



**HAL**  
open science

# Micro-architectural side channels: Studying the attack surface from hardware to browsers

Clémentine Maurice

► **To cite this version:**

Clémentine Maurice. Micro-architectural side channels: Studying the attack surface from hardware to browsers. Cryptography and Security [cs.CR]. Université de Lille, 2023. tel-04180074

**HAL Id: tel-04180074**

**<https://inria.hal.science/tel-04180074>**

Submitted on 11 Aug 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

---

---

# MICRO-ARCHITECTURAL SIDE CHANNELS: STUDYING THE ATTACK SURFACE FROM HARDWARE TO BROWSERS

---

---

*présentée et soutenue publiquement le 24 mai 2023 par*

**Clémentine Maurice**

Chargée de Recherche au CNRS

pour obtenir le

Diplôme d'Habilitation à Diriger des Recherches  
de l'Université de Lille

Spécialité Informatique

## Composition du Jury

Rapporteurs	Davide Balzarotti	Professeur, EURECOM
	Lejla Batina	Professeure, Radboud University
	Tamara Rezk	Directrice de Recherche, Inria
Examineurs	Herbert Bos	Professeur, VU Amsterdam
	Gilles Grimaud	Professeur, Université de Lille
Garant	Romain Rouvoy	Professeur, Université de Lille

# Abstract

Hardware is often represented as an abstract layer that behaves correctly, executes instructions and produces a result. However, side effects due to the execution of computations on actual computers can lead to *information leakage*. *Fault attacks* also threaten security, by generating errors in the physical system and thus bypassing security mechanisms. Typically, information leakage from *side-channel attacks* includes power consumption or electromagnetic emissions, and fault attacks include altering the system's power supply or clock. All of these attacks require physical access to the device. In contrast, my research activities since October 2015 have focused on side-channel and fault attacks that do not require physical access, and instead use the micro-architecture components of the processors. These attacks are software-based, and, therefore, remotely executable.

The main research challenge in micro-architectural attacks is to build secure systems and hardware that are immune to these vulnerabilities. In this manuscript, I give a partial view of my contributions in this domain, focusing on the attack surface, one of the main issues we face. In the first part, we cover the hardware attack surface, *i.e.*, the discovery of new side channels in unsuspected micro-architectural components. In the second part, we focus on a particular delivery method for side-channel attacks: web browsers.

# Résumé

Le matériel est souvent représenté comme une couche abstraite qui se comporte de manière correcte, exécutant des instructions et produisant un résultat. Cependant, des effets de bords dus à l'exécution des calculs sur une machine peuvent créer des *fuites d'informations* sensibles. Les *attaques par fautes* menacent également la sécurité, en générant des erreurs sur le système physique et donc en contournant les mécanismes logiciels de sécurité. Typiquement, les fuites d'informations qui découlent d'*attaques par canaux auxiliaires* incluent la consommation de courant ou les émanations électromagnétiques, et les attaques par fautes incluent la modification de l'alimentation électrique ou de l'horloge du système. Toutes ces attaques requièrent un accès physique à l'appareil. Au contraire, ma recherche depuis octobre 2015 s'intéresse aux fuites d'informations et attaques par fautes qui ne requièrent aucun accès physique, et qui utilisent les composants de micro-architecture des processeurs. Ces attaques sont réalisables par logiciel, et donc exécutables à distance.

L'enjeu principal de la recherche sur les attaques liées à la micro-architecture est de construire des systèmes et du matériel sûrs et résistants à ces vulnérabilités. Dans ce manuscrit, je donne une vue partielle de mes contributions dans ce domaine, en me concentrant sur la surface d'attaque, l'un des principaux problèmes auxquels nous sommes confrontés. Dans la première partie, nous couvrirons la surface d'attaque matérielle, c'est-à-dire la découverte de nouveaux canaux auxiliaires dans des composants micro-architecturaux insoupçonnés. Dans la seconde partie, nous nous concentrerons sur une méthode particulière de diffusion des attaques par canaux auxiliaires : les navigateurs web.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Micro-architectural security . . . . .	4
1.2	Challenges . . . . .	5
1.3	My contributions . . . . .	6
1.4	Organization of the manuscript . . . . .	7
<b>2</b>	<b>Side channels, side channels everywhere</b>	<b>9</b>
2.1	DRAM addressing . . . . .	10
2.1.1	Acquiring measurements . . . . .	11
2.1.2	Modeling DRAM addressing . . . . .	13
2.1.3	Security impact . . . . .	13
2.2	AMD cache way predictors . . . . .	16
2.2.1	Acquiring measurements . . . . .	17
2.2.2	Modeling the cache way predictor . . . . .	17
2.2.3	Security impact . . . . .	18
2.3	Intel CPU interconnect . . . . .	21
2.3.1	Acquiring measurements . . . . .	21
2.3.2	Modeling the CPU interconnect . . . . .	22
2.3.3	Security impact . . . . .	24
2.4	Discussion . . . . .	26
<b>3</b>	<b><i>Tu quoque</i>, my browser</b>	<b>27</b>
3.1	Timing is everything . . . . .	27
3.1.1	(In)security of resolution clamping . . . . .	28
3.1.2	Evolution of JavaScript’s timers in browsers . . . . .	30
3.2	Port contention side channels in browsers . . . . .	32
3.2.1	Porting port contention side channels in browsers . . . . .	33
3.2.2	Security impact . . . . .	34
3.3	Discussion . . . . .	35
<b>4</b>	<b>Conclusions and perspectives</b>	<b>36</b>
4.1	Main results . . . . .	36
4.2	Perspectives . . . . .	37
	<b>Bibliography</b>	<b>40</b>

<b>Publications from October 2015 to January 2023</b>	<b>51</b>
<b>Software related to my research from October 2015 to January 2023</b>	<b>54</b>
<b>Appendices</b>	<b>55</b>

# Chapter 1

## Introduction

The following document is a synthesis of my research activity since my PhD defense in October 2015. After a postdoc at IAIK, Graz University of Technology, Austria, I joined the EMSEC team at the IRISA lab, in Rennes, France, in October 2017. I then joined the Spirals team at the CRISTAL lab, in Lille, France, in February 2021. In the last seven years, my research activities have addressed a variety of topics, however, all related to micro-architectural security. During this time, I have had the pleasure to see the domain growing, from a few papers per year (and a domain admittedly not very bustling, if not a bit comatose, according to my own PhD supervisors) to dedicated sessions in every international top-tier security, micro-architecture, and system conferences. I have been, arguably, the main co-advisor of two successful doctorates, Thomas Rokicki (defended on November 29, 2022) and Guillaume Didier (defended January 20, 2023). I currently co-advise a third-year doctoral student, Pierre Ayoub (since 2020, co-advised with Aurélien Francillon), and a first-year doctoral student, Antoine Geimer (since 2022, main co-advisor).

### 1.1 Micro-architectural security

Hardware is often represented as an abstract layer that behaves correctly, executing instructions and producing a result. However, side effects due to the execution of computations on actual computers can create *information leakage*. Fault attacks also threaten security, by generating errors on the physical system and thus bypassing security mechanisms. Typically, information leakage from side-channel attacks includes power consumption or electromagnetic emissions, and fault attacks include altering the system's power supply or clock. All these attacks require physical access to the device and are referred to as physical attacks. In contrast, I am interested in attacks that do not require physical access, and that use micro-architectural components of processors, hereafter referred to as *micro-architectural attacks*. These attacks are all software-based, and, therefore, remotely executable. Let us depict a brief overview of the different classes of micro-architectural attacks.

In 1996, Kocher published his seminal work on *side-channel attacks* [91] by describing the first timing attacks against various cryptosystems: just by measuring the execution time of private key operations, an attacker can derive secret keys. In the following decade, the cache has been used to obtain more fine-grained information [23, 27, 119, 120, 117, 154]. Side-channel attacks are mainly due to optimizations in modern processors and operating systems, and in particular to shared micro-architectural components. The attacks use components such as the CPU cache [33, 82, 104, 117, 120, 178, 181, 179], the branch prediction unit [6, 9, 57, 58, 60], the memory management unit [67, 68,

137], floating point units [17], the ring interconnect [118], or even CPU ports [8, 10]. Indeed, these optimizations create differences in the execution time of programs, thus revealing secret information.

In contrast, *fault attacks* exploit physics and are more active attacks. While hardware operates correctly within some physical boundaries, the goal for an attacker is to push the components outside of these boundaries, using software only. Today, the most studied fault-based attack without physical access concerns DRAM and is called Rowhammer [30, 45, 46, 63, 71, 73, 78, 88, 89, 93, 99, 111, 116, 125, 128, 145, 150, 158, 177]. Internally, each DRAM cell consists of a capacitor and a transistor, which electrically implement a 0 or a 1. By repeatedly accessing cells, the charge of these cells leaks out and interacts electrically with the charge of neighboring cells. Thus, it is possible to change the value of cells in memory without ever having accessed them.

Finally, a novel class of attacks appeared in 2018, *transient execution attacks* [40, 90, 103]. They exploit transient instructions, *i.e.*, instructions that are not committed to the architectural state, due to a misprediction in speculative execution, or a fault. Functional correctness is ensured by pipeline flushes, which discard any architectural effects of pending instructions. Nevertheless, these instructions leave traces in micro-architecture — traces that we have spent a solid decade perfecting the art of retrieving through side channels. Transient execution attacks have much more severe outcomes than side-channel and fault attacks, having been shown to be able to leak, *e.g.*, kernel memory and passwords.

My research has spanned mostly side-channel attacks, moderately fault attacks (and in particular, Rowhammer attacks), and very sparingly transient execution attacks. This manuscript focuses solely on side-channel attacks.

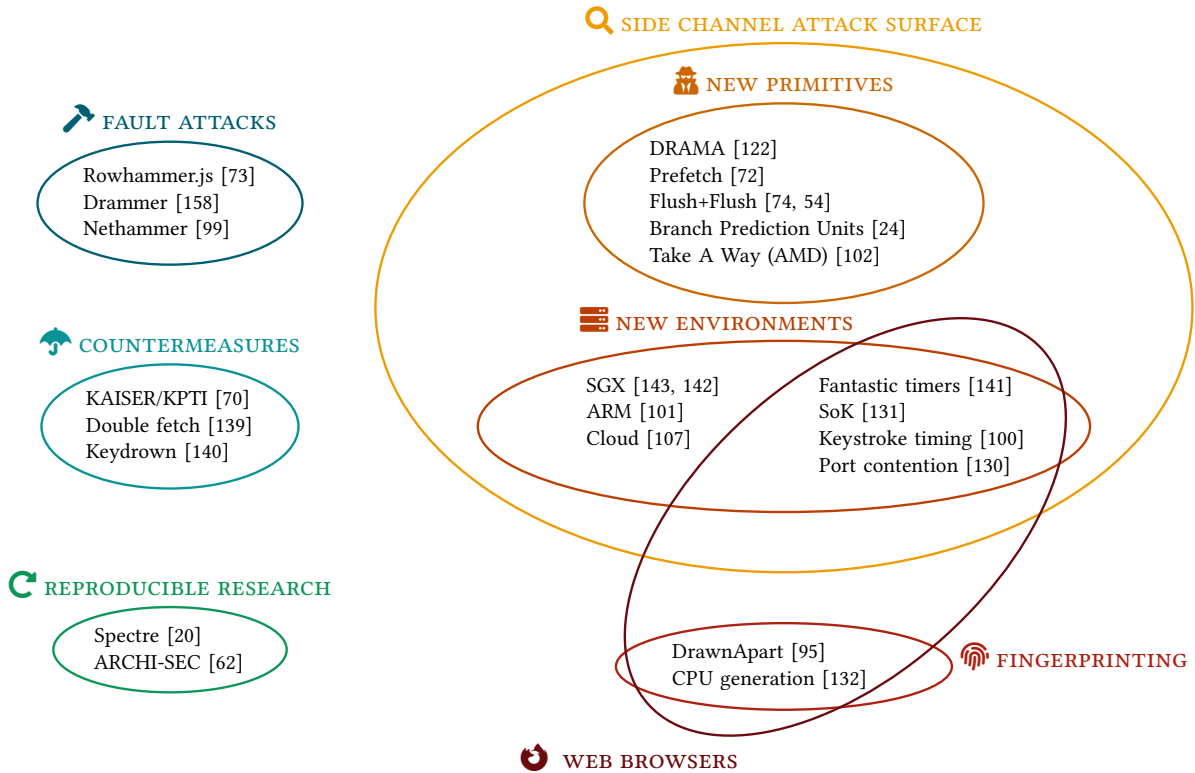
## 1.2 Challenges

The main challenge of research on micro-architectural security is to build secure systems and hardware that are immune to these vulnerabilities. Faced with this challenge, I distinguish two main issues: 1) the attack surface is not well known today, both at the hardware and software level, 2) current countermeasures suffer from an adoption problem, due to performance, cost, and scaling reasons. While I do have contributions related to countermeasures [70, 139, 140], most of my work pertains to the attack surface, this is thus the focus of this *Habilitation à Diriger des Recherches*. I believe that studying the attack surface is a necessary and preliminary step in designing efficient countermeasures. Indeed, a system that would be, *e.g.*, entirely protected against cache attacks, even with the most exquisite and efficient cache design, would not fare well, with the *same attacker model*, regarding any attack on any other component. Of course, the task of a defender is, in essence, asymmetric compared to the one of an attacker. Much like the Rebels attacking the Death Star in Star Wars, an attacker only has to find a *single* entry point, whereas a defender has to make sure that *all* entry points are secured.

The attack surface itself is, however, not a small matter. We aim at finding vulnerabilities — which, let us remember, exist whether we find them or not — to defend against them. From a defensive perspective, the task is, therefore, immense: we need to find all vulnerabilities an attacker could exploit! Furthermore, micro-architecture sits between the hardware and software interface. This means that part of the attack surface is tied to hardware, leading to a first research question “*What are the vulnerable components, and how to use them to leak secret data?*”, and part of the attack surface is tied to software, leading to a second research question “*What are the vulnerable pieces of software, and what are the different attack deliveries?*”.

### 1.3 My contributions

In this section, I give an overview of my various contributions, even though I will not detail all of them in this manuscript.



My main contributions concern the **side-channel attack surface**, at the hardware level. The first body of work I focused on is the creation of **new primitives**, to further understand the attack surface. Indeed, it is not possible to build efficient countermeasures without understanding precisely which components leak information, and how to manipulate the state of these components to leak information. As information usually leaks from the execution time of some instructions, we discovered timing leakage in novel instructions such as x86 prefetch [72] and `clflush` [74]. On some occasions, the creation of new primitives required some reverse-engineering work. This was the case when we investigated the first DRAM side-channel attack [122], the branch prediction unit [24], the AMD cache way predictor [102], and the CPU Interconnect [54].

Another body of work I focused on is investigating side-channel attacks in **new environments**. Indeed, typically, side-channel attacks are developed in native x86 environments, using native code such as C and assembly, as it makes it easier to manipulate components at a low level. Some primitives require specific instructions, e.g., the `clflush` instruction, which are only available in x86. It can therefore be challenging to port these attacks to new environments that either are not native, or with a different instruction set architecture. We demonstrated side-channel attacks from ARM devices [101], SGX enclaves [142, 143], covert channels in the cloud [107], as well as side-channel attacks in **web browsers** [100, 130, 131, 141]. Closely related to side-channel attacks in the browser is **fingerprinting**. We explored hardware fingerprinting, a novel field of research, and in particular, machine



fingerprinting leveraging GPUs with WebGL [95], and CPU generation fingerprinting leveraging port contention [132].

Different from side-channel attacks, I also focused on **software-based fault attacks**, and in particular the Rowhammer attack [89]. Here, my objective has also been to study the different environments in which Rowhammer was feasible. Indeed, much like side-channel attacks, the first attack was carried out in a native environment, using specific x86 instructions such as `clflush`. We demonstrated that, even though the attack requires surgical precision to access the right DRAM rows and to evict lines from the cache, this fault attack can be performed from JavaScript [73]. We later demonstrated that the attack can also be performed on ARM devices [158], and even remotely through network requests, given some conditions on the system [99].

While micro-architectural attacks are complicated to protect against, I strive to propose **countermeasures** in each attack paper (even though their adoption can be difficult due to performance overhead). Some work was, however, focused on countermeasures alone. We proposed a mitigation called Keydown against software-based keystroke timing side-channel attacks [140]. We also built a countermeasure dubbed KAISER [70] that provides practical kernel address isolation and protects against side-channel attacks that bypass KASLR (and in particular our previous work [72]). Interestingly, KAISER ended up being a countermeasure for the Meltdown attack by Lipp et al. [103] and was deployed in all major operating systems as kernel page-table isolation (KPTI). Finally, we also used side channels *as a countermeasure* – when they are overwhelmingly used as an attack – to detect and exploit double-fetch bugs in the kernel, eliminating them with hardware transactional memory instructions [139].

Finally, I am interested in **reproducible research** on micro-architectural security. Through the ANR ARCHI-SEC project, we are building a platform using the gem5 simulator to facilitate the reproducibility of micro-architectural attacks [62]. This is a completely new direction in this field where simulators are only used to model new hardware countermeasures. The use of a simulator for these attacks is not trivial because the processor model must be faithful to reality, and the complexity and lack of documentation can be a hindrance. We showed that visualization techniques for CPU pipeline could help the reproducibility of Spectre attacks by Kocher et al. [90], and ran data collection from gem5 and an ARM Cortex-A72 CPU to analyze the simulation accuracy. Our conclusions are optimistic about the role simulators can play in the future of micro-architectural security research [20], but much work remains.

## 1.4 Organization of the manuscript

This manuscript highlights some selected results of my research since my PhD defense in October 2015. Detailed experiments and algorithms can be found in the original papers. I have chosen to highlight two research directions: first, new side-channel primitives, and second, side channels in browsers. They seem to have little in common: the first is very close to the hardware components and will involve a fair amount of reverse-engineering, and the second is at the very top of the abstraction layers, in web browsers; yet, they complement each other.

