 64-bit of a 128-bit key. To recover the remaining key bits, we perform the second AES round attack [TOS10], exploiting the non-linear mixing in the cipher to reveal additional key information. Specially, we exploit the equation of the Rijndael specification [DR20] to compute the different indices $s_i^{(1)}$ used in four table lookups in the second round. For both rounds of AES attack, if s_i is equal to one of the indices of the monitored T-table entries, then the monitored memory line will have a very high probability of being present in the cache. We refer these samples to the hypothesis H_0. However, we refer to H_1 where s_i takes different values, and the monitored memory line is not loaded during encryption. In other words, the H_0 contains the samples when a cache hit is observed in the monitored entries. Otherwise, when the cache hit is observed in other entries, it will be referred to H_1.

3.4.3 Side-Channel Distinguishers

In order to distinguish the two cases H_0 and H_1, all that is necessary is to measure the timing for the reload of the targeted memory line. If the AES encryption accesses the targeted cache line, the reload is fast; otherwise, it takes more time. Based on this timing information, we describe and compare three distinguishers to process the side-channel data. First, we split our observations into two sets for each byte according to a hypothesis. If this hypothesis is correct, the two sets should differ, making two different distributions that can be distinguishable when there are sufficiently many observations. Otherwise, the hypotheses will be invalid when the key guess is wrong; both distributions come from the same distribution, which makes them indistinguishable. For that, we use the tree most common distinguishers in side-channel analysis to detect whether samples for hypotheses H_0 and H_1 are from different distributions.

Hit-counter based distinguisher. This distinguisher measures and compares the number of cache hits for the two hypotheses H_0 and H_1. Ideally, hypothesis H_0 has no misses on target addresses because they were accessed during the execution of AES. To create a hit counter, the reload time is compared to the threshold and assigned 1 if the data is present in the cache (hit) and 0 if the data is fetched from the main memory (miss). Based on a threshold value that we will choose empirically from our measurements, we expect to be able to distinguish main memory accesses from LLC cache accesses. For each guessed value of each key byte, we will count the number of hits, and the guessed key value with the highest number of hits has a very high probability of being the correct key byte.

Welch's t-test distinguisher. This distinguisher is common practice in side-channel analysis, especially in power analysis. However, unlike the hit-counter-based distinguisher, the Welch t-test distinguisher does not need to compute the threshold between a cache hit and a cache miss; thus, it is based on the reload time. Instead, it aims to calculate the t-value between the two hypotheses H_0 and H_1 with the formula from Equation 3.1. Thus, the most significant value of the t-test corresponds to the most likely key hypothesis.

$$t = \frac{\bar{H_0} - \bar{H_1}}{\sqrt{\frac{var(H_0)^2}{n_0} + \frac{var(H_1)^2}{n_1}}} \tag{3.1}$$

Where $\bar{H}$, $var(H)$ and n represent the mean, variance and cardinality of each sample.

Pearson's correlation coefficient. Pearson's method is widely used in statistical analysis and aims to measure a linear correlation between two random variables H_0 and H_1. Pearson's correlation coefficient is defined in Equation 3.2 and results in values from –1 to 1.

$$R = \frac{cov(H_0,\ H_1)}{\sqrt{var(H_0) * var(H_1)}} \tag{3.2}$$

Where R is null they no linear correlation between two variables. Otherwise, a higher $|R|$ indicates a higher linear correlation between H_0 and H_1.

3.5 Setting and Attack Primitives

In this section, we describe the last-level Evict+Reload cache timing attack that we have implemented and evaluated on ARMv7 architecture, targeting an OpenSSL1.0.1f implementation of AES. This concrete example of constructing cache attacks using the libMAAT's modules. In this attack, we assume that the adversary (the spy process) and victim (the encryption process) are sitting on the same physical machine but on different cores, which share the last-level cache.

3.5.1 Targeted Platforms and Settings

Targeted platform: Our target platform is FPGA Zybo-Z720, which contains a Cortext A9 core, ARMv7 architecture, 2 physical cores, and a 256KiB 8-way set-associative exclusive shared last-level cache (L2 cache). Each core contains private L1 data and instruction caches of 32KiB, 4 ways for each. The size of each cache line is 32 bytes, and both cache levels use a pseudo-random replacement policy. Thus the last-level cache has 1024 cache sets. In this setup, the execution programs run on a native Linux Debian 10 version with no virtualization.

Attacker assumptions: In our attack model, the victim program runs AES encryption using OpenSSL 1.0.1f, on which cache-based side-channel attacks were performed in the literature. We consider an AES implementation with aligned T-tables, i.e., the starting memory address of a T-table is mapped to the beginning of a cache line. A spy program runs a cache monitor simultaneously with the victim application, taking advantage of commonly available multicore or hyperthreaded environments. We assume that the spy process knows the virtual addresses of T-tables. The spy program is in user mode and has no direct interaction with the victim, except for sharing micro-architecture resources, including last-level caches. In addition, the spy can execute arbitrary code on the target platform but does not have access to the kernel or any privileged interfaces such as `/proc/self/pagemap` that provide address information for the user space of the victim program.

Attacker challenges: For the attack to be feasible in a cross-core setting, we need to address two challenges: first, we should be able to evict data in LLC and the L1 caches in different cores. In ARM micro-architectures, the last-level cache is shared and exclusive, meaning that each data is present in one cache level; evicting specific data from the memory hierarchy should be evicted from all cache levels. Thus, the main memory will serve further requests for access to the evicted data, resulting in distinguishably high latency. Second, we should be able to evict the targeted cache line in the LLC using virtual addresses.

3.5.2 Measurement Using Evict+Reload Technique

Eviction Set Construction

Since the spy process knows the T-tables virtual addresses, we lunch the spy process to construct a set of addresses mapped to the same cache sets in the LLC. We start by constructing an eviction set for each target T-table entry. As presented before, we perform a single cache line AES attack, where the attacker spies only one entry per T-table. We generate a random candidate set for each first T-table entry using 2MB transparent huge page memory. The reason for using large pages is also to avoid TLB noise, where a large number of pages accessed by a candidate sets over-stress the comparatively small TLB. Thus, allocating a set of addresses from large pages significantly reduces the number of pages visited when traversing an eviction set, reducing false positives. In addition, the 21 offset bits directly translated into the physical address are more than enough to address 1024 cache sets. We use libMAAT's cache eviction module to construct optimized eviction sets to evict a cache set from the LLC.

So concretely, we allocate a memory pool of 2MB of large transparent pages, then we randomly collect several addresses that can be used as a candidate set. In the second step, we call the optimization eviction set function, which reduces the eviction set size to contain only the congruent addresses, and removes addresses that do not participate in the eviction of the target address. Finally, we repeat the eviction set construction for each target address (the first address of each lookup table). We can effectively evict the targeted cache line from the LLC cache by accessing those selected addresses.

To avoid extra memory accesses induced by hardware prefetchers, we use a pointer-chasing eviction strategy, where the first pointer of this list is localized at the first element, and the second pointer (offset) is localized in the middle of the list. After determining the targeted cache sets and constructing their eviction sets, which results in 32 virtual addresses eviction sets for each target address. First, the spy process evicts the target T-table entry, accessing its eviction set; second, the spy process sends one random 16-byte plaintext to the victim process and waits for the encryption to complete; third, the spy process reloads the target address, and measure the access time. For each AES encryption, the spy process records the plaintext, and the reload times for the targeted cache set.

Evalution

We collect 1 million traces for each target address in the online phase, as shown in algorithm 10. At compile time, the flush instruction is replaced by the evict procedure provided by libMAAT. The attacker collects enough data from the online phase, including the plaintexts used and the time measured at each encryption.

In the offline phase, we recover the secret key using the collected data (the plaintext and the timing information). This phase is divided into a first-round attack phase, which exploits the table indices equations used in the first round, and a second-round attack phase. The attacker starts by making hypotheses H_0 and H_1 for each key byte k_i based on the timing information, and whether the target address was accessed during encryption according to the equations of each round. Then, as our attack model monitors the first entry of each T-table, the attacker can build a model for one corresponding key byte, which also applies to the other three related bytes. For example, if we monitor the first cache line of Te0, the related four key bytes are {2, 6, 10, 14}.

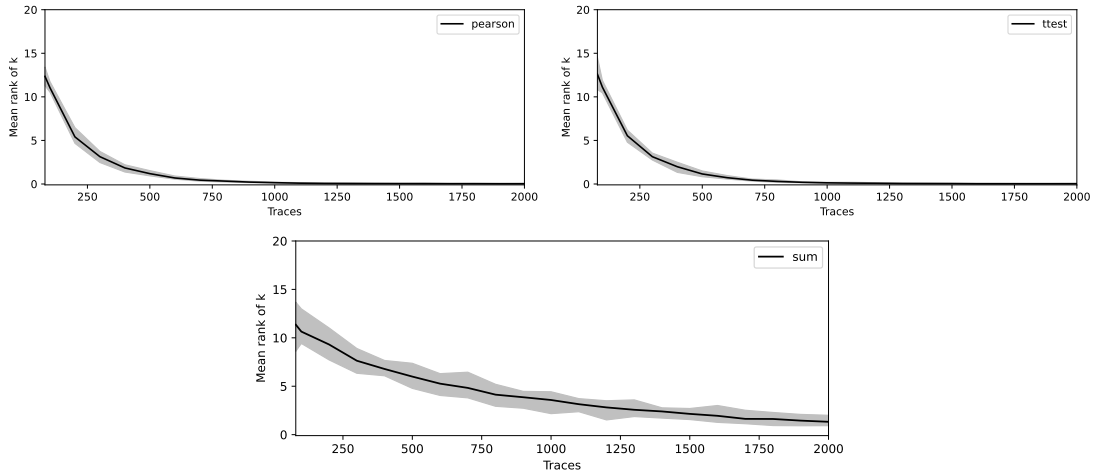


Figure 3.8 – Guessing entropy comparison of different distinguishers method for first round AES attack

To compare the different distinguishing methods presented (subsection 3.4.3), we compute a guessing entropy for each $\langle s_i^0 \rangle$ for the attack in the first round. To this end, we perform the attack 100 times on n randomly selected traces from the 1-million set of traces. The experiment then returns the rank (index) of the class in the probability vector. Thus, the guessing entropy is the mean rank of the key byte. The result is shown in Figure 3.8, where the x-axis shows the number of encryptions and the y-axis shows the variance and mean of the ranks of the different key bytes. We observe that Welch's t-test and the Person methods give identical results. Thus, half of the key AES can be recovered in 1000 traces (if the rank order of all key bytes converges to the correct key

at rank 0). Otherwise, key recovery requires more than 2000 traces with a precomputed threshold of 115 cycles when the hit-based technique is used. The increase in the number of samples for the hit-based technique is due to noise in the time calibration phase. Welch's t-test and Pearson's techniques directly test the hypothesis on the time distributions without the need to calculate an exact threshold. We use the second round of equations with the same methodology to recover the entire key AES. As shown in Figure 3.9, 400 additional samples are needed to recover the full key AES using Welch's t-test and Pearson techniques.

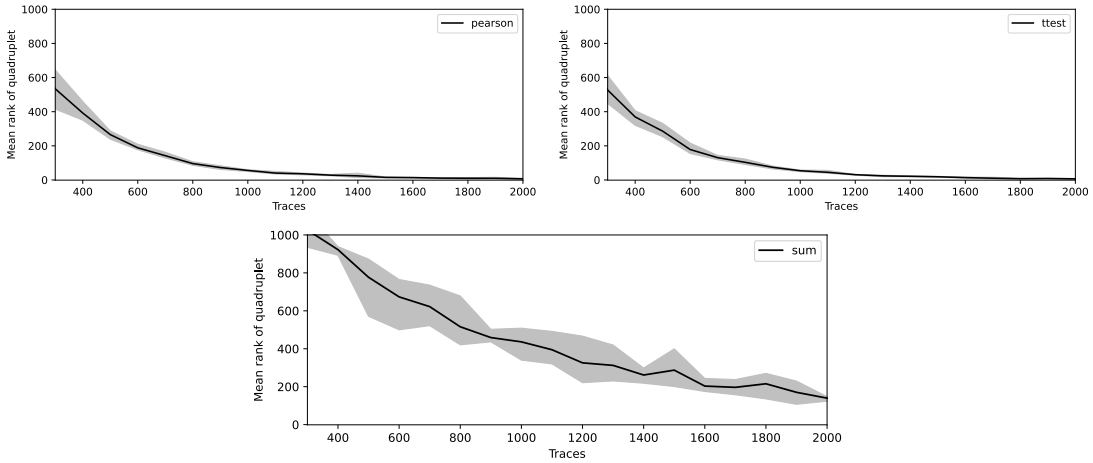


Figure 3.9 – Guessing entropy comparison of different distinguishers method for second round AES attack

3.6 Conclusion

This chapter introduced a new framework that can be used to construct micro-architectural software side-channel attacks in most processor micro-architectures. We have implemented all the techniques proposed in the literature in the form of a library called libMAAT, which allows platform-independent implementation of cache side-channel attacks for x86, ARM, and RISC-V platforms. We show that by using a more accurate timer to measure timing, the attacker can observe activities on different cache levels and not only on the LLC. LibMAAT consists of multiple modules to facilitate the construction of software attacks that target hardware vulnerabilities without requiring authorization or privileges. Unlike the two libraries, libMAAT gives attack developers more freedom to construct unprivileged-level micro-architectural side-channel attacks using the latest techniques in the literature.

Table 3.4 compares the different modules in Libflush and Mastik with our work. For

example, in Mastik and libflush, the eviction set is constructed statically by accessing the pagemap translation and then directly finding the physical addresses mapped to the same cache set. Before version 4.0 of the Linux kernel, accessing the pagemap in `/proc/self/pagemap` was possible without privileges. However, in newer kernel versions, the pagemap is accessible only with root privileges [Shu]. Therefore, both libraries require privileges to create optimized eviction sets. Nevertheless, to get the physical from a virtual address in libMAAT, we exploit the use of transparent huge pages. The Linux kernel allocates 2MB pages as often as possible to processes that map large memory blocks. By mapping a large memory block with `mmap`, we get 2MB pages for our memory when huge pages are available. This could fail if, for example, no continuous memory area of at least 2MB is available.

To benchmark our framework, we demonstrated an Evict+Reload technique on an ARM platform to retrieve the secret key AES in a real-world scenario by solving all challenges using different modules in libMAAT. One of the major challenges in constructing a cache attack on the ARM platform is to get a cache set into a known state when the flush instruction is reserved for privileged processes. In this way, we propose several strategies from the literature to easily construct an efficient eviction set. Moreover, we show that cache eviction can be performed successfully and quickly, increasing our AES attack's performance and accuracy.

