Michael Schwarz, BSc

DRAMA: Exploiting DRAM Buffers for Fun and Profit

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Prof. Stefan Mangard

Institute for Applied Information Processing and Communications

Advisor
Daniel Gruss

Graz, August 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

Cache attacks are a threat to many systems. They have shown to be a very powerful attack that can be mounted on mobile phones, computers and the cloud. This lead to research to find effective countermeasures against that kind of attacks. The central question of this thesis is whether systems are secure against information disclosure if CPUs do not leak any private information anymore.

We show that there is an attack vector outside of the CPU that is present on every computer. DRAM is commonly used as main memory in computers as well as in smartphones and servers. We highlight that the design principle of DRAM introduces a security problem which is independent of the operating system or even virtualization technologies. We developed a method to reverse engineer the DRAM address mapping that is the base for our attacks. Furthermore, we created a framework for Intel and ARM CPUs that is able to reverse engineer the address mapping fully automated within minutes.

To show the impact of this newly discovered side channel, we present attacks exploiting the address mapping. As the first attack, we demonstrate a covert channel that runs in a highly restricted environment. We show that we can utilize this cover channel to communicate from a virtual machine without network hardware to a JavaScript program running in the browser of the host system. The second attack is a keystroke logger that is able to spy on arbitrary processes without any privileges. The attacks emphasize the importance for research of countermeasures against this new kind of attack.

The content of this thesis was a major contribution to a conference paper at USENIX Security 2016 and will be presented as a talk at Black Hat Europe 2016.

Kurzfassung

Angriffe basierend auf CPU Caches sind eine Bedrohung für viele Systeme. Diese Angriffe konnten in der Vergangenheit erfolgreich auf Mobiltelefonen, Computern und Cloud-Diensten durchgeführt werden. Der Erfolg dieser Angriffe führte zu intensiven Forschungen nach Gegenmaßnahmen. Die zentrale Frage dieser Arbeit ist es, ob Systeme sicher gegen diese Art des Informationsdiebstahls sind, wenn CPUs keine privaten Informationen mehr preisgeben.

Wir zeigen, dass es außerhalb der CPU einen Angriffsvektor gibt, der auf jedem Computer vorhanden ist. DRAM wird üblicherweise als Hauptspeicher in Computern sowie in Smartphones und Servern verwendet. Wir zeigen, dass das Konstruktionsprinzip von DRAM zu einem Sicherheitsproblem führt, unabhängig von Betriebssystem oder sogar Virtualisierungstechnologie. Wir entwickelten eine Methode um die DRAM-Adresszuordnung zu rekonstruieren, welche die Basis für unsere Angriffe ist. Darüber hinaus entwickelten wir ein Framework für Intel und ARM-CPUs, welche die Adresszuordnung innerhalb weniger Minuten voll automatisiert rekonstruieren kann.

Um die Auswirkungen dieses neu entdeckten Seitenkanals zu zeigen, zeigen wir Angriffe welche diese Adresszuordnung ausnutzen. Der erste Angriff ist ein Covert Channel, der in einer stark eingeschränkten Umgebung ausgeführt wird. Wir zeigen, dass wir diesen Covert Channel nutzen können, um einen Kommunikationskanal zwischen einer virtuellen Maschine ohne Netzwerk-Hardware und einem JavaScript-Programm im Browser des Host-Systems zu etablieren. Der zweite Angriff ist ein Keylogger, der ohne Privilegien die Tastendrücke von beliebigen Prozessen auszuspionieren kann. Die Angriffe demonstrieren, dass es unabdingbar ist an Gegenmaßnahmen gegen diese neue Art von Seitenkanal zu forschen.

Der Inhalt dieser Arbeit hat zu einem großen Teil zu einer USENIX Security 2016 Publikation beigetragen und wird auch auf der Black Hat Europe 2016 vorgestellt.

Acknowledgements

First, I want to express my gratitude to my advisor Daniel Gruss for the continuous support while writing this thesis. I am very thankful for Daniel's reviews of the draft verions of this thesis and the discussions on several topics.

Second, I would like to thank my parents, Barbara and Hermann Schwarz, for supporting my fascination with computers since I was a child.

Third, I owe my thanks to Angela Promitzer for countless discussions about the thesis and reviews of draft versions of this thesis.

Finally, I want to thank Clémentine Maurice and Stefan Mangard for helpful discussions, Moritz Lipp for his ideas on the ARM port, and Anders Fogh for providing me the opportunity to be a speaker at the Black Hat Europe 2016 conference.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation 2
1.2 Structure of this document 3

2 Preliminaries **5**
2.1 DRAM 5
2.2 Cache 11

3 Reverse engineering of the addressing **15**
3.1 Basic principle 16
3.2 Timing measurement 16
3.3 Data acquisition 19
3.4 Function computation 21
3.5 ARM port 27
3.6 Results 30
3.7 Performance 33

4 Attacks **35**
4.1 Covert Channel 36
4.2 DRAM Template Attacks 51

5 Future Work **57**

6 Conclusion **59**

A Black Hat Europe 2016 **61**

Bibliography **65**

Chapter 1

Introduction

Cache attacks have been shown to be very powerful software-based side-channel attacks. As caches are present in every CPU, they are an interesting attack target. Although relatively new, there are already various techniques for cache attacks. In general, we can divide the attacks into two groups. On the one hand, there are attacks relying on shared memory, such as Flush+Reload [91] or Flush+Flush [27]. On the other hand, there are the attacks that do not require shared memory, such as Prime+Probe [81].

Cache attacks have been mounted to create covert channels [27, 59], build keyloggers [26], attack symmetric [28, 65] and asymmetric cryptography [36, 67]. Using cache template attacks, it is possible to automate those attacks to a certain degree [26]. Despite the attempts to isolate virtual machines, it has been shown that those attacks even work across virtual machines [36, 54, 94, 95].

Despite the isolation between virtual machines, it was shown that cache attacks can still be mounted in cloud environments across virtual machines [36, 54, 94, 95]. As a consequence, countermeasures were deployed. For instance, cloud providers deactivated memory deduplication. Ge et al. give an overview of the current research on finding effective countermeasures against cache attacks [23]. Disabling the sharing of CPUs by using multiple CPUs is currently the only viable countermeasure against cache attacks in the cloud.

Multiple countermeasures have already been implemented in state-of-the-art systems. Intel [51], AMD [5], and ARM [8] implemented an instruction set extension for AES. Most cryptography libraries have constant-time implementations that aim to prevent cache-based attacks. Amongst others, Microsoft has already deactivated Hyperthreading on their Azure platform [58].

Several other techniques to prevent cache attacks have already been proposed [23]. One hardware mechanism is *cache lockdown*, where the hardware provides mechanisms that allow the software to lock certain cache lines [86]. Another technique is page-coloring, where colors are assigned to physical pages. Pages with a different color do not map into the same cache set, and each process can only get pages with a

certain color [72]. However, most of the proposed mechanisms can only protect a small fragment of code and come at the cost of reduced performance.

For high-security systems, deactivating CPU caches and all forms of shared memory might be a possibility to prevent all forms of cache attacks. Due to the bad performance, this is no viable solution for consumer systems. Therefore, we expect more hardware and software countermeasures to be implemented in future systems.

In this thesis, we identify a different attack vector that does not rely on CPU caches. Since the Rowhammer attack [48], we know that DRAM is susceptible to bit flips. As it has been shown that Rowhammer can be exploited to gain root privileges [57], we started to investigate on this topic. In this thesis, we show that the main memory can be used as an alternative to CPU caches. For this, we have to reverse engineer the complex mapping of addresses to memory locations. We then demonstrate how we can mount known cache attacks using only the main memory.

The content of this thesis was a major contribution to a conference paper at USENIX Security[1] 2016 and will be presented as a talk at Black Hat Europe[2] 2016.

1.1 Motivation

We do not know whether there will be effective countermeasures against cache attacks soon. There might be no viable possibility to prevent cache attacks with the current CPU architectures. However, the motivation for this thesis is the question, whether a system is secure against microarchitectural side-channel attacks if cache attacks are not possible anymore. Nevertheless, we want to keep the effectiveness of cache attacks while circumventing security mechanisms that are there to prevent ordinary cache attacks.

Therefore, we look at another ubiquitous hardware element of computers, the main memory. Virtually every computer and smartphone use DRAM chips as main memory. Thus, it is an interesting attack target. Since the so-called Rowhammer bug, we know that stressing DRAM cells has harmful side-effects. Repeatedly accessing a row while bypassing the cache can cause bitflips. These bitflips occur in adjacent cells in neighboring rows [32, 49, 66]. The effect is even stronger when hammering on both adjacent memory rows, i.e. “sandwiching” the victim row.

To achieve this, the mapping of physical addresses to the DRAM cells has to be known. As this mapping is undocumented, we can only rely on educated guesses. The mapping could be reverse engineered using hardware probing. If you aim to do this, you need the right equipment and an exposed bus. This is not only expensive but also time-consuming, and it is infeasible for embedded systems such as mobile devices.

[1] https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[2] https://www.blackhat.com/eu-16/briefings/schedule/index.html#drama-how-your-dram-becomes-a-security-problem-4826

In this thesis, we show how to reverse engineer the DRAM mapping for any system solely in software. To achieve this task, we built a fully automated portable framework. Using the framework, we reversed the mapping for Intel and ARM CPUs with different models and microarchitectures. We did this on different setups and also verified the results by comparing it to known functions.

At first, the main goal was to improve the performance of the Rowhammer attack. However, we discovered that the buffers in DRAM could also be used as a side channel that is similar to the cache-based side channels. We found that the buffers used in DRAM show a very similar behavior as CPU caches. We exploit the timing differences of the DRAM buffers to mount attacks. Using timing differences is already known from cache-based attacks. The big advantage of DRAM attacks is that they do not require any shared memory. Moreover, in most setups, the main memory is shared between CPUs as well, meaning we can mount these attacks even in a cross-CPU scenario.

To show that this side channel works even in highly restricted environments, we demonstrate that it is possible to mount an attack in JavaScript that allows to transmit data from a virtual machine without network hardware to the browser and consequently to the internet.

These properties give us a really strong side channel that still works if countermeasures prevent cache attacks. Furthermore, we demonstrate that our reverse engineering approach works on ARM CPUs as well. We show that reverse engineering simplifies Rowhammer attacks on mobile phones. Furthermore, the reverse engineering of the DRAM mapping enabled the first Rowhammer attacks on DDR4 [55] and improved the performance of Rowhammer attacks on DDR3 significantly.

1.2 Structure of this document

Chapter 2 explains the structure of DRAM and caches as a background to understand the reverse engineering approach and the attacks based on the reverse engineering. In Chapter 3, we cover the reverse engineering method in more detail, show the setup, and present our results. Chapter 4 shows how to exploit the results of the reverse engineering by demonstrating two types of attack. Finally, Chapter 6 concludes the work and gives an outlook for future work.

Chapter 2

Preliminaries

In this chapter, we provide the background information that is required for the following chapters. We confine the overview to properties that are necessary for understanding the reverse engineering and later on the attacks. Furthermore, we introduce and define several terms that are used throughout the remainder of the thesis.

This chapter consists of two sections. The first section covers DRAM. We describe what DRAM is and how it works. Furthermore, we go into detail about the memory controller which is the connection between the CPU and the DRAM. The section concludes with an explanation of the Rowhammer attack.

In the second section, we describe the principle of caches. We show how caches work and introduce the used terminology. Finally, we show how current attacks on caches work as our attacks are based on their ideas.

2.1 DRAM

For the reverse engineering and the attacks, we first have to understand how DRAM works. DRAM stands for *dynamic random-access memory*. The main memory of a PC or mobile device is usually made of DRAM cells. The data is saved in capacitors where only one capacitor and one transistor are required per bit. This makes a DRAM cell extremely small. The capacitor is either charged or uncharged, representing a binary value. The transistor is used to access the stored value.

The DRAM cells are arranged in rows and columns as a matrix. Figure 2.1 shows a sample 16-bit DRAM matrix. Due to this arrangement, reading one cell is equivalent to reading the whole row of this cell. The memory controller opens the requested row by pulling the corresponding address line to a high level. A sense amplifier senses the signals on the column address lines. If the capacitor was charged, the column address line is high. Otherwise, the line stays low. The reading result is then stored in a latch. After reading the line, the capacitors are discharged, and the memory

controller has to refresh the row, meaning all data stored in the latch is written back to the DRAM cells. Consecutive reads from the same row can be served immediately from the latch.

Writing to a cell is equivalent to writing the complete row. The line is first opened and sensed by the sense amplifier. The sense amplifiers of the columns to be written are forced to the corresponding high or low values. This updates the capacitors' charge in the active row and therefore stores the value.

Since capacitors will slowly discharge, the memory controller has to refresh all cells periodically. The capacitance of the cell's capacitor in a state-of-the-art DRAM is 20 femtofarads [15]. Due to this low capacity, the capacitor discharges within hundreds of milliseconds to tens of seconds [45]. The JEDEC standard defines that each row has to be refreshed every 64ms or less to keep its value [46]. As the RAM loses information over time and has to be refreshed, it is described as *dynamic* [45].

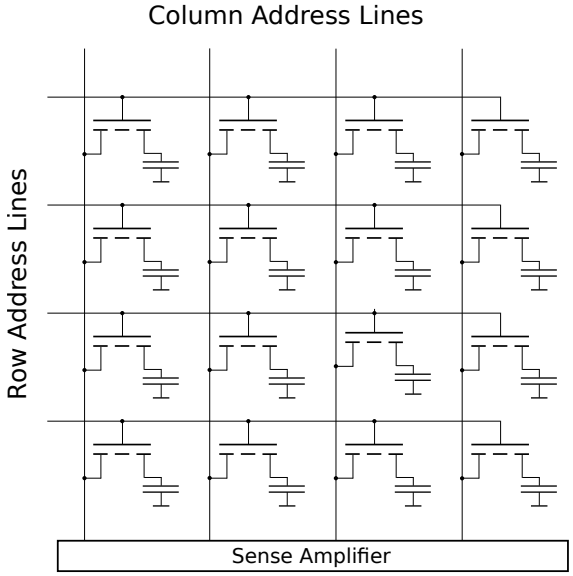


Figure 2.1: DRAM matrix structure.

In modern DRAMs, there are multiple DRAM cells. They are arranged in a strict hierarchy which we are going to introduce briefly. For desktops and servers, the memory is usually organized in DIMMs. DIMM stands for *dual-inline memory module*. Figure 2.2 illustrates a DIMM. DIMMs are connected to the mainboard through one or more *channels*. Channels are independent from each other and can be accessed in parallel which increases the bandwidth. A DIMM contains several DRAM chips. Those DRAM chips are grouped into one or more *ranks*. On most modules, each side of the DIMM corresponds to one rank. These ranks help to increase the capacity of one DIMM. Each rank is composed of multiple banks. According to JEDEC, there are 8 banks for DDR3 and 8 or 16 banks for DDR4 [46, 47]. The banks allow serving different requests simultaneously. Finally, these banks are organized in *columns* and *rows* with a typical row size of 8 KB. This structure is illustrated in Figure 2.3.

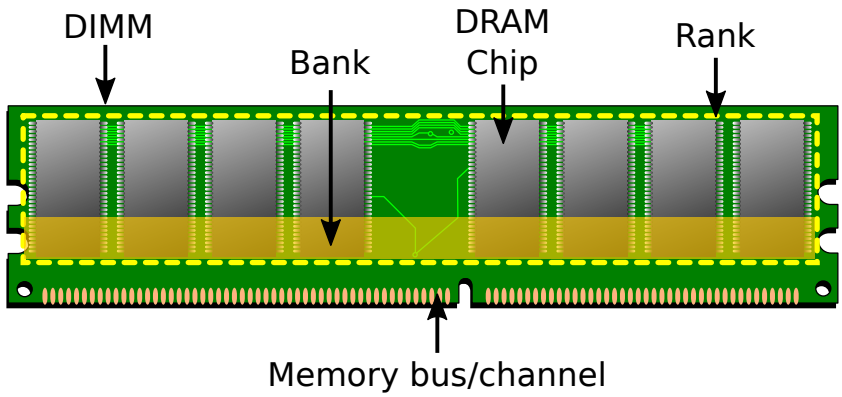


Figure 2.2: A dual-inline memory module (DIMM) containing 8 DRAM chips and the memory bus connector. The DIMM has one rank (dashed rectangle) composed of 8 DRAM chips. One of the banks is highlighted in orange.

The memory controller maps physical addresses statically to DRAM cells. As we can see, two physical addresses can only map to adjacent cells if they map to the same DIMM, channel, rank, and bank. We introduce the term *same bank* address for addresses fulfilling this property.

The mapping function that is used to map the address to DIMM, channel, rank, and bank is not publicly documented by Intel. Unlike Intel, AMD [3] discloses the mapping for their processors. Seaborn [56] reverse engineered the mapping function for Intel's Sandybridge architecture for one particular setup. However, for newer CPU models the complexity of the function increased and it is not known.

Seaborn discovered the following mapping for the Intel Sandybridge microarchitecture [56]. We denote the address bits with a_0 to a_n where a_0 is the least significant bit and a_n is the n-th bit of the physical address.

a_0 - a_2	The lower 3 bits of the byte index within a row.
a_3 - a_5	The lowest 3 bits of the column index.
a_6	This bit defines the channel.
a_7 - a_{13}	These are the upper bits of the column index, in combination with a_3 to a_5 they select the column.
a_{14} - a_{16}	Combined with the bottom 3 bits of the row number by an XOR, they give the 3-bit bank.
a_{17}	The rank bit, that selects the rank of the DIMM.
a_{18} - a_{32}	These 15 bits give the row number.

These functions are always specific to one certain configuration of the system. A different number of DIMMs, CPUs or channels can change this mapping. However, the basic principle is the same for different systems with the same CPU microarchitecture.

2.1.1 Row buffer

One part of the DIMM is the *row buffer*. This buffer is located between the DRAM cells and the memory bus. A row buffer is present in every bank which allows for bank parallelism, i.e. data from different banks can be read in parallel. Figure 2.3 shows the bank organization and the row buffer.

Due to the matrix design of DRAM, every time a value is read from the chip, the whole row has to be fetched. For this, the row is *opened* and copied to the row buffer. If data is read from an opened row, the result can be served immediately. Otherwise, if a different row needs to be accessed, the currently open row is *closed*. This means, if there is modified data in the row buffer, it has to be written back to the cells first. Then, the new row is opened and fetched. The process of closing the current row and re-opening a different row is called a *row conflict*.

A row conflict takes significantly more time than a non row conflict, as the current row needs to be written back before a new row can be fetched. It takes approximately 20ns to access data from a currently open row, whereas it takes approximately 60ns if there is a row conflict [73].

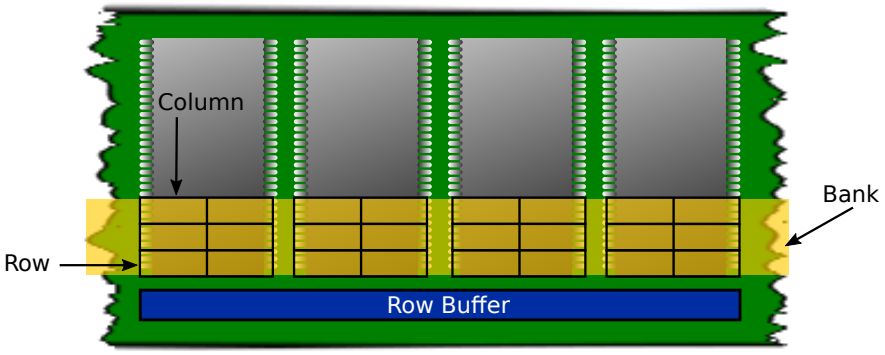


Figure 2.3: A DRAM bank organization. The bank stretches across all DRAM chips and is organized in columns and rows. The row buffer is located between the DRAM cells and the memory bus.