- ▶ Chapter 2 dives into the micro-architectural level, and focuses on the components used to build new side-channel primitives. We will explore the DRAM (Section 2.1), the AMD cache way predictor (Section 2.2), and the Intel CPU interconnect (Section 2.3). It answers our first question “*What are the vulnerable components, and how to use them to leak secret data?*”.

- ▶ Chapter 3, at a higher level, provides an overview of my work on side channels in web browsers. It shows that, even though side-channel attacks require fine-grained control over hardware components, these attacks are possible in web browsers. We will delve into the security of timers in browsers (Section 3.1), and detail a novel side channel based on CPU port contention (Section 3.2). It answers our second question “*What are the vulnerable pieces of software, and what are the different attack deliveries?*”, focusing on the delivery aspect.
- ▶ Chapter 4 provides some conclusions and research perspectives that I would like to pursue.

## Chapter 2

# Side channels, side channels everywhere

*In which we show that no micro-architectural component is spared by side channels*

---

A large part of my work has focused on finding new side channels in micro-architectural components. Indeed, when I finished my PhD thesis, the two main components that were studied in terms of side-channel attack surface were the cache [1, 2, 4, 5, 7, 23, 27, 75, 76, 83, 85, 104, 112, 117, 119, 120, 148, 151, 152, 154, 168, 171, 176, 179] and the branch prediction unit [3, 6, 9] – except for the arithmetic logic unit of the Intel Pentium 4 that had also been studied by Acicmez et al. [8] and the floating point unit that had been studied by Andryscio et al. [17].

However, the cache and the branch predictor were just the tip of the iceberg of the attack surface of a modern CPU. While caches [14, 33, 51, 56, 69, 80, 82, 109, 110, 123, 136, 153, 178, 181] and branch prediction units [58, 57, 60] have still been heavily studied in the last years, including in my own work [24, 54, 74, 101, 102, 107, 143], other components started being brought to light such as the TLB [67], the MMU [68], CPU ports [10, 66], the CPU ring interconnect [118], and the random number generator [59]. A practical axiom is that every micro-architectural component shared across processes has been, or will be, attacked.

As I explained in the introduction, our goal is to get a better, and, as much as possible, comprehensive view of the hardware attack surface. Indeed, securing one component is not enough if other components are leaking – in the same way as patching a hole on a leaking pipe that has a dozen other holes is futile. Another aspect to consider is that the scope of attack may be different from one component to another. For example, branch predictors restrict the attacker to same-core attacks (*i.e.*, the victim has to reside on the same core), the cache allows cross-core attacks but is still limited to same-CPU attacks, while DRAM attacks can be carried across CPUs too.

**Reverse-engineering.** Today, a major obstacle to a refined and complete analysis of the hardware attack surface is the lack of documentation of micro-architectural components by manufacturers. Indeed, in side-channel attack experiments, we never directly observe what we want to measure: our observations are always indirect, *e.g.*, knowing whether a line is present in the cache through timing alone. It is, therefore, crucial to have as much knowledge as possible about the environment of our

experiments, *i.e.*, the micro-architectural components, to reduce the number of uncontrolled variables. Another matter is that, as attacks are getting more sophisticated and reaching more (undocumented) components, the need for reverse-engineering increased as well. One interesting aspect is that the lack of documentation, while hindering security research, is in fact not linked to security, but rather to performance. Indeed, these components are at the heart of performance improvements from one generation to another, and are therefore intellectual property, giving a competitive advantage that is better left undocumented.

Related work in reverse-engineering focused on last-level cache addressing [84, 180], cache replacement policies [161], cache directories [178], hardware prefetchers [129, 159], the CPU Ring Interconnect [118], page table caches [138], branch predictors [155], and GPUs [174]. The measurement acquisition is usually performed in two different ways: either through timing [84, 180] or through hardware performance counters [106, 155, 161]. While hardware performance counters give more precise information as they count given events (e.g., mispredicted branches at decoding, mispredicted branches at execution), they can suffer from non-determinism [53, 169]. Moreover, not all events can be monitored on all platforms, as non-architectural events change across generations. Timing measurements have a very high resolution (usually one CPU cycle) on native environments, using the `x86 rdtsc` instruction [178], a thread counter [101, 138], or dedicated performance counters [159].


Reverse engineering micro-architectural components is a research direction that I started during my PhD with the reverse-engineering of the last-level cache addressing functions [106]. It was a major axis of my research project application for CNRS, and I have continued to develop it, including with the Ph.D. of Guillaume Didier and through the ANR JCJC MIAOUS project of which I am the PI.

### Outline

Section 2.1 presents the reverse-engineering of DRAM addressing functions and subsequent novel side channel based on DRAM. Section 2.2 presents the reverse-engineering of the AMD L1 cache way predictor, and two novel side channels based on it. Section 2.3 presents a model of the CPU Interconnect that allowed us to refine Flush+Flush cache attacks. Throughout these three examples, I will outline the general method for acquiring measurements required for reverse-engineering and modeling the different components, as well as the security implications of this reverse engineering. Section 2.4 discusses these results.

## 2.1 DRAM addressing

During my postdoc, we were interested in the Rowhammer attack [89] and we quickly discovered differences in timing due to the row buffer in DRAM. We decided to investigate this, and developed DRAMA, short for DRAM Addressing attacks. The work presented hereafter has been published in:

 Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016

Every physical memory location maps to a memory location in one out of many rows in one out of several banks in the DRAM. Considering a single access to a row  $i$  in a bank, there are two cases:

**Row hit** The row  $i$  is already *cached* in the row buffer. The reason for a *row hit* can be that unrelated code just opened row  $i$  in this bank recently or that since our last access to row  $i$  in this bank, no other access went to another row in the same bank.

**Row conflict** The row  $i$  is *not* cached in the row buffer. A *row conflict* occurs if any other row  $j \neq i$  in the same bank has been opened since the last access to row  $i$ .

Considering frequent accesses to two (or more) addresses, we distinguish three cases:

1. The addresses map to different banks. In this case, the accesses are independent and whether the addresses have the same row indices has no influence on the timing. Row hits are likely to occur for the accesses, *i.e.*, a low access time.
2. The addresses map to the same row  $i$  in the same bank. In this case, the accesses are not independent. The accesses are likely to keep the row  $i$  open. Thus, row hits are likely to occur for the accesses, *i.e.*, a low access time.
3. The addresses map to the different rows  $i \neq j$  in the same bank. In this case, the accesses are again not independent. Each access to an address in row  $i$  will close row  $j$  and vice versa. Thus, row conflicts occur for the accesses, *i.e.*, a high access time.

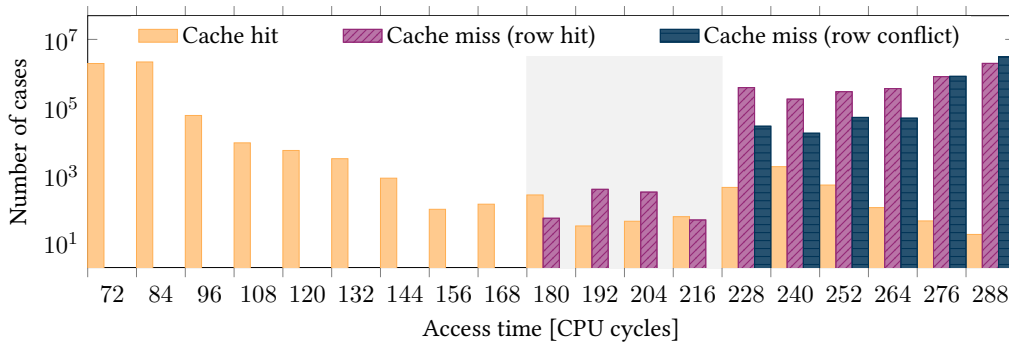


Figure 2.1: Histogram for cache hits and cache misses divided into row hits and row conflicts on an Intel i5-3230M (Ivy Bridge) CPU.

Figure 2.1 shows a comparison of standard histograms of access times for cache hits and cache misses. Cache misses are further divided into row hits and row conflicts. In the remainder, we build different attacks that are based on this timing difference between row hits and row conflicts.

## 2.1.1 Acquiring measurements

### 2.1.1.1 Physical probing

Our first approach is to physically probe the memory bus and to directly read the control signals. As shown in Figure 2.2, we use the tip of a standard passive probe to establish contact with the pin at the DIMM slot. We then use a high-bandwidth oscilloscope to measure the voltage and deduce the pins logic value. We contact all pins of interest, namely the bank-address bits (BA0, BA1, BA2 for DDR3 and BG0, BG1, BA0, BA1 for DDR4) and the chip select CS. We then record the logic levels on these pins for many randomly selected physical addresses.

This reverse-engineering approach has some drawbacks: 1) Expensive measurement equipment is needed. 2) It requires physical access to the internals of the tested machine. However, its main advantage is that the address mapping can be reconstructed for each signal individually and exactly, *i.e.*, we can determine the exact individual functions for the bus pins.

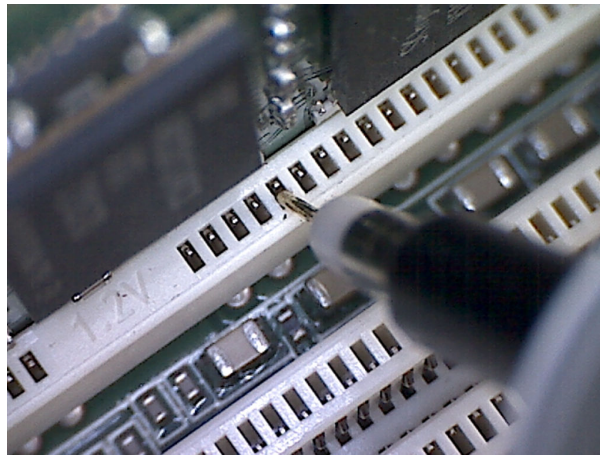


Figure 2.2: Physical probing of the DIMM slot with a standard passive probe.

### 2.1.1.2 Using row conflicts

For our second approach, we exploit the fact that row conflicts lead to higher memory access times. We use the resulting timing differences to find sets of addresses that map to the same bank but to a different row. The entire process is fully automated in software and runs without privileges.

We aim to find same-bank addresses in a large array mapped into the attacker’s address space. For this purpose, we perform repeated alternating accesses to two addresses and measure the average access time. We use `clflush` to ensure that each access is served from DRAM and not from the CPU cache. As shown in Figure 2.3, for some address pairs the access time is significantly higher than for most others. These pairs belong to the same bank but to different rows. The alternating accesses cause frequent row conflicts and consequently a high latency. The addresses are subsequently grouped into sets having the same channel, DIMM, rank, and bank.

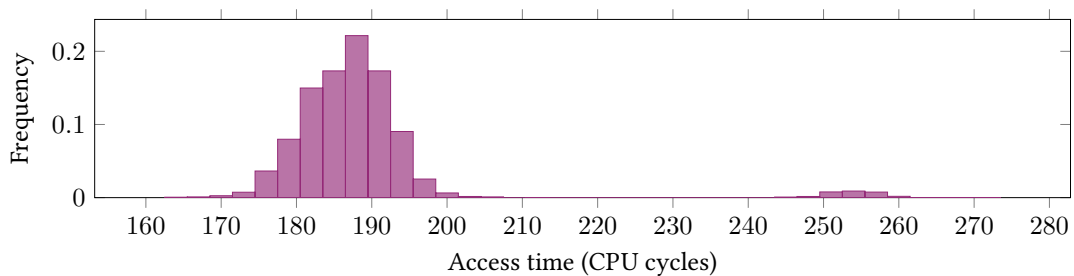


Figure 2.3: Histogram of average memory access times for random address pairs on an Intel i7-4700 (Haswell) CPU. A clear gap separates the majority of address pairs causing no row conflict (lower access times), because they map to different banks, from the few address pairs causing a row conflict (higher access times), because they map to different rows in the same bank.

## 2.1.2 Modeling DRAM addressing

### 2.1.2.1 Recovering the hash function

The reconstruction of DRAM addressing functions differs depending on the method used for acquiring measurements. In the first case where we acquired measurements with physical probing, for each DRAM addressing bit, we create an over-defined system of linear equations in the physical address bits and then solve this system using linear algebra. The solution is the addressing function for the corresponding DRAM addressing bit.

In the second case where we acquired timing measurements, we use the identified address sets to reconstruct the addressing functions. This reconstruction can either be based on 2 MB, 1 GB pages, or privileged information such as the virtual-to-physical address translation that can be obtained through `/proc/pid/pagemap`. In the case of 2 MB pages we can recover the partial functions up to bit  $a_{20}$ , as the lowest 21 bit of virtual and physical address are identical. In the case of 1 GB pages we can recover all partial functions up to bit  $a_{30}$ . This is sufficient to recover the full DRAM addressing functions on all our test systems without any privileges.

Similarly to the solving phase of the probing approach, we presume the linearity of the DRAM addressing functions. Bits  $(a_0..a_5)$  are used for addressing within a cache line. Bits  $a_{30}$  and upwards are not used. The search space is then small enough to perform a brute-force search of linear functions within seconds. For this, we generate all linear functions that use exactly  $n$  bits as coefficients and then apply them to all addresses in one randomly selected set. We verify the correctness either by comparing it to the results from the physical probing, or by performing a software-based test, *i.e.*, verifying the timing differences on a larger set of addresses.

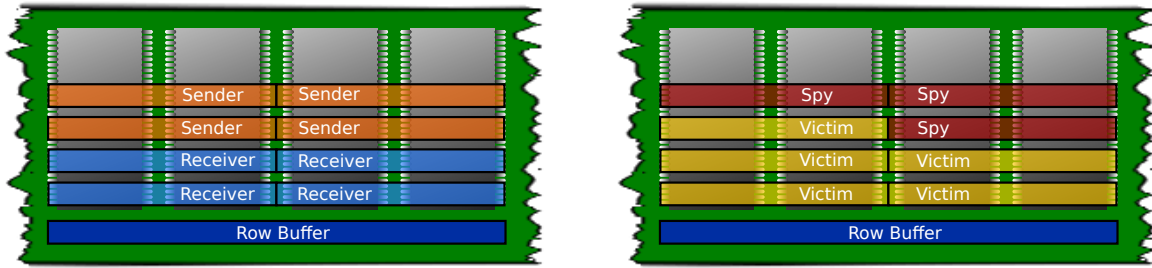
### 2.1.2.2 Results

Table 4.2 in the Appendix shows a comprehensive overview of all platforms and memory configurations we analyzed (Table 4.1 in the Appendix). As all found functions are linear, we simply list the index of the physical address bits that are XORed together.

We can observe that the detailed mapping differs with the memory setup, such as the number of DIMMs (single channel or dual channel) or CPUs. Additionally, changes can appear between different generations of CPUs, e.g., the rank and channel selection functions which are composed of a single bit on Sandy Bridge, and multiple bits from Ivy Bridge and onward. Due to DDR4's introduction of bank grouping and the doubling of the available banks (now 16), the addressing function necessarily changed again. Finally, the mapping used on the mobile platforms we analyzed is much simpler.

## 2.1.3 Security impact

This reverse-engineering work has led to a covert channel and a side-channel attack that leverage row conflicts, dubbed DRAMA (DRAM Attacks). Both these attacks do not require any shared memory, as opposed to the cache attack Flush+Reload. Unlike all cache attacks (except for [82]), their only prerequisite is that the two communicating processes (in the case of the covert channel), or the victim and the attacker (in the case of the side-channel attack), have access to the same memory module, including scenarios where these processes do not share a CPU.



(a) Covert channel scenario. The sender occupies rows in a bank to trigger row conflicts. The receiver occupies rows in the same bank to observe these row conflicts.

(b) Side-channel attack scenario. Victim and spy have memory allocated in the same DRAM row. By accessing this memory, the spy can determine whether the victim just accessed it.

Figure 2.4: Row placement in the cases of covert channel and side-channel attack scenarios.

### 2.1.3.1 Covert channel

We present a first DRAMA attack, namely a high-speed cross-CPU covert channel that does not require shared memory and can operate across CPUs.

**Protocol.** Our covert channel exploits timing differences caused by row conflicts. Figure 2.4a illustrates how rows are occupied by sender and receiver. The receiver process continuously accesses a chosen physical address in the DRAM and measures the average access time over a few accesses. If the sender process now continuously accesses a different address in the same bank but in a different row, a row conflict occurs. This leads to higher average access times in the receiver process. Bits can be transmitted by switching the activity of the sender process in the targeted bank on and off. The receiver process distinguishes the two values based on the mean access time. We assign a logic value of 0 to low access times (inactive sender) and a value of 1 to high access times (active sender).

Each (CPU, channel, DIMM, rank, bank) tuple can be used as a separate transmission channel. However, a high number of parallel channels leads to increased noise. Note that transmission channels are unidirectional, but the direction can be chosen for each one independently, thus, two-way communication is possible.

**Performance Evaluation.** We evaluate the performance of our covert-channel implementation on two systems: 1) a standard desktop PC with an Intel i7-4790 CPU (Haswell) with 2 DDR4 DIMMs, 2) a server system with two Intel Xeon E5-2630 v3 CPUs (Haswell-EP), equipped with 4 DDR4 DIMMs.

In our proof-of-concept implementation, we transmit 8 bits per block using 8 (CPU, channel, DIMM, rank, bank) tuples, and each block is sent and measured for a fixed period. The error rate varies depending on the raw bitrate, which depends on measurement intervals. On our desktop setup, the error probability stays below 1% for bitrates of up to 2 Mbps. The channel capacity reaches up to 2.1 Mbps (raw bitrate of 2.4 Mbps, 1.8% error rate). On our server setup, the maximum capacity is 1.6 Mbps (raw 2.6 Mbps, 8.7% error rate).