Library		**Mastik**	**Libflush**	**Our work**
Taget Systems		x86	x86 and ARM	x86, ARM and RISC-V
Privelges		User	Root	User and Root
Timing Sources		CPU counter	CPU counter	CPU counter
			Perf	Perf
			Monotonic	Monotonic
			Thread-based counter	-
Eviction Set Construction	**Candidate Set Finding**	Huge Pages	Huge Pages	Huge Pages
				Transparent large Pages
	Eviction Set Optimization	N/A	N/A	Single Address Elimination
				Group Address Elimination
	Eviction Set Access Strategy	N/A	Eviction Strategy	Static and Linear Access
				Eviction Strategy
				linked List
				Double linked List
	Configuratiuon	N/A	Needs to be configured for each machine (Eviction Strategy Evaluator)	Dynamically, when the evict function is called
Cache configuration		Static	Static	Dynamic
Supported CSCAs	Flush+Reload	X	X	X
	Flush+Flush	X	X	X
	Evict+Reload	-	X	X
	Prime+Probe	X	-	X
	Cache template	-	X	-

Table 3.4 – A comparison between various side-channel microarchitectural libraries

4

Security Analysis of Randomized Caches

Eviction set construction is a common step for many such attacks, and algorithms for building them are evolving rapidly. On the other hand, countermeasures are also being actively researched and developed. Cache randomization is a well-known mitigation technique against cache attacks that has a low-performance overhead. In this chapter, we attempted to determine whether address randomization on L1 caches is worth considering from a security perspective. We present the implementation of a noise-free cache simulation framework that enables an analysis of eviction set construction algorithms. We show that randomization at the L1 caches brings improvements in security but is not sufficient to mitigate all known eviction set algorithms. Nevertheless, we show that L1 randomization can be combined with a lightweight random eviction technique in higher-level caches to mitigate known conflict-based cache attacks.

4.1 Motivation and Problem Definition

Several works have shown that cache attacks are possible in all levels and all types of cache memory. The Prime+Probe [OST06] attack, for example, was initially performed on first-level data caches to attack AES [Per05; NS06; OST06] or instruction caches [Aci07]. The LLC is a more interesting attack target because adversaries and victims do not need to share the same CPU. Various extensive survey studies have listed the threats at different cache levels [Lou+21; Mus+20].

Unlike sharing-based cache attacks, conflict-based cache attacks can be applied in any cache level independently [Per05; OST06; Zha+12; Liu+15; Kay+16], even when a secure last-level cache exists. Attacking the L1 cache has some advantages because fewer load instructions are required to fill or evict the L1 cache due to its small size. Therefore, to protect a memory hierarchy against conflict-based cache attacks, the different levels of the memory hierarchy should be protected. This raises the question of which countermeasures should be deployed at each cache level. To mitigate conflict-based attacks, it is essential to prevent eviction set construction through the existing single address elimination [Liu+15], group testing [VKM19] and Prime–Prune–Probe [Pur+21] algorithms. Cache randomization is an effective technique to mitigate against such attacks (see Section section 2.8), but the existing solutions have been mainly designed for LLCs. Therefore, an analysis of the effect of low-level cache randomization on existing attacks would help to determine relevant strategies to secure the entire memory hierarchy.

4.2 Threat Model and Attacker Capabilities

We assume the existence of a victim and a spy process running on the same machine. They share lower-level caches, and upper-level caches are assumed to be inclusive. The victim process contains secret information that the spy program attempts to recover without having direct access to this information.

We ignore sharing-based cache attacks in this chapter because the solution to these would be to remove address sharing in the first place (e.g., disable memory deduplication). We focus on conflict-based cache attacks (see subsection 2.5.2), where the attacker causes set conflicts to monitor the victim's access patterns. To study the effectiveness of randomization defense techniques against conflict-based cache attacks, we assume a scenario that is favorable to the attacker and show that whether randomization are effective even under weaker assumptions. To enable a strong attacker model, we assume

that there are no sources of interferences (e.g., other processes running concurrently) that could affect the results of the eviction set construction algorithms, which improves the reproducibility of the attacks.

4.3 Noise-free Cache Simulation Framework

4.3.1 Overview

As described in section 2.1, modern micro-architectures include many optimization features, like hardware prefetching, TLBs, buses, etc. These optimizations allow programs to run faster and more effectively. However, they introduce noise into the attacker's observations, making the construction of eviction sets less reliable. The term *noise* refers to the memory operations introduced by other processes or the construction eviction set algorithm itself. It also can be due to other components such as the scheduler, Dynamic Voltage and Frequency Scaling (DVFS), etc. In this work, we adopt a noise-free cache model to analyze the complexity of systematically finding eviction sets. This cache framework deliberately ignores the implementation details of the hardware and simulates only the behavior of the memory hierarchy. The cache model was written in Python and had the following features:

Constant access latency. The latency to access a cache block is the same regardless of its location (set, way, or slice);

Ideal cache hit/miss state. Without latency, a given process could accurately determine the state of a cache block (hit or miss). This eliminates errors due to the access latency measurements on the actual processors. For each memory access, our framework returns the address state (i.e., present or not) in all cache levels of the memory hierarchy and indicates whether it was a cache miss or a cache hit;

Replacement policies. The cache framework supports two replacement policies: the least recently used (LRU) policy and the random policy. The LRU policy maintains a list containing the order in which the cache ways were accessed. When the target cache set is full, the replacement policy chooses the least used cache line in the set to be replaced. As its name suggests, the random policy randomly selects a cache line from the target set;

No translation lookaside buffer (TLB) noise. Accessing a large candidate set could trigger false positive errors when the TLB entries were mistakenly removed from the TLB [Gen+18]. To neglect this effect, we did not model the memory management unit (MMU) or the TLB component;

No cache prefetching. The hardware prefetchers improve performance by accessing the following predicted memory addresses. However, they introduce noise into the attacker's observations by accessing unnecessary memory addresses [Wan+19]. Thus, the framework did not support model prefetching;

Address randomization of caches. Address randomization can be enabled at any cache level. The framework supports differents cryptographic functions to distribute the addresses among all cache sets. It also supported a configurable *remapping period* that changed the address-to-set mapping after a certain number of memory accesses or evictions.

The cache framework also implements *performance counters* to measure the side effects of the running application. For example, they provide information about the number of memory accesses, misses, hits, evictions, writebacks, and all other information measured in the memory hierarchy. These performance counters are implemented in each cache level to evaluate and analyze the behavior of the eviction set construction algorithms.

4.3.2 Illustative Example

To simulate the behavior of an application in our cache framework, we first built a memory hierarchy by specifying the configuration of each cache level, including cache mapping, cache size, associativity, cache line size, replacement policy, etc. The example in Listing 4.1 shows the implementation of the single address elimination algorithm using the cache framework. It takes as inputs the *target* address and a candidate set. The algorithm removes a candidate address at each iteration and then calls the test eviction function (line 20) to check if the candidate set is still evicting the target address.

We use the memory access instructions from the framework to perform the algorithm's memory accesses. Both instructions, load and write, can be used to determine the cache state of the accessed address (hit or miss). For example, in line 9, the *load_state* instruction loads the target address and returns the address state (hit or miss) at each cache level.

Listing 4.1 – The implementation of the single address elimination algorithm with our cache simulation framework

```
def test_eviction(cache_simulator, target: int, eviction_set: list) -> bool
    :
    # load the target address
    cache_simulator.load(target)
    # access the eviction set
    cache_simulator.load(eviction_set)

    # The target address is reaccessed, and the simulator returns its
    # cache state (True if it is a Hit, otherwise it returns False).
    hit = cache_simulator.load_state(target)

    return not hit

def single_address_eliminaton(cache_simulator, target, candidate_set) ->
    list:
    while len(eviction_set) < self.ways:
        # remove an address from the candidate_set array
        candidate_address = candidate_set.pop()

        # check if the candidate set sitll evicting the targeted address
        evset = candidate_set + eviction_set
        miss = test_eviction(cache_simulator, target_address, evset)

        # If the test fails, it means that the "candidate_address" is
        # congruent to the target address and should be added to the
        # eviction set array
        if not miss:
            eviction_set.append(candidate_address)

    return eviction_set
```

4.4 Experimental Setup and Methodology

The baseline configuration. To investigate the eviction set construction algorithms, we create a two-level memory hierarchy (inspired from raspberry Pi4 platform) where the last-level cache is inclusive and shared. We call a baseline configuration where the target memory hierarchy did not contain any defense (as shown in Table 4.1).

We implement the different algorithms to construct an eviction set (see section 2.6) to

be executed in a noise-free environment. To evaluate the success rate of each eviction set construction algorithm in a given setting, we run it 1,000 times with different candidate sets and records the results for later analysis.

Table 4.1 – The baseline configuration.

Parameters	L1 Cache	L2 Cache
Cache size	32 kB	1 MB
Associativity	8 ways	16 ways
Cache line size	64 B	64 B
Inclusion policy	Inclusive	Inclusive
Replacement policy	LRU	LRU

Methodology. More precisely, in each simulation: ① We reset the performance counters and the content of each cache level; ② We randomly generate a target address and a candidate set with the given size. For a 32-bit address, the elements of the candidate set and the target addresses were random addresses between 0 and $2^{32} - 1$; ③ Then, we perform the eviction set construction algorithm to minimize the size of the generated candidate set; ④ We check whether the returned eviction set evicts the target address. The algorithm was considered successful when it could evict the target address from LLC; ⑤ We save the performance counters and execution time in a log file. At the end of all simulations, we computed the success rate. Then, we computed the median for each performance counter of the 1,000 experiments. The value of 1,000 was the highest number of repetitions that allowed the simulations to be performed within a few days on a 48-core machine.

4.5 Complexity of Constructing Eviction Sets in Unprotected Memory Hierarchy

In this section, we use the cache simulation framework to study the behavior of the eviction set construction algorithms. We evaluate the complexity of building an eviction set in a noise-free environment with an unprotected memory hierarchy.

4.5.1 Single Address Elimination Algorithm

Figure 4.1 shows the success rate and the execution time required to construct an eviction set for different sizes of candidate sets using the single address elimination algorithm. For each simulation evaluation, the execution time represents the median of all 1,000 samples. As shown in Figure 4.1a, the success rate increases when the size of the candidate set contains more than 10,000 addresses.

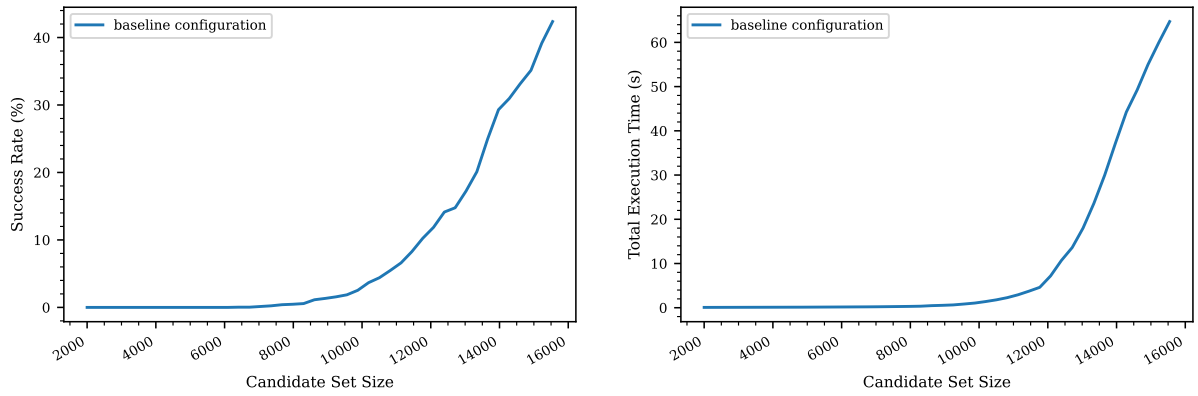


Figure 4.1 – (**a**) The success rate of the single address elimination algorithm with different candidate set sizes. (**b**) The time required to find an eviction set using the single address elimination algorithm.

Due to the quadratic complexity of the algorithm, we stop the simulation due to the long time to find an minimal eviction set. As shown in Figure 4.1b, the execution time could grow up to 160 seconds per simulation, which result in more than two days of simulation when the candidate set is larger than 20,000 addresses.

4.5.2 Group Testing Algorithm

Figure 4.2 depicts the probability of finding an eviction set as a function of the candidate set size using the group testing algorithm. As shown by the theoretical curve (see Equation Equation 2.1), the probability of finding a candidate set with at least W congruent addresses increases with its size. The results show that when the candidate set has at least W addresses, the group testing algorithm could reduce its size to construct a minimal eviction set. Furthermore, it can be observed that group testing reduction in a noise-free environment closely matched the theoretical prediction. Vila et al. [VKM19] observed similar trends in a real system.

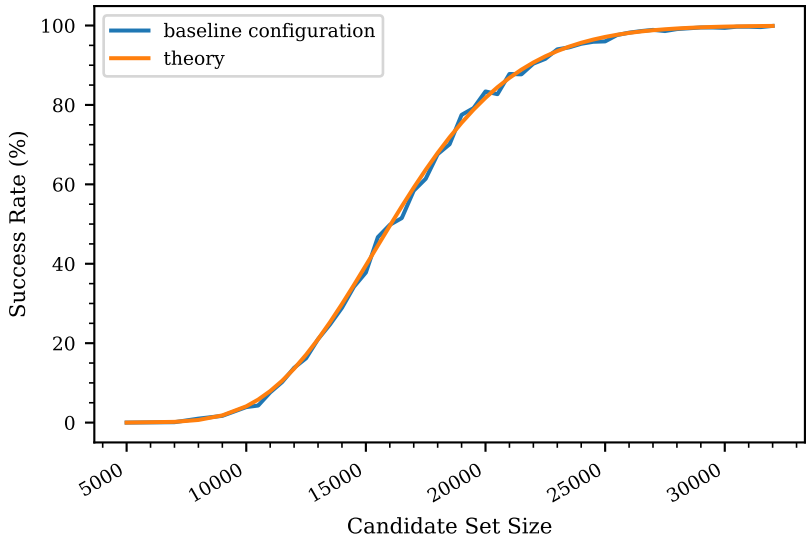


Figure 4.2 – The success rate of the group testing algorithm with different candidate set sizes.

4.5.3 Prime–Prune–Probe Algorithm

Figure 4.3 depicts the success rate of the PPP algorithm as a function of the candidate set size. As shown, the PPP algorithm built an eviction set from smaller candidate sets; therefore, accessing them can be faster to find an eviction set than using the group testing algorithm that requires a larger candidate set. The PPP algorithm captures the interferences in a small filtred candidate set and uses them to construct an eviction set. Intuitively, we expect the algorithm's success rate to keep increasing by increasing the candidate set size n. Our experiments suggest the opposite in the case of PPP, as shown in Figure 4.3. When the size of the candidate set is greater than $2N$ ($N = SW$ being the number of cache lines), we found that the algorithm fails to find an eviction set despite sufficient congruent addresses in the candidate set. This effect is not studied in [Pur+21].

By analyzing the memory accesses of the Prime–Prune–Probe algorithm, we observe that the *Prune* filter becomes more aggressive when the candidate set contains more than W congruent addresses among the n elements. It turns out that it removes all congruent addresses from the candidate set. Consequently, the algorithm cannot detect collisions within the victim's accesses.

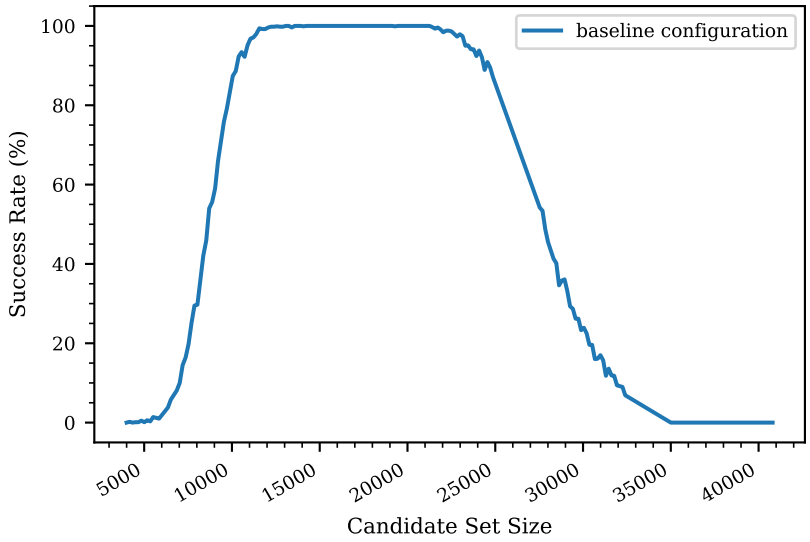


Figure 4.3 – The success rate of the Prime–Prune–Probe algorithm with different candidate set sizes.

