Jonas Juffinger

# Rowhammer Exploits are still possible

**Master's Thesis**

Graz University of Technology

Institute for Applied Information Processing and Communications

Advisor: Daniel Gruss

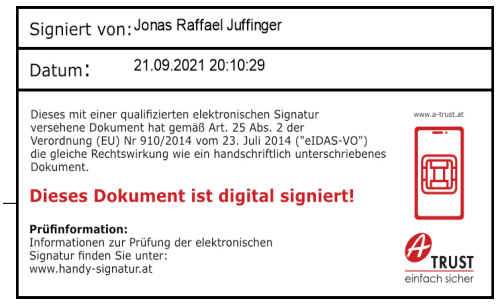
Graz, September 2021

## Statutory Declaration

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 marked all material which has been quoted either literally or by content from the used sources.

Graz, _____

Date




## Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

# Abstract

In this thesis, we present a novel Rowhammer exploit that uses a new set of techniques to attack recent of the shelf devices with active Rowhammer countermeasures reliably. Rowhammer is a hardware bug that breaks memory isolation by modifying memory locations that are not accessible to the attacking process. We categorize Rowhammer countermeasures proposed and implemented by researchers and manufacturers and explain their strengths and weaknesses. To study existing exploits we define four steps for a successful attack, *memory preparation*, *data preparation*, *hammering*, and *exploitation* for which many exploits use unique techniques that we describe in detail.

We mainly focus on recent Chromebooks, for which we developed novel solutions for all four previously defined exploit steps. For *memory preparation* we combine a timing side channel on the DRAM with the DRAM's mapping functions to detect contiguous memory areas and reverse-engineer physical address bits without requiring huge pages. To allow page table spraying in the small virtual address space of our target we use spray children for *data preparation*. Our target device uses DRAM protected with TRR and ECC to defend against Rowhammer. We use the novel half-double Rowhammer pattern to break TRR in the *hammering* step. Finally, we increase the stability of the *exploitation* with robust bit flip verification via speculative execution.

With these novel techniques our exploit is a real threat to Chrome OS's security. It can compromise a fully patched Chrome OS system in under two hours and gain full read and write access to the system's memory.

**Keywords:** Rowhammer, exploit, security, side channel, speculative execution, Linux, Chrome OS

# Kurzfassung

In dieser Arbeit präsentieren wir einen neuartigen Rowhammer Exploit der mit Hilfe neuer Techniken ein Chromebook mit aktivierten Rowhammer Gegenmaßnahmen zuverlässig übernehmen kann. Rowhammer ist eine Sicherheitslücke in der Hardware die die Modifikation von Speicherstellen erlaubt auf die der attackierende Prozess keinen Zugriff hat und somit die Speicherisolation untergräbt. Wir kategorisieren Rowhammer Gegenmaßnahmen die von Forschern und Herstellern vorgeschlagen und implementiert wurden und diskutieren Stärken und Schwächen dieser. Zur Untersuchung existierender Exploits definieren wir vier Schritte die für eine Attacke notwendig sind, *Speichervorbereitung*, *Datenvorbereitung*, *Hämmern*, *Ausnutzung*. Viele Attacken verwenden einzigartige Techniken für diese Schritte die wir im Detail beschreiben.

Für unseren Exploit haben wir auch vier neue Lösung für die vier Schritte entwickelt um unser Chromebook attackieren zu können. Für die *Speichervorbereitung* kombinieren wir eine zeitbasierte Seitenkanalattacke zusammen mit den DRAM Addressierungsfunktionen unseres Chromebooks um zusammenhängende Speicherbereiche zu erkennen und physikalische Addressbits extrahieren zu können. Um Page-Table-Spraying im kleinen virtuellen Addressraum unseres Chromebooks zu ermöglichen verwenden wir Spray-Children zur *Datenvorbereitung*. Der Speicher des Chromebooks verfügt über die Rowhammer Gegenmaßnahme TRR. Wir verwenden die neuartige Half-Double Rowhammer Technik um TRR beim *Hämmern* zu umgehen. Am Ende verwenden wir zum *Ausnutzen* der Speichermodifikation robuste Bitflip Verifikation durch spekulative Ausführung, um die Attacke stabiler zu machen.

Mit Hilfe dieser neuartigen Techniken ist unser Exploit ein reale Bedrohnung für die Sicherheit von Chrome OS. Er kann ein komplett gepatchtes Chromebook in unter zwei Stunden übernehmen and vollen Schreib- und Lesezugriff auf den Speicher erhalten.

**Stichwörter:** Rowhammer, Attacke, Sicherheit, Seitenkanalattacke, Spekulative Ausführung, Linux, Chrome OS

# Acknowledgements

I owe several contributors to this thesis my thanks. First and foremost, I want to thank my supervisor Daniel Gruss for giving me the excellent opportunity to work on this topic. You passed on your joy in research to me and taught me a lot during the reviews of this thesis.

I want to thank Moritz Lipp and Andreas Kogler. Without Moritz's knowledge about speculative execution attacks, the exploit would be a lot less performant and Andreas was of great help for the thesis.

I also want to thank Salman Qazi, Kim Yoonku, Eric Shiu and Matthias Nissler from Google. Firstly, for the interesting project and secondly for the numerous ideas to improve the exploit.

Finally, I want to thank my significant other, Sarah, as well as my Parents, Franziska and Christian. You always supported me in pursuing my dreams and achieving this degree would not have been possible without you.

Thank you.

Jonas Juffinger

# Contents

# Chapter 1

# Introduction

Rowhammer is a widespread security issue caused by the perpetual pressure to miniaturize electronic devices that was first discovered by Kim *et al.* in 2014 [52]. Rowhammer allows an attacker to intentionally cause disturbance errors in the DRAM to flip bits that are not part of its memory. Manufacturers knew about the existence of disturbance errors in the DRAM prior to the findings of Kim *et al.* [52] but it was only viewed as a reliability and not a security issue [52]. The ever higher demand in memory density and lower power consumption leads to an ongoing miniaturization of DRAM cells which in turn increases the vulnerability to Rowhammer [65].

A year after the discovery, Seaborn *et al.* [75] presented the first two Rowhammer exploits showing that Rowhammer is a real security threat that can be used to gain kernel privileges. Since then, numerous exploits and defenses were developed. The spectrum of targets attacked by these exploits spans nearly all computer systems. They were shown to be viable from Javascript, to attack mobile phones, break out of virtual machines, read memory, attack remotely over the network, use Intel SGX for hiding and many more [21, 27, 29–31, 41, 55, 58, 72, 75, 78, 80]. Also, the countermeasures that were shown throughout the years to defend against Rowhammer utilize many different techniques, from the detection and prevention of Rowhammer attacks in software or hardware to the physical isolation between the kernel and userspace memory [20,25,29,32,42,45,51,52,54]. However, every known defense was broken by at least one published exploit, therefore the problem of Rowhammer remains unsolved.