We also implemented a transmission in a cross-CPU and cross-VM setup on the server system. We obtain a transmission rate of 596 Kbps at an error rate of 0.4%, without a cross-VM synchronization mechanism. These results are thus far from what can be achieved in practice with synchronization.



### 2.1.3.2 Side-channel attack

We now present a second DRAMA attack, a side-channel attack that again exploits the row buffer, where the spy and the victim can run on separate CPUs and do not share memory. This side channel achieves a temporal resolution comparable to Flush+Reload and a higher spatial resolution than Prime+Probe. In this side-channel attack, an attacker infers the activity of a victim process by detecting *row hits* and *row conflicts*. The spy and victim need to have access to the same row (Figure 2.4b).

The spy and the victim can share a row without shared memory due to the DRAM addressing functions. The size of a row is typically 8 KB and memory is organized in 4 KB pages. Thus, the physical memory of a row contains the memory of 2 or more pages depending on the DRAM addressing functions. If no DRAM addressing functions use low address bits ( $a_0 - a_{11}$ ), the spatial resolution is 4 KB, which is the worst case. However, if DRAM addressing functions (channel, BG0, CPU, etc.) use low address bits, rows with the same row index in different banks are interleaved in the same memory region. For example, on our Haswell-EP test system, the spatial resolution of our attack is 512 B.

To run the side-channel attack on a private memory address  $t$  in a victim process, the attacker allocates a memory address  $p$  that maps to the same bank and the same row as the target address  $t$ . The attacker also allocates a row conflict address  $\bar{p}$  that maps to the same bank but to a different row. The side-channel attack then works in three steps: 1) Access the row conflict address  $\bar{p}$ , 2) Wait for the victim to compute, 3) Measure the access time on the targeted address  $p$ . If the measured timing is below a row-hit threshold (cf. the highlighted “row hit” region in Figure 2.1), the victim has just accessed  $t$  or another address in the target row. Thus, we can accurately determine when a specific non-shared memory location is accessed by a process running on another core or CPU.

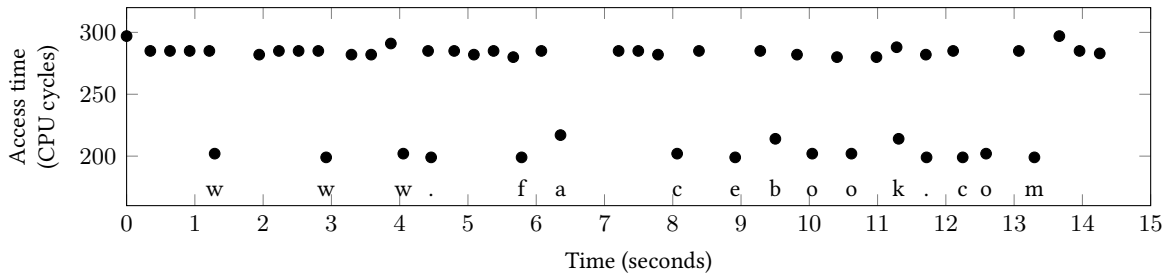



Figure 2.5: Exploitation phase on non-shared memory in a DRAMA template attack on an Intel i5-3230M (Ivy Bridge) CPU with DDR3. A low access time is measured when the user presses a key in the Firefox address bar.

Based on this attack principle, we build a fully automated DRAMA template attack (similarly to [75]). Figure 2.5 shows an access time trace for an address found in this template attack while typing in the Firefox address bar. A low access time is measured for every key the user presses.

## 2.2 AMD cache way predictors

After some investigation of AMD manuals, we decided to study AMD cache way predictors, a component that does not exist in Intel processors and that had not been studied by the security community before. The work presented hereafter has been published in:

 Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *ASIACCS*. 2020

To look up a cache line in a set-associative cache, bits in the address determine in which set the cache line is located. With an  $n$ -way cache,  $n$  possible entries need to be checked for a tag match. To avoid wasting power for  $n$  comparisons leading to a single match, Inoue et al. [81] presented way prediction for set-associative caches. Instead of checking all ways of the cache, a way is predicted, and only this entry is checked for a tag match. If the prediction is correct, the access has been completed, and access times similar to a direct-mapped cache are achieved. If the prediction is incorrect, a normal associative check has to be performed, which increases the latency.

Every cache line in the L1D cache is tagged with a linear-address-based  $\mu$ Tag [16, 64]. This  $\mu$ Tag is computed using an undocumented hash function, which takes the virtual address as input. For every memory load, the way predictor predicts the cache way of every memory load based on this  $\mu$ Tag. As the virtual address, and thus the  $\mu$ Tag, is known before the physical address, the CPU does not have to wait for the TLB lookup. If there is no match for the calculated  $\mu$ Tag, an early miss is detected, and a request to L2 is issued. Figure 2.6 illustrates AMD’s way predictor.

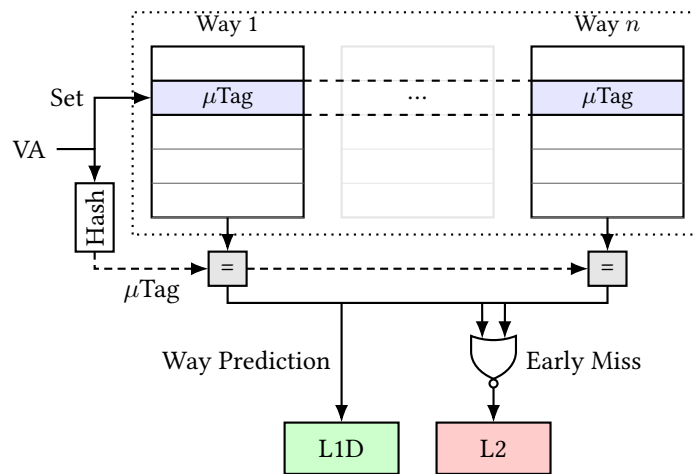


Figure 2.6: Simplified illustration of AMD’s way predictor.

The AMD’s way predictor incurs some timing differences considering the access of two addresses. We distinguish two cases:

1. The two addresses yield a conflict in the  $\mu$ Tag table, *i.e.*, they are virtual addresses that yield the same  $\mu$ Tag.
2. The two addresses are aliased, *i.e.*, the two different *virtual* addresses map to the same *physical* location.

In both cases, the second load sees an L1D cache miss and thus loads the data from the L2D cache [16].

## 2.2.1 Acquiring measurements

**High-resolution timing.** The attacker requires a method to measure timing differences in the range of a few CPU cycles. The unprivileged `rdtsc` instruction returning the current cycle count is commonly used for cache attacks on Intel CPUs. Using this instruction, an attacker can get timestamps with a resolution between 1 and 3 cycles. On AMD CPUs, this register has a cycle-accurate resolution until the Zen micro-architecture. Since then, it has a significantly lower resolution as it is only updated every 20 to 35 cycles.

The AMD Ryzen micro-architecture provides the *Actual Performance Frequency Clock Counter* (APERF counter) [15] which can be used to improve the accuracy of the timestamp counter. However, it can only be accessed in kernel mode. Although other timing primitives provided by the kernel, such as `get_monotonic_time`, provide nanosecond resolution, they can be noisier and still not sufficiently accurate to observe timing differences, which are only a few CPU cycles.

Hence, on more recent AMD CPUs, it is necessary to resort to a different method for timing measurements. In an earlier work on ARM-based cache attacks [101], we showed that *counting threads* can be used where unprivileged high-resolution timers are unavailable. A counting thread constantly increments a global variable used as a timestamp without relying on micro-architectural specifics. This is the method that we use throughout this section unless otherwise specified.

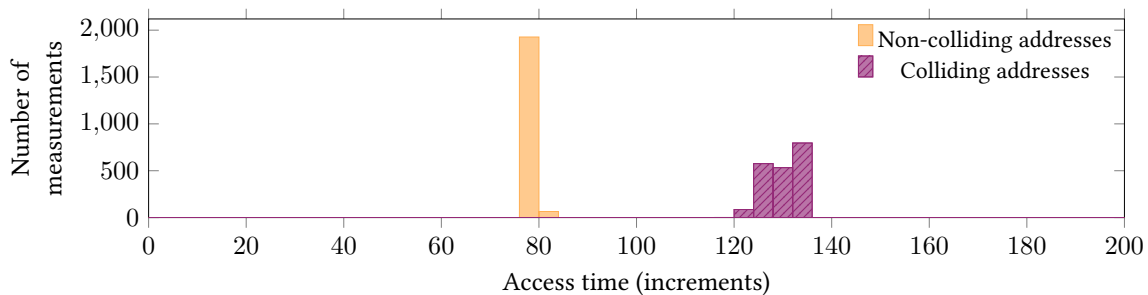


Figure 2.7: Measured duration of 250 alternating accesses to addresses with and without the same  $\mu$ Tag.

**$\mu$ Tag collisions.** We rely on  $\mu$ Tag collisions to reverse-engineer the hash function. We pick two random virtual addresses that map to the same cache set. If the two addresses have the same  $\mu$ Tag, repeatedly accessing them one after the other results in conflicts. As the data is then loaded from the L2 cache, we can measure an increased access time, as illustrated in Figure 2.7. Another method would be to measure the performance counter for L1 misses.

## 2.2.2 Modeling the cache way predictor

### 2.2.2.1 Recovering the hash function

With the ability to detect conflicts, we can build  $N$  sets representing the number of entries in the  $\mu$ Tag table. First, we create a pool  $v$  of virtual addresses, which all map to the same cache set, *i.e.*, where bits 6 to 11 of the virtual address are the same. We start with one set  $S_0$  containing one random virtual address out of the pool  $v$ . For each other randomly-picked address  $v_x$ , we measure the access time while alternatively accessing  $v_x$  and an address from each set  $S_{0\dots n}$ . If we encounter a high access

time, we measure conflicts and add  $v_x$  to that set. If  $v_x$  does not conflict with any existing set, we create a new set  $S_{n+1}$  containing  $v_x$ . In our experiments, we recovered 256 sets.

Every virtual address that is in the same set produces the same hash. We assume that this undocumented hash function is linear based on the knowledge of other such hash functions we previously reverse-engineered [106, 122]. Moreover, we expect the size of the  $\mu$ Tag to be a power of 2, resulting in a linear function. To recover the hash function, we need to find which bits in the virtual address are used for the 8 output bits that map to the 256 sets. Due to its linearity, each output bit of the hash function can be expressed as a series of XORs of bits in the virtual address. Hence, we can express the virtual addresses as an over-determined linear equation system. The solutions of the equation system are then linear functions that produce the  $\mu$ Tag from the virtual address.

While the least-significant bits 0-5 define the cache line offset, note that bits 6-11 determine the cache set and are not used for the  $\mu$ Tag computation [16]. To solve the equation system, we used the Z3 SMT solver. Every solution vector represents a function that XORs the virtual-address bits that correspond to '1'-bits in the solution vector. The hash function is the set of linearly independent functions, *i.e.*, every linearly independent function yields one bit of the hash function. The order of the bits cannot be recovered. However, this is not relevant for using the hash function, as we are only interested in whether addresses collide, not in their numeric  $\mu$ Tag value.

### 2.2.2.2 Results

Figure 4.1 in the Appendix illustrates the recovered  $\mu$ Tag hash functions on the AMD Zen, Zen+, Zen 2, Bulldozer, Piledriver, and Steamroller micro-architectures listed in Table 4.3 in the Appendix. The function illustrated in Figure 4.1a for Zen, Zen+, and Zen 2 micro-architectures uses bits 12 to 27 to produce an 8-bit value mapping to one of the 256 sets. For the Bulldozer, Piledriver, and Steamroller micro-architectures, the hash function uses the same bits but in a different combination as illustrated in Figure 4.1b.

## 2.2.3 Security impact

This reverse-engineering work has led to two novel side channels that leverage AMD's L1D cache way predictor. With Collide+Probe, we monitor memory accesses of a victim's process without requiring the knowledge of physical addresses. With Load+Reload, while relying on shared memory similarly to Flush+Reload, we can monitor memory accesses of a victim's process running on the sibling hardware thread without invalidating the targeted cache line from the entire cache hierarchy.

### 2.2.3.1 Collide+Probe

Collide+Probe exploits  $\mu$ Tag collisions in AMD's L1D cache way predictor, *i.e.*, addresses that have the same  $\mu$ Tag. As we described previously, the way predictor uses virtual-address-based  $\mu$ Tags to predict the L1D cache way. If an address is accessed, the  $\mu$ Tag is computed, and the way-predictor entry for this  $\mu$ Tag is updated. If a subsequent access to a different address with the same  $\mu$ Tag is performed, a  $\mu$ Tag collision occurs, and the data has to be loaded from the L2D cache, increasing the access time. We exploit this timing difference to monitor accesses to such colliding addresses.

**Threat model.** For this attack, we assume that the attacker has unprivileged native code execution on the target machine and runs on the same logical CPU core as the victim. Furthermore, the attacker can force the execution of the victim's code, *e.g.*, via a function call in a library or a system call.

**Attack.** The attacker first chooses a virtual address  $v$  of the victim that should be monitored for access. To mount a Collide+Probe attack, the attacker selects a virtual address  $v'$  in its own address space that yields the same  $\mu$ Tag  $\mu_{v'}$  as the target address  $v$ , *i.e.*,  $\mu_v = \mu_{v'}$ . Moreover, both  $v$  and  $v'$  have to be in the same cache set. The attack consists of three phases performed repeatedly:

**Phase 1: Collide.** The attacker accesses the pre-computed address  $v'$  and, thus, updates the way predictor. The way predictor associates the cache line of  $v'$  with its  $\mu$ Tag  $\mu_{v'}$  and subsequent memory accesses with the same  $\mu$ Tag are predicted to be in the same cache way. Since the victim's address  $v$  has the same  $\mu$ Tag ( $\mu_v = \mu_{v'}$ ), the  $\mu$ Tag of that cache line is marked invalid and the data is effectively inaccessible from the L1D cache.

**Phase 2: Scheduling the victim.** The victim is scheduled to perform its operations.

**Phase 3: Probe.** The attacker measures the access time to the pre-computed address  $v'$ . If the victim has not accessed the monitored address  $v$ , the data of the pre-computed address  $v'$  is still accessible from the L1D cache and the way prediction is correct. Thus, the measured access time is fast. If the victim has accessed the monitored address  $v$  and thus changed the state of the way predictor, the attacker suffers an L1D cache miss when accessing  $v'$ , as the prediction is now incorrect. The data of the pre-computed address  $v'$  is loaded from the L2 cache and, thus, the measured access time is slow. By distinguishing between these cases, the attacker can deduce whether the victim has accessed the targeted data.

### 2.2.3.2 Load+Reload

Load+Reload exploits the way predictor's behavior for aliased addresses, *i.e.*, virtual addresses mapping to the same physical address. When accessing data through a virtual address alias, the data is always requested from the L2 cache instead of the L1D cache [16]. Consequently, this allows one thread to evict shared data used by the sibling thread with a single load. Although the requested data is stored in the L1D cache, it remains inaccessible for the other thread and, thus, introduces a timing difference when it is accessed.

**Threat model.** For this attack, we assume that the attacker has unprivileged native code execution on the target machine. The attacker and victim run simultaneously on the same physical but different logical CPU thread. The attack target is a memory location with a virtual address  $v$  shared between the attacker and victim, *e.g.*, a shared library.

**Attack.** Load+Reload exploits the timing difference when accessing a virtual-address alias  $v'$  to build a cross-thread attack on shared memory. The attack consists of 3 phases:

**Phase 1: Load.** In contrast to Flush+Reload, where the targeted address  $v$  is flushed from the cache hierarchy, Load+Reload loads an address  $v'$  with the same physical tag as  $v$  in the first phase. Thereby, it renders the cache line containing  $v$  inaccessible from the L1D cache for the sibling thread.

**Phase 2: Scheduling the victim.** The victim process is scheduled. If the victim process accesses the targeted cache line with address  $v$ , it sees an L1D cache miss. As a result, it loads the data from the L2 cache, invalidating the attacker's cache line with address  $v'$  in the process.

**Phase 3: Reload.** The attacker measures the access time to the address  $v'$ . If the victim process has accessed the cache line with address  $v$ , the attacker observes an L1D cache miss and loads the data from the L2 cache, resulting in a higher access time. Otherwise, if the victim has not accessed the cache line with address  $v$ , it is still accessible in the L1D cache for the attacker and, thus, a lower access time

is measured. By distinguishing between both cases, the attacker can deduce whether the victim has accessed the address  $v$ .

### 2.2.3.3 Case study: covert channel

As many other cache side channels, Collide+Probe and Load+Reload can be used in different attack scenarios: 1) a covert channel between two processes to leak arbitrary data, and to leak secret data from the kernel in a Spectre attack; 2) reducing the entropy of kernel ASLR, user space ASLR, and hypervisor ALSR in a virtual machine setting; 3) a side-channel attack on a weak cryptographic implementation, e.g., the AES T-tables implementation. In this manuscript, we only detail the covert channel which uses Collide+Probe, and refer to the paper [102] for the other attack scenarios.

**Protocol.** For the most simplistic form of the covert channel, two processes agree on a  $\mu$ Tag and a cache set. These two processes must reside in the same logical core. In the initialization phase, both parties allocate their own page. The sender chooses a virtual address  $v_S$ , and the receiver chooses a virtual address  $v_R$  such that  $v_S$  and  $v_R$  are in the same cache set and yield the same  $\mu$ Tag.