Let us consider a 2-way set-associative cache. We used the Prime–Prune–Probe algorithms to construct a minimal eviction set targeting set one. Figure 4.4 illustrates this effect using a small cache memory with an LRU replacement policy. As shown in the first step, the attacker accesses the candidate set containing four congruent addresses. Since the cache was a two-way set-associative cache, the cache memory only contained the last congruent addresses (addresses C and D) in set one after the Prime step. In the Prune step, the attacker access the same candidate set again and removes the missing addresses from the candidate set to avoid any noise caused by self-eviction. Since the first two addresses, A and B, were evicted from the cache, the Prune filter found them missing after accessing them. The other two congruent addresses, C and D, were evicted from the cache due to access to addresses A and B. The Prune filter considers them as missing addresses and, therefore, removes them from the candidate set. In this case, all congruent addresses were removed from the candidate set, which explains why the Prime–Prune–Probe algorithm failed when the candidate set was large.

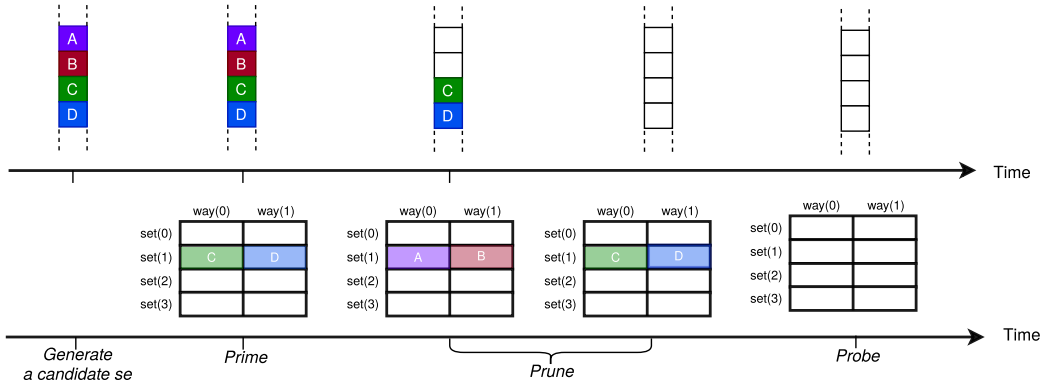


Figure 4.4 – An illustration of a Prime–Prune–Probe iteration when the candidate set contains more than W congruent addresses. Above is the state of the candidate set at the end of each step. Below is the cached state.

4.6 Randomization in Low-Level Caches

In this section, we analyze the security implications of randomizing low-level caches. This study does not consider the single address elimination algorithm because its execution takes too long in the relevant parameter range. We target a dynamic randomized low-level memory hierarchy with the same cache parameters as the baseline configuration (unprotected memory hierarchy). Starting from the baseline configuration, we apply address randomization to L1 caches in the noise-free simulation framework using a robust cryptographic scheme. To protect the memory hierarchy from conflict-based cache attacks, the cache mapping should be changed frequently to prevent eviction set construction algorithms from finding efficient eviction sets. This section estimates the remapping period for the group testing algorithm and the Prime–Prune–Probe algorithm in the baseline configuration.

4.6.1 Choice of the Addressing Function

In our experiments, we model ideal address randomization. The motivation is to prevent any "shortcut attacks" [Pur+21] that can break the randomization function and construct eviction sets statically. In this manner, we use a 128-bit AES cipher with a 128-bit AES key to perform the address randomization. The AES key is updated each time the remapping period is achieved. When the CPU access the memory, the CPU address is first randomized using the AES instance and the current AES key. The low bits of the resulting cipher are selected to index the cache set.

We stress that using such a strong randomization function in L1 caches may not be realistic in practice. Any delays introduced into L1 requests would turn into significant slowdowns. The design of a lightweight, secure and optimized hardware address randomization function is a complex topic that is not covered in this work. As an illustration, the low-latency cipher designed in the CEASER architecture [Qur19; Qur18] is completely unsafe to use [Bod+20]. Using an AES-128 function to randomize the cache mapping pushes back the problem of designing a secure and lightweight permutation function. However, its integration into a hardware cache and the design of a lightweight alternative with similar properties still open problems for the research community.

4.6.2 Remapping Interval Analysis

To protect cache structures against conflict-based cache attacks, the cache mapping should ideally be changed just after constructing an eviction set to avoid the attacker using it. Thus, the remapping period should be smaller than the minimum number of memory accesses or evictions required to construct an efficient eviction set. This section estimates the remapping interval to dynamically randomize the L1 cache. To estimate this interval, we reproduced the experiments performed in Section 4.5 to estimate the number of accesses required to find a usable eviction set. As described earlier, each experiment was repeated 1000 times to compute the success rate of each algorithm. At the end of each experiment, we obtain an array containing the number of memory accesses for each sample. After determining the median of all of the experiments for each candidate set size and their success rates, we compute the minimum number of memory accesses required to achieve a success rate of at most λ, with λ varying from 0 to 1.

In Figure 4.5, we plot the number of memory accesses needed to construct an eviction set as a function of the success rate for the group testing and Prime–Prune–Probe algorithms. For each success rate λ, we return the median of the different experiments with success rates lower than λ. Then, we compute the minimum number of memory accesses.

Since we did not want the attacker to construct an eviction set using the group testing algorithm, the remapping interval must be at most 260 K memory accesses to ensure a success rate of less than 1% (see Figure 4.5a). Regarding the number of memory accesses, we noted that the group testing algorithm requires more memory accesses to construct a valid eviction set. In contrast, PPP creates an eviction set with a small candidate set at each iteration, resulting in fewer memory accesses. As shown in Figure 4.5b, PPP takes less time to construct an eviction set than the group testing algorithm. The

minimum number of memory accesses is consistent for all experiments since the PPP algorithm constructs its eviction set from small candidate sets, requiring fewer memory accesses to construct an eviction set with a high eviction rate (success rate). This occurs because the PPP algorithm requires a small candidate set for each iteration and finds at least one congruent address for each candidate set. However, using a small candidate set in each iteration increased the success rate of the PPP algorithm. Consequently, the remapping interval of the PPP algorithm has to be fewer than 47.6 K memory accesses to guarantee a success rate of less than 1% for constructing a reliable eviction set.

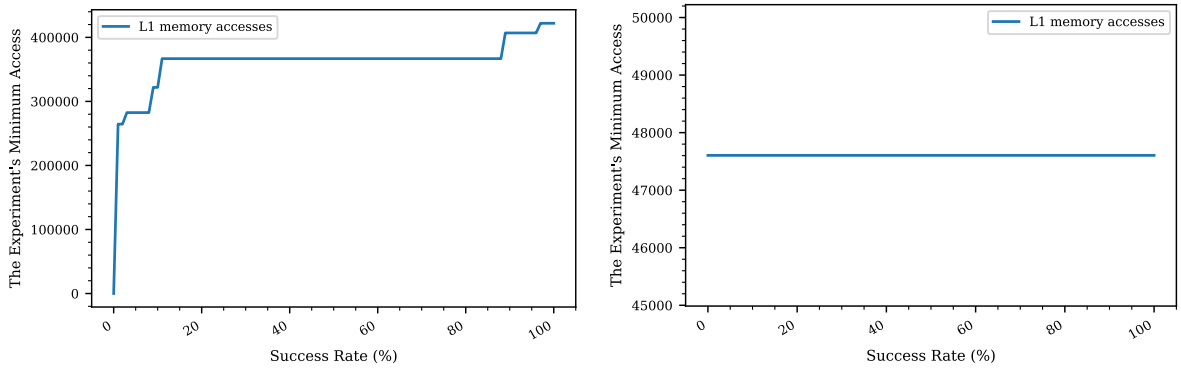


Figure 4.5 – The minimum number of memory accesses as a function of the success rate for (**a**) the group testing and (**b**) the Prime–Prune–Probe algorithms.

4.6.3 Randomization of L1 Cache

Previously, we showed that the group testing and Prime–Prune–Probe algorithms require 260 K and 47.6 K memory accesses, respectively, to construct an eviction set with a success rate of $\lambda \geq 1\%$ in the unprotected memory hierarchy.

As shown in Figure 4.6a, the group testing algorithm with the baseline configuration successfully finds an eviction set when the candidate set size contains more than 30 K addresses with a success rate of 100%. When using a dynamically randomized L1 cache with a remapping frequency of 260 K memory accesses, the results show that the group testing algorithm fails to find a useful eviction set. Indeed, we can observe a success rate of less than 1% in Figure 4.6 when the candidate set size is less than 30K addresses. This was because a single candidate set was used throughout the overall construction process. The randomization in the L1 cache interferes with the operation of this algorithm and distorts its results.

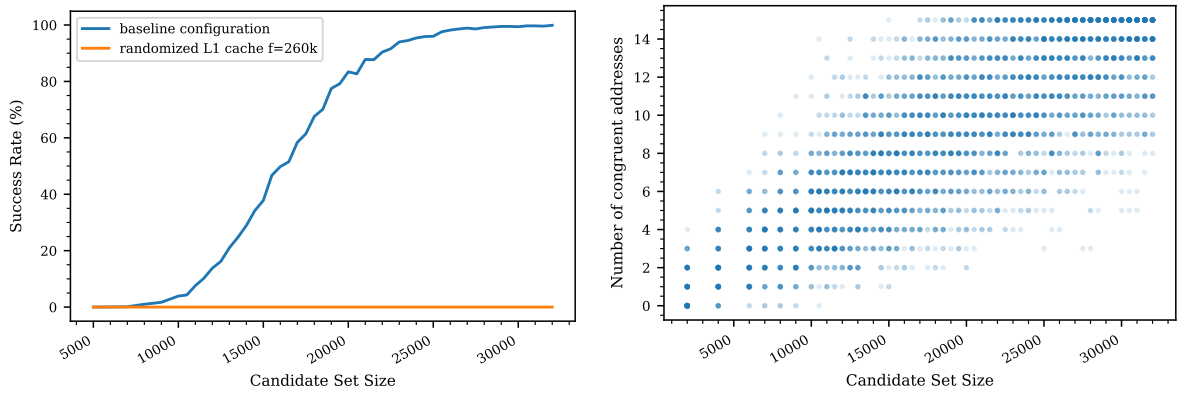


Figure 4.6 – (**a**) The success rate of the group testing algorithm with different candidate set sizes. (**b**) The number of congruent addresses in the reduced eviction set when using the group testing algorithm with a randomized L1 cache.

On the other hand, randomization in the L1 cache does not affect the success rate of the Prime–Prune–Probe algorithm (see Figure 4.7a). The PPP algorithm was developed to bypass randomized caches by generating a new candidate set at each iteration to increase the probability of finding congruent addresses. As shown in Figure 4.7a), randomizing the mapping of L1 caches using the precomputed mapping, the PPP still finds congruent addresses, even when the remapping changes serval times.

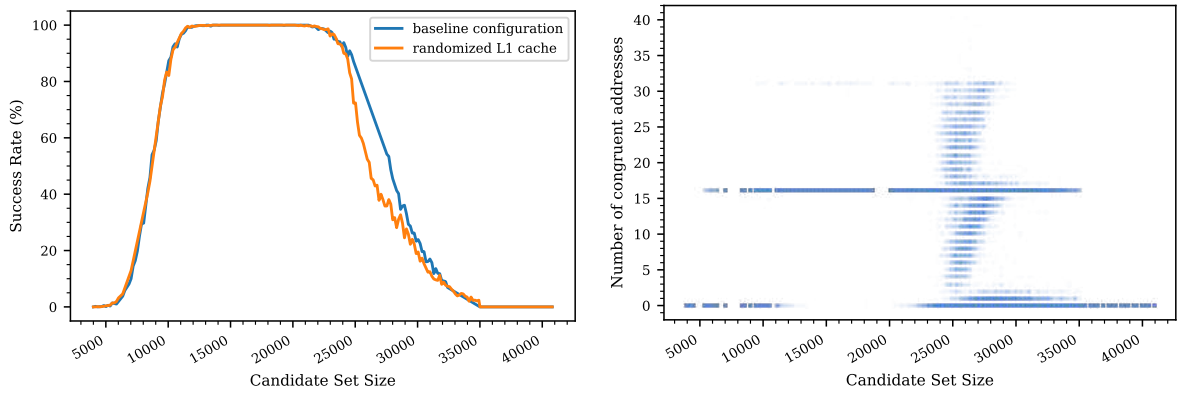


Figure 4.7 – (**a**) The success rate of the Prime–Prune–Probe algorithm with different candidate set sizes. (**b**) The number of congruent addresses in the reduced eviction set when using the Prime–Prune–Probe algorithm with a randomized L1 cache.

Note that the primary purpose of both algorithms group testing and Prime–Prune–Probe is to find at least W congruent addresses in the given candidate sets. To verify this assumption for the resulting eviction sets in Figure 4.6a and 4.7a, for each exper-

iment we compute the number of congruent addresses present in the resulting eviction set. Figure 4.6b and 4.7b show the resulting number of congruent addresses for the group testing algorithm and the Prime–Prune–Probe algorithm, respectively. Using the group-testing algorithm, we observe in Figure 4.6b that the resulting eviction sets have fewer congruent addresses than the associativity (in our case, we targeted a cache with 16 ways). Even for a large candidate set that should contain more congruent addresses, the resulting eviction sets have a success rate of less than 1 %. This lack of congruent addresses in the eviction set resulted in the target address not being evicted, which confirmed our results in Figure 4.6a. For the Prime–Prune–Probe algorithm (see Figure 4.7b), we observed that for a candidate set of between 10 K and 26 K, the algorithm could find sufficient congruent addresses to construct a valid eviction set. We also noted that the inability of this algorithm to find an eviction set when the candidate set size was greater than 32K was due to the small number of congruent addresses in the resulting eviction set.

4.7 Random Eviction Last-Level Cache

4.7.1 Overview

As described in section 4.6, the Prime–Prune–Probe bypasses randomized lower-level caches by observing the last-level cache eviction patterns. The core element of this algorithm is the *Prune* filter, that is used to remove all addresses that were subject to self-eviction. At the end of the Prune filter process, the candidate set should not contain any addresses that collided with each other. In this way, a purely deterministic decision can be made about which cache set was accessed in the Probe phase. This algorithm constructes eviction sets in the shortest possible time by bypassing randomization, even when the mapping changed more frequently. Thus, it seem natural to add random evictions in the last-level cache to disturb the Prune filter's operation. This should have made it more difficult (ideally impossible) for an attacker to construct an eviction set.

The concept of adding non-deterministic eviction is well known [ZL14; Dem+12]. The idea is to evict one or more lines randomly from the cache at a given interval of time. We can use the Prime+Probe attack as an example to understand the intuition behind it. During the Probe step, the attacker accesses the eviction set and measures the memory access latency. In this way, the attacker can determine which cache sets the victim has accessed. However, when the cache eviction is not deterministic, the attacker cannot know whether the cache line was evicted randomly by the random eviction policy or by

the victim's accesses. This can make the attacker's observations noisy and thus, mitigate against conflict-based cache attacks.