In this master's thesis, we assess the current situation regarding the threat of Rowhammer, with its ongoing conflict between new countermeasures and exploit techniques. To show that the problem is not solved yet, we present a novel Rowhammer exploit to successfully attack a recent device running the operating system Chrome OS [3]. We discuss the latest challenges of mounting such an attack and provide solutions to all of them. Finally, we try to conclude our work with an optimistic outlook on the future.

Chrome OS does not allow the user to run native applications directly on the system as it was initially designed to be a web-browser-only operating system [67]. It does however support Android apps fully since 2016, which can be used to run native code with the Android NDK [35]. Android apps run inside Linux containers directly on top of the Chrome OS kernel [33].

The LPDDR4x memory in our target device uses ECC error correction and the Rowhammer countermeasure TRR. ECC hinders memory templating because it makes bit flips partially dependent on the content of the victim row. To counter this problem, we use a new approach to hammer page tables without requiring any memory templating. TRR, on the other hand, was supposed to solve Rowhammer, but Frigo *et al.* [31] discovered that it could be broken with many-sided Rowhammer and Qazi *et al.* [71] presented another method called half-double Rowhammer. Our exploit is the first to utilize this novel half-double Rowhammer pattern.

Physically contiguous memory is essential for the half-double pattern, and our exploit supports two ways to obtain it. The first one is to use huge pages which are enabled in Chrome OS and also the Android container. This is trivial but can also easily be deactivated. The second technique uses a timing side channel together with the DRAM mapping functions to detect contiguous memory. It can further recover additional physical address bits to circumvent row scrambling.

Hammering the DRAM can have undesired consequences that can kill our exploit. To prevent this, we use a novel technique to verify bit flips with speculative execution. After obtaining full access to the whole memory, we dump the memory to disk in a matter of seconds, to scan it for private information or secret keys in a later step.

**Chapter 2** provides the fundamental knowledge the following chapters of the thesis build upon. It gives an overview of the DRAM, which is the target of the Rowhammer attack. To understand Rowhammer, knowledge about the structure of the DRAM, its operation and the design of the DRAM cells are an advantage. The chapter also describes the basics of virtual memory and how it is managed with page tables. It then gives an overview of Rowhammer, its physical root cause and how to trigger it from software. The final section covers speculative execution, which we use for our robust bit-flip verification.

**Chapter 3** gives an overview of existing Rowhammer countermeasures. It categorizes them based on their method used to defend against Rowhammer: *detection*, *neutralization* and *elimination*.

**Chapter 4** explains in detail the different techniques and concepts used by various Rowhammer exploits. The chapter describes different ways to obtain contiguous memory and how to bypass the CPU caches depending on available memory mapping options and instructions. We describe the different hammer patterns with their respective properties and provide details on ECC and TRR memory. We finally lay out different ways to use Rowhammer bit flips for privilege escalation and sandbox escapes.

**Chapter 5** presents our aforementioned novel half-double Rowhammer exploit targeting Chrome OS. We discuss the different options to run applications on Chrome OS and our choice of an Android app for our exploit. We outline the challenges our exploit is facing and detail our solution to all of them. Finally, we evaluate the real-world danger of the exploit and discuss possible countermeasures.

**Chapter 6** provides a conclusion and summary of this work.

# Chapter 2

# Background

In this chapter we, provide the necessary background knowledge to understand Rowhammer exploits and defenses. Section 2.1 covers DRAM, the main memory of every computer and the target of the Rowhammer attack. The section details the DRAM's structure, the inner workings of a single cell and the interface between CPU and DRAM. Section 2.2 covers the CPU caches. They are essential for the performance of modern computers but also an obstacle for Rowhammer. They can additionally be used as a side channel for, among others, speculative-execution attacks. Another essential building block of a computer's memory system is virtual memory, which is explained in more detail in Section 2.3. It is the basis for process isolation in every OS and an attack target in some Rowhammer exploits. Section 2.4 links all previous sections to explain the Rowhammer vulnerability in detail, including recent findings regarding the electrical cause for the vulnerability, how it is triggered and the obstacles preventing it from being triggered. Finally, Section 2.5 covers speculative execution, a performance optimization present in all modern CPUs and the basis for the Spectre CPU vulnerability [53]. Though not directly linked to Rowhammer, it but can be used in exploits to improve the chances of success.

## 2.1 Dynamic Random Access Memory (DRAM)

DRAM is typically used as the main memory of every computer, containing all actively running programs and their data, including the operating system. The reason for its wide usage is its high memory density paired with low production costs. The high memory density, which is ever more important to combat the increasing memory consumption of present-day software, is achieved by storing a single bit in a cell that consists of only two components, a transistor and a capacitor (cf. Section 2.1.2).

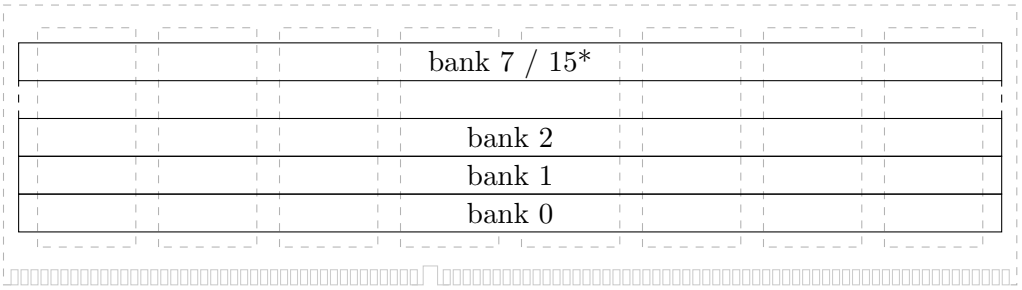



**Figure 2.1:** A DRAM rank is structured into banks.
DDR3 memory has 8 banks, DDR4 memory has 16 banks divided into groups. The dotted squares symbolize the memory chips.

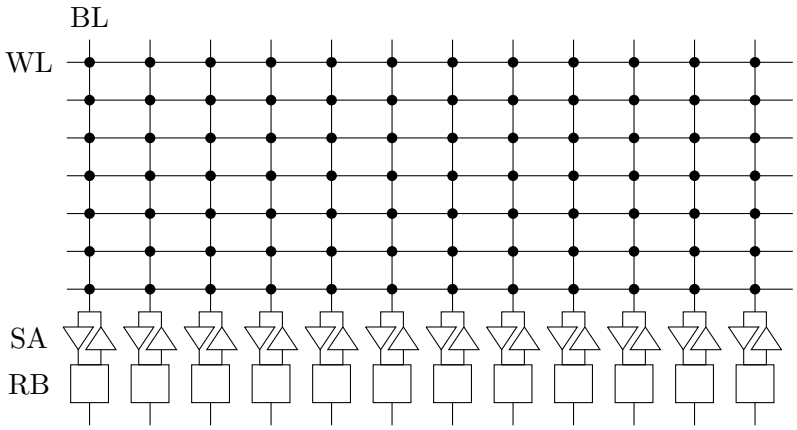



**Figure 2.2:** The DRAM cells are structured in a grid.

### 2.1.1 DRAM Structure and Operation

DRAM consists of billions of memory cells. To efficiently address, read and write them, the memory is structured into multiple levels. The highest level is a DIMM, the lowest a cell that holds a single bit.