To encode a 1-bit to transmit, the sender accesses address  $v_S$ . To transmit a 0-bit, the sender does not access address  $v_S$ . The receiving end decodes the transmitted information by measuring the access time when loading address  $v_R$ . If the sender has accessed address  $v_S$  to transmit a 1, the collision caused by the same  $\mu$ Tag of  $v_S$  and  $v_R$  results in a slow access time for the receiver. If the sender has not accessed address  $v_S$ , no collision caused the address  $v_R$  to be evicted from L1D and, thus, the access time is fast. This timing difference allows the receiver to decode the transmitted bit.


We extend this simple covert channel to transmit multiple bits in parallel by utilizing multiple cache sets. Instead of decoding the transmitted bit based on the timing difference of one address, we use two addresses in two cache sets for every bit we transmit: one to represent a 1-bit and the other to represent the 0-bit. As the L1D has 64 cache sets, we can transmit up to 32 bits in parallel without reusing cache sets.

**Performance evaluation.** We evaluate the transmission and error rate of our covert channel in a local setting and a cloud setting. We achieve a maximum transmission rate of 588.9 kBps using 80 channels in parallel on the AMD Ryzen Threadripper 1920X. On the AMD EPYC 7571 in the Amazon EC2 cloud, we achieve a maximum transmission rate of 544.0 kBps also using 80 channels. In contrast, L1 Prime+Probe achieves a transmission rate of 400 kBps [120] and Flush+Flush 496 kB/s [74]. The mean transmission rate increases with the number of bits sent in parallel. However, the error rate increases drastically when transmitting more than 64 bits in parallel.

As accesses to unrelated addresses with the same  $\mu$ Tag as our covert channel introduce noise in our measurements, we use error correction to achieve better transmission. We evaluate different hamming codes on an AMD Ryzen Threadripper 1920X. While the  $H(7, 4)$  code is slightly more robust [77], the  $H(15, 11)$  code achieves a better transmission rate of 452.2 kBps.

## 2.3 Intel CPU interconnect

Our work on the Intel CPU interconnect started serendipitously while trying to reverse-engineer another component: the hardware prefetcher. This work is part of Guillaume Didier’s PhD, and has been published in:

 Guillaume Didier and Clémentine Maurice. “Calibration Done Right: Noiseless Flush+Flush Attacks”. In: *DIMVA*. 2021

Several cache attack primitives exist with different requirements and trade-offs. Flush+Flush [73], on which I worked during my postdoc, is a stealthy and fast one using the timing of the `clflush` instruction which depends on whether a line is cached. However, one aspect that has been overlooked for this attack is the choice of the threshold to distinguish between a flush hit (slower) and a flush miss (faster). This threshold is crucial to avoid noise. When looking at the timings for a single address and on a single run, it appears that there is a good separation between the hits and the misses. However, from one run to another, the exact threshold may change, even with a fixed frequency. The threshold also differs for different addresses.

We hypothesize that the variability is due to the complex topology of sliced caches, and that accounting for these sources of variability improves significantly the quality of the channel, especially as the number of cores grows. Our experiments show that ignoring CPU topology can result in very poor error rates, e.g., in some cases up to 45% error rate. In the remainder of the section, we show that taking into account the topology and slices to compute tailored thresholds allows us to build a side channel with an error rate well under 0.01%. Flush+Flush is, therefore, contrary to what was thought before, not a noisy attack when crafted carefully.

### 2.3.1 Acquiring measurements

**Setup.** We run experiments on two single-socket systems: 1) *4-core machine*: a machine with an Intel Core i5-8365U CPU (Whiskey Lake, 4 cores, 8 threads), 2) *8-core machine*: a machine with an Intel Core i9-9900 CPU (Coffee Lake, 8 cores, 16 threads).

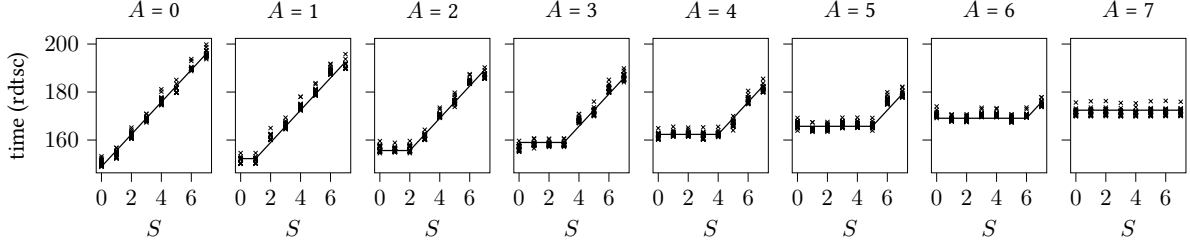
We enable hyper-threading, but disable turbo boost on those machines. The `intel_pstate` driver is set to performance mode on all cores, to stabilize the core frequencies.

**Measurements.** We investigate the factors that influence the execution time of `clflush` to improve the Flush+Flush attack, and propose a mathematical model with an associated ring topology. The only information we have from the Intel documentation is that the interconnect is a “bidirectional ring”.

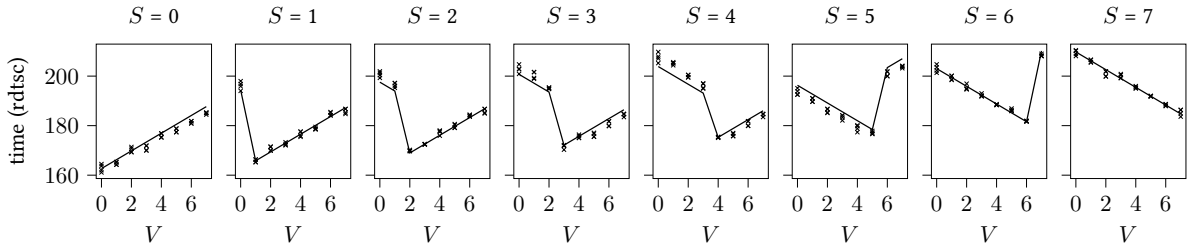
A `clflush` miss occurs when a cache line is not validly cached, which corresponds to a line in the I state. A line that has just been flushed is in the I state – the cache may have an entry in the I state or no entry at all, but it is equivalent at the cache coherency protocol level. A `clflush` hit occurs when the line is in any valid state. In practice, in a Flush+Flush attack, the cache line of interest transitions from an I to an E state when the single victim core loads the line that has just been flushed. Therefore, the two relevant timings are `clflush` of a line in the E state for a hit, and in the I state for a miss. We study these timings depending on three parameters:

1. *A*: the attacker core that executes `clflush` on an address,
2. *V*: the victim core that accesses the address and caches it in its L1 or L2,
3. *S*: the core that contains the last-level cache slice that this address maps to.

### 2.3.2 Modeling the CPU interconnect



(a) For a cache line in the I state, depending on its slice  $S$  for each attacker core  $A$ . Victim logic core has no impact on a miss.



(b) For a cache line in the E state, depending on the victim core  $V$  for each slice  $S$ , using a fixed attacker core ( $A = 0$ ).

Figure 2.8: Median timings of `clflush` on the 8-core machine depending on the victim core  $V$ , the slice  $S$ , and the attacker  $A$ , along with the fitted model according to our proposed topology.

**Topology.** For each attacker core, Figure 2.8a shows the time it takes to execute a `clflush` instruction on a cache line in the I state, depending on the slice. The first finding is that all 8 cores have a distinct timing pattern, which implies that the ring has no symmetry. For each attacker, we notice that slices with a lower core number than the attacker all have the same timing, while for slices with a higher number the time increases with the distance between the attacker and the slice. Such a pattern only makes sense if the nodes are aligned linearly, and if the attacker sends a message to the slice, which then sends a message to the system agent, and then back to the slice, and finally to the attacker. Consequently, the miss time is more variable for attackers closer to the system agent than for one further away.

Figure 2.8b shows the time it takes to execute a `clflush` instruction on a cache line in the E state. Here, we notice an asymmetry in the core, which can be explained if the recall request is always sent by the slice in the same direction without knowing in which core the line is cached.

Given that the topology is described as a ring, given the die shot in Figure 2.9a and our results, we propose the topology in Figure 2.9b, with 8 cores aligned in a linear graph with forward and backward links. For a 4-core machine, similar measurements lead to a similar topology with only cores 0-3.

**Mathematical model.** The above timing measurements can be interpreted within the proposed topology as follows, leading to a mathematical model that can be fitted and compared with the measurements. Misses result in a request to be sent on the ring from the core requesting the flush to the slice, which then sends a message to the memory, and then answers the same path in reverse, using



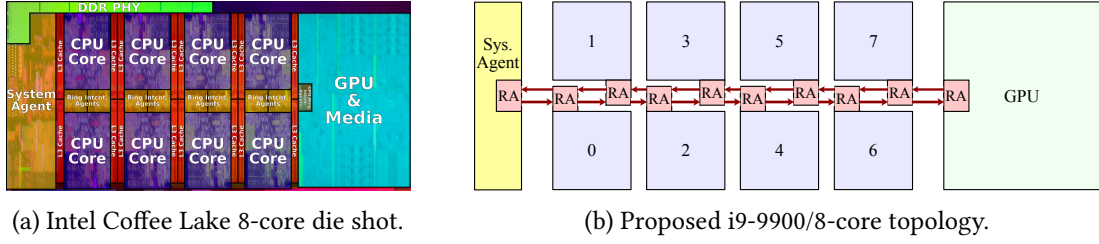


Figure 2.9: Core i9-9900 die shot and topology.

each time the shortest path. The eviction time in state I,  $t_I(A, S)$  is thus:

$$t_I(A, S) = C + h \times |A - S| + h \times |S - M|,$$

in which:

- ▶  $C$  is a constant base timing,
- ▶  $h$  is a constant corresponding to the time a round-trip hop on the ring takes,
- ▶  $M$  corresponds to the system agent location, which is  $-1$ .

Upon receipt of a request to flush a line in the E state, the slice sends a single message along the ring, in one privileged direction. For core numbered from 0 to  $\frac{n_{core}}{2}$  included, this is towards the higher numbered cores (and the GPU), otherwise, it is towards the lower numbered cores. This message is passed around the ring until the victim core  $V$  that has the line cached in its lower level cache (L1/L2) receives it. If the core is not in the initial direction, the message will follow the ring back in the other direction until it reaches the victim core. The victim core then discards the line, which is clean, and sends a reply to the slice, along the shortest path. The eviction time in state E,  $t_E(A, S, V)$  is thus:

$$t_E(A, V, S) = \begin{cases} C' + h \times |A - S| + h \times |R - (V - M)| & \text{if } S \leq \frac{N}{2} \text{ and } V < S \\ C' + h \times |A - S| + h \times |S - V| & \text{if } S \leq \frac{N}{2} \text{ and } V \geq S \\ C' + h \times |A - S| + h \times |S - V| & \text{if } S > \frac{N}{2} \text{ and } V \leq S \\ C' + h \times |A - S| + h \times |M - V| & \text{if } S > \frac{N}{2} \text{ and } V > S, \end{cases}$$

where:

- ▶  $C'$  is a different base time constant,
- ▶  $h$  is a constant corresponding to the time a round-trip hop on the ring takes,
- ▶  $N$  is the number of cores,
- ▶  $R$  is the ring diameter in hops, corresponding to how many hops there are between the system agent and the GPU, and thus, in our case,  $R = N + 1$ .

In addition to our measurements, Figure 2.8a and Figure 2.8b present the fitted model for the 8-core machine, which appears to explain the behavior consistently.

**Summary.** We have uncovered that while CPUs appeared to be arranged symmetrically in Intel's bidirectional ring, they are in fact aligned one after the other in a linear graph, with the system agent at an end and the GPU at the other end. First, the `clflush` instruction timing is always influenced by the distance between the core requesting the flush and the slice where the address lives in the last-level

cache. Second, in the I state the timing will depend on the distance between the slice and the system agent, whereas in the E state, it will depend on how long a message sent along the ring will need to reach the core that currently has the line, and then go back to the slice. These findings are consistent with the concurrent work of Paccagnella et al. [118].

### 2.3.3 Security impact

This reverse-engineering work leads to the improvement of the error rate in the Flush+Flush [74] side channel. The attack setup is therefore the same as with Flush+Flush: the attacker needs to be co-located to the same CPU as the victim to share the last-level cache, and relies on shared memory with the victim (e.g., a shared library or memory deduplication). The attacker can then carry on a side-channel attack on a vulnerable implementation, or a covert channel between two parties they control.

**Attacker model.** We define different attacker models depending on attacker capabilities.

- ▶ *Global Threshold (GT)*: The simplest model, using a single threshold that minimizes the average error rate over all triples of attacker, victim, and slice. This is a topology-oblivious attacker, as in the initial Flush+Flush attack.
- ▶ *Best A, Known V*: The attacker knows on which core the victim is running and chooses the attacker core it runs on. The attacker computes a single threshold for all addresses, therefore ignoring the impact of cache slices.
- ▶ *Best AV*: The attacker can pick the cores both the victim and the attacker are running on, e.g., in the case of a covert channel or a side-channel attack in which the attacker launches the victim process. It ignores the impact of slices.
- ▶ *Known  $\tilde{S}$* : The attacker does not know on which core they or their victim runs, but takes into account the slices, using per-slice thresholds. We use this model for comparison with the GT model.
- ▶ *Best A, Known  $\tilde{S}V$* : The attacker pins their process to the best core, knows the victim core and takes into account the slices. This is a realistic attacker model. To be compared with *Best A, Known V* model.
- ▶ *Best AV, Known  $\tilde{S}$* : This is the most powerful side-channel attacker, that can pin both the attacker and victim.
- ▶ *Best AV $\tilde{S}$* : This is the best covert channel attack model, where the attacker chooses the cores and an address in a slice that yields the best results.

**Experimental results on error rate.** We measure the error rate that can be achieved for each triple consisting of an attacker core, a victim core, and a slice. For each  $(A, V, \tilde{S})$  we make  $2^{20}$  measurements,  $2^{19}$  hits (in E state), and  $2^{19}$  misses (in I state). From these distributions, we evaluate the number of hits and misses that would be correctly or incorrectly classified using a threshold, and determine thresholds that minimize the average error rate for each model, along with the corresponding average error rate.

Table 2.1 shows the error rates for the 4-core and 8-core machines, indicating for  $A$ ,  $V$  and  $\tilde{S}$  whether they are unknown, known, or chosen in each case. For the 8-core machine, we observe a staggering difference between the 25% error rate of the *GT* attacker model, to the close to 0% error rate of the *Best AV $\tilde{S}$*  model.

Table 2.1: Results for each attacker model. “U” means Unknown, and “K” Known.

	4-core machine			8-core machine				
	Error rate	$A$	$V$	$\tilde{S}$	Error rate	$A$	$V$	$\tilde{S}$
GT	14.0%	U	U	U	25.1%	U	U	U
Best $A$ , Known $V$	6.07%	3	K	U	10.5%	7	K	U
Best $AV$	0.176%	7	0	U	0.115%	7	8	U
Known $\tilde{S}$	11.6%	U	U	K	22.8%	U	U	K
Best $A$ , Known $\tilde{S}V$	3.16%	5	K	K	7.18%	7	K	K
Best $AV$ , Known $\tilde{S}$	0.103%	7	0	K	0.0174%	1	0	K
Best $AV\tilde{S}$	$4.96 \times 10^{-3}\%$	3	3	3	$0 (< 2^{-20})$	2	7	14

**Covert channel.** We implement a framework to benchmark covert channel ideal bandwidth with different primitives. Table 2.2 shows that our carefully calibrated Flush+Flush yields a threefold increase in bandwidth on both machines compared to the naive Flush+Flush, and provides a bandwidth higher than Flush+Reload by 3 to 4 %.

Table 2.2: Covert channel benchmarking.

Channel	4-core machine			8-core machine		
	Capacity	Bit rate	Err. rate	Capacity	Bit rate	Err. rate
Naive Flush+Flush	1.01 Mbitps	2.96 Mbitps	20%	1.88 Mbitps	5.89 Mbitps	23%
Optimized Flush+Flush	2.99 Mbitps	3.03 Mbitps	0.1%	5.81 Mbitps	5.81 Mbitps	0.005%
Flush+Reload	2.88 Mbitps	2.91 Mbitps	0.1%	5.57 Mbitps	5.57 Mbitps	0.0005%

**Side-channel attack on AES.** As the AES T-tables implementation is well-known to be vulnerable to side-channel attacks, we use it as a benchmark to compare our Flush+Flush implementation to the naive version of Flush+Flush and to the Flush+Reload attack.

A naive Flush+Flush attack will show some lines with all hits or all misses, due to the threshold depending on the slice. Using a per-slice threshold allows us to achieve an accuracy similar to Flush+Reload. Again, accounting for the contribution of slices and CPU interconnect to `clflush` timing variations makes an optimized Flush+Flush channel competitive with Flush+Reload, and improves the reliability over naive Flush+Flush. We refer to the paper [54] for more details about the side-channel attack on AES T-tables implementation.

**Summary.** Choosing the attacker and victim locations significantly improves the accuracy over the very unreliable global threshold. On top of that, using a per-slice threshold provides a further boost. However, when the victim cannot be chosen, accounting for slices gives a much greater boost. Lastly, choosing the best combination of attacker, victim, and slice gives close-to-perfect error rates. We conclude that Flush+Flush with careful calibration is a compelling alternative to Flush+Reload.

## 2.4 Discussion

These works were focused on uncovering the hidden part of the iceberg that is the side-channel attack surface of modern CPUs – beyond caches and branch predictors. In all cases, we required a first phase of reverse engineering to fully understand how a particular micro-architectural component works before being able to attack it. While some people find it ingrate work, as manufacturers indeed possess the blueprints for these components that are just undocumented to the public, I believe that this is important work. Indeed, much like software obfuscation does not resist a sufficiently motivated attacker in most cases, these works (and their associated related work) show that a sufficiently motivated researcher is indeed able to reverse-engineer components – if not always precisely, at least adequately to obtain a model that allows novel or improved attacks.