We implmenent a random eviction mechanism to the last-level cache model of our simulation framework. It supports two security parameters: the eviction frequency f and the number of evicted lines n. Every f memory accesses, n cache lines are randomly evicted. For example, for a cache with the random eviction strategy parameters $(f, n) = (5, 1)$, a random cache line would be evicted every five memory accesses. Intuitively, the lower the eviction frequency, the more robust the cache. Depending on the two values of f and n, the performance counters are used to count the number of memory accesses. Every f accesses, a pseudo-random number generator generates n cache lines indexes.

4.7.2 Results and Discussion

In this section, we only study the PPP algorithm since the group testing is can mitigated by randomizing the lower cache levels. To evaluate the resilience of the random eviction strategy against the Prime–Prune–Probe algorithm, we assess the success rate of this algorithm as a function of the two parameters f and n. For this purpose, we model a two-level cache memory hierarchy. Each cache level's size, associativity, and replacement policy are the same as in the previous analysis (see section 4.5). In the last cache level, we enabled the random eviction policy.

We observe that using the random eviction module in the LLC increases the Prune filter iterations and, therefore, the PPP performances. For this reason, to evaluate the success rate for each parameter of the random eviction strategy, we perform the Prime–Prune–Probe algorithm 100 times. According to the analysis results in section 4.6, the size of the candidate set was fixed at 15K. Thus, the Prime–Prune–Probe algorithm has the best performance and a probability of 100% of being successful. Figure 4.8 shows the success rate of this algorithm, varying the eviction frequency and number of the cache lines to be evicted. As shown in Figure 4.8, the success rate is less than 1% when random cache lines are frequently evicted. Otherwise, the success rate increase with the increase in the eviction frequency. However, a small eviction frequency increases the miss rate, which drastically degrades the cache performance. Therefore, the eviction frequency and the number of evicted lines should be carefully chosen to not compromise security and performance. For example, suppose we evict 16 cache lines at each 10K memory accesses; in that case, the success rate of the Prime–Prune–Probe algorithm can be reduced from 100% to 0%.

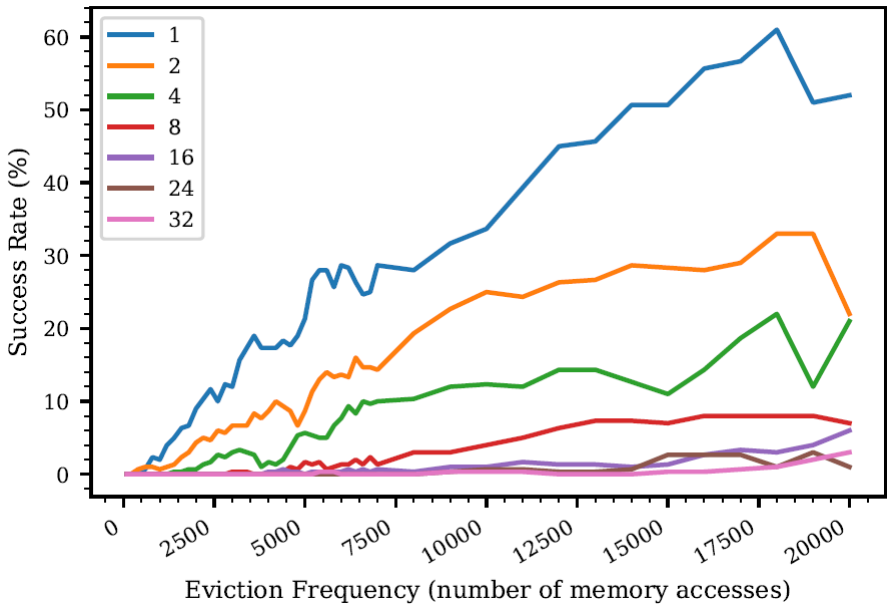


Figure 4.8 – The success rate of the Prime–Prune–Probe algorithm with different eviction frequencies and set sizes.

In the pruning phase, the Prime–Prune–Probe algorithm filter re-accesses the candidate set in three iterations at most for a cache with no countermeasures [Pur+21] in order to eliminate the addresses that evicted others. However, when the random eviction strategy is enabled, the pruning filter iterates many times due to unexpected evictions that are not caused by cache collisions. In this case, the pruning filter became aggressive and removed congruent addresses from the candidate set. Even when the attacker succeeded in constructing eviction sets and observing that the cache line had been evicted, it can not know whether the line was randomly evicted by the random eviction policy or by the victim's accesses. This prevented the attacker from creating minimal eviction sets.

4.8 Conclusion

This work provides an experimental security analysis of randomly mapped low-level caches using a noise-free cache simulator. Our study tried to determine a suitable remapping interval. We show that when the mapping changes every 260 K accesses, the success rate of finding an eviction set using the group testing algorithm is less than 1%, leading to improved security against conflict-based cache timing attacks. However, randomizing the lower cache levels does not protect against the Prime–Prune–Probe algorithm. To mitigate against this algorithm, we proposed using a random eviction

policy in the last cache level, which disrupts the attacker's observations and undermines the Prune filter. Depending on the tradeoff between security and performance, randomly evicting cache lines in time mitigate the conflict-based- cache attacks to perform the PPP algorithm to construct efficient eviction sets. We observe that using this solution, the PPP's success rate decreases from 100% to 0% when the LLC randomly evicts 16 cache lines for each 10k memory access.

5

Mitigation of Cache Attacks on lower cache levels

In this chapter, we propos the ScrambleCache, a novel dynamic randomized cache architecture that defeats cache side-channel analysis. Unlike other architectures, the ScrambleCache employs a lightweight permutation function that requires only a few logic gates, making it ideal to not increase the hit latency. The permutation function uses a pseudo-random number, which is changed at least at every context switch. We demonstrate that this countermeasure allows protecting the system against known cache-based side-channel attacks, while guaranteeing small performance and area overheads.

5.1 Motivation

Cache memories are among the most important sources of microarchitecture leaks, seriously threatening many applications. Many solutions and countermeasures exist in the literature. Nevertheless, most of them target the last-level cache since it is shared among all CPU cores, allowing cross-core cache attacks. However, lower-level caches (e.g., L1 caches) are not protected against cache attacks, even if higher-level caches are unprotected. This chapter aims to develop a feasible approach to mitigate conflict- and sharing-based attacks on lower-level caches (e.g., L1 data and instruction caches). Due to limitations in the literature, our research focuses on providing a secure dynamic randomized L1 cache. In order to develop a feasible solution, it must not only provide high security against cache attacks, but also have the following properties: ① low performance overhead, ② ease of implementation and low memory overhead, and ③ should not rely on software or OS support.

Many challenges must be considered when implementing a randomized L1 cache. First, L1 caches are in the critical path of processors. Therefore, adding a indirection layer (e.g., permutation function, redirection table, etc.) to randomize the address-to-set mapping should not increase the cache hit time. Second, for randomized caches with a write-back policy, the memory contents are no longer valid when the address-to-set mapping changes. Therefore, cache lines modified with the old mapping must be written back to main memory or moved to the correct location to maintain a consistent memory view.An intuitive solution is to identify the dirty cache lines by scanning the dirty bit of each cache line. Another solution is to flush the entire cache . Both of these solutions have a significant impact on performances if the mapping changes frequently and are therefore not scalable.

In this chapter, we propose `ScrambleCache`, a new dynamic randomized cache architecture to defend against attacks on lower-level caches. The `ScrambleCache` uses a random permutation of sets and allows the developer to adjust the security level of the system through a configurable remapping period. We demostrate that our architecture mitigates sharing-based cache attacks and conflict-based cache attacks with less performance overhead.

5.2 Detailed Architecture

5.2.1 Overview

The `ScrambleCache` implements dynamic randomization of cache sets locations. Unlike partitioning techniques, this approach allows full cache sharing, which could be beneficial to performance. The `ScrambleCache` uses a lightweight keyed permutation function π_r to permute the cache indexes. The permutation can be renewed at any time by changing the secret key r. This can be done either after a fixed number of cycles, a fixed number of accesses to the cache, interruptions, or context switches. The remapping period should be identified depending on the tradeoff between security and performance to avoid side-channel leaks in caches with less performance overhead.

Ideally, the `ScrambleCache` would change the remapping as frequently as possible in order to spread memory accesses across the whole cache sets, as shown in Figure 5.1. Interestingly, randomization tends to reduce internal cache collisions, or at least makes them harder to exploit, since addresses are no longer assigned to a fixed set anymore.

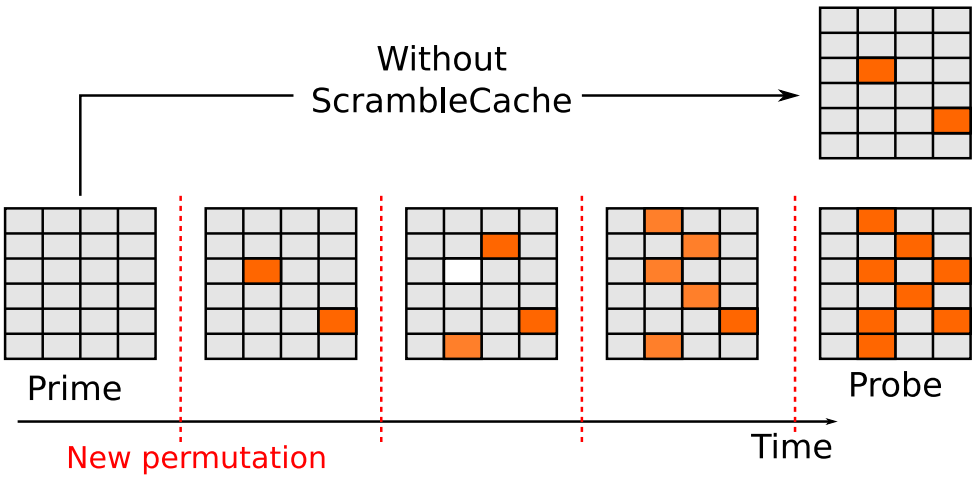


Figure 5.1 – Example of the `ScrambleCache` behavior

However, changing the cache address mapping causes some problems:

1. *coherency*: When the mapping changes, and thus any modified data (dirty lines) must be remapped to their new localization or must be written back to the higher cache level. To avoid this problem, we introduce a new mechanism to track the dirty lines and thus return their indexes when the mapping changes without the need to scan the whole cache.

2. *aliasing*: Due to the remapping change, dynamic randomization does not preserve unique data positions, and the same data may be valid in different cache sets.

`ScrambleCache` supports aliasing if the data is shared read-only. Otherwise, when the data is a writable shared memory, the cache mapping must be unique to avoid coherency problems when another process modifies the same data.

3. *performance*: Changing the cache mapping frequently leads to several cache misses. Each time the mapping changes, the whole cache content is invalid, degrading the system's performance. To avoid this issue, we propose a history mechanism to keep track of recently used mapping while preserving security properties.

The next sections will describe how these challenges are tackled to enable successful cache randomization in lower cache levels.

5.2.2 Hardware Architecture

We describe the `ScrambleCache` starting from a set-associative cache that is physically tagged and virtually indexed, using the S bits of the effective address (the set bits) to locate the cache sets. The cache contains 2^S sets, each consisting of N lines, giving a total size of $N \times 2^S$ cache words. The `ScrambleCache` architecture is depicted on Figure 5.2. Compared to a set associative cache, the `ScrambleCache` architecture contains four important elements: ① the permutation function, ② the pseudo-random number generator (PRNG), ③ the history table (HT), and ④ the rekeying management unit (RMU).

The secret permutation keys come from a cache-internal PRNG that is initially seeded when the system is reset. The behavior of `ScrambleCache` is controlled by the finite-state machine (FSM), which is also used to manage the various hardware components. The following sections describe in detail the different hardware components of the `ScrambleCache` design.

5.2.3 Address Permutation Properties

The `ScrambleCache` applies a permutation to the accessed CPU address. Any address transformation (bijective or not) can be applied since the cache continuously checks the physical tags before delivering the data to the CPU. We represent the permutation as a function $\pi_r(a)$, where r is a randomization parameter (the secret key), and a is the CPU address.

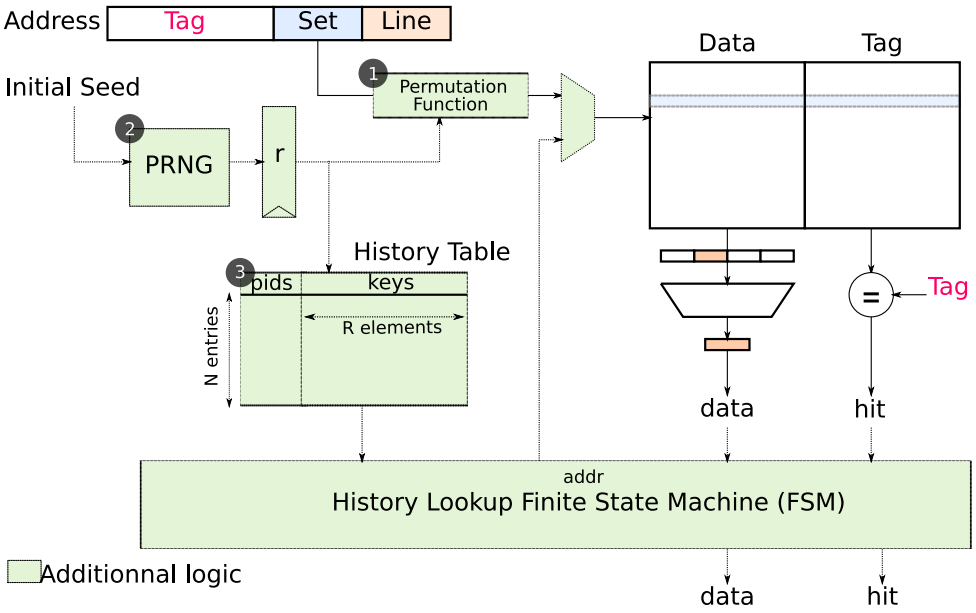


Figure 5.2 – The `ScrambleCache` Architecture

A cheap way to implement a permutation in hardware is to use an "eXclusive-OR" (XOR) function, with $\pi_r(a) = a \oplus r$. However, a XOR allows only a limited number of permutations. For example, if $S = 8$ (the cache has 256 sets), only 256 out of the $256!$ possible permutations can be reached with an XOR-based permutation. Therefore, an attacker can successfully perform an exhaustive search for the secret permutation key r. However, if the permutation changes several times between the attacker's analyzes (as shown on Figure 5.1), all intermediate random values of r must be recovered, making the search more complex. But changing the permutation comes at a cost in terms of performance.