2.1.2 Memory controller

The memory controller is the chip that manages the data transfer from CPU to main memory and back. Since Sandy Bridge, the memory controller is not in the northbridge anymore but directly integrated into the CPU. This is called Integrated Memory Controller (IMC). Compared to the northbridge, the IMC has the advantage of increasing the performance and decreasing the latency. The controller's main task is to schedule and queue the memory accesses. Furthermore, it has to convert the memory access requests to basic DRAM commands defined by the JEDEC standard.

The memory scheduling can be described by a scheduling policy [34]. A scheduling policy is a prioritization of the memory access requests. The priority can be based on multiple metrics, such as the age, the number of row hits and row misses or type of request. The memory controller is allowed to change the execution order of the received requests, as long as it does not violate the functional correctness of the program [45].

The simplest policy is *First Come First Served* (FCFS). As the name already suggests, this strategy serves the requests in the order of arrival, i.e. the prioritization is done using the age of the request. A far more optimized version of this policy is the *First Ready, First Come First Served* (FR-FCFS) policy that tries to maximize the DRAM throughput [75]. When using the FR-FCFS policy, the memory controller tries to minimize the number of row conflicts. It attempts to reorder the memory accesses in a way, that as many requests as possible can be served from an open row. To accomplish this, requests that result in row hits get a higher priority. If no request can be served from the same row anymore, the policy falls back to FCFS. Mutlu et al. [63] developed a new scheduling policy for multicore systems that can be even faster than FR-FCFS. Their *parallelism-aware batch scheduler* (PAR-BS) is aware of bank parallelism and creates batches of requests that increase the fairness and prevent starvation.

2.1.3 Rowhammer

One bug that is closely related to DRAM organization is the *rowhammer* bug [70]. It is a DRAM disturbance error, where repeated row activation can cause bit flips in adjacent rows [48]. Due to inter-cell couplings, these repeated row accesses cause accelerated leakage in adjacent rows. If the capacitor is uncharged before the row is refreshed, the bit has flipped. The rowhammer bug exists in most DRAM modules, and it is a side-effect of the increasing chip density that results in decreasing cell sizes and therefore decreasing capacitance. Moreover, due to the smaller structures, the cells are also located closer to each other. This also increases the likelihood that a cell affects a neighboring cell [70].

The rowhammer bug was first only seen as a reliability issue that can cause data corruption. Later, it has been shown that it is also a security issue, which can be exploited to gain root privileges [57]. The idea is to fill the memory with a data structure, which yields higher privileges if bitflips occur inside this structure [24]. Seaborn et al.[57] sprayed the memory with page tables. If a bit flip happens in a page-table entry, it is likely that the page-table entry points to another page table. With control over a page table, an attacker can map arbitrary memory locations with read and write access.

Figure 2.4 lists the original code to trigger the rowhammer bug. From this code we see, that there are two requirements. First, we require the `clflush` instruction to bypass the CPU cache. Second, the two addresses `X` and `Y` have to be in the

```
code1a:
  mov (X), %eax
  mov (Y), %eax
  clflush (X)
  clflush (Y)
  mfence
  jmp code1a
```

Figure 2.4: The original rowhammer code [48]

same bank but in a different row. Intel does not disclose which bits of the physical address are used to calculate the bank, whereas AMD discloses this information [4]. Therefore, on Intel CPUs, we have to pick addresses randomly. One improvement to the rowhammer code is the *double-sided* rowhammer [57]. There, both adjacent rows of the victim row are hammered to increase the likelihood of bit flips. This method has the problem that the mapping of physical addresses to DRAM positions has to be known. If a system with DDR4 memory has 16 banks, 2 ranks and 2 channels, this results in a probability of only 1/64 that two addresses are in the same bank.

2.2 Cache

Caches were introduced as the increase in memory speed could not keep up with the increasing speed of CPUs. Caches are fast, small memories that keep copies of data from the frequently used main memory locations. Whether this data is kept in the cache is based on the observations, that the same data is usually not accessed once but several times. Caches itself are transparent. That means, they work independently from the software and have their own heuristics what data to hold in the cache [45].

Caches are arranged in a hierarchy that consists of multiple cache levels. We refer to these levels as L1, L2, and L3, where L1 is the fastest but also smallest cache. The cache level furthest away from the CPU, i.e. the largest cache is called *last-level cache* (LLC). These cache levels can be *inclusive*, *non-inclusive* or *exclusive*. For inclusive cache levels, all data that resides in cache levels closer to the CPU must also be in this cache level. Exclusive caches guarantee that the data resides at most in one cache level. On Intel CPUs, the L2 cache is non-inclusive [42]. The non-inclusive L2 cache might contain data that is also included in the L1 cache but in contrast to inclusive cache levels this is not required. The last-level cache is inclusive of all cache levels nearer to the CPU, i.e. all data in the L1 and L2 cache is also in the LLC [42]. Furthermore, on multi-core systems, the LLC is the only cache that is shared among CPU cores. All other caches are per core.

A cache is divided into *sets*. Each set consists of multiple *ways*. The actual data is *tagged* with a tag and stored in one way of a set. The set is determined by the cache index. The tag decides whether a way contains the data that is requested. To determine the cache index that is used to store an address, either the physical, virtual or both addresses are used. The LLC of the Intel Core i7 is physically indexed and physically tagged (PIPT) [31], meaning that the location in the cache depends only on the physical address. Therefore, the virtual address has to be resolved to the physical address.

The access time for data in the cache is significantly lower than for data that has to be fetched from the memory. For the Intel Core i7, Intel provides the following estimates [38]:

- *L1 Cache Hit*: $\approx$ 4 cycles
- *L2 Cache Hit*: $\approx$ 10 cycles
- *L3 Cache Hit*: $\approx$ 40 cycles
- *L3 Cache Hit in another core*: $\approx$ 65 cycles
- *DRAM*: $\approx$ 60ns

From these latency estimates, we can see that a cache hit is always several times faster than a memory access. However, as the caches are small in comparison to the main memory, not all data can be held in the cache. The CPU has to implement

a *cache replacement policy* that decides which data will be kept in the cache and which data is replaced by new data.

The cache replacement policy depends on the CPU. Older CPUs used a Least-recently used (LRU) or the more efficient Pseudo-LRU (PLRU) [80]. With this policy, every cache line has an age and new entries replace the oldest entries. Intel used PLRU with a three-bit age in older CPUs [29]. Newer Intel CPUs use more complex adaptive policies, such as the Bimodal Insertion Policy (BIP) [71] or Quad-Age LRU [76]. In contrast to these complex policies, ARM uses a random replacement policy for reasons of simplicity [7]. In this policy, the data to be evicted is selected pseudo-randomly.

2.2.1 Slice Functions

With the growing number of CPU cores, the pressure on the caches increased. As a shared cache can introduce access latencies when multiple cores want to access it, the LLC became a bottleneck [44]. Therefore, the last-level cache was split into *slices*. Each of the slices belongs to one core and operates as a normal cache. These slices can be accessed in parallel to minimize latency.

To determine which slice is used by a particular physical address, Intel uses an undocumented hash function. We refer to this hash function as *slice function*. It takes an address as input and returns the corresponding slice number. The slice function should be fast as it is used for every memory access. Furthermore, it has to distribute the addresses as uniformly as possible to the individual slices to balance the workload of the cache slices. The reverse engineering of these hash functions has been studied lately [33, 36, 44, 54, 59].

Figure 2.5 shows the cache design of an Intel CPU with four cores. The input to the slice function are the most significant bits of the physical address. It outputs a two-bit slice index that decides in which of the four slices the address is cached.

2.2.2 Cache Attacks

As caches are integrated into every modern CPU, they are an attractive target for attacks. The main idea of software-based cache attacks is to exploit the timing differences between cached and non-cached data. This timing side channel is then used to spy on a victim process by deducing which data was used by the victim process. Those attacks have been shown to be very powerful [26, 27, 28, 36, 54, 81, 91].

We can divide cache attacks into two groups. One the one hand, there are the attacks which rely on shared memory, such as *Flush+Reload*. On the other hand, we have the attacks which do not require any form of shared memory, such as *Prime+Probe*. Furthermore, we explain how cache attacks can be automated. For this, we briefly introduce cache template attacks [26].

Flush+Reload Memory which is shared between attacker and victim is only cached once. This means, if the victim loads a part of a shared library into the cache, this code is also cached for the attacker. Flush+Reload relies on this property. Moreover, if the attacker flushes code of the shared library from the cache, it is also flushed for the victim.

The idea of the attack is to flush interesting parts of the shared library continuously from the cache and measure the access time afterwards. If the victim has accessed the code between flushing and accessing it, the code is again in the cache. This leads to a low access time. In contrast, if the victim did not access the code, the attacker experiences a higher access time, as the code is not cached and has to be loaded from the memory. The timing difference exposes whether the victim has accessed the flushed part of the shared code or not.

There also exist variations of this attack, such as Flush+Flush [27] or Evict+Reload [53]. However, the basic principle stays the same.

Prime+Probe The Prime+Probe attack does not require shared memory. However, the cache replacement policy has to be known for this attack. The basic principle is again based on timing differences between cached and non-cached data.

This attack works the opposite way as Flush+Reload. The idea is first to fill the cache with data. This step is called *prime*. Whenever the victim accesses data, the cache has to evict some other data from the cache. As the attacker filled the cache with its data before, the cache evicts data of the attacker. In the second step, the attacker measures the access times for the supposedly cached memory. If the access time increased for an address, this address has been evicted from the cache due to the victim accessing data.

Cache template attacks Gruss et al. [26] introduced a method to automate cache attacks. When mounting cache attacks, the attacker has to find instructions that depend on the secret information. Finding an address offset in a binary which is suitable for a cache attack is time consuming and requires knowledge of the attacked program. The idea of the cache template attacks is to automate this process. The attacker simulates the event that should later be attacked. Then, the shared library is scanned for addresses that have cache hits. The addresses found are a starting point to find a suitable address to attack.

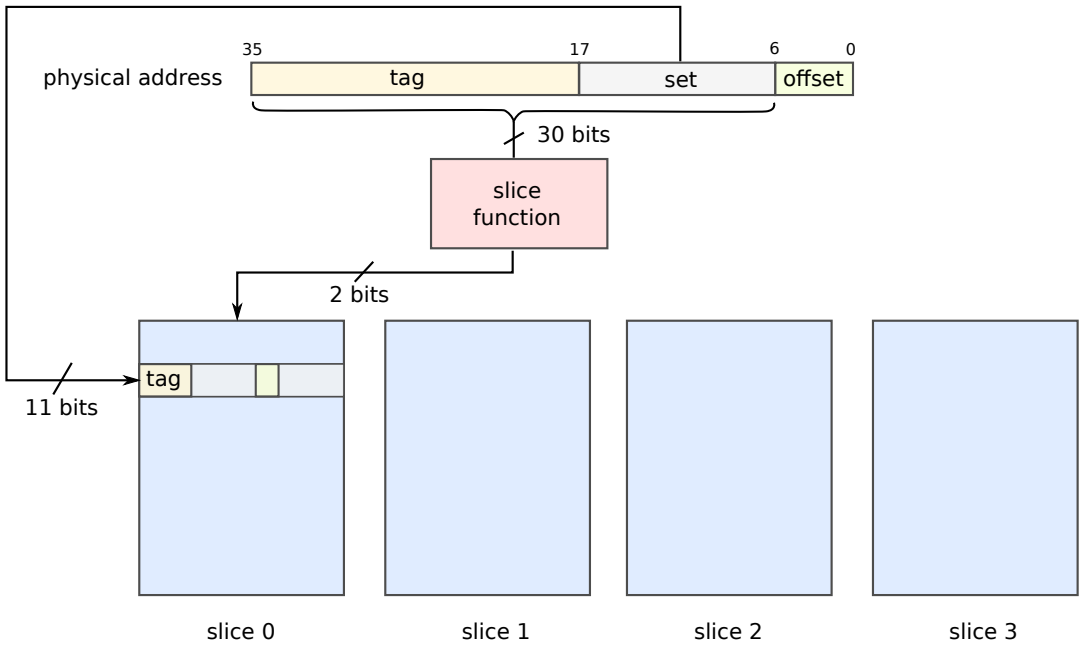


Figure 2.5: An example for a Intel CPU with 36-bit physical addresses. The physical address is divided into three parts, the tag, set and offset. The slice function uses the physical address as input and outputs the corresponding cache slice. Bits 6 to 16 select the set inside the cache slice, and the tag selects the way of the set to read.

Chapter 3

Reverse engineering of the addressing

In this chapter, we explain how to reverse engineer the DRAM addressing functions. The CPU uses these functions to map physical addresses to actual DRAM locations. We do this reverse engineering solely in software. Our method works fully automated by exploiting timing differences of DRAM accesses.

First, we discuss the basic principle of our technique. We explain how it is possible to exploit the timing differences introduced by the DRAM row buffer to reverse engineer the DRAM addressing function. Furthermore, we show which prerequisites are required for the reverse engineering and how we can fulfill them on the Intel platform.

Second, the measurement method is explained. We show how to actually measure the timing differences of row conflicts and non row conflicts. Additionally, we discuss how the memory controller might destroy our measurements and how we can circumvent this.

Third, we present the calculation of the function by using our measurements. We show how to calculate possible functions from the measurements. Moreover, we also develop a method to reduce the number of functions. This optimization step reduces the number of functions to the smallest subset of linear independent functions. It also prefers functions with a low number of coefficients.

Finally, we demonstrate that our attack mechanism also works on ARM-based CPUs. We show how to fulfill the prerequisites on ARM CPUs. We compare the results obtained from Intel and ARM CPUs and highlight the similarities and the differences of those platforms.

3.1 Basic principle

To reverse engineer the mapping function we exploit the fact that DRAM always follows the structure described in Section 2.1. Due to this organization, access times vary depending on where the actual data is located in the hardware. We are primarily interested in row conflicts. As already described, they need the most time to complete and therefore are distinguishable from all other memory accesses. Using this difference, we can identify sets of physical addresses with one common property. Every address in this set is mapping to the same DIMM, channel, bank, and rank but to a different row. We can then find functions, which yield the same result for every address in one set. These functions are then the addressing functions for DIMM, channel, bank and rank.

The number of those sets of physical addresses depend on the actual hardware organization of the DIMM, i.e. the number of banks, ranks and channels. This information can be easily retrieved using system tools such as `decode-dimms`.

The functions for the Sandy Bridge microarchitecture are a starting point for our method [56]. We expect that the basic principle of the addressing functions did not change drastically on newer microarchitectures. This assumption allows us to represent the identified physical addresses and the corresponding output as a linear equation system. Solving this equation system gives us one addressing function for DIMM, channel, bank and rank respectively.

3.2 Timing measurement

For measuring the timing differences of a row conflict and a non row conflict, we need timers with a high precision. All x86 processors since the Pentium provide the *TSC* (Time Stamp Counter) register. This 64bit register contains the number of CPU cycles since reset. This gives us the highest possible precision. The register can be read using the `rdtsc` assembly instruction [41].

There are a few crucial things to consider when measuring such small timings. Intel gives a thorough guidance on what to take into account when benchmarking [39]. For our measurement, we have to take four aspects into account.

First, we have to take into account that DRAM cells are not only cached by the row buffer but also by the CPU cache. Every access to a memory location gets cached, and further accesses are served from the cache. In our case, we have only two memory locations that are repeatedly accessed. Apart from the first access, the data will always be served from the cache. Therefore, our first priority is to bypass the CPU cache. Otherwise, we would only measure the cache access times. On x86 processors, this can be achieved using the `clflush` assembly instruction. According to Intel [40], this command invalidates every cache's cache line that contains the specified memory

address. We have to flush the cache after each measurement to force the CPU to fetch the data from the DRAM.

Second, the CPU is allowed to re-order our instructions. We need to prevent this so-called out-of-order execution to guarantee a noise-free correct result. For this purpose, we use the serializing instruction `cpuid`. Newer CPUs provide the `rdtscp` instruction which is a "pseudo" serializing version of `rdtsc` [39]. However, it only guarantees that the previous instructions have been executed, but code after the instruction can be executed before or while `rdtscp` is executed. The `cpuid` instruction guarantees, that all code up to this instruction is executed before the following instructions are executed [39]. Listing 3.2.1 shows the code. We do not have to correct any time overhead for the function call, the `cpuid` or the `rdtsc` instruction itself, as this overhead is nearly constant for all measurements.

Third, as the program runs in userspace, the scheduler can interrupt it at any time. This would destroy our timing completely. As the measurement of the access time takes a significant amount of time, it is not unlikely, that the scheduler interrupts our measurement. One possibility to deactivate the scheduler would be to implement the measurement in the kernel space. This would, however, require to create a kernel module that reduces the portability of the tool. Although there is not a guaranteed way to block the scheduler in userspace, we can minimize the probability of getting re-scheduled. By yielding several times before doing the measurement, it is unlikely that the scheduler interrupts us while doing the actual timing analysis.

Finally, we need to disallow the compiler to re-order instructions and optimize memory accesses. The optimizer could remove the whole measurement loop, as we only read from memory locations without using the results any further. As a normal memory read has no side-effects, the compiler is allowed to remove the whole loop. We prevent this by declaring all our pointers and inline assembly instructions `volatile`. This tells the compiler, that the value of the variable can change, even though there is no code that changes the variable. Therefore, the compiler cannot remove or optimize the memory access.

3.2.1 Memory controller scheduling

On some machines, considering all those things still leads to noisy measurements. One explanation for this behavior is the memory controller scheduling. The memory controller is allowed to reorder the memory accesses for performance reasons as long as the correctness of the program is still given [45]. If we encounter such memory request scheduling, we can disturb the scheduling by inserting `nop` instructions after the memory accesses. These instructions do not cause any significant overhead, as the `nop` instruction is executed in one cycle and has no side effects [42].

However, on some machines this is still not enough to prevent the memory controller from optimizing the access pattern. If this is the case, we can apply another trick. By interleaving our memory accesses with random accesses, we give the memory

controller scheduler a hard time optimizing the access pattern. One disadvantage with this method is that we add a lot of noise to the measurement. The access time does not depend only on the addresses we want to analyze, but also on the random address. Still, as the random addresses change for every measurement, the noise is distributed well over the whole measurement if we repeat the measure loop many times. Although the results are more imprecise, the results are good enough for the reverse engineering process.