### ⌘ Limitations

A striking limitation of these works, also applicable to most – if not all – related work, is that the reverse-engineering phase relies on a lot of manual work. While we strive to automate the process as much as possible, the very first parts of the research process, *i.e.*, how to choose the component to study, and how to acquire measurements, require expertise and manual intervention. A related concern is that, as CPUs become more and more efficient, either new components emerge, or known components are modified by manufacturers. These changes occur at a very fast pace, as new CPU generations are devised every year. This means that the attack surface may change every year as well, with the risk of rendering obsolete the research done on older components. A research direction that is orthogonal to this is the automated discovery of side channels [160, 170].

A second limitation, also applicable to related work, is that we establish covert channels to evaluate the reverse-engineering: they are a proof-of-concept that, by design, corroborates the correctness of the reverse-engineering. The main limitation concerns comparisons between covert channels from one paper to another. As they are mainly proofs-of-concept, the protocols are always ad-hoc and change from one paper to another. The creation of a standard benchmark for covert channels would help establish a fair comparison.

## Chapter 3

# *Tu quoque, my browser*

*In which we show that web browsers are not spared by side channels.*

---

Micro-architectural attacks are usually demonstrated in native environments, and written in low-level languages like C (often helped with inline assembly), considering that they require great control of micro-architectural components. Porting such attacks to browsers is not trivial, as several layers of abstraction separate scripts running on browsers from hardware components.

However, fundamentally, side-channel attacks only require benign actions. Indeed, all of them require a timer, and cache attacks, for example, require accessing precise regions of memory. While JavaScript executes in a strict sandbox and hides the notion of addresses and pointers, these two pre-requisites can be fulfilled, therefore allowing side-channel attacks in browsers. The first micro-architectural attack in JavaScript was shown by Oren et al. [115] in 2015, where they demonstrated a cache attack in JavaScript.

Since then, the landscape of browsers changed a lot, including in reaction to micro-architectural attacks. In this chapter, we delve into these changes and their consequences in terms of security.

### Outline

Section 3.1 presents our results on browser timers. We first investigate the (in)security of resolution clamping. We show that it is, in practice, possible to retrieve a nano-second resolution from browser vendors that clamped their timers to  $5\ \mu\text{s}$  (e.g., Chrome and Firefox) or even  $100\ \text{ms}$  (e.g., Tor Browser). We then study the evolution of JavaScript's timers in browsers and their multiple changes in the last few years, and seek to evaluate their security. Section 3.2 presents a novel side-channel attack in browsers, using port contention. Section 3.3 discusses these results.

### 3.1 Timing is everything

In response to Oren et al. cache attack, the W3C [162] and browser vendors [43, 13, 28] have clamped the `performance.now()` method to a resolution of  $5\ \mu\text{s}$ . The timestamps in the Tor browser had been even more coarse-grained, at  $100\ \text{ms}$  [108]. The rationale is simple: a timer that is clamped to  $5\ \mu\text{s}$  will output 0 to any measurement below  $5\ \mu\text{s}$ , and, in effect, does not allow differentiating between cache

hits and cache misses, which would require a resolution of 10-100 ns<sup>1</sup>.

Timers are the main, if not the only, variable over which browser vendors have some control when it comes to side-channel mitigation. It is therefore unsurprising that most countermeasures were developed from this angle.

### 3.1.1 (In)security of resolution clamping

During my postdoc, we sought to evaluate the security of resolution clamping and its impact on side-channel attacks in browsers. The work presented hereafter has been published in:

Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript”. In: *International Conference on Financial Cryptography and Data Security (FC)*. 2017

We demonstrate that reducing the resolution of timing information or even removing these interfaces is completely insufficient as an attack mitigation. We propose several new mechanisms to obtain absolute and relative timestamps. We evaluated 10 different mechanisms on the most recent versions, at the time of writing, of 4 different browsers: Chrome, Firefox, Edge, as well as the Tor browser, which took even more drastic measures. We show that all browsers leak highly accurate timing information that exceeds the resolution of official timing sources by 3 to 4 orders of magnitude on all browsers, and by 8 on the Tor browser. In all cases, the resolution is sufficient to revive the attacks that were thought mitigated.

Our main insight is that there are two ways of recovering a high-resolution timer on a browser that clamped timer resolution:

1. Recovering a high resolution from the provided High Resolution Time API, *i.e.*, `performance.now()`.
2. Creating alternative timing primitives, *implicit* timers, from scratch, using the browser API.

#### 3.1.1.1 Recovering a high resolution from `performance.now()`

It has already been observed that it is possible to recover a high resolution by observing the clock edges [105, 144, 157, 92]. We systematize these observations and derive two methods to recover a high resolution from `performance.now()`.

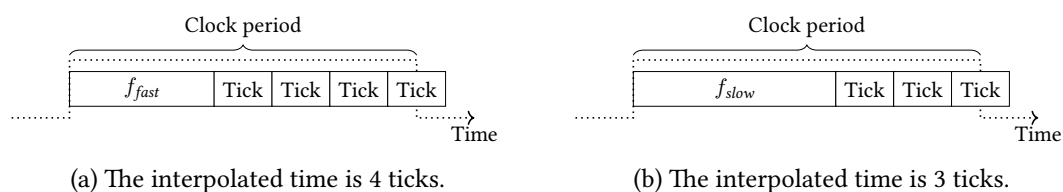


Figure 3.1: Clock interpolation: Counting the number of ticks between the end of the function to measure and the end of the clock period.

<sup>1</sup>This rationale is embodied in a commit message on July 7, 2015, from bugzilla, Mozilla’s bug tracker “Clamp the resolution of `performance.now()` calls to 5us, because otherwise we allow various timing attacks that depend on high accuracy timers” [28].

**Clock interpolation** By definition, a timer cannot measure an event shorter than its resolution. Yet, it is possible to bypass this limitation by using interpolation to retrieve an even finer-grained timer. The idea behind it is straightforward: one counts the number of times a shorter non-free running operation can be executed, e.g., increasing a custom counter by one. We refer to such an operation as a tick. By starting the function to measure at the beginning of a clock period and counting ticks when they are finished, we can conclude how fast a function is when the next clock edge is reached. Figure 3.1 illustrates a measurement using clock interpolation. It allows us to recover a resolution of 500 ns on Firefox and Chrome, and 15  $\mu$ s on Tor browser.

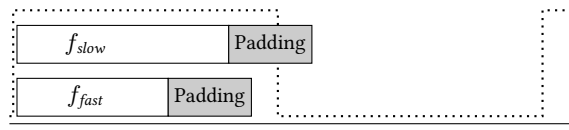


Figure 3.2: Edge thresholding: Applying padding such that the slow function crosses one more clock edge than the fast function.

**Edge thresholding** We do not require an exact timestamp in all cases, instead, it is sufficient to distinguish two operations  $f_{fast}$  and  $f_{slow}$  based on their execution time. Using multiple constant-time operations, we generate a padding after the function we want to measure. We choose the padding in such a way that  $t_{slow} + t_{padding}$  crosses one more clock edge than  $t_{fast} + t_{padding}$ , i.e., both functions take a different amount of clock edges. Figure 3.2 illustrates a measurement using edge thresholding. It allows us to recover a resolution of 2 ns on Firefox and Tor browser, and 15 ns on Chrome.

### 3.1.1.2 Creating alternative timing primitives

In cases where the High Resolution Time API [162] is not available, we have to resort to different timing primitives, as highlighted by Kohlbrenner et al. [92]. As there is no other high-resolution timer available in JavaScript and no access to any native timers, the only option is to create our own timing sources. In most cases, it is sufficient to have a fast-paced monotonically increasing counter as a timing primitive that is not a real representation of time but an approximation of a highly accurate monotonic timer. This concept was presented by Wray in 1992 [175] and demonstrated by one of our other work on ARM-based side-channel attacks [101].

We built an assortment of timers using JavaScript features such as timeouts (`setTimeout`), message passing (`postMessage`), the channel messaging API, and CSS version 3 animations. The mere fact that we can create timers out of the JavaScript API, and specifically, elements that were never designed to create a timer, is already interesting. However, the resolution of these timers remains lower than the one of `performance.now()`, rendering their usage anecdotal.

However, `SharedArrayBuffer`, an extension of web workers, experimental at the time but now fully supported, can be used to create a truly high-resolution timer. This shared resource provides a way to build a real counting thread with a negligible overhead compared to a message passing approach. This already raised concerns with respect to the creation of a high-resolution clock [98]. In this method, one worker continuously increments the value of the buffer without checking for any events on the event queue. The main thread simply reads the current value from the shared buffer and uses it as a high-resolution timestamp. The resulting resolution is close to the resolution of the native timestamp counter. On our Intel Core i5-6200U test machine, we achieve a resolution of up to 2 ns.

### 3.1.1.3 Impact on micro-architectural attacks

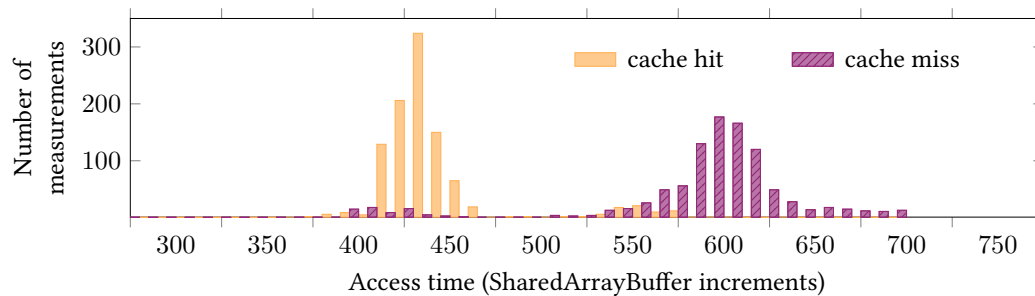


Figure 3.3: Histogram for cache hits and cache misses in a browser, using `SharedArrayBuffer` as a timer.

Figure 3.3 shows the timing difference between cache hits and cache misses, measured with the `SharedArrayBuffer` method. The ability to measure this timing difference is the building block of all cache attacks. This revives the attack of Oren et al. [115], which was only mitigated by clamping `performance.now()`. `SharedArrayBuffer` is now fully supported by all major browsers and became the *de facto* timing method for micro-architectural attacks in browsers.

### 3.1.2 Evolution of JavaScript’s timers in browsers

While we responsibly reported our findings on the insecurity of clamping `performance.now()` resolution to most major browser vendors in 2017, nothing changed with respect to timers until the beginning of 2018, with the disclosure of Spectre [90]. In the sheer panic that followed from vendors ranging from CPU vendors to browser vendors, timers were “fixed” several times in the course of a few months. With Thomas Rokicki, who started this work as a master student and followed as a PhD student, and Pierre Laperdrix, we started investigating how timers evolved, and crucially, the impact of these changes in terms of security. The work presented hereafter has been published in:

📖 Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. “SoK: In Search of Lost Time: A Review of JavaScript’s Timers in Browsers”. In: *EuroS&P*. 2021

It is important to understand the context in order to understand the choices that have been made by browser vendors. Spectre is not a side-channel attack, but rather a transient execution attack. While we will not delve into the details of this attack, one important aspect to understand is that it is far more dangerous than side-channel attacks, as it can leak information from the browser’s memory such as passwords and session cookies [135]. Another important aspect is that countermeasures are different from side-channel attacks. Timers still play an important role, but such attacks can also be mitigated by appropriate isolation.

This explains the immediate and drastic decisions that were made at the beginning of 2018. Figure 3.4 shows the changes to `performance.now()` resolution from 2015 to 2022.

#### 3.1.2.1 Isolation

Isolation is a staple of web security. Separating resources and communication between contexts reduce the threat surface drastically. The same-origin policy [48] has been a major security feature of the web



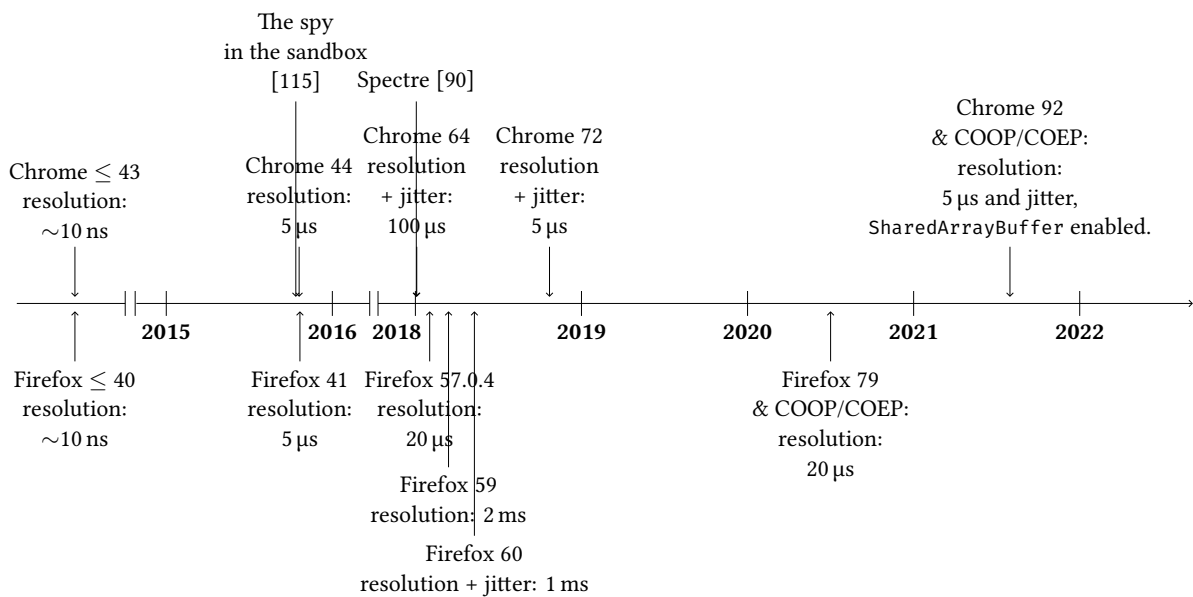


Figure 3.4: Timeline of JavaScript timers in browsers, from 2015 to 2022.

for years, well before Spectre. An origin is defined by the tuple (Protocol, Port, Host). The same-origin policy restricts read access to resources loaded from a different origin. This means that data from one website, e.g., authentication cookies, are not accessible from another website loaded in another tab. This countermeasure aims at mitigating, among others, attacks using browser resources, but it has no impact on other types of timing attacks such as micro-architectural attacks, including side-channel attacks and transient execution attacks. Two novel isolation mechanisms were developed in the wake of Spectre: Site Isolation and COOP/COEP.

**Site Isolation** In response to transient execution attacks, Chrome developed a new security measure called Site Isolation [126], which forces each website, defined by the tuple (Protocol, Host), to run in a specific process, not shared with other websites. This prevents an attacker to access the mapped memory space of another website and mitigates the effect of some JavaScript-based transient execution attacks by preventing access—including transient—to out-of-bound information. This feature was deployed in Chrome 67 and is deployed in Firefox desktop since version 95 under the code name “Project Fission” [173].

**COOP/COEP** Site Isolation was extended with the introduction of the Cross-Origin Embedder Policy (COEP) and Cross-Origin Opener Policy (COOP) [19]. COOP ensures that a top-level window is isolated from other documents by putting them in a different browsing context group. COEP complements COOP by forcing the browser to only load trusted resources. Both COOP and COEP rely on policies defined through HTTP headers and they were both added in Firefox 79 and Chrome 83 [49, 50].

### 3.1.2.2 Impact on timers

The impact on timers has been twofold, and coupled with the changes in isolation over several months: a first short-term change has been to lower even more the timer resolution and add jitter, and a second

medium-term change has been the addition of the isolation countermeasures and, subsequently, the re-appearance of high-resolution timers.

Mozilla first clamped `performance.now()` to a resolution of  $20\ \mu\text{s}$  in Firefox 57.0.4 [163] then furthermore to  $2\ \text{ms}$  in Firefox 59 [36]. However, such drastic countermeasures also had an impact on web development [38]. Browser vendors then introduced jitter to `performance.now()` in order to prevent clock interpolation: Firefox 60 set the resolution to  $1\ \text{ms}$  and added a jitter of  $1\ \text{ms}$  range [37, 47], while Chrome 64 set the resolution to  $100\ \mu\text{s}$  with a jitter of  $100\ \mu\text{s}$  [94].

Because of the security added by Site Isolation, most browsers have re-allowed high-resolution timers under certain conditions. Since Chrome 72, `performance.now()` has a resolution of  $5\ \mu\text{s}$  with a jitter of  $5\ \mu\text{s}$  under Site Isolation (which is present by default), whereas, since Firefox 79, it has a resolution of  $20\ \mu\text{s}$  without jitter, only when COOP/COEP is activated [35, 114].

The jitter being higher than the needed resolution for timing attacks, it is no longer possible to use `performance.now()` interpolation. However, jitter only creates statistical protection against timing attacks.


**Clock amplification** By repeating the measurement, an attacker can average the results and reduce the impact of jitter. This amplification allows an attacker to recover a higher resolution. Amplifying the results by repeating measurements is, however, not adapted to all attacks. Specifically, it only works when the attacker controls the event that creates the timing difference. For instance, in the case of a covert channel, the attacker can recreate the conditions for the same measurement several times.

An attentive reader will note that we are discussing the impact of changes on timers, after concluding a few pages before that the highest-resolution timer was, in fact, `SharedArrayBuffer`, *i.e.*, not a proper timer. These changes have therefore no impact on `SharedArrayBuffer`. `SharedArrayBuffer` has been disabled by default in Chrome 60 and Firefox 57.0.4 to mitigate Spectre. With the introduction of mitigations to transient execution attacks, they have been re-enabled, and continue to be supported to this day.