However, XOR-based permutation is too weak to randomize the cache sets. It was demonstrated that XOR permutation fails cache side-channel security metrics [Zha+13]. Therefore, we propose an alternative permutation family in `ScrambleCache` with a wider permutation key input. The permutation is constructed by adding a layer of bit shuffling after the XOR operations, which can be viewed as a randomized barrel shifter, adding non-linearity to the permutation function. Formally, the π function has the form as in Equation 5.1, where the CPU address a contains only the tag and set fields, and the result is masked to only contain S-bit.

$$\pi_r(a) = f(a \oplus r_0, r_1) \tag{5.1}$$

The function f is defined recursively by dividing the binary representation of its input

into two equal parts. The permutation of a 2-bit wide input is denoted by `cswap`, which stands for "conditional swap". Let $n = |s|$ be the bit length of $f(a, r)$. The function f first computes w of size n bits, where the i-th bit is defined as follows:

$$w[i] = \begin{cases} \texttt{cswap}(a[i], a[i + n/2], r[i])[0] & \text{if } i < n/2 \\ \texttt{cswap}(a[i - n/2], a[i], r[i])[1] & \text{otherwise} \end{cases} \tag{5.2}$$

Then the binary w representation is split into two equal parts w_0, w_1 and the recursive formula $f(a, r) = f(w_0, r)||f(w_1, r)$ is applied. Figure 5.3 shows the permutation graph of f for the 4-bit wide inputs.

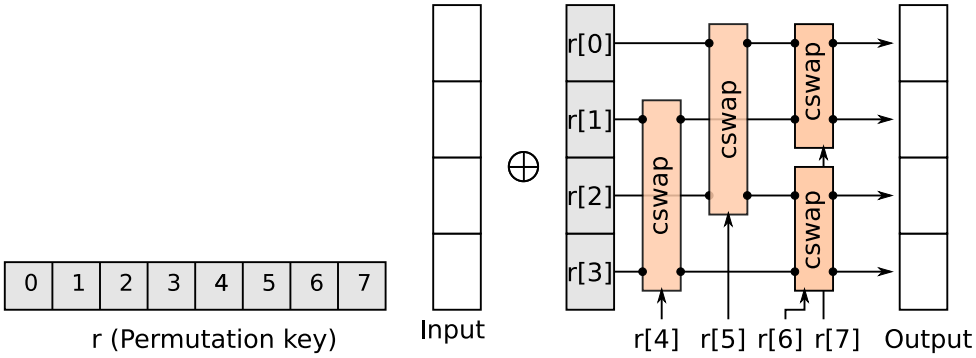


Figure 5.3 – Example of the f permutation for 4-bit wide addresses

5.2.4 History Mechanism

The history mechanism is a core element of the `ScrambleCache` design. It reduces the miss rate by keeping a history of previously generated permutation keys. For this purpose, R previous permutation key values are stored in a *history table* (see Figure 5.2). The history table is searched when the accessed data are not found in the cache with the current mapping (e.g., with the last permutation key value). Suppose the data are found with an old mapping. In this case, it will be invalidated in the cache at its old position and then moved to the new position calculated with the current value of the permutation key. Otherwise, a cache miss is triggered if the old permutation keys are scanned unsuccessfully.

To ensure strict process isolation, we assign a process identifier to each entry in the history table. Our architecture can support dynamic partitioning that provides hardware isolation between different running processes by using process identifiers in the history table. The goal of adding process identifiers is to protect cache memory from

cache side-channel attacks based on page sharing. Each entry in the history table is associated with a particular process and contains the previous R generated permutation keys associated with it. We denote N as the number of processes supported by the history table simultaneously. Unlike other dynamic partitionned caches, our partitioning is based on the history table which allow all running processes to use all available cache resources.

Algorithm 11: History table lookup algorithm

Input: $key \leftarrow$ permutation key, $pid \leftarrow$ process identifier
Output: $req \leftarrow$ read/write flag

```
1  if req is read then
2      if pid found in the history table then
3          return keys[pid], hit_ht
4      else
5          return miss_ht
6  end
7  else
8      if pid found in the history table then
9          index ← get_last_key_index(keys[pid])
10         if index > R then
11             victim_key ← keys[pid][0]
12             keys[pid][0] ← key
13             return victim_key
14         end
15         keys[pid][index] ← key
16     end
17     else
18         index ← get_old_pid_index()
19         if index > N then
20             index ← 0
21         victim_pid ← pids[index]
22         pids[index] ← pid
23         keys[pid][0] ← key
24         return victim_pid, miss_ht
25     end
26 end
```

Upon a miss, the cache yields control to the FSM, which maintains the mapping history of the N supported processes. As shown in algorithm 11, both write and read requests access the history table. In both requests, the process identifier is checked first. If the history table is accessed in reading, the accessed process pid already exists in the table. Therefore, the previous R permutation keys are returned to FSM so that they can be used to compute the cache set of the accessed data one by one. Otherwise, a history table miss is sent to the FSM to generate a cache miss. When the history table is accessed in write, and the accessed process pid already exists, the input permutation key key is stocked to its history table entry. Otherwise, a victim history table entry is allocated

when the process identifier pid is not found. In both cases, whether the accessed process exists, if the history table entries or the permutation keys associated with this process are full, a victim entry is selected and sent to FSM to write back the addresses fetched by that process.

5.2.5 Key Management and Re-Keying

When changing the mapping, we still have to write back all the cache lines that the history mechanism can no longer trace. In other words, every time the history table is full, a newly generated random permutation key must be written to the table; the history mechanism first identifies an old victim permutation key to replace. To avoid coherency problems due to the dirty cache lines mapped with the victim permutation, the associated process identifier is sent to the FSM to write back the dirty lines associeted to the couple (PID, key). A naive solution to find modified data in the cache is to scan all cache sets linearly, and all lines marked as dirty are written back to the next memory hierarchy level. This solution degrades performance because all cache lines must be scanned individually. In the best case, when there are no dirty lines, this solution requires L cycles to scan all cache lines, where L is the number of cache lines.

Algorithm 12: Re-keying management unit algorithm

Input: idx_i permuted dirty cache set index,
$pid_i \leftarrow$ process identifier,
$isDirty$ dirty flag bit
Input: $FSM_{req} \leftarrow$ request command
Output: $idx_o \leftarrow$ dirty cache set index

```
1  if FSM_req is read then
2      if is_clean(dirty_traces[pid_i]) then
3          return false
4      idx_o ← get_index_first_bit_set(dirty_traces[pid_i])
5      dirty_traces[pid_i][idx_o] ← 0
6      return idx_o
7  end
8  else if FSM_req is write then
9      if !isTracked(pid_i) then
10         setCleanEntry(pid_i)
11     end
12     /* The isDirty flag is set 0 when the cache set idx_i is written back; otherwise is
          set 1 when the set is dirty */
13     dirty_traces[pid_i][idx_i] ← isDirty
14 end
15 else if FSM_req is clean then
16     CleanEntry(pid_i)
17 end
```

To reduce the rekeying time, we use the Re-keying Management Unit (RMU), an independent component that tracks the dirty lines of each process. This way, each time a victim permutation key in a particular entry in the history table needs to be replaced, the FSM sends a signal to the rekeying management unit (RMU) to retrieve the indices of the various dirty cache lines associated with that process. The RMU component consists of R registers with S bits, where S is the number of cache sets and N is the number of allowed processes in the history table. We denote $dirty\_traces$ the matrix of NxS. On each write memory access, the permuted cache set index idx_i and the process ID pid_i are sent to the RMU. The RMU component keeps track of dirty cache sets by setting the associated bit of the permuted index to one. As illustrated in algorithm 12, each time the cache mapping of the process is changed, a read request is sent to the RMU component with its process identifier. The RMU component returns the first dirty cache localization index idx_o to the FSM and sends a write-back request for this line. This request should be sent to the RMU until no dirty cache lines associated with the requested process is found. If no dirty line is found with the requested process identifier, the RMU component returns *false* to indicate to the FSM that this process is clean.

5.2.6 Integration of ScrambleCache into Existing Microarchitectures

The `ScrambleCache` is a generic solution for building randomized lower caches that mitigate conflict- and sharing-based side-channel attacks. Depending on the target processor, the design of `ScrambleCache` can isolate different processes and security domains from each other via the pid_i input in the history table. However, this information is only known in the software, especially in the operating system. Since the `ScrambleCache` does not provide software support in order to prevent privileged processes from exploiting software vulnerabilities and thus bypassing our countermeasure. Depending on the processor architecture, an alternative to process identifiers may be found in hardware. On modern processors, Page Table Base Register (PTBR) is used to find the page directory of the running process. The PTBR is unique for each process and can be used as a process identifier.

ARM and Intel processors already support a similar mechanism for assigning identifiers to address spaces for each process at the hardware level. The x86 architecture defines Process Context IDentifiers (PCIDs) to identify running processes defined in the TLB. Similarly, the ARM architecture defines Address Space Identifiers (ASIDs) for the same purpose. On all Intel processors that support PCIDs, the size of a PCID is 12 bits. Therefore, they form the bits 11:0 of the CR3 register (PTE). Therefore, the maxi-

mum number of PCIDs that can be used simultaneously is 4096. On the other hand, the processors of ARM suggest an 8-bit ASID to support 256 processes simultaneously.

5.3 Gem5 Simulator

Gem5 [Bin+11; But+12] is a timing-accurate microarchitectural simulator resulting from the merging of two earlier projects: GEMS [Mar+05], for memory timing, and M5 [Bin+06], for accurate modeling of CPU. Unlike a trace-driven simulator such as SimpleScalar [BAK04], Gem5 focuses on timing accuracy. It executes instructions in its microarchitecture models only after all dependencies have been resolved, called execute-in-execute modeling. Currently, Gem5 supports all standard commercial ISAs (ARM, ALPHA, MIPS, Power, SPARC, and x86), and can boot a Linux operating system on three of them (ARM, ALPHA, and x86).

The Gem5 simulator supports multiple CPU architectures, memory models, system call emulation, and different modeling granularities (functional or temporal). This flexibility makes Gem5 a very powerful tool for design exploration.

- **CPU Model.** The Gem5 simulator currently provides different CPU models: *(i) AtomicSimple, (ii) TimingSimple, (iii) In-Order, and (iv) Out-Of-Order (O3)*, which differ in speed/accuracy trade-offs. AtomicSimple is a minimal single CPU model, *TimingSimple* is also a non-pipelined model but also simulates the timing of memory references. *In-Order* is a pipelined, in-order CPU, and *O3* is a pipelined model used to simulate out-of-order/superscalar, and simultaneous multithreading (SMT) CPU model. Both the O3 and InOrder models execute instructions in its microarchitecture models only after all dependencies have been resolved, called execute-in-execute modeling [Bin+06].
- **System Mode.** Two different modes are supported by Gem5 simulator: *(i) system emulation (SE) and (ii) full system (FS) mode*. The SE mode emulates avoids the need to model devices or an operating system (OS) by emulating most system-level services, resulting in a significant simulation speedup at the cost of limited support for some functionalities such as multithreading. On the other hand, the FS mode simulates a bare-metal computer. Meanwhile, FS mode executes both user-level and kernel-level instructions and models a complete system including the OS, thread scheduler and devices.
- **Memory System.** The Gem5 simulator includes two different memory system models: *(i) Classic and (2) Ruby*. The Classic model (from M5) provides a fast and

easily configurable memory system, while the Ruby model (from GEMS) provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherent memory systems.

5.4 Security Evaluation and Discussion

In the following, we investigate the security of `ScrambleCache` in terms of state-of-the-art side-channel attacks for both conflict and sharing based cache side-channel attacks.

5.4.1 Applicability of Cache Attacks

Unlike conflict-based cache side-channel attacks, sharing-based cache attacks require that the target cache line be shared between the attacker and the victim. A shared cache line is the result of a shared memory region. Conflict-based cache attacks do not require sharing but exploit the shared cache itself. Therefore, the applicability of cache attacks to `ScrambleCache` can be analyzed based on whether the target address memory is shared between the attacker and the victim.

Shared, read-only memory. Read-only memory is often shared between different processes, for example, in the case of shared libraries. ScrambleCache is secured against sharing-based cache attacks on read-only memory by adding process identifiers to each entry in the history table. Since each process uses its own permutation keys, mapping a shared read-only address is duplicated for each process. In other words, the cache lines that belong to the same shared memory area are no longer shared in the cache between different processes and security domains. Therefore, reloading data into the cache or flushing data from the cache does not provide information about other processes' access to the shared memory cache lines. It should be noted that the lack of sharing between the victim and attacker processes implies the impossibility of sharing-based cache attacks (e.g., Flush+Reload, Flush+Flush, and Evict+Reload).

Shared, writable memory. The exchange of data between processes requires shared writable memory. To ensure cache consistency, writeable shared memory regions must always use the same cache line and, thus, the same process identifier for that particular memory region. For this reason, the ScrambleCache uses a dedicated permutation key for these addresses without considering the process ID entering the cache. At the same

time, cache attacks on these shared memory regions require the use of flush instructions. For this reason, we recommend that writable shared memory be used only as an interface for non-secret data transfer rather than for sensitive computations. Cache attacks that rely on sharing without the need to use a cache maintenance instruction, such as the Evict+Reload technique, using dynamic randomization caches increase the difficulty of building efficient eviction sets.

Unshared memory. Unshared memory regions never share the same cache line, making attacks such as Flush+Reload, Flush+Flush, and Evict+Reload infeasible. However, as the cache resources are shared, conflict-based cache attacks such as Prime+Probe are still possible. However, as we showed in chapter 4, these attacks cannot be avoided as long as the cache is shared between the attacker and the victim process. Furthermore, the ScrambleCache dramatically increases the complexity of conflict-based cache attacks by increasing the complexity of constructing an eviction set.

5.4.2 Side-channel Vulnerability Factor Evaluation

Considering the conflict-based attacks, we evaluate the security using the Side-channel Vulnerability Factor (SVF) [Dem+12]. We stress that we do not treat it as an absolute security indicator but rather as a tool to discuss the effect of the parameters of the architecture. The SVF metric compares *oracle traces*, with the *attacker traces* (i.e., side-channel measurements). To model a Prime+Probe attack, the oracle and attacker traces (denoted $\vec{v}_o$, $\vec{v}_a$) are both binary vectors that contain 1 if the set was accessed at least once during a fixed period and 0 otherwise. The trace $\vec{v}_a$ is usually deduced by measuring the time of memory accesses and comparing them to a determined threshold between hit and miss time. By running a program on the system, one obtains k traces for the oracle and the attacker: $(\vec{v}_o(1), \ldots, \vec{v}_o(k))$ and $(\vec{v}_a(1), \ldots, \vec{v}_a(k))$. From these observations, the SVF starts by building a similarity matrix:

$$M_{i,j} = \begin{cases} D(\vec{v}_o(i), \vec{v}_a(j)) & \text{if } i < j \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

The matrix (Equation 5.3) uses a distance function $D(\cdot, \cdot)$ to compare two traces. We stick to the choice of Demme et. al [Dem+12] and use the Euclidean distance for D, which intuitively quantifies how much two cache probings differ. Then, the SVF is evaluated by computing Pearson's correlation between the two matrices element-wise.

To compute the SVF, we run a quicksort algorithm that sorts a random array of 32K elements on the Gem5 system model using system emulation. This program is interesting since it makes both linear and random memory accesses. The `ScrambleCache` model in Gem5 is instrumented to build the oracle and the attacker trace. The cache is probed on a fixed number of cycles fixed to 5000 in our experiments. The SVF is computed by grouping 256 consecutive cache probings. As our program contains more than 15k cache probings, we retain the highest SVF value observed.