A computer can have multiple DIMMs connected to the motherboard. Every physical side of these modules is called a rank. DIMMs can have memory chips on one or both sides and therefore one or two ranks. A rank is split into multiple banks as shown in Figure 2.1. In DDR4 memory, the banks are logically divided into two or four bank groups. Accesses to different rows in the same bank are slow because of additional delays caused by closing and opening rows. These banks work independently, which can decrease access times. The figure also shows how the banks are distributed over the typical eight memory chips of non-ECC memory.

In a bank, cells are organized in a grid with word lines (WL) connecting the cells horizontally and bit lines (BL) connecting the cells vertically with the sense amplifiers (SA)

of the row buffer (RB) at one end. The bank structure is shown in Figure 2.2, showing cells as dots at each crossing of a word line and a bit line.

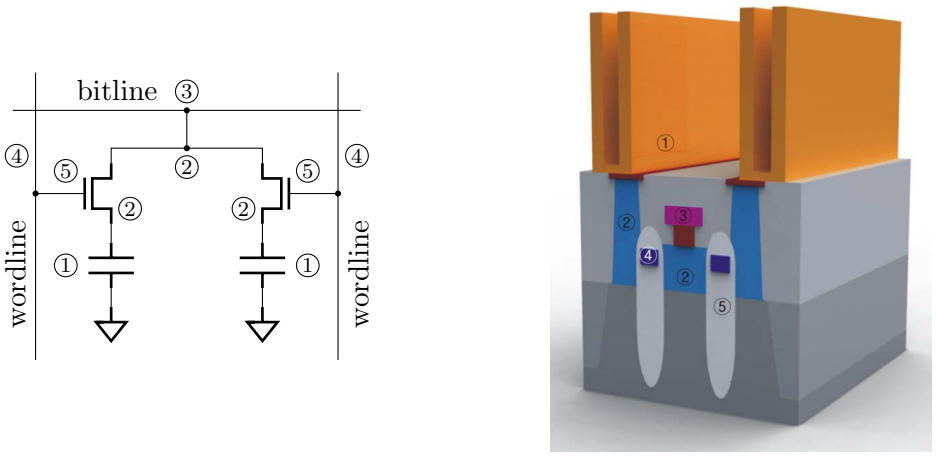
Mapping functions are used to map the physical address of a memory location to a specific channel, rank, bank and row (Section 2.1.3). These mapping functions are different in most CPU-architectures and usually not published. To access a memory location, the word line corresponding to the row is activated, connecting all cells' capacitors in this row with the sense amplifiers of the row buffer. This process is called *opening a row* and the row that is in the row buffer is called the *active row*. The tiny capacitors, with a capacity of less than 10 fF, are completely discharged when sensed by the amplifiers of the row buffer, erasing the information from the row, it is therefore a destructive read. Subsequent reads from the same row are then served directly from the row buffer, increasing performance. If however, data stored in another row is requested, the currently active row must be restored by writing its data back from the row buffer (row closing). Only then a new row can be activated and moved into the row buffer. As a consequence, reading from two different rows on the same bank also leads to writes to these rows.

The accessed row stays activated on most CPUs until another row is activated to benefit of the faster access to an active row. This method is only effective if multiple accesses with close proximity can be expected, which is usually the case with normal desktop computers with a low number of cores. However, it can lead to worse performance if the majority of subsequent accesses target varying rows because then every access has the additional latency from closing the row. In these cases, it is faster to close the rows after every access immediately. This is called a *closed-row policy*, in contrast to the *open-row policy* explained prior, and it is used in many CPUs with a high core count.

### 2.1.2 DRAM Cell Structure

To understand the Rowhammer vulnerability, a closer look at the silicon structure is required. A single bit is stored in one cell, which consists of a MOSFET and a capacitor [77]. The ongoing process optimization of these cells, lead to a design where two cells share one active region on the silicon. Figure 2.3a shows the schematic of one active region of this design. Figure 2.3b shows a rendering of the actual structure in silicon. The orange area ① shows the two capacitors of the cells. Their height in this rendering is not to scale. The blue areas ② are the drain and source of the two transistors. The bit line is shown in purple ③ and connected to the two transistors. The two word lines in dark blue ④ are embedded into the two gates in gray ⑤.

The long vertical oval-like shape of the gates allows for a long channel length with a smaller footprint when compared with the classical planar MOSFET design. MOSFETs with very short channels have a higher sub-threshold current which leads to a faster draining of the capacitors which cannot not be tolerated for DRAM memory [77]. Even with the optimized channel shape the small leakage current can significantly impact the stored charge in a short amount of time, because the capacitors are tiny, with capacitance values below 10 fF. As a result, DRAM cells must be refreshed periodically to keep the




**(a)** Schematic **(b)** 3D view from Shin *et al.* [76]

**Figure 2.3:** A single DRAM cell consists of a MOSFET and a capacitor. Two cells share an active region in the silicon.

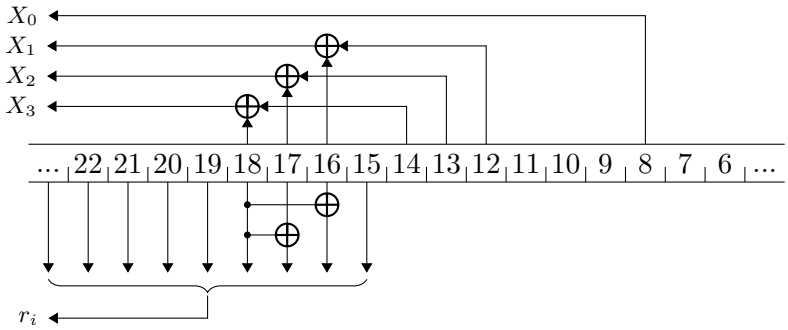



**Figure 2.4:** The reverse engineered DRAM mapping functions of our target device.

capacitors above their digital `1` threshold voltage [82].[1] This is why this memory type is called *dynamic* random access memory in contrast to *static* random access memory (SRAM) that does not require refreshes to keep its content. Most modern DRAM modules have a refresh interval of 64 ms meaning that every cell is refreshed every 64 ms. The rows are not refreshed simultaneously but successively by rotating through all rows and refreshing one every few microseconds.

### 2.1.3 DRAM Mapping

The division of the memory into many levels, from rows to channels, increases the parallelism of memory accesses and therefore the memory throughput.