Listing 3.2.1 shows the complete code to measure the access time. To further reduce noise, we perform the measurement multiple times and take the average timing. Using this code, we are able to accurately measure the timing of accessing two memory locations alternately.

```
uint64_t rdtsc() {
  uint64_t a, d;
  asm volatile ("xor %%rax, %%rax ::: "rax");
  asm volatile ("cpuid" ::: "rax","rbx","rcx","rdx");
  asm volatile ("rdtsc" : "=a" (a), "=d" (d) : : );
  a = (d << 32) | a;
  return a;
}
```

Listing 3.1: Timestamp counter with serializing

```
uint64_t get_timing(volatile void* addr1, volatile void* addr2, int num_reads) {
  for (int yield = 0; yield < 10; yield++)
    sched_yield();

  uint64_t start = rdtsc();
  int number_of_reads = num_reads;
  while (number_of_reads-- > 0) {
    *addr1;
#ifdef DRAM_SCHEDULING
    asm volatile("nop\n" "nop\n" "nop\n" "nop\n");
#elseif DRAM_SCHEDULING_V2
    *(addr1 + number_of_reads);
#endif
    *addr2;
#ifdef DRAM_SCHEDULING
    asm volatile("nop\n" "nop\n" "nop\n" "nop\n");
#elseif DRAM_SCHEDULING_V2
    *(addr2 + number_of_reads);
#endif
    asm volatile("clflush (%0)" : : "r" (addr1) : "memory");
    asm volatile("clflush (%0)" : : "r" (addr2) : "memory");
  }
  uint64_t res = (rdtsc() - start) / (num_reads);
  return res;
}
```

Listing 3.2: Timing measurement code

3.3 Data acquisition

Using the setup from Section 3.2, we are now able to create a timing histogram. For the histogram, we always take two addresses and measure the average time it takes to access them alternately.

At first, we need a large pool of physical addresses. We `mmap` a large piece of memory and calculate the physical addresses using the page map. The minimum amount of memory we have to map depends on the hardware configuration of the machine. The memory block must be distributed over all DIMMs, i.e. every DIMM provides some addresses for the memory segment. This ensures us that we do not miss any bit that is used for the addressing function. Moreover, we also have an upper limit of memory that we have to take into account. The remaining system needs enough free memory to not start swapping. Swapping would lead to wrong measurements, as swap-in requests are slow. Additionally, the mapping from virtual address to physical address can change when swapping the memory. On most systems with two DIMMs of the same capacity, we get good results with a memory percentage of 60% to 75%. Figure 3.1 shows the relation between mapped memory and the average percentage of functions that could be reverse engineered.

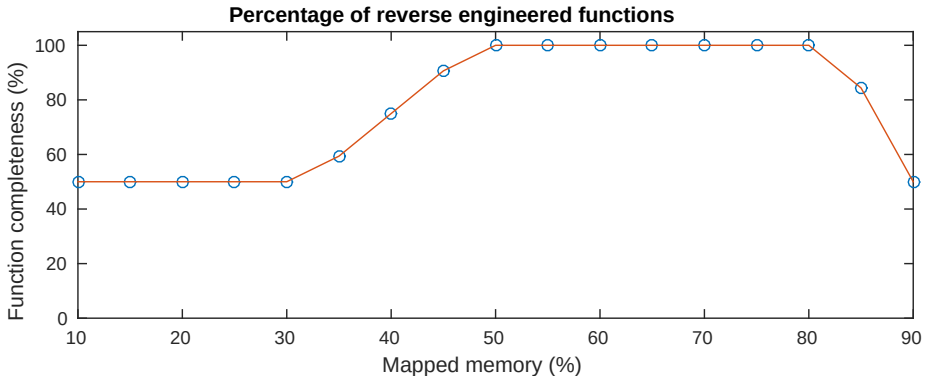


Figure 3.1: Measurement on an i7-3630QM (Ivy Bridge) with 8GB Hynix DDR3 HMT351S6CFR8C-PB. It shows that the best results, i.e. a recovery of all functions was obtained when mapping between 50% and 80% of main memory. If we used more memory, the operating system started to swap or kill the process.

From the mapped memory, we choose a random subset of addresses that form our address pool. The number of addresses for the address pool is a trade-off between accuracy and speed. The more addresses we have in the pool, the longer it takes to measure the access time for all of them. However, more measurements reduce the noise and therefore decrease the chance of errors. In our empirical tests, depending on the hardware, values between 1000 and 5000 were good choices as long as the addresses are uniformly distributed over the whole mapped memory segment. As a rule of thumb, we want to have around 100 addresses per set. The number of sets is given by the number of DIMMs, ranks, channels and banks. For each of these variables we can choose one possible value. We can write this as

$$sets = \binom{DIMMs}{1} \times \binom{channels}{1} \times \binom{ranks}{1} \times \binom{banks}{1}.$$

We can simplify the equation to

$$sets = DIMMs \times channels \times ranks \times banks.$$

For example, if we have two DIMMs, two ranks, one channel and eight banks, we get a total of 32 different sets. As we expect the addresses to be uniformly distributed over the memory, we have to choose at least 100 times the number of sets to have around 100 addresses per set. Due to measurement errors we should increase the value slightly. Furthermore, as we cannot map the whole memory, the addresses are not exactly distributed uniformly. In our tests it turned out, that choosing a pool size of $125 \times sets$ was reliable.

First, we choose one random address from our address pool. We refer to this as the *base address*. We do not know to which set the base address belongs, but we know that there will be around 100 addresses that belong to the same set. For every other address in the pool, we calculate the timing when accessing this address and the base address alternately. This leads to a histogram as shown in Figure 3.2.

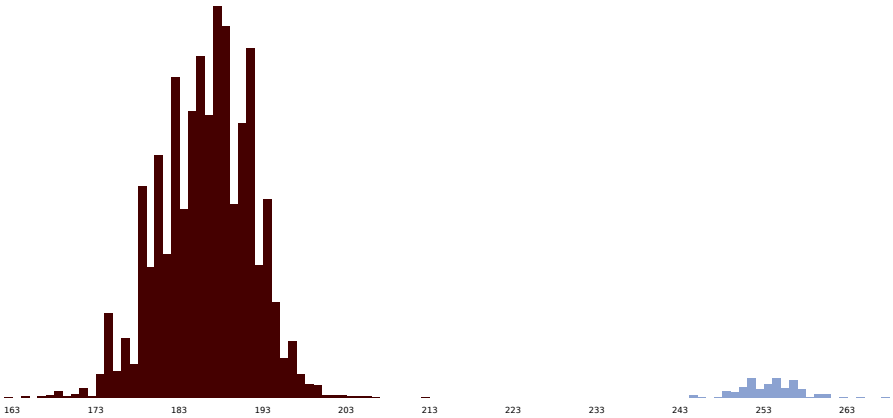


Figure 3.2: Measurement on an i7-3630QM (Ivy Bridge) with 8GB Hynix DDR3 HMT351S6CFR8C-PB. The red bars on the left are the non row conflicts, the blue parts on the right are row conflicts. The x-axis shows the number of cycles for one alternating access of two addresses.

In this histogram, the difference between row conflicts and non row conflicts can be seen clearly. A row conflict takes significantly longer than a non row conflict. We see that the span between the fastest and slowest access time is approximately 100 CPU cycles. Considering a CPU frequency of 2.4GHz, this equals 40ns which matches the specifications described in Section 2.1.1. The normal distribution for the left peak is caused by the different physical locations of the addresses. There is the rank, channel

and bank parallelism that leads to slightly different access times. Combined with the measuring inaccuracy we get this normal distribution.

We are only interested in the addresses that correspond to the right part of the histogram. These are the addresses that have a row conflict with our base address. For these addresses, we know that they are in the same DIMM, channel, rank and bank. All the addresses causing a row conflict form one set. Therefore, we split the histogram into two parts so that both parts contain one peak. The separation should be as close to the right peak as possible. The reason for this is that we do not want to have non row conflicts identified as row conflicts. Having such addresses in the set would lead to a problem when solving the equation system. With one wrong equation we do not get a solution anymore, as this equation will not satisfy the correct function. However, it is acceptable to miss some row conflicts as this just makes our set smaller.

After we successfully identified a set, we remove all of its addresses from the address pool. Ideally, there is no remaining address in the pool that belongs to the same set as the one just identified. In practice, due to a conservative decision of which address belongs to the set, there will still be a few addresses left in the pool. As long as we do not choose such an address as a new base address, this does not matter at all. Although the probability that we choose such an address as base address is small, it might happen. However, we can easily detect that case. If the number of addresses that have a row conflict with our base address is much smaller than the expected number, we can discard the measurements and choose a new base address.

We repeat this procedure until all sets are identified, no new set is found, or the address pool is empty. Due to the randomness of the address choosing, it is not always possible to identify all sets. There are multiple reasons why we do not identify all sets. Either, we chose the size of the mapped memory segment too small or the number of addresses in the address pool too low. Or, the memory controller reschedules our accesses, leading to wrong measurement as it can be seen in Figure 3.3. It might also be the case that there is too much background noise on the machine.

However, to compute the addressing function, we will see that it is not necessary to find all sets. If we identify only a subset of all sets, we are still able to solve the equation system. Still, we might get additional spurious addressing functions. These are functions that are valid for the found sets but they would not be valid for the remaining, undiscovered sets. In this case, we can run the program multiple times and take the intersection of the identified functions.

3.4 Function computation

We now have a number of sets where each set has the property that all addresses in that set are same bank addresses. From the work of Seaborn [56], we know that the function implemented in Sandy Bridge is a linear function using an XOR of address

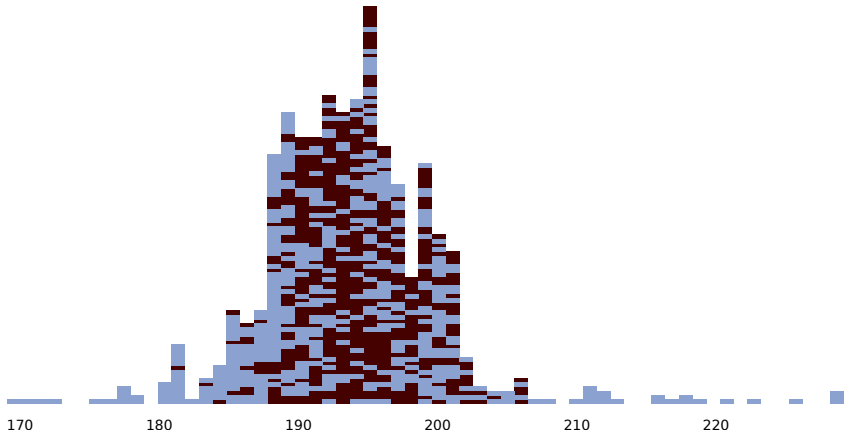


Figure 3.3: Measurement with active DRAM memory controller scheduling. The memory controller reduces the number of row conflicts and increases the number of row hits. We see one peak where the sets are not distinguishable anymore. Only a few row misses can be seen on the right, but they are not clearly separated from the peak.

bits. A linear function makes sense for various reasons. One important aspect is that it is extremely fast to compute, both in software and in hardware. As the function has to be evaluated for every DRAM access, there should be no performance impact from the addressing function. Additionally, XOR gives us a uniform distribution of the output if the inputs are uniformly distributed. We can combine as many input bits as necessary and as long as the input bits are uniformly distributed the output is too. Linear XOR functions are also used for caches [33, 36, 44, 54, 59] and interleaved multibank memory [17].

We assume that the addressing functions are also linear for newer CPU models. Especially, if the number of sets is a power of two, the XOR functions give a uniform address distribution over all sets. Under this assumption, we can represent the addresses and their corresponding set as a linear equation system. As the number of equations equals the number of addresses, it is rather small. Moreover, the maximum number of coefficients is the number of bits of an physical address. These limitations allow us to find the solution very quickly without having to solve the equation system over the Galois field GF(2). Instead of systematically solving the equations, we generate every possible function and verify whether it yields the same result for each address in the set.

The upper limit of coefficients is determined by the number of bits of the highest physical address. This, again, depends on the amount of memory installed in the machine and the I/O mapping of the machine. Usually, the physical memory does not have a contiguous address space. Figure 3.4 shows a possible memory layout of a computer with a 64-bit Intel CPU and 8GB memory. On Linux, the exact memory map is provided in `/proc/iomem`. However, we can use an optimistic guess for the highest physical address, as the remaining memory mapped segments are

small compared to the main memory. Usually, we can use $\lceil \log_2(\text{memory size}) \rceil$ for the maximum number of bits that are used by physical addresses. The number of bits can be reduced further. The first 6 bits of an address are the lower 6 bits of the byte index in the row and used for addressing within the cache [56]. Therefore, we have at most $\lceil \log_2(\text{memory size}) \rceil - 6$ bits that can be used for the addressing function. For example, on a 64-bit system with 8GB memory we have a maximum of 27 bits that can be used.

Starting from $n = 1$ to $n = \lceil \log_2(\text{memory size}) - 6 \rceil$ we generate all bitmasks with exactly n bits set. Algorithm 3.1 provides an algorithm to calculate the next higher number after a given number with the same number of 1-bits set [87].

Algorithm 3.1 Next higher number with same number of bits set

function NEXTHIGHER(x)
  $s \leftarrow x \;\&\; -x$
  $r \leftarrow s + x$
  $y \leftarrow r \mid (((x \oplus r) \gg 2) \div s)$
  return y
end function

We start with the bitmask $(1 << n) - 1$ and use the function to generate all bitmasks iteratively. The 1-bits in the bitmask determine which bits of the address to combine using XOR.

Then, we apply these bitmasks to every address in a set. Applying a bitmask to an address is again very efficient. We combine the bitmask and the address with a logical AND and take the bitcount modulo 2. The bitcount is usually a assembler instruction that is provided by the `gcc` built-in command `__builtin_popcountl` [22]. If the applied bitmask yields the same result for every address in the set, we save it as *function candidate*. Here we see why it is important to be sure that non row conflict addresses are not part of the set. Those addresses would not yield the same result. Therefore, the function, although correct, would be discarded.

After we iterated through all bitmasks, we have several candidate functions that are valid for one set at a time. Each function gives us exactly one output bit that is the combination of several input bits. Due to the uniform distribution of the random bits, the resulting bit follows a uniform distribution too. We can use this property of the XOR function to further reduce the set of candidate functions. When we apply all potential functions to the addresses from every set, we expect the result to be equally distributed, i.e. the function yields 0 for half of the sets and 1 for the other half. If we have identified all sets, we have exactly this distribution. Otherwise, this might deviate slightly, but not much. If the result deviates too much from a uniform distribution, the function is removed from the candidate functions.

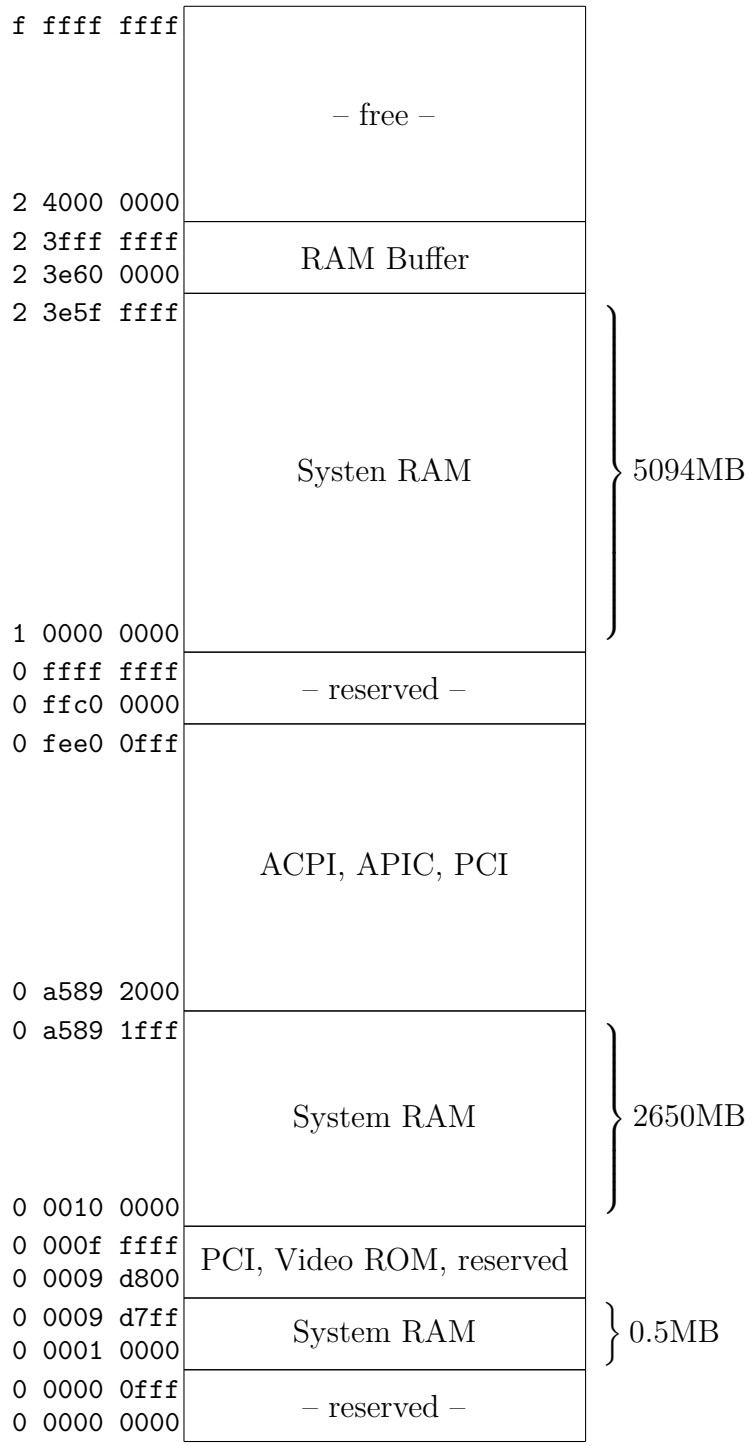


Figure 3.4: A sample physical memory layout of a 64bit machine with 8GB memory.

3.4.1 Function optimization

We now have several candidate functions left. From the remaining function candidates, we prefer functions with a small number of terms. To achieve this, we remove all functions that are linear combinations of other functions. To do this, we represent the candidate functions as a matrix, where each function is one row of the matrix. The columns of the matrix represent the bits used in the XOR function and contain either a 1 if the bit is used or a 0 if it is not. In each row, the highest coefficient is in the leftmost column, the lowest coefficient in the rightmost column. For example, if we have the functions $f_1(a) = a_7$, $f_2(a) = a_9$, and $f_3(a) = a_7 \oplus a_9$, we would represent them as the matrix

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccc} 9 & 8 & 7 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array}. \tag{3.1}$$

The *rank* of the matrix gives us the number of linearly independent rows. This is equal to the number of addressing functions to which we can reduce the recovered functions. Using linear algebra, we can eliminate the functions that do not contribute to the rank of the matrix. We can use the Gauss-Jordan elimination to compute the reduced row echelon form. With this algorithm, we also get the *column rank* of the matrix. The column rank is, however, the same as the rank of the transposed matrix which is in turn the same as the rank of the matrix [79]. To get the rows that are linearly independent, we can therefore apply the Gauss-Jordan elimination to the transposed matrix. The columns that contain the leading variables in the reduced row echelon form correspond to the linearly independent columns in the matrix [21]. Applying Gauss-Jordan elimination to the transposed matrix from Equation (3.1) gives us

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix}^T \rightarrow \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \xrightarrow{Gauss-Jordan} \begin{pmatrix} \mathbf{1} & 0 & 1 \\ 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.2}$$

The columns with the leading variables shown in bold are 1 and 2. These are the rows of the original matrix that are linearly independent. In this example, these are rows 1 and 2 which correspond to the functions $f_1(a) = a_7$ and $f_2(a) = a_9$. The function $f_3(a) = a_7 \oplus a_9$ can be represented as linear combination of f_1 and f_2. Note that all operations have to be performed in GF(2) as we only work with bits. Listing 3.2 shows the pseudo-code of this algorithm.