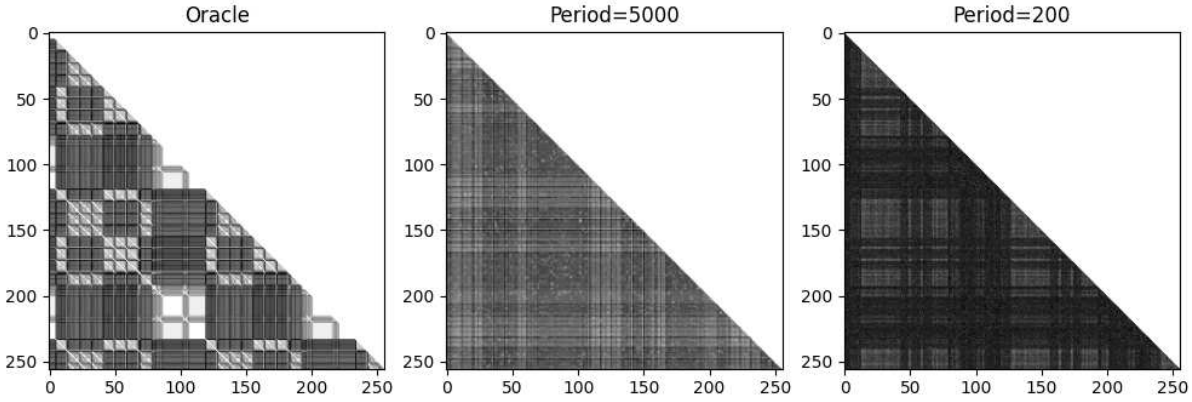


Figure 5.4 – Similarity Matrix between the oracle and attacker observations for a 32kB cache

Some similarity matrices are shown in Figure 5.4. It can be observed that the similarity patterns are drastically reduced with the `ScrambleCache` when the remapping period increases. Indeed, the matrices become darker, meaning that all probings become very distinct.

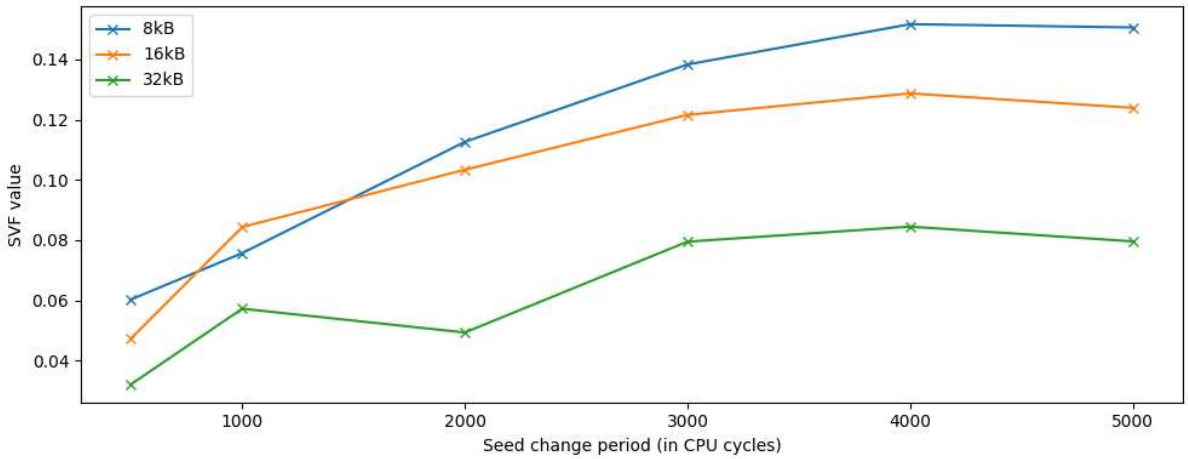


Figure 5.5 – Evolution of SVF with the remapping period for different cache sizes

The evolution of the SVF with the remapping period is shown in Figure 5.5. As ex-

pected, the more frequently the permutation key changes, the higher is the SVF. One can also see the effect of the cache size on SVF. Namely, bigger caches appear more secure considering the SVF metric.

5.4.3 Complexity of Prime+Probe Attack

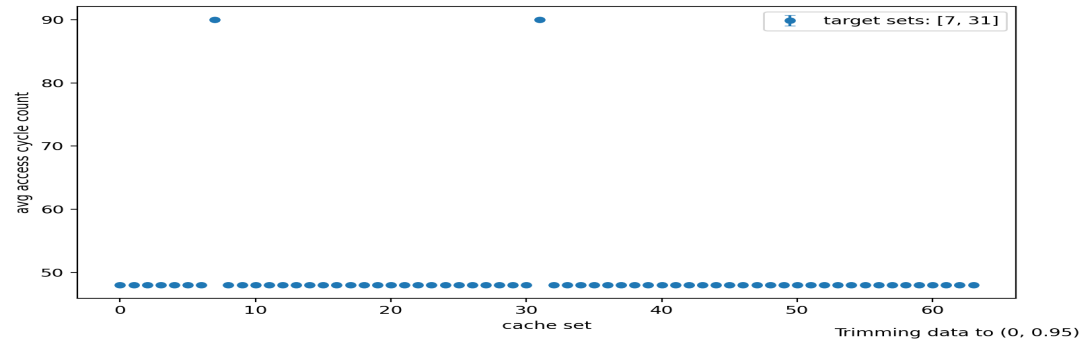
For conflict-based cache side-channel class of attacks (such as the Prime+Probe technique), the `ScrambleCache` adds extra noise to the victim memory access patterns to make them unexploitable, and thus, making the construction of eviction sets harder. The noise level is directly related to the remapping period. To validate this statement, we perform a simple Prime+Probe attack on the `ScrambleCache` (L1 data cache) using the Gem5 model. The victim and the attacker run in the same process. Between Prime and Probe, the victim reads two fixed memory locations once.

We traget a system with 4-ways L1 caches sized 32kB, and the L1 data cache use ScrambleCache design with a history table of 4 entries and support 16 permutation key per process. Figure 5.6 shows the average cache access times measured by the attacker after the victim execution for each cache set. Times are expressed in cycles and obtained by the x86 instruction `rdtsc`. As measurements are noisy (especially with the `ScrambleCache`), we repeat the experiment 100 times on the unprotected cache and 10,000 times on the `ScrambleCache` with a remapping period of 10,000 memory accesses. For the latter, we plot the mean (as a circle) and the standard deviation (as a bar) of the measurement. It can be observed by comparing Figure 5.6a and Figure 5.6b that the `ScrambleCache` completely hides the memory access pattern, making Prime+Probe and address conflicts attacks impractical.

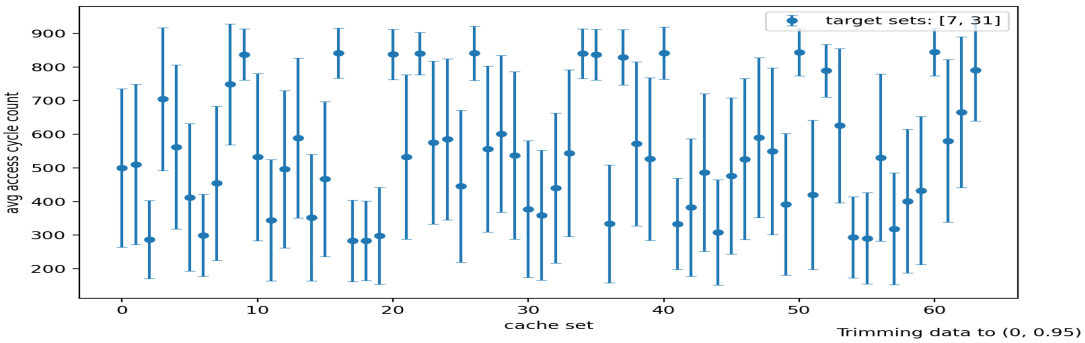
5.5 Performace Analysis

5.5.1 Experimental Methodology and Configuration

The `ScrambleCache` significantly increases the effort of attackers to perform cache-based side-channel attacks. However, a countermeasure must not degrade performances to be practical as well. In this section, we analyze the performances of `ScrambleCache` using the Gem5 simulator.



(a) Baseline set-associative configuration



(b) ScrambleCache configuration

Figure 5.6 – Single cache line PRIME+PROBE attack on both ScrambleCache and unprotected cache

Targeted System

We performed our cache evaluation using the Gem5 full system simulator in 32-bit x86 mode. In particular, we used the CPU model O3 (Out-Of-Order) together with a cache architecture such as commonly used in enbaded ARM CPUs: the cache line size was chosen to be 64 bytes, the 4-way L1 data and instruction caches are each sized 32 kB. We adapted the Gem5 simulator such as to support ScrambleCache for the L1 data cache. This allows to evaluate the impact of different cache configuration. The main memory is a timing model of a single channel DDR3-1600 DRAM.

Workloads

The *Mibench*[Gut+01] benchmark suite, is used for evaluating the performance impact of the `ScrambleCache` architecture on general-purpose workloads. It is composed of applications from different categories and provides two profiles: small and large. The small data set represents a lightweight version of the benchmark, useful embedded ap-

plications. In contrast, the large data set provides a more stressful real-world workload, which contains more than 750M dynamic instructions (Table 5.1).

Benchmark	Instruction Count (Small)	Instruction Count (Large)
basicmath	65,459,080	1,000,000,000
bitcount	49,671,043	384,803,644
qsort	43,604,903	595,400,120
susan.corners	1,062,891	586,076,156
susan.edges	1,836,965	732,517,639
susan.smoothing	24,897,492	1,000,000,000
dijkstra	64,927,863	272,657,564
patricia	103,923,656	1,000,000,000
CRC32	52,839,894	61,659,073
FFT	52,625,918	143,263,412

Table 5.1 – Benchmark Sizes

To evalute the performances of our architecture, we first evalute the baseline system without countermeasures, to obtain a reference time. Then, for different parameters, we replace the L1 data cache with the Scramble Cache and compare the performance results against the baseline configuration. Our analysis focuses on the following parameters: the size of the L1 cache, the depth history table, and the permutation change interval (expressed in number of cache accesses).

5.5.2 Sensitivity to Remapping Period

The remapping period has a direct impact on the security of the `ScrambleCache`. The smallest acceptable value for this parameter should be selected in order to maximize security. However, changing the permutation key more frequently implies more misses, which at some point cannot be compensated with a deeper history table. Thus, for the remapping period selection, a trade-off between security and performance is necessary. It can be seen from Figure 5.7 that when the remapping period is long enough, the `ScrambleCache` approaches the performances of the baseline cache without countermeasures. However, if the remapping is changing very frequently, the performances are decreasing very quickly.

5.5.3 Sensitivity to the History Table Depth

To evaluate the effect of the history table depth, we vary this parameter on a 32 KB 8-way L1 data cache. Figure 5.8 shows the runtime overhead of the `ScrambleCache` for different history table depths. As expected, the overhead tends to decrease with a deeper

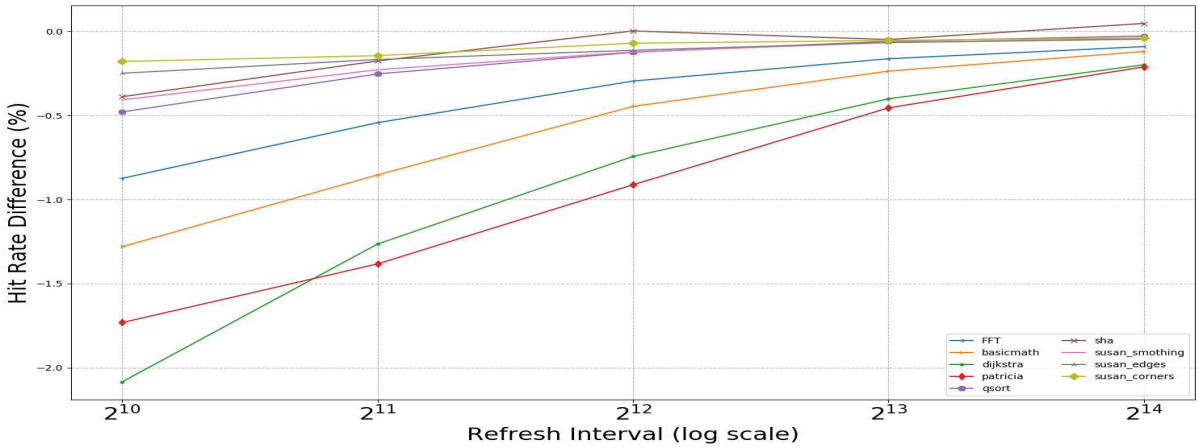


Figure 5.7 – Impact of remapping frequency on slowdown of `ScrambleCache`

history table. Indeed, increasing depth should reduce the number of misses caused by a permutation change. This is true up to a certain point. As we can observe on Figure 5.8, when the history table stores more than eight elements, the benefits are not always visible. At that point, the history table lookup (a linear search) is not a cheap operation anymore and it is almost as costly as reading directly from memory. In the case of a single element history table, at each time mapping change all cache content is considered invalid, so the number of requests to the main memory increases which impacts significantly the execution time. The use of a larger table can compensate this degradation.

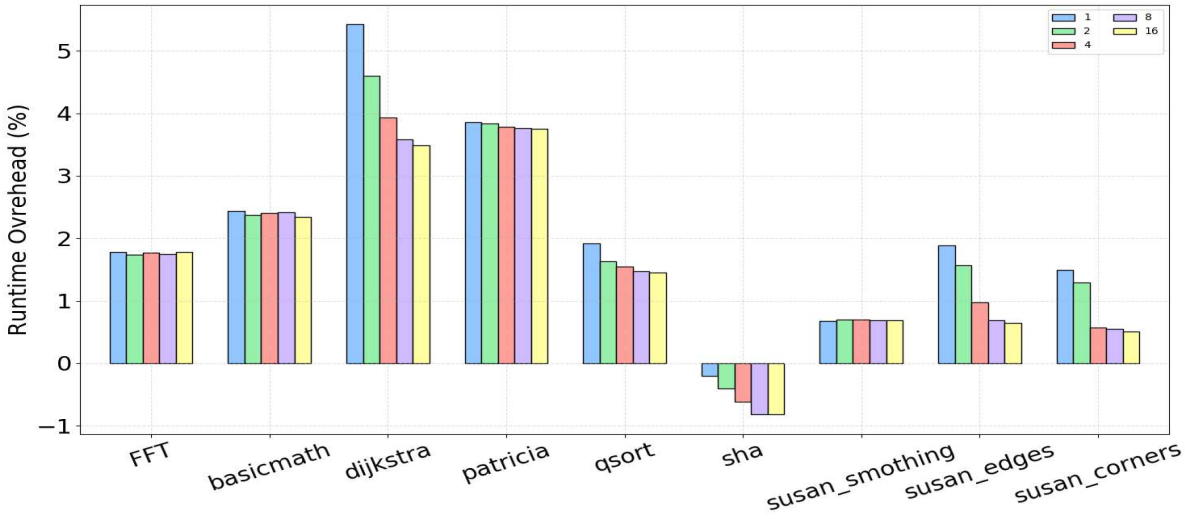


Figure 5.8 – Impact of history table depth on `ScrambleCache` performance

5.5.4 Sensitivity to the Cache Capacity

Figure 5.9 shows the hit rate change in a 4kB, 8kB, 16kB, and 32kB L1 data cache for different workloads. A higher hit rate overhead is better and 0% denotes no degradation. In comparison with the baseline cache, the average performance loss by the `ScrambleCache` using is 0.49% in the worst case on the patricia benchmark. These results suggest that as the cache size increases, the `ScrambleCache` needs more time to write back all the dirty lines when the history table is full. With the exception of two workloads, which perform better because of the reduction in conflict misses by changing the memory-to-cache mapping.