[1] A charged capacitor can denote a `0` or a `1`, as discussed in Section 2.4.1.

To make use of the parallelism in the DRAM structure as efficiently as possible, the accesses to the different rows should be distributed seemingly randomly. CPU manufacturers developed the so called *DRAM mapping functions* that map physical addresses to banks, ranks and channels seemingly randomly. These functions are different for most CPU families and usually not published by the CPU manufacturers [66].

Knowledge of this mapping can however enable a variety of attacks like a cross-CPU row-buffer covert channel [66] and Rowhammer attacks [50, 83]. DRAMA [66] is a tool to reverse engineer the mapping by probing a large contiguous memory area for timing differences due to row-buffer hits and conflicts. Figure 2.4 shows the reverse-engineered mapping of our target device, a recent Chromebook with a MediaTek 8183 CPU. We will use this mapping in our exploit (cf. Section 4.1.2) to detect physically contiguous memory blocks and obtain additional physical address bits of the pages in these contiguous memory areas. With these additional physical address bits we can almost completely circumvent the row-scrambling Rowhammer countermeasure implemented in our target device.

### 2.1.4 Error Correcting Code (ECC)

For many critical applications like server infrastructure, medical devices, or financial services, random memory errors can have a huge negative impact. The reasons for these errors are manifold, ranging from cosmic rays [44] to the aging of memory cells [73]. Schroeder *et al.* [73] studied the effects of random errors on computer systems by observing the majority of Google's servers for 2.5 years from 2006. They concluded that over 99.7% of all random errors in a DIMM are correctable by ECC memory [73] making ECC memory an effective countermeasure against the low volume of random memory errors. While ECC memory was used almost only in servers and specialized hardware a few years ago, newer memory standards like LPDDR4 and LPDDR4x brought ECC memory into everyday mobile devices. One reason for the inclusion of ECC memory in mobile devices is the lower power consumption. The error correction allows the refresh rate to be decreased to approximately one quarter of what is specified for the same memory module without ECC [63].

## 2.2 CPU Caches

The DRAM is too slow to keep up with the memory read and write operations performed by the CPU under a normal workload. To counter this imbalance, CPUs come with integrated caches that are a lot smaller than the DRAM, but also faster and can therefore provide faster access to recently accessed data to the CPU. Inside the CPU the caches are further divided into typical two or three levels with increasing size and latency [18]. Additionally, there are special purpose caches like the translation lookaside buffers (TLB) [18].

Caches are divided into cache lines. On ARMv8 their size is variable, typical is a size of 64 B, e.g., in the CPU of our target device [17,19]. The L1 cache is split into two caches, one I-cache containing instructions and one D-cache for data. The most simple cache implementation would map each memory block to exactly one cache line (direct mapping). This can lead to cache trashing, rendering the cache practically useless. Therefore, ARMv8 implements set-associative caches where each memory block is mapped to a set which consists of multiple ways. Inside a set, the cache replacement policy decides which cache line to evict if all ways are filled [18].

The CPU also contains other caches, for example, the TLB to store recently resolved virtual-to-physical page translations. In some cases, programs must be able to flush cache lines from various caches, e.g., because ARM does not guarantee coherence between L1 data and instruction caches [18]. For this, CPUs provide cache maintenance instructions for, e.g., self modifying code.

## 2.3 Virtual Memory

A computer runs many processes simultaneously that all share the resources of this computer, including its DRAM. This imposes the risk of processes manipulating the memory of other processes or the operating system, with the consequence of crashes and data corruption. To prevent this, the concept of virtual memory was developed. In systems with virtual memory, every process has its own memory area which is usually a lot larger than the actual available physical memory. On ARMv8, the virtual address space is currently implemented with a size of $2^{48} = 256\,\text{TB}$ but can be increased to $2^{52} = 4\,\text{PB}$ with a architecture extension [16]. Every process is allowed to use every byte of its virtual memory and it's the task of the operating system and the CPU to map virtual memory to physical memory on demand. To do this, the memory is divided into smaller parts, so-called pages, with special data structures called page tables.

### 2.3.1 Memory Paging

Memory paging is a memory management concept where the memory of a computer is split up into pages. Every physical page can be mapped to a page in the virtual memory of one or multiple processes, making this virtual memory page usable. The CPU and operating system store additional information for every page, for example, whether the page is accessible from the user space or only the kernel, whether the page can be read, written and executed or whether it has been accessed recently. Pages can also be backed by a file on a computer's disk. The smallest page size depends on the CPU architecture and its configuration. x86-64 supports only a 4 kB pages as the smallest page size [14], AArch64 on the other hand supports 4 kB, 16 kB and 64 kB pages [18].

**Demand Paging.** When a process wants to use a virtual memory area, it has to ask the operating system to map it into the physical memory. At this point, however, the operating system does not map any physical memory pages. It only extends the list

of reserved virtual addresses of the calling process. When this process later actually accesses the requested memory, the CPU raises an interrupt because the virtual address is not mapped. The operating system catches this interrupt and checks if the memory was reserved by the process in which case it maps the page to physical memory and continues the process's execution. This procedure is called *demand paging* and happens completely transparent to the process. *Demand paging* allows all processes on a computer to reserve more virtual memory than there is physical memory, as long as not every process is actually using all the memory. It also increases performance and reduces memory consumption because unused pages are never mapped.

**File-Backed Pages.** File-backed pages reflect the content of a file on a disk of the computer. Processes can map files into their virtual address space and access it through memory read and write operations. This allows for easy file access by the process and the underlying CPU. Binaries and libraries of running processes are mapped into their virtual address space and executed from there. Together with demand paging, file-backed pages add more flexibility to the management of physical memory for the operating system. Because their content is always retrievable from the disk, they can be removed from the physical memory whenever needed, for example, when not enough memory is available. When they are accessed again by the process, the CPU raises an interrupt and the operating system simply reloads the page from the disk.

**Shared Pages.** Another advantage of memory paging is that a single physical page can be mapped into the virtual memory of multiple processes or in the virtual memory of one process more than once. This is used for all files mapped read-only by multiple processes, e.g., shared libraries or program binaries that are started more than once. However, it is also used to share pages between processes for fast data exchange.

All these benefits and the flexibility made memory paging the de facto standard for all computers. But the flexibility comes with the cost of a quite complex mapping system to be able to manage the millions of pages with their associated permission and property bits. This mapping configured via data structures called page tables.

### 2.3.2 Page Tables

Page tables are pages containing multiple page-table entries that store the mapping between virtual and physical memory pages, including additional information like access permissions and other properties of the respective pages. To allow the mapping of the possible 256 TB of virtual memory without too much memory overhead, the mapping is split up into multiple levels in a tree structure. x86-64 has four or five levels [14], AArch64 can have three to five [16]. Both architectures have PTEs that are 64-bit wide, resulting in a capacity of $^{4\,\mathrm{kB}}/_{64\,\mathrm{bit}} = 512$ PTEs in one page table.