After removing the linearly dependent functions, we are left with a much smaller number of functions. Ideally, these functions contain only the real DRAM addressing functions. However, depending on the random address selection, we get some additional, although incorrect, functions. This is the case if we did not discover all sets in our measurement phase. The reason for this is that we either guessed a two small number of sets or we did not get enough addresses in a set. In both cases, we are missing some sets for which the computed functions might be wrong. To

Algorithm 3.2 Gauss-Jordan elimination to get the linearly independent columns of a matrix over GF(2)

```
function LinearlyIndependentColumns(matrix, rows, cols)
    row ← 0
    col ← 0
    idx ← []
    while row < rows and col < cols do
        (val, index) = max(matrix[*][col])    ▷ Maximum value of current column
        if val == 0 then
            col ← col + 1                              ▷ Column is zero, goto next
        else
            idx.append(col)                            ▷ Remember column index
                                                ▷ Swap current with maximum row
            (matrix[row], matrix[index]) ← (matrix[index], matrix[row])
                                ▷ Subtract multiple of pivot row from other rows
            for r ∈ {[0, rows] \ row} do
                pivot ← matrix[r][col]
                for c ∈ [col, cols] do
                    matrix[r][c] ← matrix[r][c] ⊕ (pivot ∧ matrix[row][c])
                end for
            end for
            row ← row + 1
            col ← col + 1
        end if
    end while
    return idx
end function
```

eliminate these spurious functions, we can run the measurement and identification phase multiple times. The intersection of the functions from all runs will result in only correct functions with a high probability.

One limitation of this software-based reverse engineering is that we cannot infer the labeling of the functions. In other words, we will get the functions for DIMM, channel, rank and bank, but we do not know which function is used for what. We can, however, infer this function labeling from known functions of similar systems [56].

3.5 ARM port

Due to the ball-grid array structure of mobile devices, we cannot use hardware to analyze the memory bus on ARM platforms. This makes it a very interesting target for our software approach on reverse engineering the address mapping. Again, we know the number of bits used for the DIMM, rank, channel and bank from the specifications and can calculate the corresponding number of sets. We assume, that the ARM platform uses simple XOR functions as well. In fact, many modern microcontrollers use XOR address mapping for bank interleaving [52, 96]. Bank interleaving reduces the DRAM latency by mapping addresses that would normally be in the same bank but in a different row, to a different bank [30]. For this platform, we had no indication of the mapping function. We guessed it to be similar to functions used on Intel CPUs.

In contrast to PCs, mobile devices do not use DDR RAM. They use a slightly modified type called LPDDR. The main difference concerning our attack is the lower power consumption and the temperature controlled refresh rate, *i.e.* the refresh rate is lowered with decreasing temperature. However, we can apply the same method to the ARM platform as well. As the hardware of a mobile phone cannot be changed, these functions are valid for all phones of this model. Most likely, they are the same for all phones that use the same hardware configuration.

On ARM we face different challenges when porting our tool. There is of course a different instruction set, meaning we have to change the inline assembly parts. Moreover, ARM microarchitectures before ARMv8 do not provide unprivileged instruction to flush the cache [11]. For the timing measurement, we can use the Performance Monitors Cycle Count Register (PMCCNTR). It is equivalent to the TSC on Intel. However, the performance monitor is only accessible in privileged mode [9].

Figure 3.5 shows a possible memory layout on a mobile device. As we can see, the memory layout is different, but similar. This allows us to use the same guess for the maximum number of bits per physical address as described in Section 3.4 for the Intel CPU.

ARMv8 We decided to use the ARMv8 microarchitecture as the first target. The reason is that ARMv8 provides the cache flush instruction `DC CIVAC` [10]. As operating system we use Android as it is widespread and based on Linux. Furthermore, Android provides access to the hardware cycle counter [53]. The tool is implemented using the Android Native Development Kit (NDK)[1] which allows to implement parts of applications in native code instead of Java. Until recently, the file `/proc/self/pagemap`, that we use to translate virtual to physical addresses was accessible for any user. Recently, this has been patched in the Linux kernel to only allow privileged userspace applications to access the page map [50]. Therefore, we

[1] https://developer.android.com/ndk/index.html

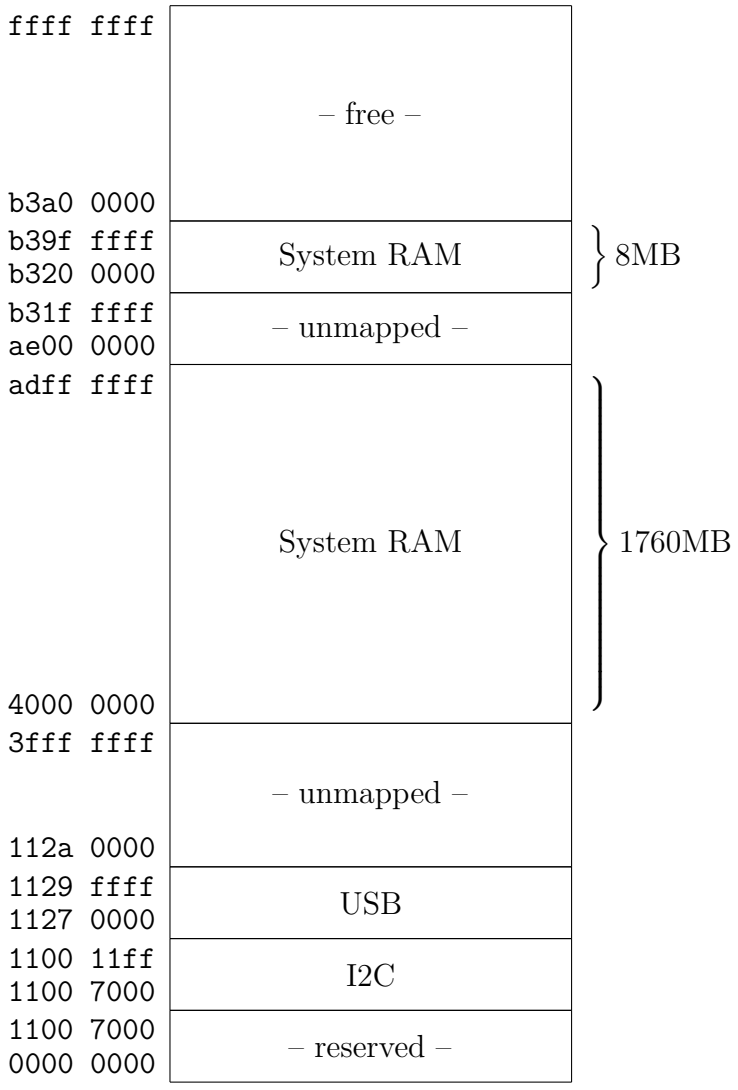


Figure 3.5: A sample physical memory layout of an ARMv8 device with 2GB memory.

require a rooted phone if we want to use the reverse engineering tool on a recent Android version.

For devices where the performance counters or flush instruction is not userspace accessible, we implemented a kernel module. As the kernel module runs in privileged mode, we can use all privileged instructions. Furthermore, we can reduce the noise by disabling the scheduler while doing the measurement. The advantage of the kernel module is that it gives us very accurate measurement results. However, the disadvantage is that if we do this on a mobile phone, we have to run a custom firmware in order to compile the kernel module. As the reverse engineering is only done once for a processor, this is only a negligible limitation.

ARMv7 On the ARMv7 microarchitecture, we cannot use the flush instructions in userspace. To circumvent this limitation, we can use cache eviction [53]. We used the implementation by Lipp et al. to evict the cache from our userspace application [60]. However, this only works for devices where the eviction strategy is known. Moreover, cache eviction introduces more noise than cache flushing which increases the number of measurements we have to make.

We again decided to implement a kernel module to have a generic software solution that does not depend on any hardware or software configurations. In the kernel module, we can use the flush instruction, access the performance counters and the page map. This gives us the same accuracy for the measurements as on ARMv8. Again, this comes with the drawback that we require a custom firmware to build the kernel module.

3.5.1 Verification

Due to the small chips and multi-layered circuit boards on mobile devices, we cannot verify the results by hardware probing. Furthermore, prior to this work, there were no published results of a reverse engineered DRAM addressing function of any ARM CPU. However, to rule out any errors of the tool, we would like to verify the results at least once.

As we cannot verify the results directly, we have to verify them using a side channel. One of the motivations for the reverse engineering was to increase the performance of the Rowhammer attack. With the correct addressing functions we should able to select a row and its two adjacent rows where all rows are in the same bank. This configuration allows us to mount the double-sided rowhammer attack. If we assume that we have 8 banks, the probability is 1/64 that three random addresses are from the same bank. We can improve this naïve address selection by selecting one address x and the two supposedly adjacent memory locations $x + sizeof(ROW)$ and $x - sizeof(ROW)$. Using the reverse engineered addressing functions, we can now choose the rows selectively to be sure that they are in the same bank. This gives us a higher probability of success for the rowhammer attack. The exact performance

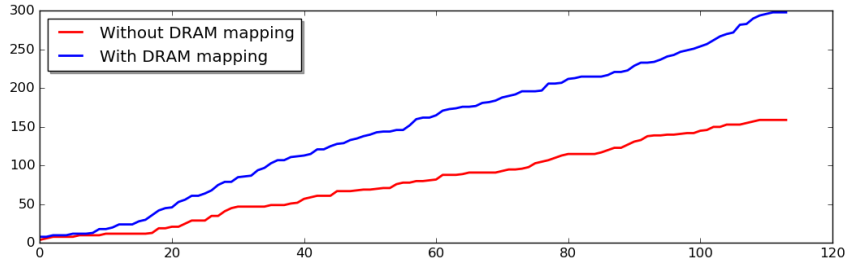


Figure 3.6: Bit flip rate on a Nexus 5 running Android 6.0.1 with and without reverse engineered addressing functions. The x-axis are the seconds, the y-axis is the number of bit flips.

gain depends on how the bank interleaving is implemented. However, we are not interested in the performance gain itself. As long as we see a significant performance gain, we know that the reverse engineered functions have to be correct.

Figure 3.6 shows the bit-flip rate on a Nexus 5. The two graphs show the number of bit flips with supposedly adjacent memory addresses and real adjacent memory addresses. We chose the real adjacent addresses based on the beforehand reverse engineered DRAM addressing functions. The number of bit flips doubled when choosing the correct rows using the addressing function. This is a strong indicator, that the reverse engineering works on ARM as well and yields the correct results.

3.6 Results

In this section, we analyze different Intel systems as well as smartphones. All results are obtained using the software-based reverse engineering method presented in this chapter. Pessl et al. [68] verified and published the results by hardware probing where this was possible. As hardware probing is infeasible on mobile devices, we verified the results using the rowhammer attack. For ARM, we tried to induce the labelling from the known labelling of Intel CPUs. For our tests, we used the machines described in Table 3.1.

The basic scheme is always the same although the details differ for each system [1, 45, 56, 68]. Figure 3.7 shows this basic schema. The address bits a_0 to a_2 are never used, they serve only as a byte index. As memory locations are byte-addressable but the memory bus has 64 bit, these bits are never sent over the memory bus and just select the correct byte. The next bits are used for the column inside a row. At least one bit in between is used for the channel selection. This leads to an equal loads on then channels when addresses are accessed sequentially. Due to the channel parallelism, this results in a lower latency. The next bits determine the DIMM, rank and bank. The exact usage depends on the system and is shown in Table 3.2. Finally, the remaining upper bits are used for the row selection. As banks can be accessed in parallel, it makes sense to change the bank before the row number. This

CPU / SoC	Microarch.	Mem.
i5-2540M	Sandy Bridge	DDR3
i5-3230M	Ivy Bridge	DDR3
i7-3630QM	Ivy Bridge	DDR3
i7-4790	Haswell	DDR3
i7-6700K	Skylake	DDR4
Samsung Exynos 5 Dual	ARMv7	LPDDR3
Samsung Exynos 7420	ARMv8-A	LPDDR4
Qualcomm Snapdragon S4 Pro	ARMv7	LPDDR2
Qualcomm Snapdragon 800	ARMv7	LPDDR3
Qualcomm Snapdragon 820	ARMv8-A	LPDDR3

Table 3.1: Experimental setups.

bit arrangement has the property that same bank addresses have a preferably large distance to each other.

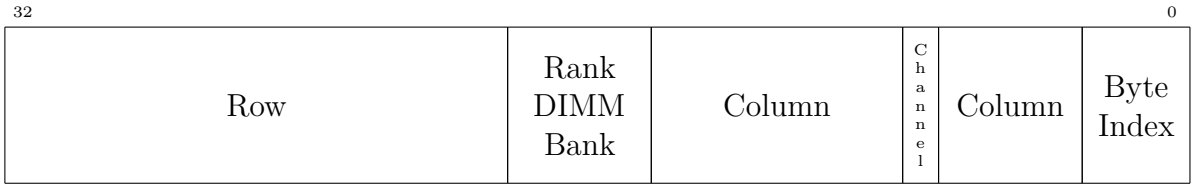


Figure 3.7: The basic scheme of a physical address

(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⋆	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†	2	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	8, 9, 12, 13, 18, 19
Skylake†	1	1	7, 14	15, 19	17, 21	18, 22	16, 20	-	-

⋆ Mark Seaborn
† Software analysis only. Labeling of functions is based on results of other platforms.

Table 3.2: Reverse engineered DRAM mapping on all Intel platforms and configurations we analyzed. These tables list the bits of the physical address that are XORed. For instance, the entry (13, 17) describes an XOR of address bit 13 and 17, i.e. $a_{13} \oplus a_{17}$.

Figure 3.8 shows the distribution of the bits for a machine with an Intel Core

i7 Ivy Bridge CPU and 8GB RAM in dual-channel. Compared to Sandy Bridge, the mapping function became indeed more complex with Ivy Bridge. The channel function uses an XOR combination of seven different bits instead of one bit on the Sandy Bridge microarchitecture. Intel presented this feature as *channel hashing* to distribute memory accesses more evenly across channels [76]. Still, they are linear functions using XOR. This is plausible, as the functions have to be implemented in hardware and they should not introduce any overhead.

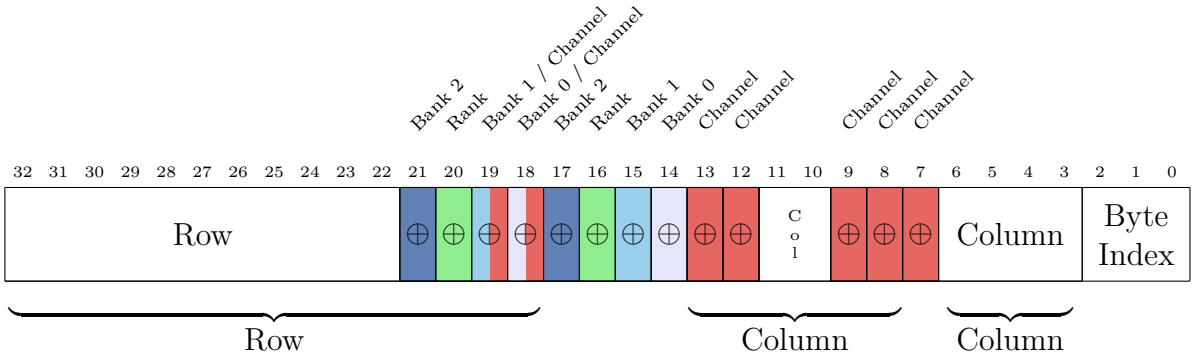


Figure 3.8: Bits of the physical address for a Ivy Bridge CPU with 2 channels and 1 DIMM with 4GB per channel.

We found that ARM indeed uses a similar, although simplified, schema. Here, only one- and two-bit functions are used. As we do not get the labelling using the software-based reverse engineering, our labelling is based on the basic scheme and the results from the Intel platform. Figure 3.9 shows the bit distribution of a Nexus 5 with ARMv7 and 2GB LPDDR3 memory from SK Hynix. This DRAM module has 8 banks and uses 15 bits for the row and 10 bits for the column [35]. The rank is interleaved with the column to utilize the rank parallelism when accessing sequential addresses. The following bits are used for bank selection to again switch the bank before the row number changes. As the bus width is 32 bits, only 2 bits are used as byte index. On the newer Samsung Exynos 7420 ARMv8 CPU, the rank and the channel use an XOR of two bits to distribute the memory accesses more evenly across the ranks and channels. Table 3.3 shows the reverse engineered functions of all analyzed ARM microarchitectures.

(a) LPDDR2

CPU	Ch.	BA0	BA1	BA2	Rank	Channel
Qualcomm Snapdragon S4 Pro	1	13	14	15	10	-

(b) LPDDR3

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

(c) LPDDR4

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

Labeling of functions is based on results of Intel microarchitectures.
† Software analysis only, without Rowhammer verification.

Table 3.3: Reverse engineered DRAM mapping on all ARM platforms and configurations we analyzed. These tables list the bits of the physical address that are XORed. For instance, the entry (7, 12) describes an XOR of address bit 7 and 12, i.e. $a_7 \oplus a_{12}$.

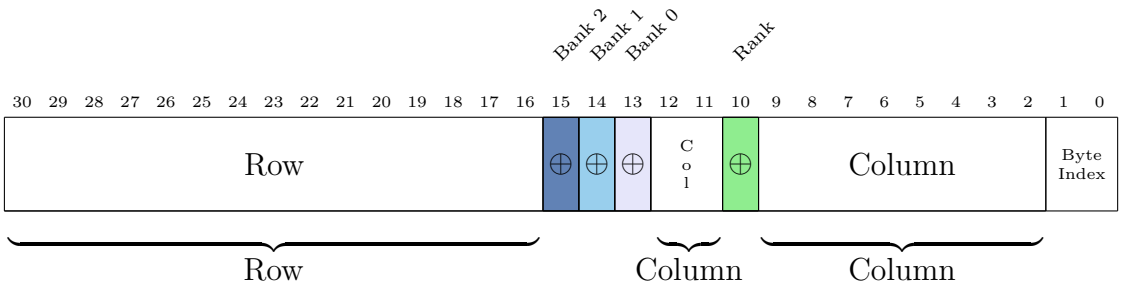


Figure 3.9: Bits of the physical address for a ARMv7 CPU with 1 channel and 1 DIMM with 2GB (Nexus 5).

3.7 Performance

The effort of reverse engineering the functions using the tool is much less compared to the probing approach. Still, it takes some time to measure the timings and reconstruct all the functions. Altogether, the reverse engineering of a new platform is in the range of minutes. As the framework does the measurement and calculation automatically, one only has to start the tool and come back to get the results. Depending on the random address selection and the amount of background noise on the machine, the process has to be repeated a few times to get reliable results.

Measurement. The time it takes to do all the measurements heavily depends on the CPU speed, RAM speed and the random selection of the addresses. Table 3.4 can be taken as a reference point. If the tested machine has more memory, it takes longer as the random address pool has to be larger.

Calculating the function. Although parts of the function calculation rely on a brute-force approach, it is by far faster than the measurement itself. The actual time depends on how many addresses where found per set. The worst-case occurs if there are only a few addresses per set. This gives many possible functions and the optimization step takes up to ten minutes. On average, the calculation takes between one and five seconds.