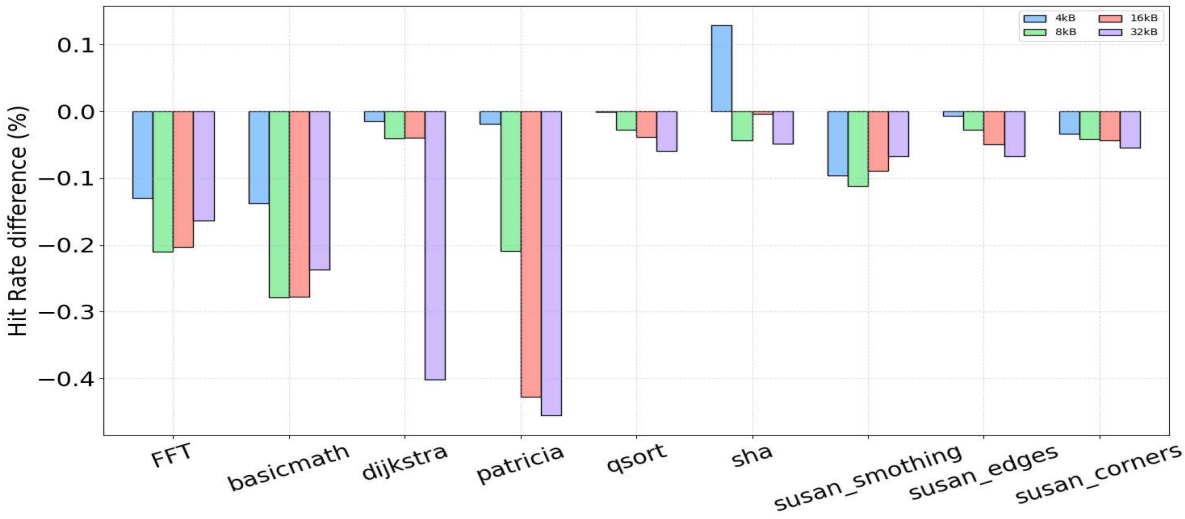


Figure 5.9 – Impact of L1 data capacity on `ScrambleCache` performance

5.6 Conclusion

In this chapter, we presented `ScrambleCache` an architecture that implements efficiently a lightweight set permutation. The core elements of the `ScrambleCache` are its dirty cache lines tracking and history mechanisms, both allowing to frequently change the permutation. Thanks to the `ScrambleCache` modeling in the Gem5, we observed that conflict-based cache attacks are made much harder. The benchmarks demonstrated that the history table depth allows a trade-off between security and performances (under the assumption that changing the mapping more frequently improves security). Indeed, with the permutation changing every 8192 accesses, the overhead on the execution time is below 4%, and we already observed security improvements against conflict- and sharing-based cache timing attacks. Furthermore, its access latency is drastically

reduced thanks to a cheap permutation, making this architecture usable as a first level cache even in constrained environments such as embedded systems.

6 Conclusion

6.1 Summary and Conclusion

In the microarchitecture industry, from embedded systems to high-end systems, the security of hardware components is fundamental to ensure the confidentiality of the sensitive data they contain. Indeed, some malicious programs are able to retrieve information about other running programs by exploiting hardware vulnerabilities at the memory hierarchy level. This thesis aims to propose protection schemes against cache-based side-channel attacks. This thesis has made the following contributions, which hopefully will help the research field gain a better understanding of the threats in cache memories and develop new secure caches against cache-based side-channel attacks.

We first studied the various cache-based side-channel attacks and proposed a modular framework, libMAAT, that contains the primitives needed to efficiently implement cache attacks on different processor architectures. The flexibility of this library is provided by a modular implementation that can be easily deployed on different target systems. In particular, this approach allows the primitives for microarchitectural attacks to be refined and added to over time.

LibMAAT has allowed us to accurately understand the threats present in cache memories. LibMAAT was also developed as an experimental tool to evaluate the feasibility of microarchitectural side-channel attacks on microarchitectures with secure caches. Leakage analysis of different levels of cache memory using libMAAT provided insight into cache vulnerabilities. We showed that a malicious process can put a cache line into a known state by using a cache maintenance instruction, such as the clflush instruction in x86 architectures or an eviction set with a group of congruent addresses. If the victim process and the malicious process share the same cache, the attacker uses an un-

privileged time source to observe the victim process's activities in the cache. We have shown that by exploiting those leakages, an attacker is able to retrieve sensitive information such as the encryption key AES. This analysis was performed on various ARM processors. We found that noise caused by various optimization techniques in modern microarchitectures and noise caused by other processes using the same memory hierarchy can complicate the analysis of conflict-based cache attacks. This led us to implement a noise-free model to explore the various algorithms used in the literature to construct a conflict-based cache attack.

Our proposed cache simulator fills a gap in the experimental security evaluation of inclusive memory hierarchy. It provides a flexible framework that supports secure cache architectures, particularly randomized caches. We used this framework to perform an experimental analysis of the resilience of randomized caches at different levels against algorithms for constructing eviction sets. In a two-level cache system, we show that by dynamically randomizing cache mapping at the lower level with a carefully predefined remapping period, the success rate of single address elimination and group testing algorithms drops dramatically to less than 1%. However, randomization of the lower cache levels does not provide protection against the Prime–Prune–Probe algorithm in other cache levels. In this context, we proposed to randomly evict a number of cache lines from the LLC to disrupt the attacker's observations and thus mitigate the impact of the Prime–Prune–Probe algorithm. We conclude that cache randomization is an effective defense that is sufficiently secure and has good chances of being used in the next generations of processors since it introduces less overhead into the existing cache structure. It should be noted that dynamically randomized caches increase the performance overhead whenever the remapping is changed. Hence a careful choice of the remapping period is necessary to achieve a performance security tradeoff.

Designing lower-level randomized caches, especially L1 caches, is challenging because they are very close to the CPU pipeline and therefore have the greatest requirements in the access time. In addition, adding a randomization function, such as a hash function or encryption scheme, can significantly impact cache access time and reduce system performance. Another challenge in developing dynamically randomized caches is managing the dirty cache lines whenever the cache mapping is changed to avoid consistency issues. In this work, we propose ScrambleCache, a new first-level cache architecture based on a lightweight permutation function. The main feature of the ScrambleCache is its history mechanism, which records the previously used mappings for each process that uses the cache. We have shown that this mechanism can improve the performance of dynamically randomized caches without compromising security features. Our security analysis on the Gem5 simulator found that ScrambleCache in a

single-processor system precludes cache attacks based on sharing, thanks to the process identifiers. It also avoids conflict-based attacks by dynamically randomizing the cache mapping for each process. Still, those security improvements have a small impact on the performances with the same ScrambleCache configuration. We obtained a performance degradation of 1.8% on average and 4% in the worst case.

6.2 Future Works and Improvements

The work presented in this thesis offers several perspectives:

Chapter 3 - LibMAAT This work aims to evaluate the resilience of the proposed countermeasures against micro-architectural software side-channel attacks. It aims to bridge the gap between cache design evaluation and exploitation of actual leaks in experimental key recoveries. The toolkit focuses on cache-based side-channel attacks. Developing other micro-architectural side-channel attacks, such as those targeting prefetchers, out-of-order execution, speculative execution, etc., is possible. For this reason, identifying and implementing the primitives needed to prototype microarchitectural side-channel attacks, such as Spectre or Rowhammer techniques, in different CPU microarchitectures.

Chapter 4 - Noise-free analysis In this work, we focus only on analyzing the security of randomization in lower-level caches and exploring its limitations. An interesting research direction for the future is to design different randomized caches in the literature to evaluate their security against conflict-based cache attacks. In addition, finding other metrics to assess the security instead of the success rate of the target algorithms will also be interesting, e.g., integrating available side-channel metrics such as Side-channel Vulnerability Factor (SVF) or Signal-to-Noise Ratio (SNR).

Based on current trends and research, it is clear that new software side-channel attacks on microarchitectures will continue to emerge in the future. An important future research direction could be to systematically discover microarchitectural side channels, which could help the community anticipate and target their solutions accordingly. For this reason, investigating the causes of these side channels and their impact on information leakage could be helpful for future research in understanding microarchitectural side-channel attacks. Such an analysis would help to understand whether the information leaking through different side channels can add up or whether the side channels

interfere with each other.

Chapter 5 - ScrambleCache Architecture There are several directions for further improvements of ScrambleCache design. First, they are related to the permutation function and the remapping period; thus, an increase in the remapping period should be accompanied by a strong permutation function, as shown in chapter 4. In addition, the ScrambleCache architecture is inefficient when the cache supports multiple processes, as is the case with the last-level cache, due to the isolation added in the history table while does, not allow scalability in the LLC. Another improvement direction is to look for new strategies to implement randomized caches with process isolation in the LLC. Note that using both randomization and isolation to develop a secure cache is a prominent solution that should be explored to mitigate known cache-based side-channel attacks.

FOREWORD

Patents

-

International Publications

- Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Noise-free security assessment of eviction set construction algorithms with randomized caches", *in*: *Applied Sciences* **12**.*5* (2022), p. 2415
- Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation", *in*: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 621–626

Oral presentations and Posters

- Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Efficient Cache Architecture for Randomized Set Permutation in L1 Cache", *in*: RISC–V organization, Paris, France, 2022
- Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Strategies for Securing a Memory Hierarchy Against Software Side-Channel Attacks", *in*: Journées Codage & Cryptographie JC2 Day, 2022
- Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Cache attacks and Countermeasures", *in*: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, CNRS–TIMA, 2020

BIBLIOGRAPHY

[AAA17] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin, "When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks", *in*: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, IEEE, 2017, pp. 8–13.

[ABG10] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher, "New results on instruction cache attacks", *in*: *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2010, pp. 110–124.

[Aci07] Onur Aciiçmez, "Yet another microarchitectural attack: exploiting I-cache", *in*: *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[Acı+09] Onur Acıçmez et al., "Microarchitectural attacks and countermeasures", *in*: *Cryptographic Engineering*, 2009.

[Ago+07] Giovanni Agosta et al., "Countermeasures against branch target buffer attacks", *in*: *Workshop on fault diagnosis and tolerance in cryptography (FDTC 2007)*, IEEE, 2007, pp. 75–79.

[AK06] Onur Acıiçmez and Çetin Kaya Koç, "Trace-driven cache attacks on AES (short paper)", *in*: *International Conference on Information and Communications Security*, Springer, 2006, pp. 112–121.

[Alm+16] José Bacelar Almeida et al., "Verifying {Constant-Time} Implementations", *in*: *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.

[AS08] Onur Acıiçmez and Werner Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL", *in*: *Cryptographers' Track at the RSA Conference*, Springer, Berlin, Heidelberg, 2008, pp. 256–273.

[ASK07] Onur Acıiçmez, Werner Schindler, and Çetin K Koç, "Cache based remote timing attack on the AES", *in*: *Cryptographers' track at the RSA conference*, Springer, 2007, pp. 271–286.

[BAK04] Doug Burger, Todd M Austin, and Stephen W Keckler, "Recent extensions to the simplescalar tool suite", *in*: *ACM SIGMETRICS Performance Evaluation Review* **31**.*4* (2004), pp. 4–7.

[Bar+03] Paul Barham et al., "Xen and the art of virtualization", *in*: *ACM SIGOPS operating systems review* **37**.*5* (2003), pp. 164–177.

[Bar+14] Gilles Barthe et al., "System-level non-interference for constant-time cryptography", *in*: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.

[Bas14] VMware Knowledge Base, "Security considerations and disallowing inter-virtual machine transparent page sharing", *in*: *VMware Knowledge Base* **2080735** (2014).

[Ben+14] Naomi Benger et al., ""Ooh Aah... Just a Little Bit": a small amount of side channel can go a long way", *in*: *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2014, pp. 75–92.

[Ber+94] Brian N Bershad et al., "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*, 1994, pp. 158–170.

[Ber05] Daniel J Bernstein, "Cache-timing attacks on AES", *in*: 2005.

[Bin+06] Nathan L Binkert et al., "The M5 simulator: Modeling networked systems", *in*: *Ieee micro* **26**.*4* (2006), pp. 52–60.

[Bin+11] Nathan Binkert et al., "The Gem5 simulator", *in*: *ACM SIGARCH computer architecture news* (2011).

[BK07] Johannes Blömer and Volker Krummel, "Analysis of countermeasures against access driven cache attacks on AES", *in*: *International Workshop on Selected Areas in Cryptography*, Springer, 2007, pp. 96–109.

[BM06] Joseph Bonneau and Ilya Mironov, "Cache-collision timing attacks against AES", *in*: *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2006, pp. 201–215.

[Bod+20] Rahul Bodduna et al., "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser", *in*: *IEEE Computer Architecture Letters* (2020), pp. 9–12.

[Bou+20] Thomas Bourgeat et al., "Casa: End-to-end quantitative security analysis of randomly mapped caches", *in*: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 1110–1123.

[Bri+06] Ernie Brickell et al., "Software mitigations to hedge AES against cache-based software side channel vulnerabilities", *in*: *Cryptology ePrint Archive* (2006).

[Bri+19] Samira Briongos et al., "Cache misses and the recovery of the full AES 256 key", *in*: *Applied Sciences* **9**.*5* (2019), p. 944.

[BS15] Chongxi Bao and Ankur Srivastava, "3D integration: New opportunities in defense against cache-timing side-channel attacks", *in*: *2015 33rd IEEE International Conference on Computer Design (ICCD)*, IEEE, 2015, pp. 273–280.

[But+12] Anastasiia Butko et al., "Accuracy evaluation of Gem5 simulator system", *in*: *Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012.

[CD16] Victor Costan and Srinivas Devadas, "Intel SGX explained", *in*: *Cryptology ePrint Archive* (2016).

[Cra+15] Stephen Crane et al., "Thwarting cache side-channel attacks through dynamic software diversity.", *in*: *NDSS*, 2015, pp. 8–11.

[CSY16] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters", *in*: *Applied Soft Computing* **49** (2016), pp. 1162–1174.

[Dem+12] John Demme et al., "Side-channel vulnerability factor: A metric for measuring information leakage", *in*: *International Symposium on Computer Architecture (ISCA)*, 2012.

[Dem+13] John Demme et al., "A quantitative, experimental approach to measuring processor side-channel security", *in*: *IEEE Micro* **33**.*3* (2013), pp. 68–77.

[DFS20] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments", *in*: *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 451–468.

[Dis+17] Craig Disselkoen et al., "{Prime+ Abort}: A {Timer-Free}{High-Precision} L3 Cache Attack using Intel {TSX}", *in*: *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[DM21] Guillaume Didier and Clémentine Maurice, "Calibration Done Right: Noiseless Flush+ Flush Attacks", *in*: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2021, pp. 278–298.

[Dom+12] Leonid Domnitser et al., "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks", *in*: *Transactions on Architecture and Code Optimization (TACO)* (2012).

[Doy+15] Goran Doychev et al., "Cacheaudit: A tool for the static analysis of cache side channels", *in*: *ACM Transactions on information and system security (TISSEC)* **18**.*1* (2015), pp. 1–32.

[DR20] Joan Daemen and Vincent Rijmen, "Specification of Rijndael", *in*: *The Design of Rijndael*, Springer, 2020, pp. 31–51.

[DR99] Joan Daemen and Vincent Rijmen, "AES proposal: Rijndael", *in*: (1999).

[DXS18] Shuwen Deng, Wenjie Xiong, and Jakub Szefer, "Cache timing side-channel vulnerability checking with computation tree logic", *in*: *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[DXS19] Shuwen Deng, Wenjie Xiong, and Jakub Szefer, "Analysis of secure caches using a three-step model for timing-based attacks", *in*: *Journal of Hardware and Systems Security* **3**.4 (2019), pp. 397–425.