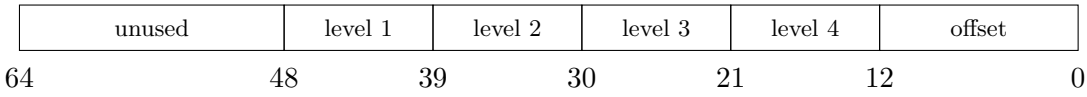



**Figure 2.5:** A virtual address consists of multiple indices of the page-table entry in each page-table level and an offset.

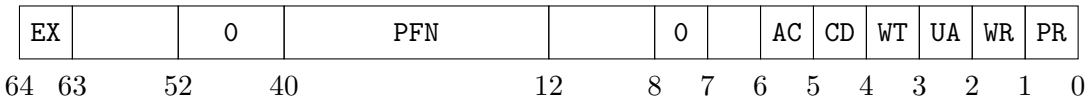



**Figure 2.6:** A x86-64 page-table entry containing the page-frame number and property bits.

The translation from virtual to physical addresses works by splitting up the virtual addresses into indices for the page table levels and a final offset within the page, as shown in Figure 2.5.

Every process has one level-1 page table. This first page table distinguishes the virtual memory areas of the processes. Bits 39 to 47 are the index of the entry within this level-1 PT, pointing to the physical page containing the level-2 PT. Bits 38 to 30 contain the index of the entry within the level-2 PT pointing to the level-3 PT. It continuous like this to the level-4 PT, which points to the page containing the data. When the MMU has to translate a virtual- to a physical address, it has to go through all four levels to find the physical address of the mapped page. This process is called a page-table walk and all modern CPUs cache recent translations in the so-called translation lookaside buffers to speed up this process.

### 2.3.3 Page-Table Entries (PTEs)

The page-table entries store mappings between individual virtual- and physical pages and additional properties. Figure 2.6 shows the format of page-table entries used in 64-bit x86 CPUs [14]. The lower bits are used to store status and configuration information for the MMU and the operating system, cf. Table 2.1. The page-frame number of the physical page is stored in the PFN field.

### 2.3.4 Memory Management Unit (MMU)

To keep the added performance overhead by the virtual memory system as small as possible, its management and translation is performed by the CPU. All CPU instructions work with virtual addresses which are translated, transparently to the program and operating system, to physical addresses by the MMU. The MMU also checks various bits of the PTE such as read, write, execute, or user-accessible permissions and the present bit. It raises interrupts for the operating system if virtual addresses are not mapped or accesses are not permitted.

**Table 2.1:** Example property bits of a page-table entry and their meaning.

| | Bit | Meaning | Description |
|---|---|---|---|
| PR | 0 | present | If 0, not mapped to a physical page |
| WR | 1 | write | If 0, the page is read-only |
| UA | 2 | user accessible | If 0, the page is only accessible by the kernel |
| CD | 4 | cache disabled | If 1, all accesses to the page are not cached |
| EX | 63 | execution disabled | If 1, execution is disabled |

## 2.4 The Rowhammer Bug

DRAM cells are like gates in a CPU, under the constant pressure of miniaturization to increase memory density and reduce power consumption. That led to the effect now known as Rowhammer. The capacitors in the cells hold tiny charges that can be changed through electrical disturbances induced by adjacent cells.

The problem that cells are electrically coupled and accesses to one can flip neighboring cells was long known by DRAM manufacturers but was classified as a reliability issue and not a security vulnerability [52]. This shift in awareness came in 2014, when Kim *et al.* [52] showed that some cells can be flipped repeatably from software. Attacks exploiting random flips in the DRAM caused by, e.g., cosmic rays were already known back then and the ease of flipping bits with Rowhammer made them a lot more dangerous [36]. Figure 2.7 shows the double-sided Rowhammer pattern. The two red rows are the aggressors, the blue rows the victims. Accessing the aggressors repeatedly can cause bits in the victim rows to flip.

Since discovery of Rowhammer, a variety of exploits and countermeasures were presented by the research community targeting different CPU architectures, devices and applications.

### 2.4.1 The Physics Behind Rowhammer

Walker *et al.* [82] did the first in-depth analysis of the Rowhammer bug on the physical level. They identified two effects that are the main contributors to this phenomenon.

Figure 2.3b shows the 3D view of two DRAM cells that share an active region. Upon access to a cell, electrons are injected into the p-well and can wander to other storage nodes. Almost 70 % of the injected electrons are captured by the cell sharing the same active region and the nearest storage node not sharing the same active region collects about 30%. The second cause for Rowhammer is capacitive crosstalk between the word lines. Rowhammer can only discharge charged cells, this is obfuscated by bit flips from $0 \rightarrow 1$ and $1 \rightarrow 0$ because every DRAM uses *true-cells* and *anticells*. *True-cells* store a logic one by being charged and *anticells* by being discharged.

Walker *et al.* [82] argue that physical countermeasures are possible and also describe possible solutions in the paper. The most crucial element is to have the source and drain

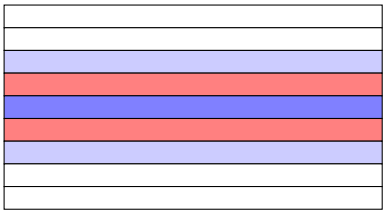


**Figure 2.7:** Depiction of the double-side Rowhammer pattern. The red rows are the aggressors, the blue rows the victims. In the dark blue row probability of a bit flipping is higher than in the light blue rows.

of the transistor not embedded in the same active region. This hinders the injected electrons from wandering freely to the storage nodes. The capacitive crosstalk can also be reduced by putting a metal shield between word lines. These changes have, however, also downsides. If the source and drain do not share the same active region, floating body effects arise that must be dealt with. To a metal should between the word lines, a different transistor design is required.

### 2.4.2 Triggering Rowhammer from Software

Accessing a row causes electrons to be injected into the P-well from the bit line. These electrons can wander into other storage capacitors and cause a change in charge large enough to flip the bit. For this to happen, a large number of accesses is required in one refresh period, which was named maximum-activation-count (MAC) by the DRAM manufacturers.

The last accessed row is usually cached in the row buffer to make subsequent accesses faster and more energy-efficient (Section 2.1.1). Because of that, alternating accesses to at least two addresses that lay in different rows in the same bank are required. The CPU, however, tries to make as few DRAM accesses as possible by caching recently accessed memory. So after the first access, all following accesses would be served from the cache and never reach the DRAM. To prevent this, different techniques like uncachable memory, cache flushing and cache eviction can be used (Section 4.3.1) in Rowhammer attacks.