We performed a longitudinal study of `performance.now()` interpolation, `performance.now()` amplification, and `SharedArrayBuffer`. Advances in isolation are a great step forward in terms of security, but are not sufficient alone to mitigate the vast majority of timing attacks. We conclude that the recent increase of timer resolution coupled with the reactivation of `SharedArrayBuffer` represents a massive step back in security where contention-based side-channel attacks can be run in similar conditions to the ones in 2015, *i.e.*, unmitigated.

## 3.2 Port contention side channels in browsers

Cache attacks are starting to be well understood, including in browsers. During Thomas Rokicki's PhD, and with Yossi Oren and Marina Botvinnik from Ben Gurion University (Israel), we were interested in exploring the attack surface of web browsers in terms of micro-architectural attacks, and investigating port contention attacks. The work presented hereafter has been published in:

 Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. "Port Contention Goes Portable: Port Contention Side Channels in Web Browsers". In: *ASIACCS*. 2022

Port contention side-channel attacks have been introduced by Aldaya et al. [10]. This attack on Intel CPUs is based on port contention, where CPU ports act as a bottleneck in the execution pipeline.

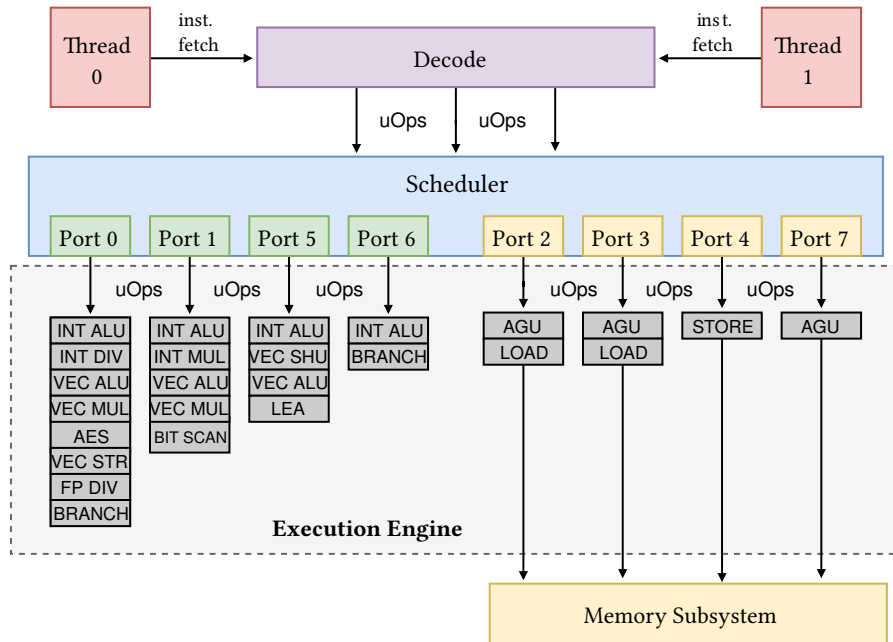


Figure 3.5: Execution engine of an Intel CPU (Figure modified from [10]).

By sharing ports with the victim, the attacker can exploit timing differences caused by the contention of different instructions.

When compared to cache attacks such as Prime+Probe, native port contention attacks offer better speed and spatial resolution, do not require a complex cache profiling step, and are more resistant to noise. Mounting a port contention attack in a browser setting, therefore, delivers an advantage to attackers. Performing such an attack, however, is far from trivial. Port contention requires an attacker process that is co-located with the victim on the same processor core and executes assembly language instructions carefully chosen to conflict with the victim’s instructions. This is highly challenging in a web browser environment:

- C1 In this setting, the attacker’s code is written in a highly-abstracted language that is translated into machine code by a just-in-time compiler.
- C2 The attacker has no control over the physical core selected by the browser to execute the attack.
- C3 Finally, web-based timers have a lower resolution than native hardware-based timers, increasing the attacker’s measurement noise.

We will not delve into C3, as we already covered timers in the previous section.

### 3.2.1 Porting port contention side channels in browsers

**Tackling C1.** We use WebAssembly, an open-sourced binary instruction format designed to be deployed on the web. It functions as a low-level, assembly-like, program. It supports two main formats: binary, which is directly interpretable by the engine, and the text format, human-readable format, allowing to read and modify compiled WebAssembly code. We develop PC-detector, a Selenium-based framework to dynamically detect and characterize the port usage of WebAssembly instructions. Out of 100 WebAssembly instructions evaluated, we have found 21 causing contention on P1 or P5.

**Tackling C2.** We perform our attack on multiple cores simultaneously by using the Web Workers API, JavaScript multi-threading implementation, which creates a sub-thread running in a different process. This lets the attacker create as many attacker processes as physical cores, and as they all have a high workload, they are spread to different physical cores by the operating system scheduler. Then, one of the attacker processes runs on the same core as the victim process, able to monitor it.

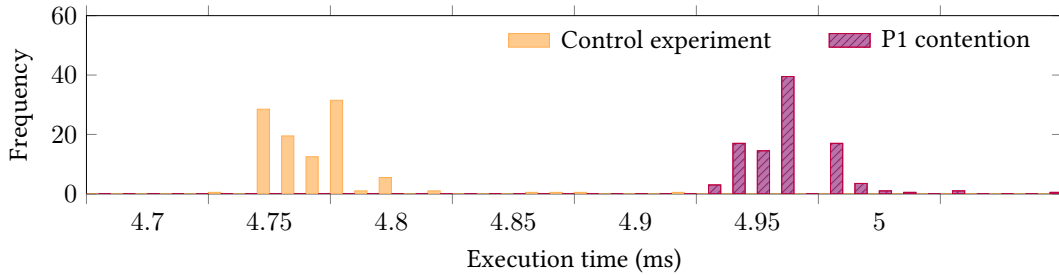


Figure 3.6: Port 1 contention experiment on `i64.ctz` for 1 000 000 instructions.

**Proof of concept.** Figure 3.6 shows a proof-of-concept illustrating the contention on P1 caused by the WebAssembly `i64.ctz` instruction. In this experiment, we time the execution of 1 000 000 WebAssembly `i64.ctz` instructions. In parallel, we run a sender program written in native code and pinned to the same physical core. In the P1 contention experiment, the native sender runs the Intel instruction `crc32` in a loop. This assembly language instruction is known to cause contention on P1. In the control experiment, the native sender runs a simple loop designed not to cause port contention. As the figure shows, the timings measured during the P1 contention experiment are on average 5% higher than the control experiment, allowing us to distinguish the two distributions.

### 3.2.2 Security impact

We built two different attacks to evaluate port contention in browsers: a side-channel attack on artificial targets (to evaluate the spatial resolution), and a covert channel (to evaluate the temporal resolution).

**Side-channel attack.** Our synthetic and generic example, implemented in native assembly code, executes instructions creating contention either on P1 (`popcnt`) or P5 (`vpbroadcastd`), depending on a secret bit. It shows how a program, whose execution depends on secret information, is vulnerable to WebAssembly port contention. The victim process is an unprivileged native process. The attacker is a JavaScript and WebAssembly script, running inside the browser’s sandbox. In our implementation, an attacker, running code inside the browser’s sandbox, monitors the victim’s execution with a spatial resolution of 1024 native instructions, *i.e.*, 3072 bytes. This spatial resolution is of the same order of magnitude as other microarchitectural attacks in the browser, *e.g.*, Prime+Probe, which has a resolution of a cache set (typically 12 to 20 cache lines), *i.e.*, 1280 bytes on our system.

**Covert channel.** We then evaluate our covert channel in different scenarios:

**Native-to-web** A native sender running unprivileged C code with a receiver inside the JavaScript sandbox. This covert channel has a bit rate of 200 bps (1% error rate) without noise, 120 bps (3% error rate) when running `stress -m 2`, and 25 bps (5% error rate) when running `stress -m 3`.

We observe that the performance only drops when a noisy thread runs on a physical core shared either by the clock or the receiver (*i.e.*, with stress on 3 cores). This is a difference from cache attacks, which are affected by any noise on the cache, regardless of the core.

**VM-to-web** A native sender running inside a virtual machine with a receiver inside the JavaScript sandbox on the host. This covert channel has a bit rate of 80 bps without noise. The main difference from the native-to-web scenario is that the sender cannot know nor control which core it is running on, which presents an additional difficulty.

**Cross-browser** Both the sender and the receiver are inside the JavaScript sandbox. This covert channel has a bit rate of 200 bps without noise. In this scenario, the sender also cannot know nor control which core it is running on.

### 3.3 Discussion

More and more applications are moving to Web-based services. Web applications ease initial deployment (no need for installation), updates (server side pushes new code), and portability (compilation happens just in time). This process is enabled by recent advances in browser technology with new features previously only available to native software. At the same time, we have shown that side-channel attacks can be performed inside the JavaScript sandbox, thereby increasing the side channel attack surface. Indeed, an attacker now does not have to install malware on the victim's system, but merely just needs to lure their victim to a malicious website (or on a website with malicious advertisement), and push seemingly arbitrary code to a large user base.

These attacks are very hard to patch from the perspective of browser vendors, as the root cause lies several abstraction layers below the browser. One, if not the only, leverage is timers, as a high-resolution timer is a pre-requisite for all micro-architectural attacks. However, as new features are pushed to browsers — such as `SharedArrayBuffer` objects that allow concurrency in JavaScript and are therefore very useful to web developers —, they get repurposed by attackers to create new high-resolution timers that evade countermeasures.

Ultimately, I believe it is fair to say that there will always be resource contention side channels visible from browsers, and that will need to be able to live with it at the browser level. One avenue to better protect sensitive information is the ability to better isolate secrets in hardware, and better coordination between hardware and software. The interplay between micro-architecture and web browsers is something that we will continue to explore in the next few years with the ANR-DFG PRCI project FACADES (“Fingerprinting And CPU Attack and Defense Exploration from browser Scripts”), with the Spirals group and colleagues from CISP and Saarland University in Germany.

#### Limitations

As browsers are ever-changing, at an even faster pace than CPUs, the precise results of these studies will not necessarily hold for future browser versions. In all our work, we aim at creating frameworks that will be reusable to reproduce our results in the future. An example is the PC-detector framework, designed to characterize the port usage of WebAssembly instructions: between the submission of the article and the camera ready, WebAssembly had released more instructions, which we were able to characterize as well and add to the article swiftly.

## Chapter 4

# Conclusions and perspectives

*A ship in port is safe; but that is not what ships are built for.*

---

Grace Murray Hopper

### 4.1 Main results

My research revolves around micro-architectural security. In this manuscript, we have first tackled the research question “*What are the vulnerable components, and how to use them to leak secret data?*” by working very close to the hardware, which involves reverse-engineering undocumented micro-architectural components and building novel side channels (Chapter 2). We uncovered two novel side channels in DRAM and AMD cache way predictor. DRAM side-channel attacks have pushed the envelope in terms of attacker model in micro-architectural attacks, as it was the first one — and still one of the very rare — where an attacker does not require to be co-located on the same CPU as their victim, e.g., in dual socket systems. The AMD cache way predictor is also a good target for attackers, as it allows stealthier attacks that do not induce last-level-cache evictions. Finally, we refined a cache side channel by modeling the ring interconnect. This showed the viability of the Flush+Flush attack on the last-level cache, as it is not as prone to noise as the original work suggested.

Second, we tackled the research question “*What are the vulnerable pieces of software, and what are the different attack deliveries?*” by working much higher in terms of abstraction layers, and investigating the applicability of side channels in web browsers (Chapter 3). We have shown that clamping the resolution of timers does not prevent attacks that rely on high-resolution timers, and that the advances in isolation and subsequent increase in timer resolution represented a massive step back in the protection against contention-based side channels. We have also demonstrated that CPU port contention attacks can be ported to Web browsers and are less vulnerable to noise than cache attacks, with a faster bit rate.

These two research directions have emerged since the end of my PhD — when micro-architectural security was mostly a matter of cryptanalysis, and caches were the main side-channel vector that was studied — to now. This research paved the way for a better understanding of the attack surface in terms of hardware — no, no micro-architectural component is spared by side-channel attacks —, and in terms of attack delivery — yes, the web is concerned too.

This research has been a collaborative endeavor, with my students and my colleagues around the

world. While manuscript redaction has been a relatively solitary experience, the results presented here could not have happened all by myself. Collaborations are at the heart of this research if only by the range of expertise they require: security, micro-architecture, operating systems, software, cryptography, web... I am very happy and thankful to have met so many skillful students and colleagues – and last but not least, to have shared so many coffees and laughs, essential fuel for good research.

## 4.2 Perspectives

Among other things, there are three research directions that I would like to pursue in the next years.

**Automated detection and correction of side-channel vulnerabilities.** Until now, my research has focused on both how to extract information from micro-architectural components, and on the delivery of the attacks, but not on finding side-channel vulnerabilities in software, which is another crucial aspect for end-to-end attacks.

In recent years, and especially since 2017, there has been a trend to try to automatize the detection of side-channel vulnerabilities [21, 25, 26, 34, 41, 42, 52, 66, 87, 113, 127, 147, 149, 164, 165, 166, 172], with many open sourced tools. However, in the same time, there has been no shortage of novel vulnerabilities found (mostly manually) in cryptographic libraries [10, 11, 12, 18, 31, 32, 44, 61, 65, 79, 121, 133, 134, 156], despite the fact that constant-time programming is a well-known paradigm that is (tentatively) applied in all major libraries.

This paradox has been studied by Jancar et al. [86] to understand the awareness and usage of automated detection tools by developers of cryptographic libraries. This study found that all 44 participants, developers of 27 popular cryptographic libraries or primitives, were aware of timing attacks, but many do not have a systematic approach to address them. Participants of this study cite the lack of usability of formal analysis tools, perceiving them as requiring too much effort for the guarantees. However, they were more positive about dynamic instrumentation tools like ctgrind.

The question of *usability* should therefore be a major concern when designing tools, while most of the time, mainly the number of new vulnerabilities found is taken into account in scientific publications. Another problem rises during the evaluation of these tools: there does not exist any reference benchmark to compare them, which is problematic even when tools are open-sourced. Finally, the evaluation of these tools is mostly done on cryptographic primitives, while the most recent vulnerabilities found in the wild concern either protocols or cryptographic modules such as random number generation, secret key generation, or key parsing. It is therefore unclear whether an automated detection tool would have found these vulnerabilities, and highlights the need for *scalability*.

I would like to focus on these issues that impede practical and impactful results as of now. My objective is first to create a reference benchmark to compare detection tools against each other. Second, an in-depth evaluation should highlight the strengths and limitations of existing tools, uncover whether the tools are able to find manually found vulnerabilities on protocols or cryptographic modules, and expose features that might be missing from existing tools. Similar endeavors have already been started by Barbosa et al. [22] around formal verification tools for the design, analysis, and implementation of cryptography, and by Buhan et al. [39] around automated leakage detection of physical side channels. However, the tools we consider are different, as is their evaluation method. From there, I believe that we will see more approaches that aim at automatically protecting or repairing libraries vulnerable to side channels, in the vein of Borello et al. [29] or Rane et al. [124].

One open question is how to protect software against side-channel attacks like Hertzbleed [167], *i.e.*, frequency side channels. Under certain circumstances, the CPU frequency depends on the data

being processed due to dynamic voltage and frequency scaling (DVFS). Wang et al. show an attack on a constant time implementation of SIKE, a post-quantum key encapsulation mechanism. One thing to note is that the threat model is the same as other micro-architectural attacks, but Hertzbleed can exploit software that was previously believed to be secure.

**Side-channel attacks and fingerprinting.** Fingerprinting can be seen as a class of side-channel attacks. Both find their origins in the numerous abstraction layers in complex systems and the implementation of complex standards. For a side-channel attack, an attacker tries to retrieve an inaccessible secret while using non-functional information (e.g., execution timing) to leak a secret. For fingerprinting, an attacker tries to retrieve or construct a unique identifier while using attributes that depend on the software and hardware configuration of the machine on which the web page is displayed. In both cases, an attacker relies on auxiliary information on the program's execution, whether it is a cryptographic library or displaying a web page.

In the state of the art, the vast majority of fingerprinting techniques are based on attributes sent by the browser itself [96, 97]. My objective is to infer hardware characteristics from side channels via micro-benchmarks, instead of querying static attributes from the browser API. This is a research direction that I want to pursue since the end of my PhD and that I finally started to work on. With a collaboration between the Spirals group, Ben Gurion University (Israel), and Adelaide University (Australia), we studied the GPU as a source of hardware fingerprinting [95]. In this work, we have shown that the difference in execution time between different execution units in individual GPUs can serve as a unique and robust fingerprint. With my PhD student Thomas Rokicki and Michael Schwarz from CISPA (Germany), we also investigated the difference in mapping between  $\mu\text{ops}$  and CPU ports, which changes from one CPU generation to another [132]. This as well can serve as a fingerprint, albeit less unique.

I believe there is still a lot of interesting work in that direction. One key point to be able to use side-channel fingerprinting is that, contrary to browser API attributes, the result of micro-benchmarks varies (even if slightly) from one execution to another. Therefore, current techniques that hash the fingerprint will fail at properly linking the same device, as the hashes will differ from one execution to another. We therefore need to quantify the reliability that is needed in a side-channel fingerprinting method, and to design new approaches to link these hardware fingerprints.

This is a project I will lead within the Spirals group, and with the help of the international project ANR-DFG FACADES with our colleagues from CISPA and the Saarland University in Germany. Indeed, several members in the Spirals group are experts in browser fingerprinting, and developed and maintained the AmIUnique platform<sup>1</sup> to study the impact of fingerprinting on a large scale.