Device	Setup	Measurement	Computation	Overall
i5-3230M, 4GB RAM	$\approx$ 5 minutes	$<$ 25 minutes	$<$ 5 second	$\approx$ **30 minutes**
i7-3630QM, 8GB RAM	$\approx$ 5 minutes	$\approx$ 30 minutes	$<$ 10 seconds	$\approx$ **45 minutes**
Samsung Exynos 7420, 3GB RAM	$\approx$ 30 minutes	$\approx$ 20 minutes	$<$ 10 seconds	$\approx$ **60 minutes**

Table 3.4: Approximate timings to reverse engineer a platform.

Chapter 4

Attacks

In the previous chapter, we developed a method to reverse engineer the DRAM addressing functions fully automatically for a wide range of platforms. These addressing functions are the fundament for the attacks we present in this chapter.

DRAM is used in virtually every computer as main memory. Moreover, the main memory is one resource, which will most likely be shared among CPUs and CPU cores. CPU caches, on the contrary, are not shared among CPUs at all, and only the last-level cache is shared between CPU cores. The fact that all CPU cores of all CPUs can access the main memory makes it an excellent target for attacks.

In this chapter, we develop attacks that are similar to CPU cache attacks. However, we lower the requirements compared to cache attacks based on CPU caches. We do not require shared memory at all which is a requirement for several cache attacks, such as Flush+Reload or Flush+Flush [27, 91]. Moreover, our attacks also work in virtualized environments and across multiple CPUs.

This is especially interesting on servers, where multiple CPUs and virtual machines are prevalent: cloud providers used to enable kernel same-page merging (KSM) to reduce the necessary amount of physical memory [6]. With KSM, the hypervisor continuously searches for pages with the same content and merges them to one shared page. Especially on systems where multiple instances of the same operating system run, this technique can save a considerable amount of memory. However, many cloud providers have disabled this feature due to security concerns [68]. It was shown, that enabled KSM can lead to information disclosure [78, 89]. Moreover, it provides the basis for breaking ASLR and enables cross-VM cache attacks that are based on shared memory [12, 43].

One of the great strengths of our attacks is that they do not require KSM or any other form of shared memory. This makes them extremely powerful in settings where attacker and victim run on a cloud provider that does not provide any shared resources.

4.1 Covert Channel

Covert channels are unintended channels for communications that bypass access controls and restrictions established by the isolation of shared resources [88]. Exploiting shortnesses of this isolation allows two entities to communicate with each other. Previous research has shown, that covert channels are a potential leakage of data in the cloud [85, 88, 90, 93].

In this section, we develop a covert channel [68] that does not depend on shared memory or CPU caches. Moreover, our covert channel works in virtualized environment and also cross-CPU. The only requirement is a DRAM-based main memory that both the sender and receiver can access. We can find this setting on every cloud provider where multiple VMs run on the same hardware, i.e. where sender and receiver do not run on dedicated separate hardware.

The communication relies on the timing differences between row conflicts and row hits. As described in Section 2.1.1, there is a measurable timing difference between accessing data that is in the row buffer and data that is not in the row buffer. The idea is to use these timing differences to send bits from a sender to a receiver.

Figure 4.1 illustrates the basic principle of the covert channel. Both sender and receiver require an address in their address space that maps to the same bank but to a different row.

Receiver The receiver will continuously access its address and measure the access time. The access time tells the receiver whether there was a row conflict or not. As long as no process accesses a memory address in a different row of the same bank, the receiver's row will stay in the row buffer and the access time will be low. If another process accesses a memory location that is in the same bank but different row, the receiver's access will result in a row miss and consequently in a higher access time.

Sender The sender can provoke a row conflict for the receiver's address. By accessing its address which is in the same bank but in a different row, the sender's address will be fetched into the row buffer. This results in a row conflict the next time the receiver accesses its address. We use this timing difference to send our bits in the following way.

1 bit The sender accesses its row. The next read access of the receiver will be a row conflict which the receiver interprets as a 1.

0 bit The sender does nothing. For the next read access of the receiver, its row will most likely still be in the row buffer. The resulting fast access time is interpreted as a 0.

To increase the speed, we can also do this in parallel for all banks. For 8 banks, this allows us to transmit up to 8 bits in parallel.

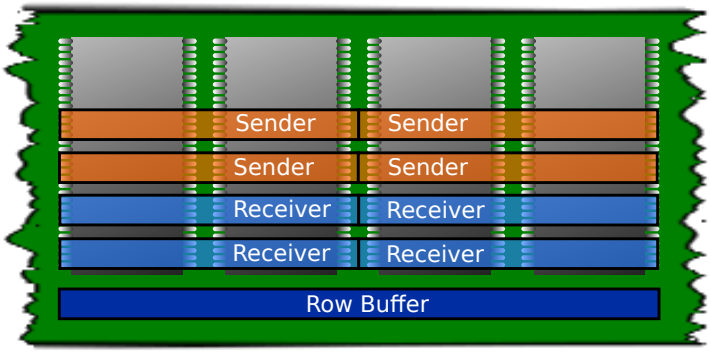


Figure 4.1: The sender and receiver both have an address in different rows of the same bank.

4.1.1 Cross-core covert channel

The first covert channel we build is a simple cross-core covert channel. It is a reliable implementation that works if the sender and receiver run on the same host and the same CPU. This assumption simplifies the synchronization between the two entities which is an important factor for reliable communication.

Synchronization As both sender and receiver run on the same CPU, we can synchronize them via the CPU. The idea is to have specific points in time where bits are sent and received. The advantage of this approach is that bits cannot get lost. There are exactly as many bits received as sent, transmission errors only show up as bits with a wrong value. To have exact points in time that are the same for both programs, we use the Time Stamp Counter (TSC). This register which is available on Intel CPUs contains the number of CPU cycles since startup. As both programs use the same CPU, and this value is the same for all cores, we have an accurate timestamp which we can use for synchronization.

Communication Channel Prior to any communication, sender and receiver have to agree on a communication channel, i.e. a tuple of (DIMM, channel, rank, bank) in which the both have access to a memory location in different rows. We introduce the term *bank address* for this tuple. Algorithm 4.1 shows the function that we use to calculate the bank address. It simply concatenates the DIMM, channel, rank, and bank to get one number.

Algorithm 4.1 Location hash of a physical address

```
function BankAddress(addr)
    return DIMM(addr) || channel(addr) || rank(addr) || bank(addr)
end function
```

This bank address can be hard-coded both in the sender and receiver. One way to get the addressing functions for DIMM, channel, rank and bank is to calculate them using the reverse engineering method from Chapter 3. It is also possible to pre-compute them if the target system is known.

Address Selection At startup, both sender and receiver require a memory location with the agreed bank address. As the addressing functions use physical addresses, we first require a method to translate virtual addresses to physical addresses. Prior to Linux kernel 4.0, the pagemap in `/proc/self/pagemap` was readable by anyone without privileges [50]. The pagemap is a kernel interface that provides an abstract representation of the page tables. Therefore, this can be used to translate virtual to physical addresses in kernel versions before 4.0.

However, with more recent kernel versions, the pagemap is only accessible with root privileges. To still be able to get the physical from a virtual address, we can exploit the usage of *transparent huge pages*. The Linux kernel will allocate 2MB pages for processes mapping large blocks of memory as often as possible [74]. By mapping a large block of memory using `mmap`, we will get 2MB pages for our memory if there are huge pages available. This could fail if, for example, no continuous memory area of at least 2MB is available. To verify that we got huge pages for our memory segment, we can check the value of `AnonHugePages` in `/proc/self/smaps`. If this is not zero, our memory segment uses huge pages.

With huge pages, there is no page table and the page directory points to the page directly. The last 21 bits of the virtual address are used as page offset. Therefore, the last 21 bits of the virtual address are equal to the last 21 bits of the physical address. Most systems do not use any bit above bit 21 for the DRAM addressing function. Nevertheless, even if one bit above bit 21 is used, we can successfully build the covert channel by sending the data to both addresses. As the address will most likely be a bank or rank bit, the two transmissions do not interfere.

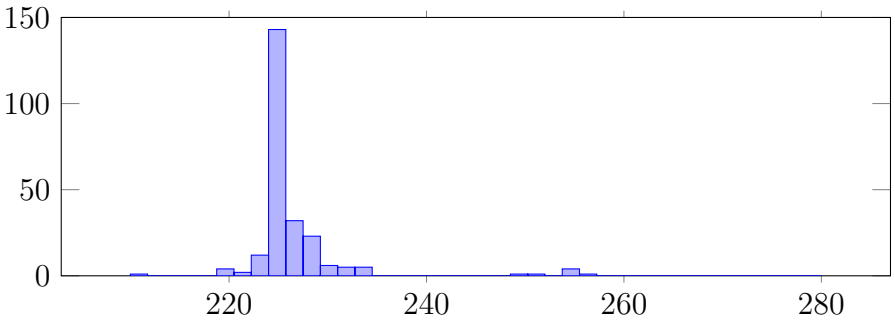
We can, therefore, use virtual addresses in 2MB pages to calculate the bank address. Note that the memory segment might not start with a 2MB page. Therefore, we can only assume that the first address with the lowest 21 bits set to zero is the start of a 2MB page.

Transmission For the actual transmission, we first start with an initialization phase. In this phase, the sender sends a sequence of alternating 0s and 1s. The receiver measures the access times and builds a histogram similar to the one in Figure 4.2c. Using this histogram, the receiver can calculate the threshold above which the measured access time is interpreted as a 1.

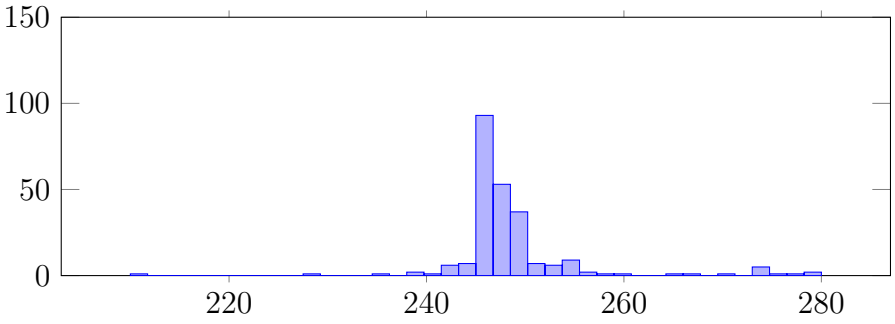
The sender and receiver synchronize the time of measurement using the TSC. Both use small time windows to send and receive the bits. Synchronizing the time window is accomplished by a *time barrier*. The time barrier busy-waits until the last n bits

of the TSC are zero, where the transmission speed depends on n. The higher n, the more reliable, but also slower the communication.

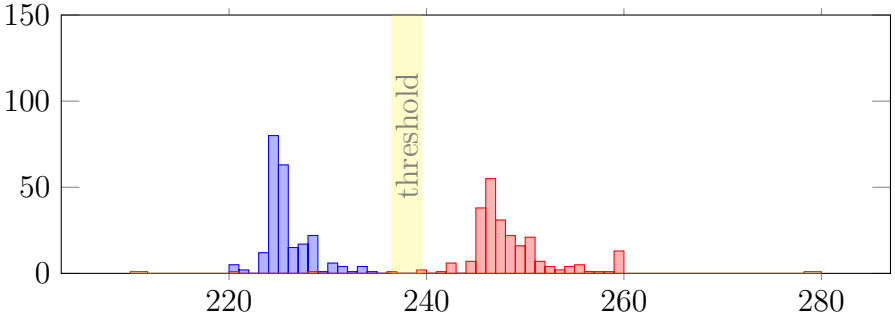
In our tests on the Intel Core i7 Ivy Bridge, we get a transfer rate of up to 5kbit per second with a bit-error rate below 1%.



(a) Transmitting a 0, i.e. no sender activity



(b) Transmitting a 1, i.e. sender is active



(c) Difference between 0 (≈ 225 cycles) and 1 (≈ 247 cycles)

Figure 4.2: Histogram of access times for the receiver's address on the Intel Core i7 Ivy Bridge machine. If the sender is active, the access time is approximately 20 cycles higher on average.

4.1.2 Cross-VM covert channel

A cross-VM covert channel is a more interesting and also realistic scenario for a covert channel. Most cloud providers use virtual machines for their customers. Moreover, the covert channel allows us to circumvent working isolation methods which would normally prevent any communication between virtual machines. However, when using a virtual machine, we face several additional obstacles compared to a cross-core covert channel.

Synchronization When running in a virtual machine, we cannot use the TSC for synchronizing the communication. Most hypervisors emulate the `rdtsc` instruction for multiple reasons [20].

First, virtual machines can be paused and resumed at any time. If the hardware TSC is used, this will lead to jumps in the time stamp value. There must also be a starting offset that is correct since not every virtual machine is started at the same time. Otherwise, the TSC would not represent the time since startup.

Second, older symmetric multiprocessing (SMP) systems might not have a TSC that is synchronized across multiple CPUs. On such SMP systems, the time stamp counter is not guaranteed to be monotonically increasing. This is especially a problem when virtual machines can be moved from one CPU to another. This also means that we have different time stamps if the VMs are running on different CPUs on the same system.

Third, as the time stamp counter always increases with the maximum CPU frequency, independent of the current CPU speed, this leads to problems when migrating the VM to a different machine with a different CPU speed. For example, if the VM is migrated from a 3GHz machine to a 1.5GHz machine, the time stamp counter suddenly increases with only half the frequency as before.

Fourth, letting the user access the real TSC can lead to security risks. A malware can use the TSC to detect that it is running in a virtual machine. Moreover, access to the hardware TSC simplifies cache attacks in virtual machines [62, 82].

These are the main reasons that the hypervisor usually emulates the time stamp counter. Therefore, we cannot use the time stamp counter as a synchronization mechanism anymore.

Instead of synchronizing the sender and receiver, we take advantage of the Nyquist-Shannon sampling theorem.
Theorem 1. *If a function contains no frequencies higher than W Hz, it is completely determined by giving its ordinates at a series of points spaced 1/(2W) seconds apart. [77]*

If the condition is fulfilled that our sampling rate is at least twice as high as the signal rate, we can perfectly reconstruct a measured signal. Applied to our covert channel, this states that if the receiver measures the access time at least twice as

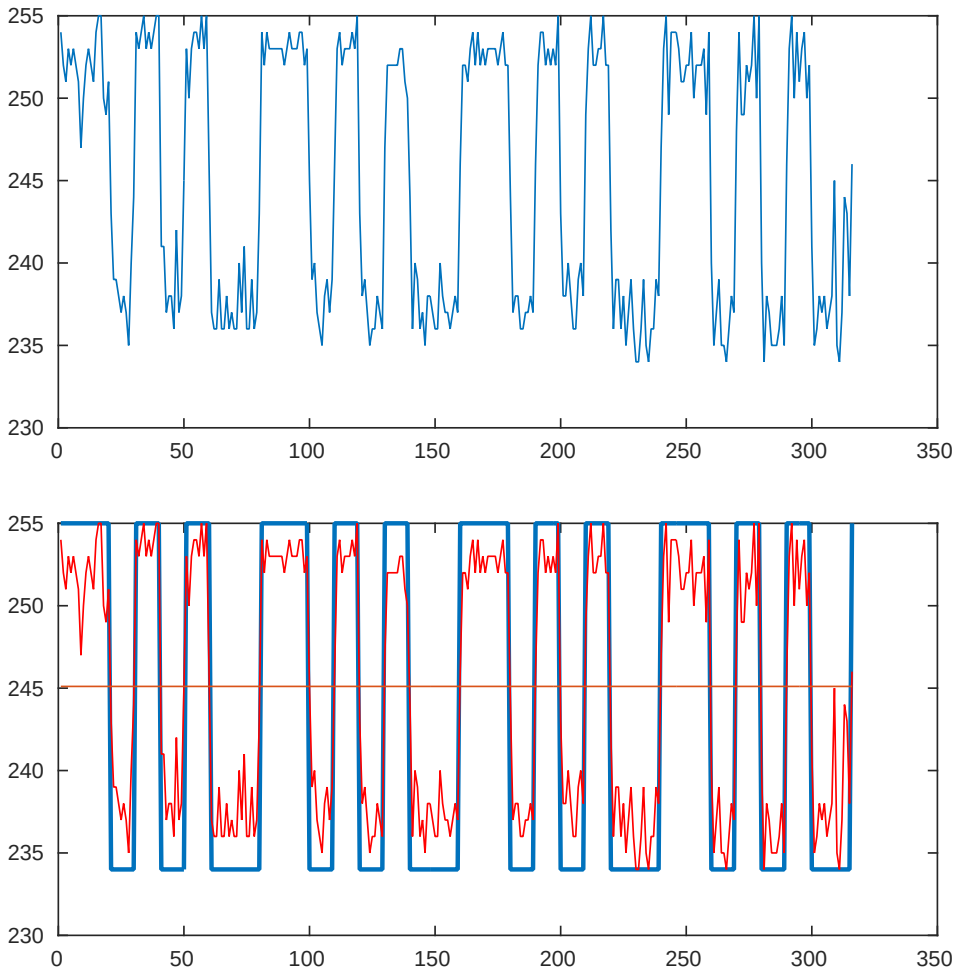


Figure 4.3: Measurement of the access times when repeatedly sending the bit-sequence `11010100` over the covert channel

often as the sender accesses the addresses, we will be able to reconstruct the sent signal fully.

To further reduce the noise, we measure with a rate that is ten times as high as the senders access rate. Figure 4.3 shows the measurement of a transmission and the reconstructed signal.

To get an exact time, we do not use the `rdtsc` instruction. Instead, we use the highest resolution clock that is provided by the operating system. In C++ 2011, we can use the high resolution clock `std::chrono::high_resolution_clock` [16]. This clock does not provide an absolute time, but it can be used to measure durations with a high accuracy.

Address Selection There is also a difference in the theory of address selection. As we run in a virtualized environment, we do not have only *virtual* and *physical*

addresses anymore. Instead, we have to distinguish between *guest virtual*, *guest physical*, *host virtual* and *host physical* addresses. We are again only interested in the host physical address, as this is the one that is used to determine the DRAM location. However, the virtualized operating system only knows guest virtual and guest physical addresses.

Depending on the implementation of the hypervisor and the underlying CPU, there are two methods which are used to translate guest physical to host physical addresses [13]. We describe both of them and show the implications on the cross-VM covert channel.

Extended Page Tables On newer CPUs, the hardware has special support for an additional paging layer for virtualized pages. Intel calls this technique *extended page tables* (EPT) [37]. AMD has a similar technique as well which is called *nested page tables* (NPT) [2]. It is an extension of the paging mechanism, where the memory management unit (MMU) does the translation from guest virtual to host physical addresses by walking through the guest page tables and looking up every guest physical address in the nested page tables. Due to this structure, the mechanism is also called *two-dimensional page walk* [14]. Figure 4.4 shows a translation from a 64-bit guest virtual address to a host physical address. The *nested walk* is a normal paging look-up in the nested page tables to convert the guest physical address to a host physical address.

Although the paging is implemented in hardware, it requires multiple memory lookups which are the bottleneck in the address translation. Therefore, one or more *translation lookaside buffers* (TLBs) are used to cache the results of the address translation. To reduce the number of memory accesses required to translate from a guest virtual to a host physical address, the host can utilize large pages. For example, the use of 2MB pages instead of 4KB pages reduces the paging structure by one level.