All programs running on a computer work with virtual memory, which can be mapped anywhere in the physical memory. To access specific rows, the virtual-to-physical mapping or at least parts of it are required. Prior to the first Rowhammer exploit by Seaborn and Dullien [75], Linux had a unprivileged-user-readable proc-file with all virtual-to-physical translations, which is now only readable by `CAP_SYSADMIN` users [13].

```
1 for (i = 0; i < loops; ++i) {
2   *aggressor1;
3   *aggressor2;
4   flush(aggressor1);
5   flush(aggressor2);
6 }
```

**Listing 2.1:** The basic Rowhammer loop accessing and flushing two rows.

Newer exploits use huge pages, special DMA memory functions, memory massaging and contiguous memory detection, among others, to get (at least partial) information about the physical address.

Listing 2.1 shows a simple Rowhammer loop using a cache flush instruction. The two pointers `aggressor1` and `aggressor2` are first both accessed, causing each row they are pointing to, to be opened and closed. Afterward, both addresses are flushed from the CPU caches. When writing native code for x86 or ARMv8, this is easy because both instruction sets have cache flush instructions. On ARMv8 they are available from unprivileged code if enabled in the kernel, on x86 they are always available from unprivileged code and cannot be disabled. For attacks from non-native code like, for example, JavaScript specifically crafted access patterns can be used to evict the cache (cf. Section 4.3.1).

## 2.5 Speculative Execution

On the architectural level directly observable by a running program, the CPU executes every instruction in the strict order defined by the program. However, on the microarchitectural level, the CPU has multitude of optimizations to increase the performance, fully transparent to the running programs. Instruction have different latencies, often depending on the complexity. An ARM integer divide (`DIV`), for example, takes up more clock cycles than an integer add (`ADD`) instruction [18]. Instructions that access data from the DRAM have variable latencies because the data can be cached or not. To minimize the effect of these latencies on the execution speed of the program, CPUs can execute multiple instructions without any data dependency in parallel. They can also reorder independent instructions to fill all execution units as efficiently as possible and execute them out of order.

Another optimization targets conditional jumps like branch instructions which can depend on the delayed outcome of arithmetic instructions, memory accesses or returns from procedures that require the return address from the stack. In these cases, among others, CPUs speculate on the outcome of the branch or return instruction and execute instructions without knowing the actual outcome. When the required result becomes available and the prediction was not correct, the CPU reverts all speculatively executed instructions and continues with the correct path. If the prediction was correct, the CPU

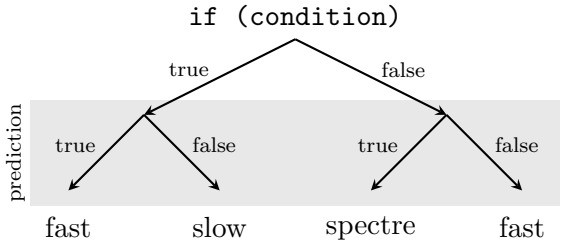



**Figure 2.8:** A branch condition can have two possible outcomes. Depending on the prediction, the execution can be fast or slow and lead to spectre.

saved clock cycles and improved the performance.

In 2017, five research groups discovered that speculative execution leaves microarchitectural traces that can be detected through side channels. This breaks the important security assumption that speculative execution has no impact on program execution except its performance. When memory loads are executed speculatively, the loaded values are cached. Due to that, the information if specific memory locations were accessed during speculative execution can be extracted afterward by measuring the time it takes to access the value. If it is below the threshold for cached accesses, the memory was accessed during speculative execution. This can be used maliciously to extract secret information like keys from other processes.

The CPU uses complex branch prediction circuits to execute the correct branch outcome as often as possible. These branch predictors consider the outcomes of previously executed branches and learn from these. Consequently, the branch predictors can be mistrained to execute branches speculatively in an attacker-controlled way. This mistraining is simplified by the implementation on all tested CPU that only uses parts of the virtual address to remember branches. So it is possible to mistrain the branch predictor in the attacker's processes and then have an attacker-controlled path taken in the victim process.

Listing 2.2 shows the most simple victim code of a conditional branch that can be exploited with spectre to leak an inaccessible value. `x` is an attacker-controllable value, e.g., user input and `array2` is shared memory with the attack process. The attacker then executes this code many times with values for `x` that are smaller than `array1_size`, training the branch predictor into believing that this condition is always true. After the training is done, the attacker flushes `array2` from the cache and sets `x` greater than `array1_size`. The CPU then executes the code speculatively which leads to a out-of-bounds read of `array1` and caches a cache line of `array2` depending on the out-of-bounds value. The attacker iterates over the pages in `array2` and measures the access time to retrieve the value. It is smaller for one index of `array2` because it was cached. This index equals the the value that was read outside the bounds of `array1`.

```
1 if (x < array1_size)
2     y = array2[array1[x] * 4096];
```

**Listing 2.2:** Example code of a conditionnal branch that can be exploited with spectre.

Branch prediction can also be used to speculatively execute gadgets to build custom chains of instructions similar to return-oriented programming. For this, the attacker first searches for gadgets in the victim binary and then builds the chain in his own memory, mimicking the virtual addresses of the gadgets in the victim. The branch predictor is then mistrained by executing the chain in the attacker process and is speculatively executed in the victim. The result is retrieved again through a side channel, e.g., the cache. This is very powerful because it also works on victims that do not contain instruction sequences like in Listing 2.2 that can be exploited directly.

In this thesis, we will present a new way to use speculative execution and exploit its exception suppression capabilities to make Rowhammer attacks more reliable when attacking ECC memory. (cf. Section 5.5).

# Chapter 3

# Rowhammer Countermeasures

Rowhammer is a real threat for computer systems, shown by the variety of exploits. This in turn, led to the development of many countermeasures to defend against Rowhammer exploits. They range from software that is detecting ongoing attacks to hardware changes and aim at every step in an exploit's run time [20, 25, 29, 32, 42, 45, 51, 52, 54].

Gruss *et al.* [39] identified three methodologies to classify Rowhammer countermeasures: *detection*, *neutralization* and *elimination*. *Detection* countermeasures examine, for example, performance counters or memory access patterns to detect ongoing Rowhammer attacks. Another method is to search binaries for suspicious instructions or instruction sequences often used by Rowhammer exploits. *Neutralization* countermeasures allow Rowhammer bit flips to happen, but ensure that they cannot be exploited. This is possible by isolating the memory of the attacker and the victim. *Elimination* countermeasures try to eliminate Rowhammer bit flips altogether. This requires a hardware change in most cases and is therefore not easily implemented on existing systems.

## 3.1 Detection-Based Countermeasures

Rowhammer attacks leave traces while running, like an unusual high cache miss rate and special memory access patterns. Some countermeasures use these characteristics to detect ongoing attacks and stop them. The other possibility is to try to detect malicious programs by analyzing the binary for suspicious instructions and instruction sequences.