**Towards more reproducible micro-architectural attacks.** Another research direction close to my heart is the reproducibility of micro-architectural attacks. Although these attacks are entirely software-based and several techniques are now known to improve the reproducibility of software experiments, e.g., by using containers such as Docker, these techniques do not solve a number of problems specific to micro-architectural attacks. Indeed, many parameters, both software and hardware, influence the success or failure of an attack, e.g., the CPU frequency, the cache organization, the version and optimizations of the compiler, and the deployment of particular countermeasures on the target machine. It is therefore not uncommon for code that works on one machine to produce different results on another, with no easy way of identifying the root cause. Another related problem is that the

---

<sup>1</sup><https://amiunique.org/>



---

complexity of micro-architecture increases generation after generation and that a vulnerability is sometimes linked to a micro-architectural component that is only available on a few models of machines, and therefore not necessarily within the reach of all researchers in the field.

This profoundly impacts the research domain, because it may be that some of the underlying assumptions about a vulnerability turn out to be wrong – in other words, we have built an attack, but we think it depends on a different component than it actually does. It is also complex or impossible to vary some parameters (especially hardware ones) on a machine to test multiple configurations or develop a new countermeasure.

With the national project ANR ARCHI-SEC, we propose to use the gem5 simulator to overcome some of these problems. This is a completely new direction in this field where simulators are only used to model new hardware countermeasures. The use of a simulator for hardware-related attacks is not trivial, because the processor must be faithful to reality, and the complexity and the lack of documentation can be detrimental to this. Nevertheless, with Pierre Ayoub, master student at the time and now PhD student, we have started to carry out experiments that are so far rather conclusive on the use of gem5 [20]. An interesting point that was raised during this study was the usefulness of visualization techniques (of the CPU pipeline in this case) during the reproduction of attacks. It would be very interesting to continue this visualization effort for other components, such as the cache or the branch predictor, to better understand and facilitate the development of attacks. We can also imagine the creation of a reference benchmark of attacks available to the community. Finally, I would like to study countermeasures in this context of a controlled environment where it is easier to change some parameters than on a physical machine.

Reproducibility is also an aspect that I try to push forward in the international security community since 2019. I have indeed created the first Artifact Evaluation at the WOOT workshop in 2019 (Workshop on Offensive Technologies, co-located with the USENIX Security Symposium at the time), being then Program Chair. An Artifact Evaluation allows, after the acceptance of the papers by the program committee, to validate, by another committee, that the artifacts submitted by the authors are consistent with the paper, complete, well documented, and easy to reuse. An article that passes this evaluation then earns one or multiple badges. This practice has been common for several years in some computer science communities, for example in software engineering, but was then still almost non-existent in the computer security community. I continued my involvement by being a co-chair of the Artifact Evaluation of USENIX Security Symposium, one of the top four international conferences in the field, for the 2021 and 2022 editions. These are the second and third editions respectively. In this third edition, we have improved the process to encourage authors to submit better-quality artifacts. Indeed, the conference only offered one badge to indicate that an artifact had passed the evaluation. We aligned ourselves with ACM, which offers three badges to indicate that an artifact: 1) is available online; 2) is functional; 3) can reproduce the results of the paper. We also introduced a standardized appendix allowing authors to describe their artifact and how it works. These two actions required a very important time investment from us because it is a new process for the organizers as well as for the authors and reviewers. To continue my actions on reproducibility, I am also a member of the Software and Source Codes College from the Committee for Open Sciences of the French Higher Education and Research Ministry (*Ministère de l'enseignement supérieur et de la recherche*).

# Bibliography

- [1] Onur Aciçmez. “Yet Another MicroArchitectural Attack: Exploiting I-cache”. In: *ACM Computer Security Architecture Workshop (CSAW)*. 2007.
- [2] Onur Aciçmez, Billy Bob Brumley, and Philipp Grabher. “New Results on Instruction Cache Attacks”. In: *CHES*. 2010.
- [3] Onur Aciçmez, Shay Gueron, and Jean-pierre Seifert. “New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures”. In: *IMA International Conference on Cryptography and Coding*. 2007.
- [4] Onur Aciçmez and Çetin Kaya Koç. “Trace-Driven Cache Attacks on AES”. In: *IACR Cryptology ePrint Archive* (2006). URL: <http://eprint.iacr.org/2006/138>.
- [5] Onur Aciçmez and Çetin Kaya Koç. “Trace-Driven Cache Attacks on AES (Short Paper)”. In: *ICICS*. 2006.
- [6] Onur Aciçmez, Çetin Kaya Koç, and Jean-pierre Seifert. “On the Power of Simple Branch Prediction Analysis”. In: *ASIACCS*. 2007.
- [7] Onur Aciçmez, Werner Schindler, and Çetin Kaya Koç. “Cache Based Remote Timing Attack on the AES”. In: *CT-RSA*. 2007.
- [8] Onur Aciçmez and Jean-Pierre Seifert. “Cheap Hardware Parallelism Implies Cheap Security”. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2007.
- [9] Onur Aciçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. “Predicting secret keys via branch prediction”. In: *CT-RSA*. 2007.
- [10] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. “Port Contention for Fun and Profit”. In: *S&P*. 2019.
- [11] Alejandro Cabrera Aldaya, Cesar Pereida García, and Billy Bob Brumley. “From A to Z: Projective coordinates leakage in the wild”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.3 (2020), pp. 428–453.
- [12] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. “Cache-Timing Attacks on RSA Key Generation”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.4 (2019), pp. 213–242.
- [13] Alex Christensen. *Reduce resolution of performance.now*. [https://bugs.webkit.org/show\\_bug.cgi?id=146531](https://bugs.webkit.org/show_bug.cgi?id=146531). 2015.
- [14] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. “Amplifying side channels through performance degradation”. In: *ACSAC*. 2016.
- [15] AMD. *AMD64 Architecture Programmer’s Manual*. 2017.

- 
- [16] AMD. *Software Optimization Guide for AMD Family 17h Processors*. June 2017.
- [17] Marc Andryscio, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. “On subnormal floating point and abnormal timing”. In: *S&P*. 2015.
- [18] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. “LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage”. In: *CCS*. 2020.
- [19] Anne van Kesteren Artur Janc Charlie Reis. *COOP and COEP explained*. [https://docs.google.com/document/d/1zD1fvfTJ\\_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k/edit](https://docs.google.com/document/d/1zD1fvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k/edit).
- [20] Pierre Ayoub and Clémentine Maurice. “Reproducing Spectre Attack with gem5: How To Do It Right?” In: *EuroSec@EuroSys*. 2021.
- [21] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. “Abacus: Precise Side-Channel Analysis”. In: *ICSE*. 2021.
- [22] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. “SoK: Computer-Aided Cryptography”. In: *S&P*. 2021.
- [23] Daniel J. Bernstein. *Cache-Timing Attacks on AES*. <http://cr.yp.to/antiforgery/cachetiming-20050414.pdf>. 2004.
- [24] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. “Branch Prediction Attack on Blinded Scalar Multiplication”. In: *IEEE Transactions on Computers* 69.5 (2020), pp. 633–648.
- [25] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation”. In: *ESORICS*. 2017.
- [26] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. “Vale: Verifying High-Performance Cryptographic Assembly Code”. In: *USENIX Security Symposium*. 2017.
- [27] Joseph Bonneau and Ilya Mironov. “Cache-collision timing attacks against AES”. In: *CHES*. 2006.
- [28] Boris Zbarsky. *Reduce resolution of performance.now*. <https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab>.
- [29] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. “Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization”. In: *CCS*. 2021.
- [30] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector”. In: *S&P*. 2016.
- [31] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. “Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild”. In: *ACSAC*. 2020.
- [32] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. “PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild”. In: *CCS*. 2021.
- [33] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. “RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks”. In: *USENIX Security Symposium*. 2020.
- [34] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *S&P*. 2019.

- 
- [35] Bugzilla. *Check crossOriginIsolated for all nsRFPService::ReduceTimePrecision\* callers*. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1586761](https://bugzilla.mozilla.org/show_bug.cgi?id=1586761). Apr. 2020.
- [36] Bugzilla. *Reduce Timer Resolution to 2ms*. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1435296](https://bugzilla.mozilla.org/show_bug.cgi?id=1435296). Feb. 2018.
- [37] Bugzilla. *Set Timer Resolution to 1ms with Jitter*. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1451790](https://bugzilla.mozilla.org/show_bug.cgi?id=1451790). Apr. 2018.
- [38] Bugzilla. *Unanticipated security/usability degradation from precision-lowering of performance.now() to 2ms*. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1440863](https://bugzilla.mozilla.org/show_bug.cgi?id=1440863). Feb. 2018.
- [39] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. “SoK: Design Tools for Side-Channel-Aware Implementations”. In: *AsiaCCS*. 2022.
- [40] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security Symposium*. 2019.
- [41] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. “Quantifying the Information Leakage in Cache Attacks via Symbolic Execution”. In: *ACM Trans. Embed. Comput. Syst.* 18.1 (2019), 7:1–7:27.
- [42] Jia Chen, Yu Feng, and Isil Dillig. “Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic”. In: *CCS*. 2017.
- [43] Chromium. *window.performance.now does not support sub-millisecond precision on Windows*. <https://bugs.chromium.org/p/chromium/issues/detail?id=158234>. 2015.
- [44] Shaanan Cohny, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. “Pseudorandom Black Swans: Cache Attacks on CTR\_DRBG”. In: *S&P*. 2020.
- [45] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. “Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers”. In: *S&P*. 2020.
- [46] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. “Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks”. In: *S&P*. 2019.
- [47] MDN Contributors. *Performance.now() API*. <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now>. Sept. 2020.
- [48] MDN Contributors. *Same-Origin-Policy*. [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy).
- [49] MDN contributors. *Cross-Origin-Embedder-Policy*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy>.
- [50] MDN contributors. *Cross-Origin-Opener-Policy*. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Opener-Policy>.
- [51] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. “CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 171–191.
- [52] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In: *S&P*. 2020.

- 
- [53] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. “SoK: The challenges, pitfalls, and perils of using hardware performance counters for security”. In: *S&P*. 2019.
- [54] Guillaume Didier and Clémentine Maurice. “Calibration Done Right: Noiseless Flush+Flush Attacks”. In: *DIMVA*. 2021.
- [55] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. “Characterizing Prefetchers using CacheObserver”. In: *SBAC-PAD*. 2022.
- [56] Craig Disselkoben, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *USENIX Security Symposium*. 2017.
- [57] Dmitry Evtuyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *MICRO*. 2016.
- [58] Dmitry Evtuyushkin, Dmitry Ponomarev, and Nael Abu-ghazaleh. “Understanding and Mitigating Covert Channels Through Branch Predictors”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.1 (2016), pp. 1–24.
- [59] Dmitry Evtuyushkin and Dmitry V. Ponomarev. “Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations”. In: *CCS*. 2016.
- [60] Dmitry Evtuyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *ASPLOS*. 2018.
- [61] Luca De Feo, Bertram Poettering, and Alessandro Sorniotti. “On the (In)Security of ElGamal in OpenPGP”. In: *CCS*. 2021.
- [62] Quentin Forcioli, Jean-Luc Danger, Clémentine Maurice, Lilian Bossuet, Florent Bruguier, Maria Mushtaq, David Novo, Loïc France, Pascal Benoit, Sylvain Guilley, and Thomas Perianin. “Virtual Platform to Analyze the Security of a System on Chip at Microarchitectural Level”. In: *EuroS&P Workshops*. 2021.
- [63] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “TRRespass: Exploiting the Many Sides of Target Row Refresh”. In: *S&P*. 2020.
- [64] W. Shen Gene and S. Craig Nelson. *MicroTLB and micro tag for reducing power in a processor*. Oct. 2006.
- [65] Daniel Genkin, Luke Valenta, and Yuval Yarom. “May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519”. In: *CCS*. 2017.
- [66] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. “ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures”. In: *NDSS*. 2020.
- [67] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks”. In: *USENIX Security Symposium*. 2018.
- [68] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *NDSS*. 2017.
- [69] Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. “AutoLock: Why Cache Attacks on ARM Are Harder Than You Think”. In: *USENIX Security Symposium*. 2017.

- 
- [70] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *International Symposium on Engineering Secure Software and Systems (ESSoS)*. 2017.
- [71] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. “Another Flip in the Wall of Rowhammer Defenses”. In: *S&P*. 2018.
- [72] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *CCS*. 2016.
- [73] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *DIMVA*. 2016.
- [74] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *DIMVA*. 2016.
- [75] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *USENIX Security Symposium*. 2015.
- [76] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *S&P*. 2011.
- [77] Richard W Hamming. “Error detecting and error correcting codes”. In: *The Bell system technical journal* (1950).
- [78] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. “Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications”. In: *MICRO*. 2021.
- [79] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. “Déjà Vu: Side-Channel Analysis of Mozilla’s NSS”. In: *CCS*. 2020.
- [80] Mehmet S Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cache Attacks Enable Bulk Key Recovery on the Cloud”. In: *CHES*. 2016.
- [81] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. “Way-predicting set-associative cache for high performance and low energy consumption”. In: *Symposium on Low Power Electronics and Design*. 1999.
- [82] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Cross processor cache attacks”. In: *ASIACCS*. 2016.
- [83] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “\$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES”. In: *S&P*. 2015.
- [84] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors”. In: *DSD*. 2015.
- [85] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. “Lucky 13 Strikes Back”. In: *ASIACCS*. 2015.
- [86] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. “”They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks”. In: *S&P*. 2022.

- 
- [87] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. “Cache Refinement Type for Side-Channel Detection of Cryptographic Software”. In: *CCS*. 2022.
- [88] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yaglikçi, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. “Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques”. In: *ISCA*. 2020.
- [89] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *ISCA*. 2014.
- [90] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. “Spectre attacks: Exploiting speculative execution”. In: *S&P*. 2019.
- [91] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *CRYPTO*. 1996.
- [92] David Kohlbrenner and Hovav Shacham. “Trusted Browsers for Uncertain Times”. In: *USENIX Security Symposium*. 2016.
- [93] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. “RAMBleed: Reading Bits in Memory Without Accessing Them”. In: *S&P*. 2020.
- [94] Sami Kyöstilä. *Clamp performance.now() to 100us*. <https://chromium-review.googlesource.com/c/chromium/src/+849993>. Jan. 2018.
- [95] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Romain Rouvoy Yossi Oren, Walter Rudametkin, and Yuval Yarom. “DrawnApart: A Device Identification Technique based on Remote GPU Fingerprinting”. In: *NDSS*. 2022.
- [96] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. “Browser Fingerprinting: A Survey”. In: *ACM Trans. Web* 14.2 (2020), 8:1–8:33.
- [97] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. “Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints”. In: *S&P*. 2016.
- [98] Lars T Hansen. *Shared memory: Side-channel information leaks*. [https://github.com/tc39/ecmascript\\_sharedmem/blob/master/issues/TimingAttack.md](https://github.com/tc39/ecmascript_sharedmem/blob/master/issues/TimingAttack.md). 2016.
- [99] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. “Nethammer: Inducing Rowhammer Faults through Network Requests”. In: *EuroS&P Workshops*. 2020.
- [100] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript”. In: *ESORICS*. 2017.
- [101] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *USENIX Security Symposium*. 2016.
- [102] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *ASIACCS*. 2020.
- [103] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *USENIX Security Symposium*. 2018.

- 
- [104] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. “Last-Level Cache Side-Channel Attacks are Practical”. In: *S&P*. 2015.
- [105] Robert Martin, John Demme, and Simha Sethumadhavan. “TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks”. In: *ISCA*. 2012.
- [106] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *RAID*. 2015.
- [107] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *NDSS*. 2017.
- [108] Mike Perry. *Bug 1517: Reduce precision of time for Javascript*. <https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517>. 2015.
- [109] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX”. In: *CT-RSA*. 2018.
- [110] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *CHES*. 2017.
- [111] Onur Mutlu and Jeremie S. Kim. “RowHammer: A Retrospective”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.8 (2020), pp. 1555–1571.
- [112] Michael Neve and Jean-Pierre Seifert. “Advances on Access-Driven Cache Attacks on AES”. In: *SAC*. 2006.
- [113] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. “DiffFuzz: differential fuzzing for side-channel analysis”. In: *ICSE*. 2019.
- [114] *nsRFPService.cpp, Firefox sourcecode*. <https://hg.mozilla.org/mozilla-central/file/tip/toolkit/components/resistfingerprinting/nsRFPService.cpp>. Oct. 2020.
- [115] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications”. In: *CCS*. 2015.
- [116] Lois Orosa, Abdullah Giray Yaglikçi, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. “A Deeper Look into RowHammer’s Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses”. In: *MICRO*. 2021.
- [117] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: the Case of AES”. In: *CT-RSA*. 2006.
- [118] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *USENIX Security Symposium*. 2021.
- [119] Dan Page. “Theoretical use of cache memory as a cryptanalytic side-channel”. In: *Cryptology ePrint Archive, Report 2002/169* (2002).
- [120] Colin Percival. “Cache missing for fun and profit”. In: *BSDCan*. 2005.
- [121] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. “Certified Side Channels”. In: *USENIX Security Symposium*. 2020.