We can see, that the extended page tables do not influence the page offset. Therefore, if both the host and guest use large pages, i.e. they use page sizes of 2MB or even 1GB, at least the last 21 bits of the guest virtual address are again equal to the 21 bits of the host physical address. As long as only the lower 21 bits of the physical address are used, we can calculate the bank address from the virtual address. If 1GB pages are used, the lowest 30 bits of the virtual address correspond to the lowest 30 bits of the physical address. This is sufficient for all microarchitectures we have reverse engineered.

Shadow Page Tables If either CPU or hypervisor do not support extended page tables, or extended page tables are disabled, *shadow page tables* are used. With shadow page tables, the virtual machine holds a copy of the virtualized operating system's page tables. The shadow page tables directly map guest virtual to host physical addresses. When resolving a guest virtual address, the hardware uses these copies, i.e. the `CR3` register points to the shadow page tables. Figure 4.5 illustrates the method. The guest operating system has its own virtual `CR3` register that is not

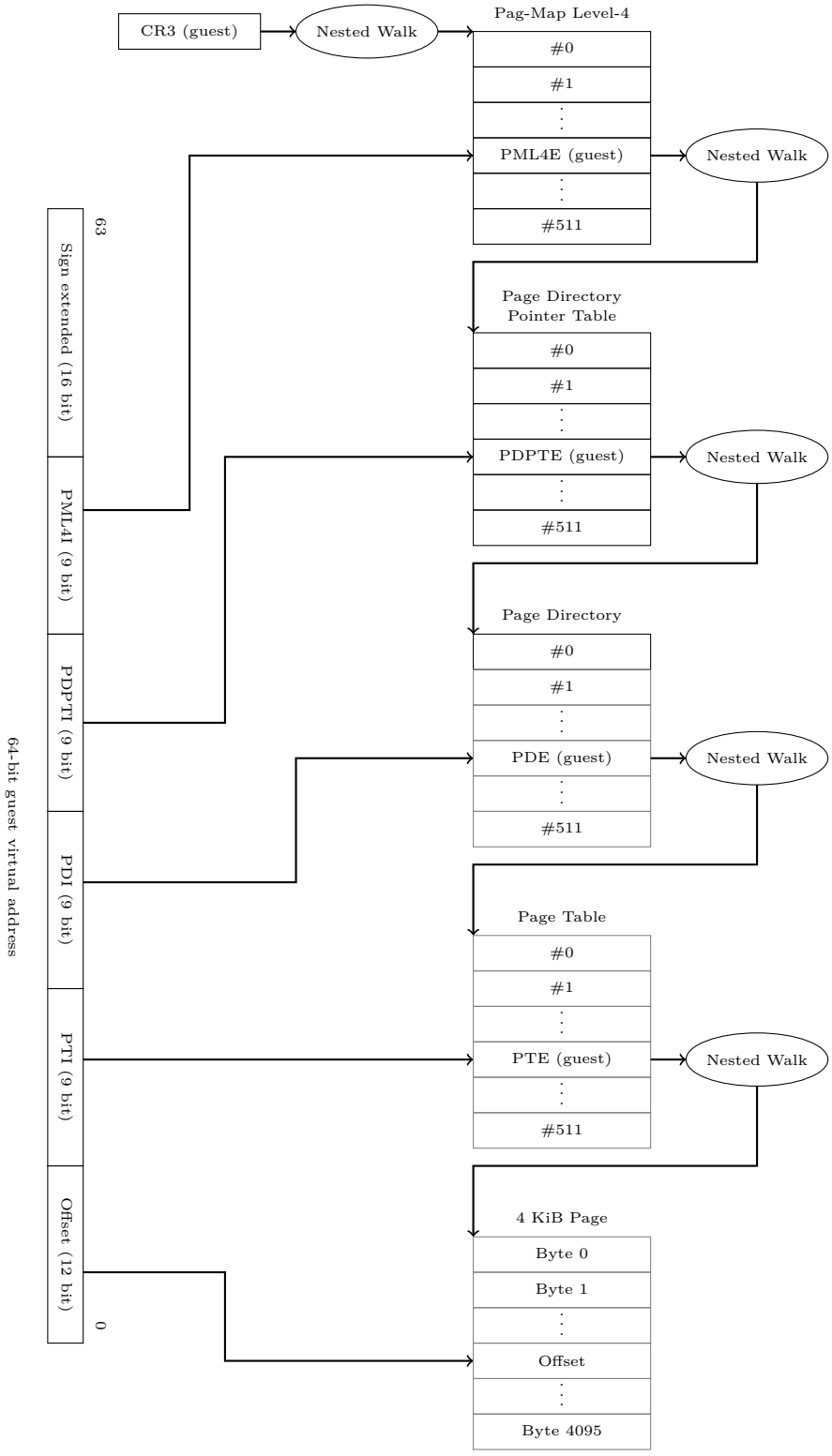


Figure 4.4: Guest virtual to host physical using nested page tables

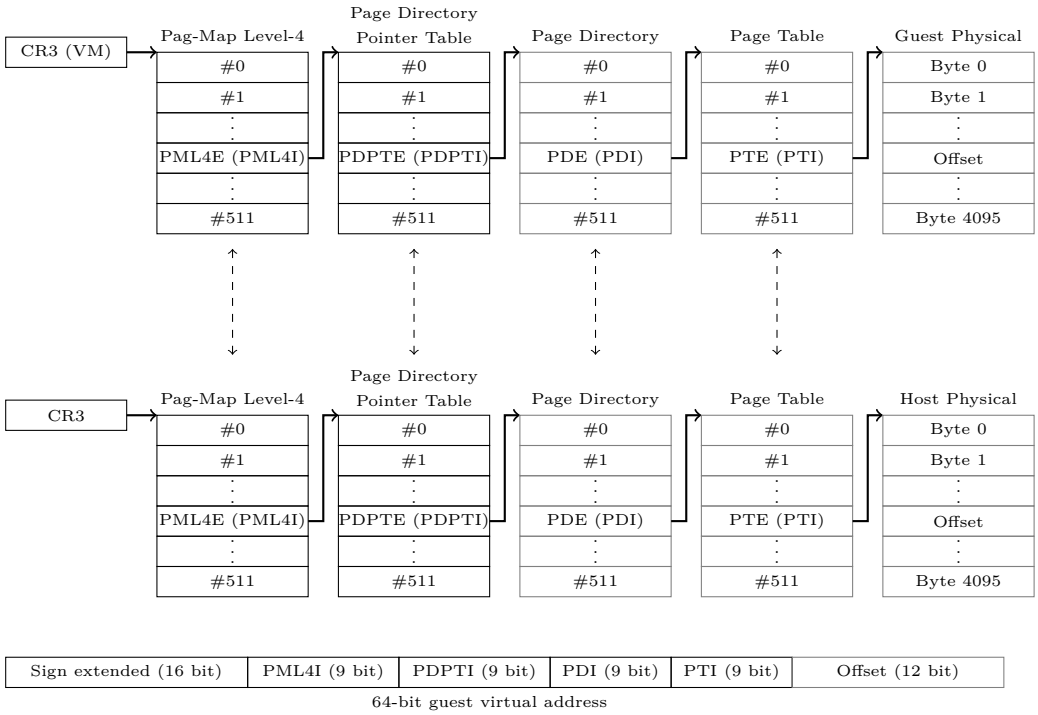


Figure 4.5: Address translation using shadow page tables

used by the MMU but only by the operating system. The MMU uses the real `CR3` register that points to the shadow page tables.

The virtual machine has to keep the shadow page tables synchronized with the normal page tables. Whenever the guest operating system updates its page tables, the shadow page tables are updated as well. The virtual machine can either do this when the operating system invalidates the TLB or set all page tables to read-only and handle the page fault whenever the operating system modifies the page table. Switching the `CR3` register is a privileged instruction that is also handled by the virtual machine. If the operating system does a task switch, it traps into the virtual machine, and the virtual machine can switch to the correct shadow page tables.

Again, we can see that this technique does not have an influence on the page offset. Therefore, we can also use the lower 21 bits of the guest virtual address if both the guest and host use 2MB pages. For our attack, it does not matter whether the virtual machine uses extended page tables or shadow page tables as long as large pages are used.

Transmission The actual transmission does not differ much from the cross-core covert channel. We first start with an initialization phase. In this phase, the sender sends a sequence of alternating 0s and 1s. The receiver measures the access times and builds a histogram similar to the one in Figure 4.2c. Using this histogram, the receiver can calculate the threshold above which the measured access time is interpreted as a

1.

As we cannot synchronize the time in a reliable way anymore, we use a higher rate for receiving as for sending. While the receiver sends one bit, the receiver measures the access time n times. To fulfill the Nyquist-Shannon condition, n must be greater or equal 2. A higher value for n means a slower, but more reliable communication. To cope with inevitable noisy measurements, we set n to a higher value and consider the system as a K-out-of-n-system [69]. This means, we measure n times and if we get at least k-times the same value, we accept it as the correct value. Otherwise, we cannot say anything about the received bit, meaning we have a bit error in the transmission.

Figure 4.6 shows the transmission speed on our Intel Core i7 Ivy Bridge test machine. With a tranmission rate of 3.3kbit/seconds we get a bit-error rate of approximately 5%. Using the transmission rate and the error rate, we can calculate the channel's capacity [19]. The channel capacity C is given as

$$C = T \cdot (p \log_2 p + (1-p) \log_2 (1-p) + 1), \tag{4.1}$$

where T is the raw transmission rate and p is the bit-error probability. The resulting maximum channel capacity is 2.3kbit/second.

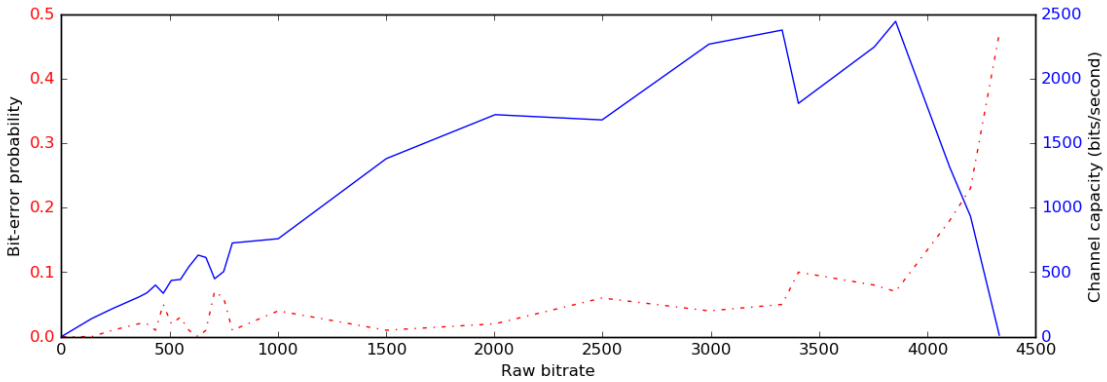


Figure 4.6: The tranmission rate and corresponding channel capacity and bit-error probability.

4.1.3 JavaScript covert channel

The cross-VM covert channel is primarily applicable in cloud settings where multiple virtual machines are hosted on one server. In this section, we develop a covert channel that is useful if the victim is a normal user. We consider a scenario, where a user runs a virtual machine from his company on his computer. As the virtual machine contains sensitive data, it has no network hardware configured. This configuration should prevent any data leakage. Nevertheless, the user's computer is connected to the internet.

The idea is again to use the DRAM row buffer to build the covert channel between the virtual machine and the computer. However, we show that it is possible to implement the receiver purely in JavaScript, meaning that the user does not have to run any program. We are then able to send data from the virtual machine via the covert channel to the internet and leak private data from the virtual machine.

In JavaScript, we have even more limitations than on a virtual machine. We cannot execute any native code such as `clflush` or `rdtsc`. Moreover, we neither have access to physical nor virtual addresses as the JavaScript code is sandboxed and the language does not use the concept of pointers [64].

Timing Measurement In JavaScript, we do not have direct access to the TSC. Therefore, we need a different method to measure timings with a very high precision. The highest resolution timer we have access to is the High Resolution Time API [84]. This API provides a sub-millisecond time stamp that is monotonically increasing. The standard recommends that the time should be accurate to 5 microseconds if the hardware supports this accuracy. However, until Firefox 36, the `window.performance.now()` function returned the time stamp with nanoseconds resolution on Linux. On a machine with 2GHz CPU, we get an accuracy of 2 cycles which is almost as exact as the native time stamp counter. Recently, all new browser versions decreased the resolution for security reasons [18]. Table 4.1 shows the accuracy on different operating systems and browser versions.

	Windows 7/10	Linux	Mac OS X
Firefox 36	1μs	**1ns**	**1ns**
Firefox 47	1μs	1μs	1μs
Internet Explorer 11	1μs	-	-
Chrome 37	10μs	1μs	1μs
Chrome 51	5μs	5μs	5μs

Table 4.1: Resolution of `window.performance.now()` on various browsers and operating systems

We will show that decreasing the accuracy does not prevent our covert channel from working. We can circumvent the lower accuracy by increasing the number of

measures. This leads to a much lower performance, but we are still able to transmit data.

Address Selection As JavaScript does not know the concept of pointers, we have neither access to virtual nor to physical addresses. However, to find the bank address, we require at least the lowest 21 bits of the virtual address of a 2MB page. We can, however, use a timing side channel to infer the page borders [25].

We observed that large arrays are backed by contiguous 2MB pages if they contain only primitive types, such as integers. This is true for Firefox and Chrome. As the memory is mapped on-demand, we will trigger a page fault whenever we access a new page. These accesses have a significant timing difference. Figure 4.7 shows the access times in nanoseconds for the array locations when iterating over the array. We can clearly detect the beginning of a new physical page as the access time is in the order of microseconds, whereas a normal memory access completes within a few nanoseconds. As the timing difference is that large, we can even do this with modern browsers that have a reduced resolution of only 1μs or 5μs.

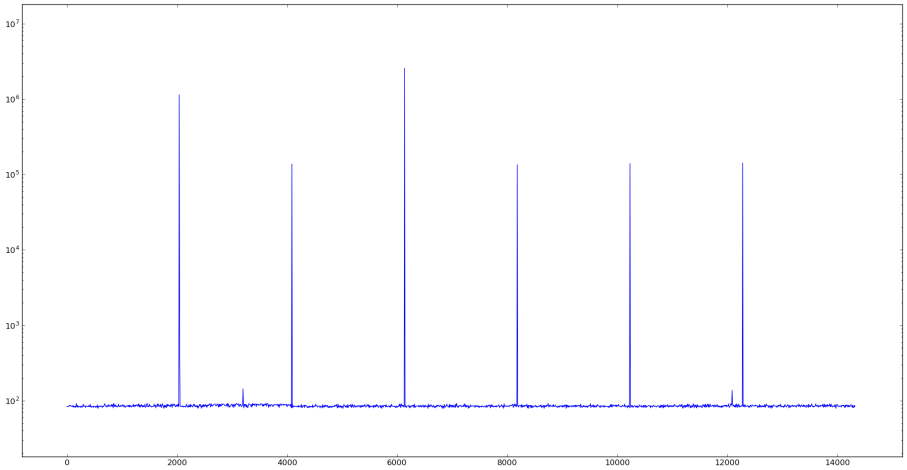


Figure 4.7: Timing peaks when accessing a new 2MB page.

As we know that the page faults occur at the beginning of a page, we can analyze the array indices that cause the large timings. Those indices are always 262144 elements apart, resulting in an element size of 8 bytes. This seems plausible, considering the fact that JavaScript stores integers as floating point numbers.

As a consequence, we have the array index that starts at a page border, meaning the lowest 21 bits are zero. Furthermore, we can access all 8-byte aligned addresses on that page. This is sufficient to find a memory location with a given bank address. If we find two indices i_0 and i_1 with a difference of 262144, i.e. $i_0 + 262144 = i_1$ which both have high timings, we can calculate the lowest 21 bits of the physical address

for any array index on this page. Given the array index x with $i_0 \leq x < i_i$, those bits are $(x - i_0) \cdot 8$.

Eviction In JavaScript, there is no possibility to flush the CPU cache. However, this is one of the prerequisites for the covert channel. Each memory read has to fetch the data directly from the DRAM and not from the CPU cache to exploit the timing difference in the row buffer. As we cannot execute any native code in JavaScript, we have to fallback to a different technique, cache eviction.

With cache eviction, an address is not flushed explicitly from the cache. Instead, different addresses are accessed that fill up the cache. If enough addresses have been accessed, the address that shall be flushed will not be in the cache anymore. The naïve implementation suggested by Hund et al. [33] is to fill a memory segment that is as large as the cache. However, this method is extremely slow. We are looking for a faster and still reliable way to evict an address from the cache.

Gruss et al. [25] describe how to reduce the number of addresses that we have to access to flush our address from the cache. They suggest several eviction strategies for different CPU microarchitectures. We can use the same strategies as they are fast, and the algorithm can be implemented in JavaScript, as it does not depend on pointers. Furthermore, we will also use the notation introduced by Gruss et al. to denote the eviction strategy. They describe each strategy in the form $\mathcal{P} - C - D - L - S$. This eviction strategy can be implemented using the code in Listing 4.8. In this code, `memory` is a large array and `address` is the eviction set. The eviction set contains S congruent addresses.

```
1 for(var s = 0; s <= S - D; s += L)
2   for(var c = 0; c <= C; c += 1)
3     for(var d = 0; d <= D; d += 1)
4       memory[address[s + d]];
```

Figure 4.8: Cache eviction loop [25]

On our Ivy Bridge test machine, the most efficient eviction strategy is $\mathcal{P} - 4 - 5 - 5 - 20$. Figure 4.9 shows the access pattern of this eviction strategy. The JavaScript implementation of this eviction strategy takes approximately 22μs to evict one address from the cache.

Even though the eviction rate is not 100%, it is sufficient to measure most of the access times correctly. If the eviction fails, we might miss one measurement, but this will only give us one bit error in the worst case.

Synchronization In JavaScript, we have the same limitations for the synchronization as we have for the cross-VM covert channel described in Section 4.1.2. There is no shared accurate clock between sender and receiver.

However, we can again use the Nyquist-Shannon Theorem in this setting. In the time span where one bit is sent, we measure the access time at least twice. To reduce the noise, we can measure the access time n times and interpret it as 1-bit if k out of those n measurements indicate a row conflict. We can choose k freely in the range $1 \leq k \leq n$. The higher k, the more measurements must be the same in order to accept the measurement as correct. Therefore, we have a higher chance that the bits are correct with the downside that we might discard some correct bits.

Transmission Transmitting data works in the same way as for the cross-VM covert channel. Again, there is an initialization phase where alternating bits are sent to calibrate the receiver. After calibration, the receiver can distinguish between 0-bits and 1-bits using the time it takes to access the variable.

One difference is that the flushing step is replaced by the cache eviction step. As the eviction is not as accurate as a cache flush, we might get some wrong access times here. However, we can increase the number of measures, denoted by n to filter out these incorrect timings.

With Firefox up to version 36, measuring the access time is the same as in the cross-VM covert channel. However, in newer versions and also in Chrome or Internet Explorer, the time is a lot less accurate. We cannot measure the difference between a row conflict and a non row conflict. Preventing an attacker to measure such small timing differences was one of the reasons for reducing the timer accuracy [18].