### 3.1.1 Detection Through Runtime Traces

Rowhammer requires direct access to the DRAM and must therefore constantly bypass the caches either through special instructions or eviction sets. Every DRAM access is caused by a cache miss, which can be detected by the CPUs performance counters. Performance counters are a functionality of many CPUs that can be used by application

developers to find performance-related issues in their software, which among others can be frequent cache misses.

Seaborn *et al.* [75] already discussed the idea of using performance counters to detect Rowhammer attacks in their first exploit blog post. Aweke *et al.* [20] then presented a working implementation called ANVIL. What distinguishes a normal running application that causes many cache misses, because it is for example reading large amounts of memory, from Rowhammer is the high spatial locality of the Rowhammer accesses. A high spatial locality and a high cache miss rate are unlikely under normal circumstances by design. Another criteria for most Rowhammer attacks, except one-location Rowhammer, is that at least two rows in the same bank show these high cache miss rates. By taking these characteristics into consideration, ANVIL achieves a very low false-positive detection rate that translates into an average overhead of only 1.17 % with a 100 % detection rate on their test exploits.

To counter the attack, ANVIL accesses the neighboring rows of the aggressors, which refreshes the charges in the cells. Aweke *et al.* [20] note that a clever attacker cannot use the selective refresh to hammer other DRAM rows due to low selective refresh rate. This assumption was just proven wrong by Qazi *et al.* [71] with the discovery of half-double Rowhammer.

### 3.1.2 Detection Through Static Code Analysis

Static code analysis searches the binary for suspicious instructions and instruction sequences to detect potential malware before it is executed. Irazoqui *et al.* [45] developed MASCAT which aims to detect Rowhammer and cache or DRAM side-channel attacks. Most of them share similar code features to bypass the cache or access high precision timers which are uncommon for benign software. Bypassing the cache is usually done with a cache flush instruction or cache eviction. The cache flush instruction can be detected easily and is very suspicious when executed in a tight loop together with accesses to the same memory locations. Cache eviction is often implemented with pointer chasing to improve performance which is another strong indication for malware code.

MASCAT was proposed as a part of the approval process for an app store where every app is scanned automatically and a human can verify the results to reduce the false positive rate. However, this only works if the hammer code is in the binary at the time when it is checked. Code obfuscation or encryption can circumvent the detection. To use code encryption the attack must be able to run dynamically generated code. This is possible on Linux and also Android with the NDK by mapping writable and executable pages, in JavaScript code can be executed with `eval()`.

## 3.2 Neutralization-Based Countermeasures

Neutralization-based countermeasures take a different approach by allowing Rowhammer bit flips to happen but preventing exploiting them by separating potential aggressor rows from victim rows.

Brasser *et al.* [25] achieve that by isolating the kernel memory from the userspace memory. For this they extended the memory allocator in the Linux kernel. By isolating kernel from userspace memory, CATT protects only against exploits that are targeting the kernel from the userspace. If an attack is able to hammer directly in the kernel, e.g., through syscalls or network traffic, or gain root privileges by flipping bits in the user space, CATT cannot prevent them.

ZebRAM is extending this methodology by not only protecting the kernel from the userspace, but all processes on a system from each other [54]. Konoth *et al.* [54] split the physical memory into safe and unsafe rows, which are distributed over the entire memory in a zebra pattern. This ensures that the safe rows, which are used for memory mappings for the kernel and userspace, are never directly adjacent to each other. As this alone would halve the usable memory, ZebRAM utilizes the unsafe rows as integrity checked and ECC protected swap space. The discovery of half-double Rowhammer by Qazi *et al.* [71] requires a reevaluation of this protection mechanism because it allows hammering over the distance of two rows if it is possible to control the content in the unsafe rows.

## 3.3 Elimination-Based Countermeasures

Elimination-based countermeasures aim to prevent Rowhammer bit flips from happening. Some try to get a Rowhammer-free DRAM, which would be the best solution to the problem while others try to make it infeasible to hammer specific victims. However, all elimination-based countermeasures have one of two problems. They are either not designed for Rowhammer and therefore not really effective and can be circumvented, or they are implemented in hardware, therefore hard to distribute and not adaptable if broken.

### 3.3.1 Shorter DRAM Refresh Interval

For Rowhammer to flip bits, aggressor rows must be activated more often than a cell-dependent threshold, now called the maximum activation count (MAC) in one refresh interval. The refresh interval has therefore a direct influence on the number of Rowhammer induced bitflips in every DRAM module. Kim *et al.* [52], therefore, investigated the feasibility of decreasing the refresh interval as a countermeasure against Rowhammer and came to the conclusion that it is not enough. It was, however, still proposed by many hardware manufacturers as a countermeasure [8–10, 37].

The refresh interval of typical DRAM is 64 ms. During every refresh, the DRAM cannot read or write any memory, so decreasing the interval adds additional latency to DRAM operations and lowers performance. Additionally, the power consumption is increased because every row must be read, amplified and written back more often. To eliminate Rowhammer on all modules tested by Kim *et al.* [52], the refresh interval must be so low that the DRAM would spend 11.0 %–35.0 % refreshing, which is an impractical value. Taking into consideration that modern DDR4 memory modules require as little as only 50k aggressor activations in comparison to 139k on DDR3 modules measured by Kim *et al.* [52] makes this countermeasure even less feasible on modern memory [31, 52].

### 3.3.2 Error Correcting Code (ECC) DRAM

ECC memory is great at correcting random bit errors and, therefore, used in almost all server infrastructure, medical devices or financial services [73]. It additionally can reduce the power consumption of the DRAM in mobile devices and is, therefore, included in many LPDDR4(x) memory modules.

Although Rowhammer induced bit flips are seemingly similar to random bit flips, ECC memory is not an effective countermeasure against Rowhammer. This was already observed by Kim *et al.* [52]. All modules tested by them that were vulnerable to Rowhammer had victim rows with more than one bit flip per code word.

Most ECC DRAMs implement single error correction, double error detection (SECDED) codes. This means that single bit flips are corrected, but two bit flips in one code word are only detected. If more than two bits flip, even detection is not possible in all cases. What makes matters worse is that many systems simply ignore double bit errors. The CPU would be able to handle these detections and inform the operating system which could, for example, reboot. However, most systems checked by Lanteigne *et al.* [57] simply ignored double bit flips or acted only when the number of detections was above a certain threshold. Our target device does also have LPDDR4x DRAM with ECC. Single bit flips are corrected, which is why we always see flips in pairs or rarely triplets, but these are ignored by the CPU and OS.

### 3.3.3 Disabling Huge-Pages

Huge pages allow programs to get a 2 MB contiguous block of memory which improves performance because fewer page table walks are necessary. However, it also gives additional physical address bit information and contiguous memory that usually spawns over multiple neighboring rows. Both things are very helpful for Rowhammer. This enables many Rowhammer attacks after access to `/proc/self/pagemap` was disabled in the Linux kernel as a countermeasure.