[ElG85] Taher ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *in*: *IEEE transactions on information theory* **31**.4 (1985), pp. 469–472.

[Fos+05] James C Foster et al., "Buffer overflow attacks", *in*: *Syngress, Rockland, USA* (2005).

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn, "Cache games - bringing access-based cache attacks on AES to practice", *in*: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505, URL: http://dx.doi.org/10.1109/SP.2011.22..

[Ge+18] Qian Ge et al., "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware", *in*: *Journal of Cryptographic Engineering* **8**.1 (2018), pp. 1–27.

[Gen+18] Daniel Genkin et al., "Drive-by key-extraction cache attacks from portable code", *in*: *International Conference on Applied Cryptography and Network Security*, Springer, 2018, pp. 83–102.

[GKT10] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall, "Improved trace-driven cache-collision attacks against embedded AES implementations", *in*: *International Workshop on Information Security Applications*, Springer, 2010, pp. 243–257.

[GMM16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript", *in*: *International conference on detection of intrusions and malware, and vulnerability assessment*, Springer, 2016, pp. 300–321.

[Gre+17] Marc Green et al., "{AutoLock}: Why cache attacks on {ARM} are harder than you think", *in*: *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1075–1091.

[Gru+16] Daniel Gruss et al., "FLUSH+FLUSH: a fast and stealthy cache attack", *in*: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2016, pp. 279–299.

[Gru+17] Daniel Gruss et al., "Strong and efficient cache side-channel protection using hardware transactional memory", *in*: *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 217–233.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches", *in*: *Proceedings of the 24th USENIX Conference on Security Symposium. SEC'15. Washington, D.C.* 2015, URL: http://dl.acm.org/citation.cfm?id=2831143.2831200.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis", *in*: *Advances in Cryptology – CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA*, Springer, 2014, pp. 444–461.

[Gul+15] Berk Gulmezouglu et al., "A faster and more realistic FLUSH+RELOAD attack on AES", *in*: *International Workshop on Constructive Side-Channel Analysis and Secure Design*, Springer, 2015, pp. 111–126.

[Gut+01] Matthew R Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite", *in*: *International workshop on workload characterization. WWC-4*, 2001.

[GZ14] Michael Godfrey and Mohammad Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment", *in*: *IEEE transactions on cloud computing* **2**.4 (2014), pp. 395–408.

[HC12] Gael Hofemeier and Robert Chesebrough, "Introduction to intel AES-NI and intel secure key instructions", *in*: *Intel, White Paper* **62** (2012).

[HL17] Zecheng He and Ruby B Lee, "How secure is your cache against side-channel attacks?", *in*: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.

[Hol09] ARM Holdings, "ARM security technology: Building a secure system using trustzone technology", *in*: *Retrieved on June* **10** (2009), p. 2021.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz, "Practical timing side channel attacks against kernel space ASLR", *in*: *2013 IEEE Symposium on Security and Privacy*, IEEE, 2013, pp. 191–205.

[IES15a] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar, "S $A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES", *in*: *Symposium on Security and Privacy*, 2015.

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar, "Systematic reverse engineering of cache slice selection in Intel processors", *in*: *2015 Euromicro Conference on Digital System Design*, IEEE, 2015, pp. 629–636.

[IES16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar, "MASCAT: Stopping microarchitectural attacks before execution", *in*: *Cryptology ePrint Archive* (2016).

[Int21] Intel, *Intel® 64 and IA-32 Architectures, Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 (Order Number: 325462-076US)*, 2021.

[Ira+14] Gorka Irazoqui et al., "Wait a minute! A fast, Cross-VM attack on AES", *in*: *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2014, pp. 299–319.

[Jal+10a] Aamer Jaleel et al., "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies", *in*: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2010, pp. 151–162.

[Jal+10b] Aamer Jaleel et al., "High performance cache replacement using re-reference interval prediction (RRIP)", *in*: *ACM SIGARCH Computer Architecture News* (2010), pp. 60–71.

[JHD20] Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Cache attacks and Countermeasures", *in*: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, CNRS–TIMA, 2020.

[JHD21] Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Scramble Cache: An Efficient Cache Architecture for Randomized Set Permutation", *in*: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 621–626.

[JHD22a] Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Efficient Cache Architecture for Randomized Set Permutation in L1 Cache", *in*: RISC–V organization, Paris, France, 2022.

[JHD22b] Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Noise-free security assessment of eviction set construction algorithms with randomized caches", *in*: *Applied Sciences* **12**.*5* (2022), p. 2415.

[JHD22c] Amine Jaamoum, Thomas Hiscock, and Giorgio Di Natale, "Strategies for Securing a Memory Hierarchy Against Software Side-Channel Attacks", *in*: Journées Codage & Cryptographie JC2 Day, 2022.

[Kay+16] Mehmet Kayaalp et al., "A high-resolution side-channel attack on last-level cache", *in*: *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[KH92] Richard E Kessler and Mark D Hill, "Page placement algorithms for large real-indexed caches", *in*: *ACM Transactions on Computer Systems (TOCS)* (1992).

[Kir+18] Vladimir Kiriansky et al., "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.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential power analysis", *in*: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '99.* London: Springer, 1999, pp. 388–397, URL: http://dl.acm.org/citation.cfm?id=646764.703989.

[KMO12] Boris Köpf, Laurent Mauborgne, and Martıén Ochoa, "Automatic quantification of cache side-channels", *in*: *International Conference on Computer Aided Verification*, Springer, 2012, pp. 564–580.

[Koc+19] Paul Kocher et al., "Spectre attacks: Exploiting speculative execution", *in*: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[KPM12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz, "{STEALTHMEM}: System-level protection against cache-based side channel attacks in the cloud", *in*: *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 189–204.

[Lip] Moritz Lipp, "libflush, 2016", *in*: *URL https://github.com/IAIK/armageddon.(cited on p. 29)* ().

[Lip+16] Moritz Lipp et al., "Armageddon: Cache attacks on mobile devices", *in*: 2016, pp. 549–564.

[Lip+18] Moritz Lipp et al., "Meltdown", *in*: *arXiv e-prints 1801.01207* (2018).

[Liu+15] Fangfei Liu et al., "Last-level cache side-channel attacks are practical", *in*: *Symposium on Security and Privacy*, 2015.

[Liu+16a] Fangfei Liu et al., "Catalyst: Defeating last-level cache side channel attacks in cloud computing", *in*: *International symposium on High Performance Computer Architecture (HPCA)*, 2016.

[Liu+16b] Fangfei Liu et al., "Newcache: Secure cache architecture thwarting cache side-channel attacks", *in*: *IEEE Micro* (2016).

[Lou+21] Xiaoxuan Lou et al., "A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography", *in*: *arXiv preprint arXiv:2103.14244* (2021).

[Mar+05] Milo MK Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset", *in*: *ACM SIGARCH Computer Architecture News* **33**.*4* (2005), pp. 92–99.

[Mat+70] Richard L. Mattson et al., "Evaluation techniques for storage hierarchies", *in*: *IBM Systems journal* **9**.*2* (1970), pp. 78–117.

[Mau+15a] Clémentine Maurice et al., "C5: cross-cores cache covert channel", *in*: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2015, pp. 46–64.

[Mau+15b] Clémentine Maurice et al., "Reverse engineering Intel last-level cache complex addressing using performance counters", *in*: *International Symposium on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 48–65.

[Mau+17] Clémentine Maurice et al., "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.", *in*: *NDSS*, vol. 17, 2017, pp. 8–11.

[MDS12] Robert Martin, John Demme, and Simha Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks", *in*: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2012, pp. 118–129.

[Muk+20] M Asim Mukhtar et al., "Flush+ prefetch: A countermeasure against access-driven cache-based side-channel attacks", *in*: vol. 104, Elsevier, 2020, p. 101698.

[Mus+20] Maria Mushtaq et al., "Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA", *in*: *Information Systems* **92** (2020), p. 101524.

[NS06] Michael Neve and Jean-Pierre Seifert, "Advances on access-driven cache attacks on AES", *in*: *International Workshop on Selected Areas in Cryptography*, Springer, 2006, pp. 147–162.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang, "A refined look at Bernstein's AES side-channel analysis", *in*: *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, 2006, pp. 369–369.

[Org] Linux Kernel Organization, *Transparent Hugepage Support (Linux Kernel)*, URL: `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer, "Cache attacks and countermeasures: the case of AES", *in*: *Proceedings of the 2006 The Cryptogra-*

phers' Track at the RSA Conference on Topics in Cryptology. San Jose, CA: Springer-Verlag, 2006, URL: http://dx.doi.org/10.1007/11605805%5C%5C%5C_1.

[Pag02] Dan Page, "Theoretical use of cache memory as a cryptanalytic side-channel.", *in*: *IACR Cryptology ePrint Archive* (2002).

[Pag03] Daniel Page, "Defending against cache-based side-channel attacks", *in*: *Information Security Technical Report* **8**.*1* (2003), pp. 30–44.

[Pag05] Dan Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism.", *in*: *IACR Cryptology ePrint archive* (2005).

[Pao10] Gabriele Paoloni, "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures", *in*: *Intel Corporation* **123** (2010), p. 170.

[Per05] Colin Percival, *Cache missing for fun and profit*, 2005.

[Pol+21] Nikolaos Foivos Polychronou et al., "MaDMAN: Detection of Software Attacks Targeting Hardware Vulnerabilities", *in*: *2021 24th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2021, pp. 355–362.

[PSY15] Joop van de Pol, Nigel P Smart, and Yuval Yarom, "Just a little bit more", *in*: *Cryptographers' Track at the RSA Conference*, Springer, 2015, pp. 3–21.

[Pur+21] Antoon Purnal et al., "Systematic analysis of randomization-based protected cache architectures", *in*: *42th IEEE Symposium on Security and Privacy*, 2021.

[PV19] Antoon Purnal and Ingrid Verbauwhede, "Advanced profiling for probabilistic Prime+ Probe attacks and covert channels in ScatterCache", *in*: *arXiv preprint arXiv:1908.03383* (2019).

[QS01] Jean-Jacques Quisquater and David Samyde, "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards", *in*: *zSmart Card Programming and Security: International Conference on Research in Smart Cards, E-smart 2001 Cannes, France*, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 200–210.

[QTP05] Moinuddin K Qureshi, David Thompson, and Yale N Patt, "The V-Way cache: demand-based associativity via global replacement", *in*: *32nd International Symposium on Computer Architecture (ISCA'05)*, IEEE, 2005, pp. 544–555.

[Qur18] Moinuddin K Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping", *in*: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 775–787.

[Qur19] Moinuddin K Qureshi, "New attacks and defense for encrypted-address cache", *in*: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2019, pp. 360–371.

[Sea15] Mark Seaborn, *L3 cache mapping on Sandy Bridge CPUs*, 2015, URL: `http://lackingrhoticity.blogspot.com.es/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html`.

[Shi+11] Jicheng Shi et al., "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring", *in*: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE, 2011, pp. 194–199.

[Shu] Kirill A Shutemov, "pagemap: do not leak physical addresses to non-privileged userspace, 2015", *in*: *URL https://git. kernel. org/cgit/linux/kernel/git/torvalds/linux. git/commit* (), pp. 27–38.

[SK11] Daniel Sanchez and Christos Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning", *in*: *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 57–68.

[SL19] Wei Song and Peng Liu, "Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for {Side-Channel} Attacks against the {LLC}", *in*: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 427–442.

[Son+20] Wei Song et al., "Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It", *in*: *arXiv preprint arXiv:2008.01957* (2020).

[SQ21] Gururaj Saileshwar and Moinuddin Qureshi, "MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design", *in*: *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[Tan+20] Qinhan Tan et al., "PhantomCache: Obfuscating Cache Conflicts with Localized Randomization.", *in*: 2020.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir, "Efficient cache attacks on AES, and countermeasures", *in*: *Journal of Cryptology* (2010), pp. 37–71, URL: `http://dx.doi.org/10.1007/s00145-009-9049-y`.

[Tsu+03] Yukiyasu Tsunoo et al., "Cryptanalysis of DES implemented on computers with cache", *in*: *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer, 2003.

[Van+16] Victor Van Der Veen et al., "Drammer: Deterministic rowhammer attacks on mobile platforms", *in*: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.

[VKM19] Pepe Vila, Boris Köpf, and José F Morales, "Theory and practice of finding eviction sets", *in*: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 39–54.

[VRS14] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift, "Scheduler-based Defenses against {Cross-VM} Side-channels", *in*: *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 687–702.

[Wal02] Carl A Waldspurger, "Memory resource management in VMware ESX server", *in*: *ACM SIGOPS Operating Systems Review* **36**.*SI* (2002), pp. 181–194.

[Wan+16] Yao Wang et al., "SecDCP: secure dynamic cache partitioning for efficient timing channel protection", *in*: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2016, pp. 1–6.

[Wan+17] Wenhao Wang et al., "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX", *in*: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[Wan+19] Daimeng Wang et al., "Papp: Prefetcher-aware prime and probe side-channel attack", *in*: *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[Wan+20] Limin Wang et al., "Analyzing the security of the cache side channel defences with attack graphs", *in*: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2020, pp. 50–55.

[Wer+19] Mario Werner et al., "Scattercache: Thwarting cache attacks via cache set randomization", *in*: *28th USENIX Security Symposium*, 2019.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf, "A cache timing attack on AES in virtualization environments", *in*: *International Conference on Financial Cryptography and Data Security*, Springer, 2012, pp. 314–328.

[WL07] Zhenghong Wang and Ruby B Lee, "New cache designs for thwarting software cache-based side channel attacks", *in*: *Proceedings of the 34th annual international symposium on Computer architecture*, 2007.

[Won13] Henry Wong, "Intel Ivy Bridge cache replacement policy", *in*: *Url: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement* (2013).

[Yar16] Yuval Yarom, *Mastik: A micro-architectural side-channel toolkit*, 2016.

[YB14] Yuval Yarom and Naomi Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+ RELOAD cache side-channel attack", *in*: *Cryptology ePrint Archive* (2014).

[YK14] Yuval Yarom and Falkner Katrina, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack", *in*: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732, URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[ZAF07] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin, "Non-inclusion property in multi-level caches revisited", *in*: *International Journal of Computers and Their Applications* **14**.2 (2007), p. 99.

[ZAM12] Danfeng Zhang, Aslan Askarov, and Andrew C Myers, "Language-based control and mitigation of timing channels", *in*: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 99–110.

[Zha+12] Yinqian Zhang et al., "Cross-VM side channels and their use to extract private keys", *in*: *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 305–316.

[Zha+13] Tianwei Zhang et al., "Side channel vulnerability metrics: the promise and the pitfalls", *in*: *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[Zha+15] Danfeng Zhang et al., "A hardware design language for timing-sensitive information-flow security", *in*: *Acm Sigplan Notices* **50**.4 (2015), pp. 503–516.

[ZL14] Tianwei Zhang and Ruby B Lee, "New models of cache architectures characterizing information leakage from cache side channels", *in*: *Proceedings of the 30th annual computer security applications conference*, 2014, pp. 96–105.

[ZRZ16] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang, "A software approach to defeating side channels in last-level caches", *in*: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 871–882.

[ZZL18] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee, "Analyzing cache side channels using deep neural networks", *in*: *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 174–186.