- 
- [122] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *USENIX Security Symposium*. 2016.
- [123] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. “Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks”. In: *CCS*. 2021.
- [124] Ashay Rane, Calvin Lin, and Mohit Tiwari. “Raccoon: Closing Digital Side-Channels through Obfuscated Execution”. In: *USENIX Security Symposium*. 2015.
- [125] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. “Flip Feng Shui: Hammering a Needle in the Software Stack”. In: *USENIX Security Symposium*. 2016.
- [126] Charles Reis, Alexander Moshchuk, and Nasko Oskov. “Site Isolation: Process Separation for Web Sites within the Browser”. In: *USENIX Security Symposium*. 2019.
- [127] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?” In: *DATE*. 2017.
- [128] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. “SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript”. In: *USENIX Security Symposium*. 2021.
- [129] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. “Reverse Engineering the Stream Prefetcher for Profit”. In: *EuroS&P Workshops*. 2020.
- [130] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: *ASIACCS*. 2022.
- [131] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. “SoK: In Search of Lost Time: A Review of JavaScript’s Timers in Browsers”. In: *EuroS&P*. 2021.
- [132] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. “CPU Port Contention Without SMT”. In: *ESORICS*. 2022.
- [133] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. “The 9 Lives of Bleichenbacher’s CAT: New Cache Attacks on TLS Implementations”. In: *S&P*. 2019.
- [134] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. “Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure”. In: *CCS*. 2018.
- [135] Stephen Röttger and Artur Janc. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>. 2021.
- [136] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin K. Qureshi. “Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion”. In: *ASPLOS*. 2021.
- [137] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think”. In: *USENIX Security Symposium*. 2018.
- [138] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. “RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches”. In: *EuroSec@EuroSys*. 2017.
- [139] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. “Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features”. In: *ASIACCS*. 2018.

- 
- [140] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. “KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks”. In: *NDSS*. 2018.
- [141] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript”. In: *International Conference on Financial Cryptography and Data Security (FC)*. 2017.
- [142] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Abusing Intel SGX to conceal cache attacks”. In: *Cybersecurity* 3 (Jan. 2020).
- [143] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *DIMVA*. 2017.
- [144] Mark Seaborn. *Comment on ECMAScript Shared Memory and Atomics*. [https://github.com/tc39/ecmascript\\_sharedmem/issues/1#issuecomment-144171031](https://github.com/tc39/ecmascript_sharedmem/issues/1#issuecomment-144171031). 2015.
- [145] Mark Seaborn. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. <http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html>. Mar. 2015.
- [146] Mark Seaborn. *How physical addresses map to rows and banks in DRAM*. <http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html>. May 2015.
- [147] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. “Enforcing Fine-grained Constant-time Policies”. In: *CCS*. 2022.
- [148] Raphael Spreitzer and Thomas Plos. “Cache-Access Pattern Attack on Disaligned AES T-Tables”. In: *Constructive Side-Channel Analysis and Secure Design (COSADE)*. 2013, pp. 200–214.
- [149] Chung-ha Sung, Brandon Paulsen, and Chao Wang. “CANAL: a cache timing analysis framework via LLVM transformation”. In: *ASE*. 2018.
- [150] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Throwhammer: Rowhammer Attacks over the Network and Defenses”. In: *USENIX ATC*. 2018.
- [151] Kris Tiri, Onur Aciçmez, Michael Neve, and Flemming Andersen. “An analytical model for time-driven cache attacks”. In: *International Workshop on Fast Software Encryption*. 2007.
- [152] Eran Tromer, Dag Arne Osvik, and Adi Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. In: *Journal of Cryptology* 23.1 (July 2010), pp. 37–71.
- [153] Po-An Tsai, Andrés Sánchez, Christopher W. Fletcher, and Daniel Sánchez. “Safecracker: Leaking Secrets through Compressed Caches”. In: *ASPLOS*. 2020.
- [154] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. “Cryptanalysis of DES implemented on computers with cache”. In: *CHES*. 2003.
- [155] Vladimir Uzelac and Aleksandar Milenkovic. “Experiment flows and microbenchmarks for reverse engineering of branch predictor structures”. In: *ISPASS*. 2009.
- [156] Mathy Vanhoef and Eyal Ronen. “Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd”. In: *S&P*. 2020.
- [157] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. “Eliminating fine grained timers in Xen”. In: *ACM workshop on Cloud computing security workshop (CCSW)*. 2011.

- 
- [158] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *CCS*. 2016.
- [159] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. “Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest”. In: *S&P*. 2022.
- [160] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data”. In: *ISCA*. 2021.
- [161] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. “CacheQuery: learning replacement policies from hardware caches”. In: *PLDI*. 2020.
- [162] W3C. *High Resolution Time Level 2*. <https://www.w3.org/TR/hr-time/>. 2016.
- [163] Luke Wagner. *Mitigations landing for new class of timing attack*. <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/>. Jan. 2018.
- [164] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation”. In: *USENIX Security Symposium*. 2019.
- [165] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. “CacheD: Identifying Cache-Based Timing Channels in Production Software”. In: *USENIX Security Symposium*. 2017.
- [166] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. “Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries”. In: *RAID*. 2019.
- [167] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86”. In: *USENIX Security Symposium*. 2022.
- [168] Zhenghong Wang and Ruby B Lee. “Covert and Side Channels due to Processor Architecture”. In: *ACSAC*. 2006.
- [169] Vincent M Weaver, Dan Terpstra, and Shirley Moore. “Non-determinism and overcount on modern hardware performance counter implementations”. In: *ISPASS*. 2013.
- [170] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. “Osiris: Automated Discovery of Microarchitectural Side Channels”. In: *USENIX Security Symposium*. 2021.
- [171] 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 (FC)*. 2012.
- [172] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *ACSAC*. 2018.
- [173] Mozilla Wiki. [https://wiki.mozilla.org/Project\\_Fission](https://wiki.mozilla.org/Project_Fission).
- [174] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. “Demystifying GPU microarchitecture through microbenchmarking”. In: *ISPASS*. 2010.
- [175] John C Wray. “An analysis of covert timing channels”. In: *Journal of Computer Security* 1.3-4 (1992), pp. 219–232.

- 
- [176] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. “An exploration of L2 cache covert channels in virtualized environments”. In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW’11)*. 2011.
- [177] Abdullah Giray Yaglikçi, Haocong Luo, Geraldo F. de Oliveira, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S. Kim, Lois Orosa, and Onur Mutlu. “Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices”. In: *DSN*. 2022.
- [178] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World”. In: *S&P*. 2019.
- [179] Yuval Yarom and Katrina Falkner. “Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *USENIX Security Symposium*. 2014.
- [180] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. “Mapping the Intel Last-Level Cache”. In: *IACR Cryptol. ePrint Arch.* (2015), p. 905.
- [181] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: a timing attack on OpenSSL constant-time RSA”. In: *J. Cryptogr. Eng.* 7.2 (2017), pp. 99–112.

# Publications from October 2015 to January 2023

The following is my list of publications, since my PhD defense in October 2015. The majority have taken place either in a supporting role to a doctorate student, for example, during my postdoctorate, or directly in a supervising role since I obtained my position as a CNRS researcher.

- [1] Guillaume Didier, Clémentine Maurice, Antoine Geimer, and Walid J. Ghandour. “Characterizing Prefetchers using CacheObserver”. In: *Proceedings of the 34th International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD. IEEE, Nov. 2022.
- [2] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. “CPU Port Contention Without SMT”. In: *Proceedings of the 27th European Symposium on Research in Computer Security*. ESORICS. Springer, Sept. 2022, pp. 209–228.
- [3] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: *Proceedings of the 17th ACM ASIA Conference on Computer and Communications Security*. ASIACCS. ACM, May 2022, pp. 1182–1194.
- [4] Walid Ghandour and Clémentine Maurice. “DITTANY: Strength-Based Dynamic Information Flow Analysis Tool for x86 Binaries”. In: *Binary Analysis Research Workshop*. BAR @ NDSS. Apr. 2022.
- [5] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Romain Rouvoy Yossi Oren, Walter Rudametkin, and Yuval Yarom. “DrawnApart: A Device Identification Technique based on Remote GPU Fingerprinting”. In: *Proceedings of the Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, Apr. 2022.
- [6] Quentin Forcioli, Jean-Luc Danger, Clémentine Maurice, Lilian Bossuet, Florent Bruguier, Maria Mushtaq, David Novo, Loïc France, Pascal Benoit, Sylvain Guilley, and Thomas Perianin. “Virtual Platform to Analyze the Security of a System on Chip at Microarchitectural Level”. In: *Workshop on the Security of Software / Hardware Interfaces*. SILM @ EuroS&P. IEEE, Sept. 2021, pp. 96–102.
- [7] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. “SoK: In Search of Lost Time: A Review of JavaScript’s Timers in Browsers”. In: *Proceedings of the 6th IEEE European Symposium on Security and Privacy*. EuroS&P. IEEE, Sept. 2021.
- [8] Guillaume Didier and Clémentine Maurice. “Calibration Done Right: Noiseless Flush+Flush Attacks”. In: *Proceedings of the 18th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA. Springer, July 2021, pp. 278–298.

- 
- [9] Pierre Ayoub and Clémentine Maurice. “Reproducing Spectre Attack with gem5: How To Do It Right?” In: *Proceedings of the 14th European Workshop on Systems Security*. EuroSec @ EuroSys. ACM, Apr. 2021, pp. 15–20.
- [10] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. “Nethammer: Inducing Rowhammer Faults through Network Requests”. In: *Workshop on the Security of Software / Hardware Interfaces*. SILM @ EuroS&P. June 2020, pp. 710–719.
- [11] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. “Take A Way: Exploring the Security Implications of AMD’s Cache Way Predictors”. In: *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security*. ASIACCS. ACM, June 2020.
- [12] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. “Branch Prediction Attack on Blinded Scalar Multiplication”. In: *IEEE Transactions on Computers* 69.5 (2020), pp. 633–648.
- [13] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Abusing Intel SGX to conceal cache attacks”. In: *Cybersecurity* 3.1 (2020).
- [14] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. “Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features”. In: *Proceedings of the 13th ACM Asia Conference on Computer and Communications Security*. ASIACCS. ACM, May 2018.
- [15] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. “KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks”. In: *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, Feb. 2018.
- [16] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript”. In: *Proceedings of the 22nd European Symposium on Research in Computer Security*. ESORICS. Springer, Sept. 2017.
- [17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. “KASLR is Dead: Long Live KASLR”. In: *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems*. ESSoS. Springer, July 2017.
- [18] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. DIMVA. Springer, July 2017.
- [19] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *Proceedings of the 21st International Conference on Financial Cryptography and Data Security*. FC. Springer, Apr. 2017.
- [20] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. “Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud”. In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium*. NDSS. The Internet Society, Feb. 2017.

- 
- [21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. “Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR”. In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. CCS. ACM, Nov. 2016.
- [22] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. CCS. ACM, Nov. 2016.
- [23] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. “AR-Mageddon: Cache Attacks on Mobile Devices”. In: *Proceedings of the 25th USENIX Security Symposium*. USENIX Security. USENIX Association, Aug. 2016, pp. 549–564.
- [24] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks”. In: *Proceedings of the 25th USENIX Security Symposium*. USENIX Security. USENIX Association, Aug. 2016, pp. 565–581.
- [25] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript”. In: *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. DIMVA. Springer, July 2016, pp. 300–321.
- [26] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. “Flush+Flush: A Fast and Stealthy Cache Attack”. In: *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. DIMVA. Springer, July 2016, pp. 279–299.

# Software related to my research from October 2015 to January 2023

The following is a list of prototype software related to my research, since my PhD in October 2015. I have not written a lot of code, but I participated to their design and evaluation.

- ▶ <https://github.com/IAIK/rowhammerjs> - [73]
- ▶ [https://github.com/IAIK/flush\\_flush](https://github.com/IAIK/flush_flush) - [74]
- ▶ <https://github.com/IAIK/armageddon> - [101]
- ▶ <https://github.com/IAIK/drama> - [122]
- ▶ <https://github.com/vusec/drammer> - [158]
- ▶ <https://github.com/IAIK/prefetch> - [72]
- ▶ <https://github.com/IAIK/CJAG> - [107]
- ▶ <https://github.com/IAIK/KAISER> - [70]
- ▶ <https://github.com/IAIK/interruptjs> - [100]
- ▶ <https://github.com/IAIK/keydrown> - [140]
- ▶ <https://pierreay.github.io/reproduce-spectre-gem5/> - [20]
- ▶ <https://github.com/MIAOUS-group/calibration-done-right> - [54]
- ▶ <https://github.com/thomasrokicki/in-search-of-lost-time> - [131]
- ▶ <https://github.com/drawnpart/drawnpart> - [95]
- ▶ <https://github.com/MIAOUS-group/web-port-contention> - [130]
- ▶ <https://github.com/MIAOUS-group/port-contention-without-smt> - [132]
- ▶ <https://github.com/MIAOUS-group/CacheObserver> - [55]



# Appendices

## Experimental setups and reverse-engineering for DRAM mapping

Table 4.1: Experimental setups.

CPU / SoC	Micro-architecture	DRAM
i5-2540M	Sandy Bridge	DDR3
i5-3230M	Ivy Bridge	DDR3
i7-3630QM	Ivy Bridge	DDR3
i7-4790	Haswell	DDR3
i7-6700K	Skylake	DDR4
2x Xeon E5-2630 v3	Haswell-EP	DDR4
Samsung Exynos 5 Dual	ARMv7	LDDDR3
Samsung Exynos 7420	ARMv8-A	LPDDR4
Qualcomm Snapdragon S4 Pro	ARMv7	LPDDR2
Qualcomm Snapdragon 800	ARMv7	LPDDR3
Qualcomm Snapdragon 820	ARMv8-A	LPDDR3

Table 4.2: Reverse engineered DRAM mapping on all platforms and configurations we analyzed via physical probing or via software analysis. These tables list the bits of the physical address that are XORed. For instance, for the entry (13, 17) we have  $a_{13} \oplus a_{17}$ .

(a) DDR3

CPU	Ch.	DIMM/Ch.	BA0	BA1	BA2	Rank	DIMM	Channel
Sandy Bridge	1	1	13, 17	14, 18	15, 19	16	-	-
Sandy Bridge [146]	2	1	14, 18	15, 19	16, 20	17	-	6
Ivy Bridge/Haswell	1	1	13, 17	14, 18	16, 20	15, 19	-	-
	1	2	13, 18	14, 19	17, 21	16, 20	15	-
	2	1	14, 18	15, 19	17, 21	16, 20	-	7, 8, 9, 12, 13, 18, 19
	2	2	14, 19	15, 20	18, 22	17, 21	16	7, 8, 9, 12, 13, 18, 19

(b) DDR4

CPU	Ch.	DIMM/Ch.	BG0	BG1	BA0	BA1	Rank	CPU	Channel
Skylake <sup>†</sup>	2	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	8, 9, 12, 13, 18, 19
2x Haswell-EP (interleaved)	1	1	6, 22	19, 23	20, 24	21, 25	14	7, 17	-
	2	1	6, 23	20, 24	21, 25	22, 26	15	7, 17	8, 12, 14, 16, 18, 20, 22, 24, 26
2x Haswell-EP (non-interleaved)	1	1	6, 21	18, 22	19, 23	20, 24	13	-	-
	2	1	6, 22	19, 23	20, 24	21, 25	14	-	7, 12, 14, 16, 18, 20, 22, 24, 26

(c) LPDDR2

CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Qualcomm Snapdragon S4 Pro <sup>†</sup>	1	13	14	15	10	-

(d) LPDDR3

CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Samsung Exynos 5 Dual <sup>†</sup>	1	13	14	15	7	-
Qualcomm Snapdragon 800/820 <sup>†</sup>	1	13	14	15	10	-

(e) LPDDR4

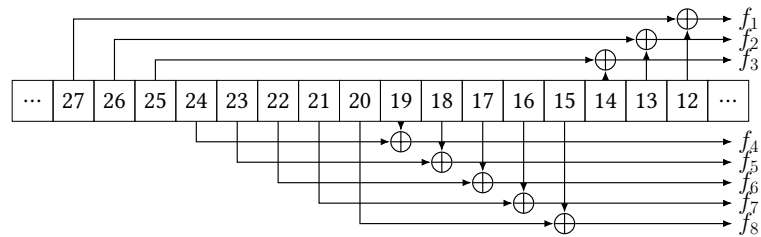
CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Samsung Exynos 7420 <sup>†</sup>	2	14	15	16	8, 13	7, 12

<sup>†</sup> Software analysis only. Labeling of functions is based on results of other platforms.

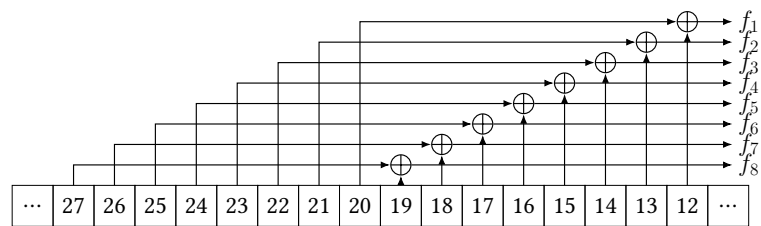
## Experimental setups and reverse-engineering for AMD cache way predictor

Table 4.3: Tested CPUs with their micro-architecture and whether they have a way predictor (WP).

Setup	CPU	Micro-architecture	WP
Lab	AMD Athlon 64 X2 3800+	K8	○
Lab	AMD Turion II Neo N40L	K10	○
Lab	AMD Phenom II X6 1055T	K10	○
Lab	AMD E-450	Bobcat	○
Lab	AMD Athlon 5350	Jaguar	○
Lab	AMD FX-4100	Bulldozer	●
Lab	AMD FX-8350	Piledriver	●
Lab	AMD A10-7870K	Steamroller	●
Lab	AMD Ryzen Threadripper 1920X	Zen	●
Lab	AMD Ryzen Threadripper 1950X	Zen	●
Lab	AMD Ryzen Threadripper 1700X	Zen	●
Lab	AMD Ryzen Threadripper 2970WX	Zen+	●
Lab	AMD Ryzen 7 3700X	Zen 2	●
Cloud	AMD EPYC 7401p	Zen	●
Cloud	AMD EPYC 7571	Zen	●



(a) Zen, Zen+, Zen 2



(b) Bulldozer, Piledriver, Steamroller

Figure 4.1: The recovered hash functions use bits 12 to 27 of the virtual address to compute the  $\mu$ Tag.