Still, we can circumvent this limitation by repeating the measurements multiple times. As the sender accesses the address continuously while sending a 1-bit, we will always get a row conflict when accessing the address. Therefore, we can measure the summed access time of multiple accesses. This method is of course much slower. On a recent version of Firefox, we can transmit 0.5 bits per second. This shows that decreasing the timer resolution does not prevent these kinds of attacks, but only reduces the performance.

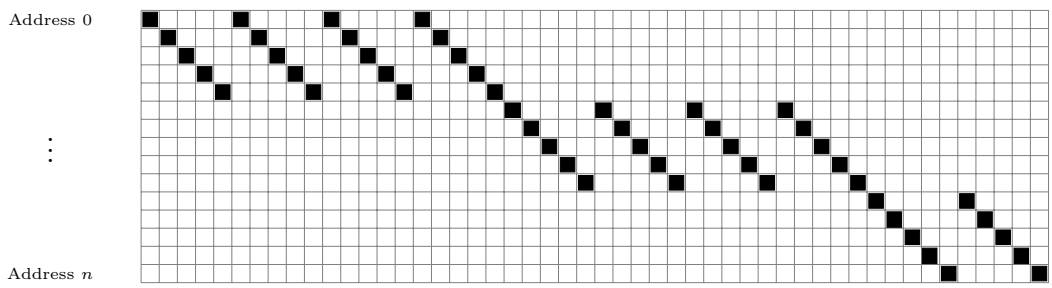


Figure 4.9: Eviction strategy on Ivy Bridge. Each row represents an address, and each column is one time interval. A filled rectangle represents an access to the address

4.2 DRAM Template Attacks

The covert channel from Section 4.1 exploited the timing difference between a row conflict and a non row conflict. As both the sender and the receiver settled on a common bank, they were able to induce row conflicts and measure them. This works as long as sender and receiver work together.

However, the row buffer can also be used to spy on a process that is not under our control. The idea is to use a similar approach as in Flush+Reload [91]. Based on the time it takes to access an address, we want to be able to detect whether our victim accessed a certain address or not [68].

Instead of exploiting row conflicts, we will focus on row hits. If we access an address that falls in the currently open row, i.e. into the row that is loaded in the row buffer, the request can be served immediately. The memory controller does not have to access the DRAM cell but only the faster row buffer. We call a row *shared* if the row contains both the attacker's and the victim's data. Figure 4.10 shows four rows where one of them is shared between the attacker and the victim.

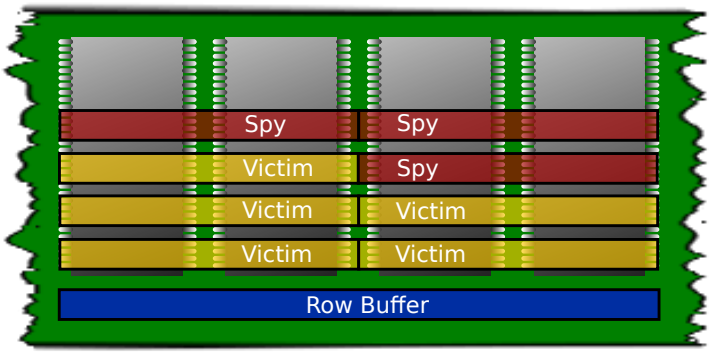


Figure 4.10: In the row hit attack scenario, the attacker (spy) has a 4KB page mapped in the same row as the victim. Whenever the victim accesses its page in this row, the row is loaded into the row buffer and the spy experiences a lower access time.

Basic principle The requirements for the attacker are one address a in a shared row and one address $\bar{a}$ that causes a row conflict with the shared row. The attacker can repeat the following steps

1. Access address $\bar{a}$.
2. Wait for victim to access its data.
3. Measure access time for address a.
4. Flush addresses a and $\bar{a}$ from the cache.

When accessing the address a, we either have a row hit or a row conflict. If the attacker has accessed the data after we have accessed $\bar{a}$, the row containing address a is loaded into the row buffer, and we have a row hit. Otherwise, $\bar{a}$ or any other address from a different row is in the row buffer, and we get a row conflict when accessing address a.

Despite the term shared row, this attack works without any form of shared memory. One row in the DRAM usually has 8KB, meaning that there are at least two 4KB pages in one row. If the lowest 12 bits of the physical address are not used for the DRAM addressing, the whole 4KB page is in one row. Then, exactly two pages share one row. This case is also illustrated in Figure 4.10. In such a case, the accuracy of our attack is 4KB as we can only detect whether the victim process has accessed its page or not.

Spatial accuracy However, on most systems, address bits below bit 12 are used for the DRAM addressing function as well. In this case, we get an even higher accuracy. If any bits of the lowest 12 bits are used to select the bank, rank or channel, one page stretches over multiple rows. Conversely, if one page stretches over multiple rows, one row contains multiple pages as well. For example, on our Ivy Bridge test machine, the channel selection bits can be found in the least significant 12 bits. This results in a page spanning across two rows. One part is accessed through the first channel. The other part is accessed through the second channel. As a consequence, one 8KB row contains four 4KB pages.

Skylake uses another bit in the lowest 12 bits for the addressing function. On this microarchitecture, address bit a_7 influences the bank group and a_8 is part of the channel selection. Therefore, one page stretches over four rows, and one row contains eight pages. Figure 4.11 illustrates how one 4KB page on Skylake maps to the corresponding rows. The lowest bit that is used on Skylake is a_7, meaning blocks of 128 consecutive bytes on the page map to 128 consecutive bytes in the row. However, consecutive blocks map in fact to the same row number but to a different row on the DRAM as either the bank group 0 or the channel changes. The page is therefore distributed over four different locations on the DRAM.

As each page is distributed over four DRAM rows, each page occupies 1KB in one row. Therefore, each row contains parts of eight different 4KB pages. As a consequence, we can spy on a victim's page as long as at least one of the other seven pages is mapped by our attacking program. The spatial accuracy is, therefore, 1KB.

Depending on how many of the bits a_0 to a_{11} are used for the addressing function, we get a spatial accuracy between 4KB if no bits are used and 64 bytes if six bits are used. The latter is the case if channel, DIMM, rank, bank, bank group, and CPU use bits from those low bits. Table 4.2 shows the spatial accuracy of our test machines.

As we can see, DDR4 ram provides a higher accuracy as the number of banks increased from 8 to 16. Furthermore, having more DIMMs potentially increases the accuracy as well, as more bits are used.

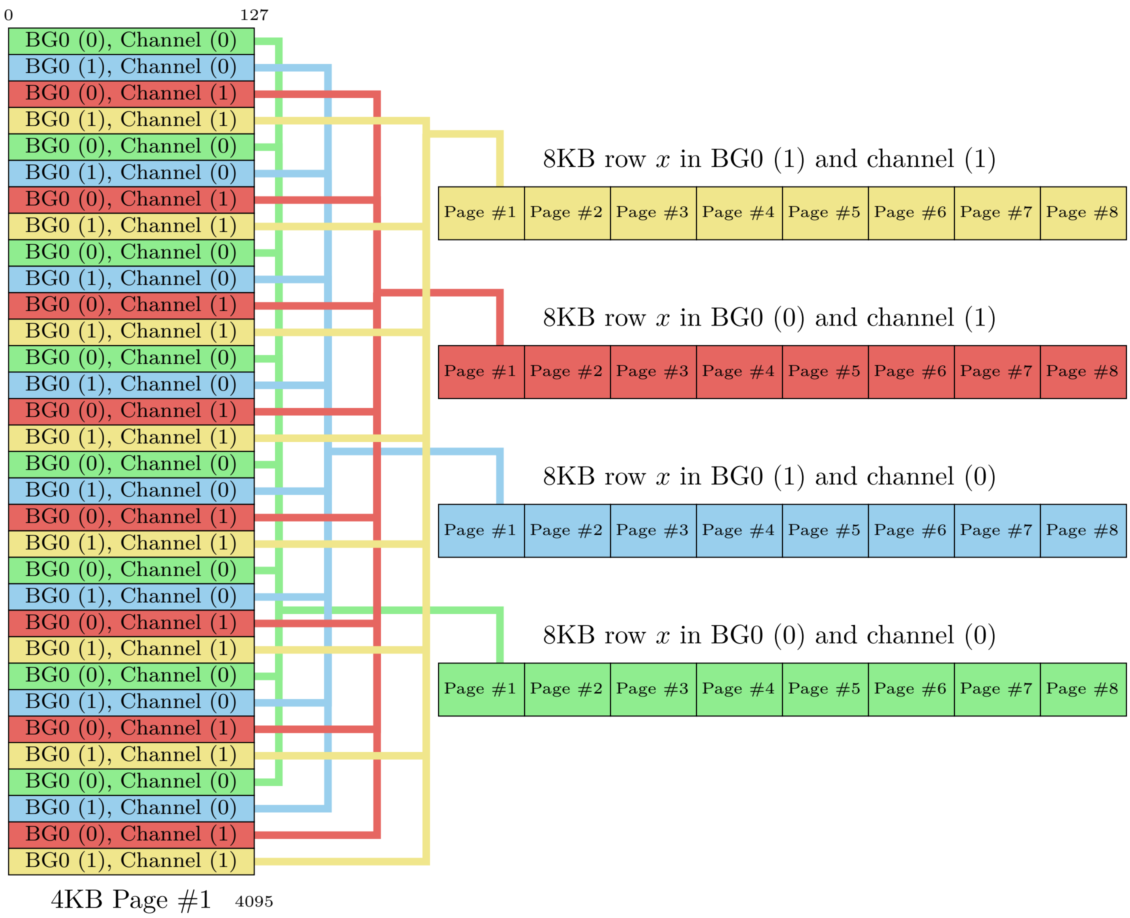


Figure 4.11: The mapping between a 4KB physical page and an 8KB DRAM row on Skylake. Each row contains eight 1KB parts from eight different pages. The 128-byte blocks on each page are distributed over four different physical rows.

(a) DDR3

CPU	Ch.	DIMM/Ch.	Bits $\leq a_{11}$	Pages/Row	Accuracy
Sandy Bridge	1	1	-	2	4KB
Sandy Bridge⋆	2	1	Channel	4	2KB
Ivy Bridge/Haswell	1	1	-	2	4KB
	1	2	-	2	4KB
	2	1	Channel	4	2KB
	2	2	Channel	4	2KB

(b) DDR4

CPU	Ch.	DIMM/Ch.	Bits $\leq a_{11}$	Pages/Row	Accuracy
Skylake†	2	1	BG0, Channel	8	1KB

⋆ Mark Seaborn
† Software analysis only. Labeling of functions is based on results of other platforms.

Table 4.2: Our test machines and the corresponding spatial accuracy.

Comparison We can compare our accuracy to known cache attacks. Flush+Reload has a spatial accuracy of 64 bytes which is the best case of our attack. In most cases, the accuracy will be slightly lower but still comparable to Flush+Reload. However, in contrast to Flush+Reload, we do not require any form of shared memory.

4.2.1 Spying on keystrokes

Similar to cache template attacks [26], we can mount an attack to spy on keystrokes of a different process. We want to spy on a certain event that occurs in the victim. In this case, our target are keystrokes inside the victim.

The basic idea is first to profile the victim program. In this *profiling phase*, we induce the event we later want to spy on. During this phase, we scan the memory for addresses that have a row hit when accessing them. After we have found a set of addresses, we stop the event and check whether we still get row hits when accessing them. We can eliminate all addresses where there are row hits without the induced events. Those addresses are not linked to the event. Figure 4.12 shows the timing difference between a row hit and a row conflict.

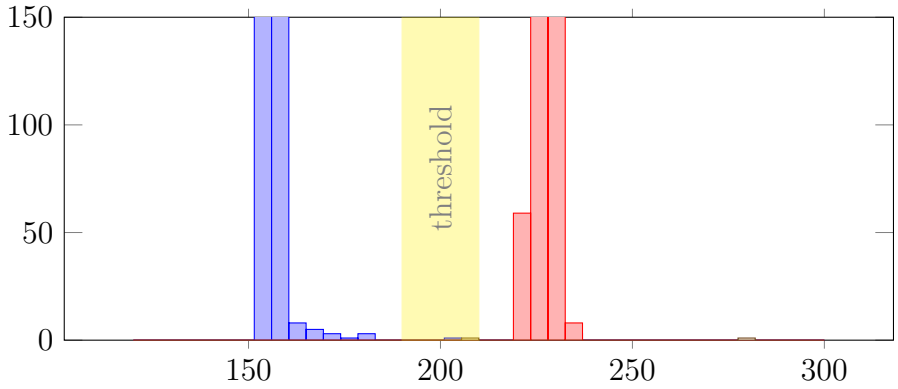


Figure 4.12: Histogram of access times for row hits and row conflicts. It shows the difference between row hits on the left (≈ 150 cycles) and row conflicts on the right (≈ 247 cycles)

In the *exploitation phase*, we use the found addresses to spy on the victim. We monitor the time it takes to access those addresses. If the victim receives the event, it accesses an address in the same row, and we experience a row hit for our address. From this, we deduce that the event, i.e. a keystroke, was processed by the victim.

We tested the attack on the address bar of Mozilla Firefox. As a prerequisite, we require pages in the same row as our victim. For this, we allocated approximately 90% of the computer's main memory. The more memory we map, the higher the probability that we get pages in the same row as the victim. After mapping the memory, we start the two phases of our attack.

In the *profiling phase*, we inject keystrokes into the address bar. We simultaneously scan the addresses in our allocated memory for row hits. Every address that has a row hit is saved as a possible offset for the attack. In this phase, we find approximately 1000 addresses. These addresses include multiple false positives which we filter in the next phase.

In the *exploitation phase*, we do not inject keystrokes. For every address which we identified during the profiling phase, we measure the access time. If the address is a row hit, we can remove this address as it is a false positive and does not correlate with the keystroke. The remaining addresses should only have row hits if the user enters a key into the address bar. Nevertheless, it is possible that there are still false positives. We can eliminate them by repeating the profiling and exploitation phases. If we did not get a page in the same row as a vulnerable page of the victim, we would stop without any address that is suitable for the attack. In this case, we have to restart the victim and start the profiling phase again.

Finally, we have a set of addresses where each address has a row hit whenever the user presses a key inside the address bar. Depending on the spatial accuracy, there a nearly no false positives. If the attacker's page and the victim's page are the only pages in the row, only the attacker can load the page into the row buffer. Figure 4.13 shows the access times of one address when entering an address into the address bar.

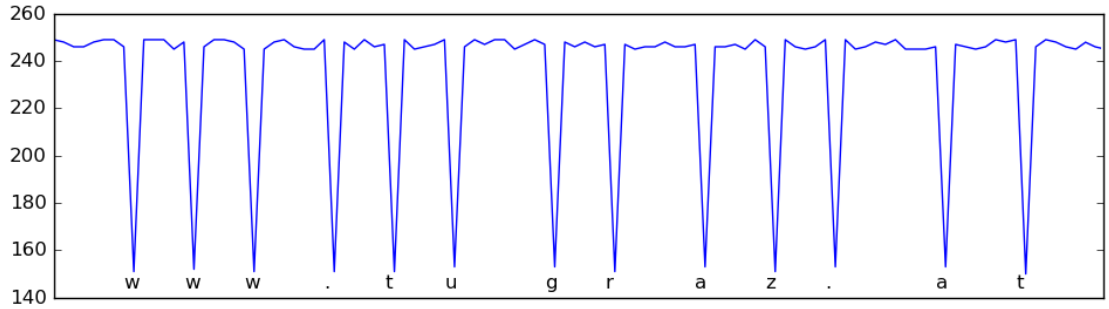


Figure 4.13: The access times of an attacker's address while typing inside the address bar of Firefox. Keystrokes lead to row hits and therefore lower access times.

Chapter 5

Future Work

In this chapter, we show what problems remain and which improvements to the presented methods will be made in future work.

Reverse Engineering In the reverse engineering approach, we assumed that the DRAM mapping function is always a linear function. On all our test machines, this assumption held. However, there are most likely machines where this is not the case. These would be machines where there exists at least one target used in the addressing function where the number is not a multiple of two. Specifically, if either the number of CPUs, DIMMs or channels are not a power of two, the mapping function differs.

In such a setup, the addressing function cannot consist of XORs only, as linear XOR function can only distribute the values uniformly if the number of outputs is a power of two. As we know that such configurations exist, we assume that they are handled in the same way as with cache slices. Inci et al. [36] and Yarom et al. [92] analyzed the case of non-linear addressing functions for caches. They proposed that the non-linear function is a two-stage function. First, a linear function consisting of XORs is generated. Then, this linear function is compressed using a non-linear function. We expect that the non-linear DRAM addressing functions will follow a similar pattern.

Although the measurement step would be the same, neither the brute-force approach nor the removal of linear dependent functions would work on non-linear functions. Furthermore, as we do not know the exact form, we could only reconstruct a generic representation in a normal form such as the ANF.

Covert Channel Our cross-VM covert channel can be improved by using multiple threads to transmit data. As DRAM banks can work in parallel, it is possible to transmit multiple bits at the same time. Depending on the number of CPU cores and available channels, this could give a major performance improvement.

One of the biggest limitations for the JavaScript covert channel is the transmission speed in modern browsers. Due to security concerns, the resolution of the high-resolution timing API was reduced to prevent side-channel attacks which rely on timing. Nevertheless, we demonstrated that it is possible to mount such attacks by measuring the accumulated time of multiple accesses.

However, the W3C published a working draft for the Web Worker API [83]. The Web Worker API allows JavaScript to spawn multiple threads with true parallelism. As soon as this is implemented, we are again able to get a high precision timing using the approach suggested by Lipp et al. [53]. We can spawn one counting thread, which continuously increments a global variable. This global variable is then used as a simulated TSC.

As global variables are not accessible by web workers, we have to use a different shared resource. At the time of writing (June 2016), the only shared resources are shared array buffers which are still experimental and not supported in any browser [61].

DRAM Template Attacks We showed that the DRAM template attack could be as fine-grained as Flush+Reload while not requiring any form of shared memory. As a proof-of-concept, we demonstrated a keystroke logger for Firefox. It should be possible to extend those attacks to get even more private information from a victim process. For example, it might be possible to get certain keys on systems with a high spatial accuracy.

Chapter 6

Conclusion

In this thesis, we presented an automated software approach to reverse engineer DRAM mappings. This method turned out to be an inexpensive and fast alternative to hardware bus probing. Bus probing took multiple days, whereas the software reverse engineered the function within minutes. The framework is written in portable C++ and is applicable to all Intel systems. It runs as a userspace application, as it does not need any privileged instructions. The framework can be fine-tuned to all circumstances we encountered while testing. This gives fast results and does not require a sophisticated setup.

We also ported the framework to the ARM platform. Our proof-of-concept is implemented on various smartphones running Android. Although the code does not change, the framework has to be compiled for every new system. This requires a custom kernel or, at least, the kernel sources. Nevertheless, we were able to reverse engineer the DRAM addressing function for all tested phones.

We verified the correctness of the tool by comparing the results with already known mapping functions for Intel CPUs [56, 68]. Furthermore, this increased the performance of Rowhammer attacks on mobile devices with ARM CPUs and on computers with DDR4 memory. Until now, this was slow or even infeasible due to the unknown mapping functions. We demonstrated that the success rate for the Rowhammer attack doubled compared to random hammering. Therefore, we are confident to say that the reverse engineered functions are also correct for ARM CPUs.

Based on this work, we have developed DRAMA (DRAM addressing) attacks [68]. We developed proof-of-concept attacks based on row conflicts and row hits. We demonstrated that co-operative malicious processes could build a covert channel which even works in sandboxed environments such as browsers. Furthermore, we are also able to spy on other processes in a cache attack like manner.

The multitude of parameters for calibrating our reverse engineering tool allows it to be adapted to future, even yet unknown, platforms. With the DRAM mapping function, the attacks are easily reproducible on new platforms. This emphasizes the importance of research on countermeasures against this new type of side channel.

Appendix A

Black Hat Europe 2016

We submitted the following form to the Black Hat Europe 2016 conference. Our submission with the title "DRAMA: How Your DRAM Becomes a Security Problem" was accepted as a 50-minute briefing. Together with Anders Fogh, I will present the content of this thesis on the Black Hat Europe 2016 conference in London.

Abstract

In this talk, we will present our research into how the design of DRAM common to all computers and many other devices makes these computers and devices insecure. Since our attack methodology targets the DRAM, it is mostly independent of software flaws, operating system, virtualization technology and even CPU.

The attack is based on the presence of a row buffer in all DRAM modules. While this buffer is of vital importance to the way DRAM works physically, they also provide an attack surface for a side channel attack. These side channel attacks allow an unprivileged user to gain knowledge and spy on anybody sharing the same system even when located on a different CPU or running in a different Virtual Machine.

We will show that we can exploit this side channel even in the limited environment of a sandboxed JavaScript application despite the countermeasures implemented in modern browsers. We will demonstrate the attack by sending data from a virtual machine without network hardware to the internet via the DRAM row buffer. The JavaScript library to exploit this attack vector will be made open source.

Further these attacks enabled us to reverse engineer the complex addressing function of the CPU. This knowledge has real world implication for other software attacks on hardware, such as the row hammer attack. We will discuss how our finding led to moving the row hammer attack to DDR4 ram and how this research enabled other researchers to do software based fault injection attacks on cryptographic keys.

We present an easy-to-use tool that can reverse engineer the CPUs addressing function fully automated. This tool is open source and can be used to reproduce the presented attacks, improve existing rowhammer-based attacks and to find new attacks.

Presentation Outline

1. **Introduction to DRAM & Caches,**
   This will establish a basic understanding of how DRAM and caches work. We will briefly describe the DRAM architecture as far as it is necessary to understand the attacks. The vocabulary used throughout the presentation is also part of the introduction.

2. **The row buffer as a side channel objective,**
   We will explain the prominent role of the row buffer for all our attacks. This includes how the row buffer becomes a medium for side channels and the basic principle to exploit this side channel.

3. **Spying on processes,**
   Using the memory mapping functions, we can give a detailed analysis which data is held in the row buffer. This explains how the side channel gets access to private information and how we can exploit this fact. We present the DRAM template tool to show how we can detect keystrokes without any noise.

4. **Covert channel,**
   The second attack we present is a covert channel which uses the row buffer. We show that the only shared resource required is the main memory.

5. **Covert channel in JavaScript,**
   Given the limitations of sandboxed JavaScript and the inability to execute native code, we demonstrate that it is still possible to implement the covert channel in a browser. We show how we circumvented all the limitations and present libdramajs which implements all necessary functions in JavaScript.

6. **Demo of covert channel from VM to JavaScript,**
   This demonstrates that we can send data from a VM without network hardware to the internet.

7. **Reverse engineering the CPU,**
   This explains the idea of how to map physical addresses to a geographical location on the DRAM purely in software. We introduce our automated reverse-engineering tool that anyone can use as a starting point for future attacks and to easily reproduce the presented attacks on their own computer.

8. **Impact on row hammer,**
   We explain how this helps to move rowhammer to DDR4 and how it improves the performance of existing rowhammer attacks. Furthermore, we explain how this helps other researchers do software fault inject attack on private keys.

9. **Closing remarks,**

Attendee Takeaways

1. Raise public awareness for the previously unknown DRAM attack vector and show that this is a highly realistic attack scenario.
2. Detail how to reverse-engineer hardware behavior without having physical access to the system.
3. We reveal an easy-to-use new open-source reverse-engineering tool.

What's New

We reveal an entirely new hardware side channel that is based on DRAM row hits and row conflicts. We show how it can be exploited purely in software – even in sandboxed environments.

Why Black Hat?

Our DRAM side channel is an entirely new attack vector that was neither known nor exploited previously. The reverse-engineering and subsequent attacks work on smartphones, personal computers and servers alike and even in virtual machines running in cloud environments. Black Hat attendees will find that they work on their systems as well and are scaringly easy to reproduce. Our reverse-engineering tool is open source and can be used by anyone to attack and research the security of such systems.

Changes & Updates if this talk was presented before

The academic paper (http://arxiv.org/abs/1511.08756) on the theoretic perspective will be presented at USENIX Security 2016 (`https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl`).

This BlackHat talk will focus on the reverse-engineering tool that we will present here and practical applications of the side channel. One new application we present is: Using the side channel to send data to the internet from a VM with no network hardware. In this attack, no native code or binary is running on the host system or any other VM.

Bibliography

[1] Aater Suleman. What every Programmer should know about the memory system, 2011. URL http://www.futurechips.org/chip-design-for-all/what-every-programmer-should-know-about-the-memory-system.html. (cited on p. 30)

[2] Advanced Micro Devices. AMD-V™ Nested Paging, July 2008. URL http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf. (cited on p. 43)

[3] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2013. URL: http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf. (cited on p. 7)

[4] AMD. (BKDG) for AMD Family 15h Models 10h-1Fh Processors, 2015. URL http://support.amd.com/TechDocs/42300_15h_Mod_10h-1Fh_BKDG.pdf. (cited on p. 10)

[5] AMD. AMD FX processors, 2016. URL http://www.amd.com/en-us/products/processors/desktop/fx. (cited on p. 1)

[6] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009. (cited on p. 35)

[7] ARM. *L210 Cache Controller Technical Reference Manual*. 2006. (cited on p. 12)

[8] ARM. *ARMv8 Instruction Set Overview*. 2011. (cited on p. 1)

[9] ARM. *ARM® Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. 2012. (cited on p. 27)

[10] ARM. Cache coherency, 2015. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/BABIJAAI.html. (cited on p. 27)

[11] ARMDeveloper. Cache maintenance operations, 2016. URL https://developer.arm.com/docs/dui0646/latest/4-cortex-m7-peripherals/48-cache-maintenance-operations. (cited on p. 27)

[12] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. Cain: Silently breaking aslr in the cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015. (cited on p. 35)

[13] Bartosz Milewski. Virtual Machines: Virtualizing Virtual Memory, Dec. 2011. URL https://corensic.wordpress.com/2011/12/05/virtual-machines-virtualizing-virtual-memory/. (cited on p. 43)

[14] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. DOI 10.1145/1346281.1346286. URL http://doi.acm.org/10.1145/1346281.1346286. (cited on p. 43)

[15] J. N. Burghartz. *Guide to State-of-the-Art Electron Devices*. Wiley-IEEE Press, 2013. ISBN 1118347269. (cited on p. 6)

[16] C++ 2011. std::chrono::high_resolution_clock, 2011. URL http://en.cppreference.com/w/cpp/chrono/high_resolution_clock. (cited on p. 42)

[17] S.-J. Cho, U.-S. Choi, Y.-H. Hwang, and H.-D. Kim. Design of new XOR-based hash functions for cache memories. *Computers & Mathematics with Applications*, 55(9):2005–2011, may 2008. DOI 10.1016/j.camwa.2007.07.008. URL http://dx.doi.org/10.1016/j.camwa.2007.07.008. (cited on p. 22)

[18] Chromium. window.performance.now does not support sub-millisecond precision on Windows, Dec. 2015. URL https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110. (cited on p. 47, 50)

[19] T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012. (cited on p. 46)

[20] Dan Magenheimer. TSC Mode How-To, 2013. URL http://xenbits.xen.org/docs/4.3-testing/misc/tscmode.txt. (cited on p. 41)

[21] Francis J. Narcowich. Methods for Finding Bases, 2016. URL http://www.math.tamu.edu/~fnarc/psfiles/find_bases.pdf. (cited on p. 25)

[22] GCC. Other Built-in Functions Provided by GCC, 2016. URL https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. (cited on p. 23)

[23] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. (cited on p. 1)

[24] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 154–165. IEEE, 2003. (cited on p. 9)

[25] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. *arXiv preprint arXiv:1507.06955*, 2015. (cited on p. 48, 49)

[26] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating

attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015. (cited on p. 1, 12, 13, 54)

[27] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. DIMVA, 2016. (cited on p. 1, 12, 13, 35)

[28] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505, May 2011. DOI 10.1109/SP.2011.22. (cited on p. 1, 12)

[29] J. Handy. *The cache memory book*. Morgan Kaufmann, 1998. (cited on p. 12)

[30] M. Hassan, A. M. Kaushik, and H. Patel. Reverse-engineering embedded memory controllers through latency-based analysis. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. Institute of Electrical & Electronics Engineers (IEEE), apr 2015. DOI 10.1109/rtas.2015.7108453. URL `http://dx.doi.org/10.1109/RTAS.2015.7108453`. (cited on p. 27)

[31] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728. (cited on p. 11)

[32] R.-F. Huang, H.-Y. Yang, M. C.-T. Chao, and S.-C. Lin. Alternate hammering test for application-specific DRAMs and an industrial case study. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 1012–1017, 2012. (cited on p. 2)

[33] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013. (cited on p. 12, 22, 49)

[34] Hyesoon Kim. High-Performance Computer Architecture, 2011. URL `http://www.cc.gatech.edu/~hyesoon/fall11/`. (cited on p. 9)

[35] S. Hynix. *H9CKNNN8GTMPLR LPDDR3-S8B 8Gb(x32) Datasheet*. 2013. (cited on p. 32)

[36] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Technical report, IACR Cryptology ePrint Archive, 2015. (cited on p. 1, 12, 22, 57)

[37] Intel Corporation. Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d, June 2009. URL `https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d`. (cited on p. 43)

[38] Intel Corporation. *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors*. 2009. (cited on p. 11)

[39] Intel Corporation. *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. September 2010. (cited on p. 16, 17)

[40] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. Number 325383-057US. December 2015. (cited on p. 16)

[41] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-Z*. Number 253667-057US. December 2015. (cited on p. 16)

[42] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-032. January 2016. (cited on p. 11, 17)

[43] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. *Wait a Minute! A fast, Cross-VM Attack on AES*, pages 299–319. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11379-1. DOI 10.1007/978-3-319-11379-1˙15. URL http://dx.doi.org/10.1007/978-3-319-11379-1_15. (cited on p. 35)

[44] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 629–636. IEEE, 2015. (cited on p. 12, 22)

[45] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007. ISBN 0123797519. (cited on p. 6, 9, 11, 17, 30)

[46] JEDEC. Standard No. 79-3F. DDR3 SDRAM Specification., 2012. URL https://www.jedec.org/standards-documents/docs/jesd-79-3d. (cited on p. 6)

[47] JEDEC. Standard No. 79-4A. DDR4 SDRAM Standard, 11 2013. URL http://www.jedec.org/standards-documents/results/jesd79-4a. (cited on p. 6)

[48] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014. (cited on p. 2, 9, 10)

[49] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture – ISCA*, pages 361–372, 2014. (cited on p. 2)

[50] Kirill A. Shutemov. pagemap: do not leak physical addresses to non-privileged userspace, 2015. URL https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce. (cited on p. 27, 38)

[51] Leslie Xu. Securing the enterprise with intel AES-NI, 2010. URL http://www.intel.com/content/www/us/en/enterprise-security/enterprise-security-aes-ni-white-paper.html. (cited on p. 1)

[52] W.-F. Lin, S. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. Institute of Electrical & Electronics Engineers (IEEE), jan 2001. DOI 10.1109/hpca.2001.903272. URL `http://dx.doi.org/10.1109/HPCA.2001.903272`. (cited on p. 27)

[53] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard. ARMageddon: Last-Level Cache Attacks on Mobile Devices. *ArXiv e-prints*, Nov. 2015. (cited on p. 13, 27, 29, 58)

[54] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015. (cited on p. 1, 12, 22)

[55] Mark Lanteigne. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware, 2016. URL `http://www.thirdio.com/rowhammer.pdf`. (cited on p. 3)

[56] Mark Seaborn. How physical addresses map to rows and banks in DRAM, May 2015. URL `http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html`. (cited on p. 7, 16, 21, 23, 26, 30, 31, 54, 59)

[57] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges., 2015. URL `http://googleprojectzero.blogspot.co.at/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`. (cited on p. 2, 9, 10)

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

[59] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses*, pages 48–65. Springer, 2015. (cited on p. 1, 12, 22)

[60] Moritz Lipp. libflush, 2016. URL `https://github.com/IAIK/armageddon`. (cited on p. 29)

[61] Mozilla Developer Network. SharedArrayBuffer, May 2016. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer`. (cited on p. 58)

[62] D. Mukhopadhyay and R. S. Chakraborty. *Hardware Security: Design, Threats, and Safeguards*. Chapman & Hall/CRC, 1st edition, 2014. ISBN 143989583X, 9781439895832. (cited on p. 41)

[63] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH*

Computer Architecture News, volume 36, pages 63–74. IEEE Computer Society, 2008. (cited on p. 9)

[64] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015. (cited on p. 47)

[65] D. Osvik, A. Shamir, and E. Tromer. Full aes key extraction in 65 milliseconds using cache attacks. In *Crypto*, 2005. (cited on p. 1)

[66] K. Park, S. Baeg, S. Wen, and R. Wong. Active-Precharge Hammering on a Row Induced Failure in DDR3 SDRAMs under 3x nm Technology. In *Proceedings of the 2014 IEEE International Integrated Reliability Workshop Final Report (IIRW'14)*, pages 82–85, 2014. (cited on p. 2)

[67] C. Percival. Cache missing for fun and profit, 2005. (cited on p. 1)

[68] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. Reverse Engineering Intel DRAM Addressing and Exploitation. *ArXiv e-prints*, Nov. 2015. (cited on p. 30, 35, 36, 51, 59)

[69] F. P. Philip J. Boland. The reliability of k out of n systems. *The Annals of Probability*, 11(3):760–764, 1983. ISSN 00911798. (cited on p. 46)

[70] R. Qiao and M. Seaborn. A new approach for rowhammer attacks. 2016. URL `http://seclab.cs.sunysb.edu/seclab/pubs/host16.pdf`. (cited on p. 9)

[71] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. DOI 10.1145/1250662.1250709. URL `http://doi.acm.org/10.1145/1250662.1250709`. (cited on p. 12)

[72] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009. (cited on p. 2)

[73] Rajeev Balasubramonian. CS/ECE 7810 Advanced Computer Architecture - DRAM Basics, 2014. URL `http://www.eng.utah.edu/~cs7810/pres/11-7810-12.pdf`. (cited on p. 8)

[74] Red Hat. How to use, monitor, and disable transparent hugepages in Red Hat Enterprise Linux 6?, 9 2015. URL `https://access.redhat.com/solutions/46111`. (cited on p. 38)

[75] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM, 2000. (cited on p. 9)

[76] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, Ryan Wells. Power

Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge, 2012. URL `http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf`. (cited on p. 12, 32)

[77] C. E. Shannon. Communication in the presence of noise. *Proc. Institute of Radio Engineers*, 37(1):10–21, 1949. (cited on p. 41)

[78] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, page 1. ACM, 2011. (cited on p. 35)

[79] G. Teschl and S. Teschl. *Mathematik für Informatiker: Band 1: Diskrete Mathematik und Lineare Algebra (eXamen.press) (German Edition)*. Springer, 2008. ISBN 3540708243. (cited on p. 25)

[80] C. M. R. Thimmannagari. 2005. DOI 10.1007/b102502. URL `http://dx.doi.org/10.1007/b102502`. (cited on p. 12)

[81] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. (cited on p. 1, 12)

[82] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop - CCSW '11*. Association for Computing Machinery (ACM), 2011. DOI 10.1145/2046660.2046671. URL `http://dx.doi.org/10.1145/2046660.2046671`. (cited on p. 41)

[83] W3C. Web Workers, 2015. URL `https://www.w3.org/TR/workers/`. (cited on p. 58)

[84] W3C. High Resolution Time Level 2, Feb. 2016. URL `https://www.w3.org/TR/hr-time/`. (cited on p. 47)

[85] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *null*, pages 473–482. IEEE, 2006. (cited on p. 36)

[86] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 494–505. ACM, 2007. (cited on p. 1)

[87] H. S. Warren. *Hacker's Delight (2nd Edition)*. Addison-Wesley Professional, 2012. ISBN 0321842685. (cited on p. 23)

[88] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 159–173, Bellevue, WA, 2012. USENIX. ISBN 978-931971-95-9. URL `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu`. (cited on p. 36)

[89] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security implications of memory

deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, June 2013. DOI 10.1109/DSN.2013.6575349. (cited on p. 35)

[90] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011. (cited on p. 36)

[91] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`. (cited on p. 1, 12, 35, 51)

[92] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. Mapping the intel last-level cache. Technical report, Cryptology ePrint Archive, Report 2015/905, 2015. (cited on p. 57)

[93] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *security and Privacy (SP), 2011 IEEE Symposium on*, pages 313–328. IEEE, 2011. (cited on p. 36)

[94] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012. (cited on p. 1)

[95] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014. (cited on p. 1)

[96] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 32–41. ACM, 2000. (cited on p. 27)

List of Figures

2.1 DRAM matrix structure. 6
2.2 DIMM 7
2.3 DRAM bank organization 8
2.4 The original rowhammer code 10
2.5 Cache slices 14

3.1 Relation between mapped memory and functions reverse engineered . 19
3.2 Timing histogram 20
3.3 Timing histogram with DRAM memory controller scheduling 22
3.4 A sample physical memory layout of a 64bit machine with 8GB memory. 24
3.5 A sample physical memory layout of an ARMv8 device with 2GB memory. 28
3.6 Bit flip rate on Nexus 5 30
3.7 The basic scheme of a physical address 31
3.8 Bits of the physical address for a Ivy Bridge CPU with 2 channels and 1 DIMM with 4GB per channel. 32
3.9 Bits of the physical address for a ARMv7 CPU with 1 channel and 1 DIMM with 2GB (Nexus 5). 33

4.1 Covert channel principle 37
4.2 Histogram of access times 40
4.3 Covert channel measurement 42
4.4 Guest virtual to host physical using nested page tables 44
4.5 Address translation using shadow page tables 45
4.6 Covert channel tranmission rate 46
4.7 Page borders 48
4.8 Cache eviction loop 49
4.9 Eviction strategy on Ivy Bridge 50
4.10 Row hit attack 51
4.11 Skylake mapping between page and row 53
4.12 Histogram of row hits and row conflicts 55
4.13 Access times of keystrokes 56

List of Tables

3.1 Experimental setups. 31
3.2 Reverse engineered DRAM mappings (Intel) 31
3.3 Reverse engineered DRAM mappings (ARM) 33
3.4 Approximate timings to reverse engineer a platform. 34

4.1 Resolution of `window.performance.now()` on various browsers and operating systems 47
4.2 Our test machines and the corresponding spatial accuracy. 54