It is however, still possible to use timing side channels in combination with reverse-engineered DRAM mapping functions to detect contiguous memory blocks figure out additional physical address bits. We use two methods to achieve that in our exploit and

describe them in more detail in Sections 5.2 and 5.2.2. But there are also new exploits that are still dependent on huge pages like SMASH [29]. De Ridder *et al.* [29] name disabling huge pages in browsers as an effective countermeasure against their attack.

### 3.3.4 Target Row Refresh (TRR) DRAM

The basic idea behind TRR was already proposed by Kim *et al.* [52] in the paper first describing the Rowhammer vulnerability. In theory, TRR counts the accesses to all rows in the DRAM and refreshes neighboring rows of rows that were accessed above a specific threshold since the last refresh. This threshold called the MAC is determined by the DRAM manufacturer. In reality, the energy and space cost is too high to count the accesses to all rows in the DRAM and the DRR specification does not allow to refresh unlimited rows when required. [31]

Frigo *et al.* [31] did an in-depth analysis on the behavior of TRR implementations of all major DRAM manufacturers and came to the conclusion that there are many different TRR implementations and none of them are documented openly. Intel also developed a TRR method called pseudo TRR (pTRR) which is implemented in the memory controller of the CPU. It is however, only available on very few systems because no consumer CPUs and only a few server CPUs from Intel support pTRR.

Typical TRR implementations consist of a sampler and an inhibitor [31]. The sampler counts accesses to rows and tries to detect aggressor rows. The inhibitor then refreshes a victim to protect it when a neighboring aggressor was accessed more often than defined in the MAC. Frigo *et al.* [31] used SoftMC, an FPGA memory controller, to analyze the inner workings of TRR on multiple memory modules of different manufacturers. They found that the samplers of all tested modules can only count a few rows. If more aggressors are hammered, the samplers overflow and bitflips can occur. The samplers on modules of one manufacturer were synchronized with the `REFRESH` command and only counted the accesses of the first few rows that were accessed after the refresh. Some samplers also have a dependency between the addresses of multiple aggressors presumable due to internal optimizations. The inhibitor on most modules could only refresh one victim per `REFRESH` command because DDR is a synchronous protocol. This means that the modules have only `tRFI` nanoseconds to complete the normal DRAM refresh plus a TRR refresh before the memory controller sends the next command.

### 3.3.5 Row Scrambling

All Rowhammer attacks that try to hammer precisely selected victim pages or use more advanced hammer patterns like double-sided Rowhammer require knowledge about the physical layout of rows inside a bank respectively knowledge about which addresses map to physically neighboring rows. Row scrambling is a countermeasure that complicates finding neighboring rows and relatively simple to implement by DRAM manufacturers. It works by applying a logic function to the row index bits that rearranges the order of the rows. The row scrambling function of our target device is shown in Equation 3.1.

$$r_i \oplus (r_i[3] \ll 2) \oplus (r_i[3] \ll 1) \tag{3.1}$$

On our target device, the row index starts at physical address bit 15, so the scrambling function uses physical address bits 16, 17 and 18. With 2 MB huge pages, it can be easily be circumvented because there are enough physical address bits available. We also managed to reverse engineer physical address bits 16 and 17 with contiguous memory detection and the DRAM mapping functions to correctly guess the scrambling in 50 % of all cases (Section 5.2).

# Chapter 4

# Exploit Concepts

Since the discovery of Rowhammer by Kim *et al.* [52] in 2014, researchers presented a large variety of exploits attacking Linux [27, 31, 75, 80], web browsers [21, 29, 30, 41, 75], Intel SGX [39], OpenSSH [55], remote systems over the network [58, 78], Hypervisors [72] and TRR DRAM [29, 31]. In this chapter, we want to give an overview of the different ways these exploits operate.

Rowhammer exploits can be split into four steps, *memory preparation* (cf. Section 4.1), *data preparation* (cf. Section 4.2), *hammering* (cf. Section 4.3) and *exploitation* (cf. Section 4.4). *Memory preparation* is the first step with the goal to allocate physically adjacent rows that are required for most Rowhammer patterns. To exploit Rowhammer bit flips, the attack must hammer vulnerable data, ranging from kernel data structures to instructions inside the attack. In the second step, *data preparation*, this vulnerable data is placed in the victim rows with the help of many different techniques. When the data has successfully been placed, the attack *hammers* the victim row. Depending on the attack scenario, the hammer pattern and cache bypass method can vary widely. Finally, the flipped bit in the vulnerable data is *exploited* to achieve the attack goal, like privilege escalation or a virtual machine escape. In order to give a clearer picture, the descriptions of the steps include descriptions of exploits that perform the respective step.

## 4.1 Memory Preparation

Having access to physically adjacent rows allows the attack to hammer more efficiently with more effective patterns and have more control over the corresponding victim row. Obtaining the knowledge about which memory locations correspond to adjacent rows is not straightforward because of virtual memory and complex DRAM mapping functions. However, with the combination of different techniques like timing side channels, huge pages, contiguous memory detection and row unscrambling it can be achieved by an unprivileged attacker.

### 4.1.1 Rows On Same Bank

The first requirement for most Rowhammer patterns is the access to at least two rows on the same DRAM bank. Repeatedly accessing only one row would usually[1] keep this row in the row-buffer of the bank and, therefore, cause no electrical disturbances to flip bits. There are two ways to get rows on the same bank. The first one is with the DRAM mapping functions that determine the mapping of the rows to the banks and and the second one through a timing side channel.

#### DRAM Mapping Functions

If the DRAM mapping functions and all required physical address bits are available, it is trivial to map the rows to the corresponding banks. Reverse engineering the mapping functions is possible with the DRAMA tool developed by Pessl *et al.* [66]. The mapping of our device is shown in Figure 2.4. Two rows are on the same bank if the values $X_1 - X_4$ have the same value. The difficulty with this approach is to get the required physical address bits. Prior to the first Rowhammer kernel privilege escalation the virtual to physical address mapping could be read from `/proc/self/pagemap`, but this file was made only root-readable as a response to this attack. Another option is to use huge pages (cf. Section 4.1.2). In the huge page, the lowest 21 virtual address bits are the offset and, therefore, equal to the physical address bits. An advantage of using the DRAM mapping functions is that not only the bank but also the location within the bank is known. This simplifies the search of neighboring rows.

#### Timing Side Channel

The second approach uses a timing side channel on the access latency between the accesses to two rows. If a row is accessed, its content is moved into the row buffer, the row is opened, and stays in this buffer until another row is in the same bank is accessed[1]. In this case, the old row must be closed before the new row can be opened. The additional latency can be measured by alternatingly accessing two rows a fixed amount of times and measuring the time the accesses take. If the rows are located in different banks, they stay in the row buffer. If the rows are located in the same bank, the rows must be opened and closed each time and the accesses are measurably slower than in the first case. For the timing side channel to work, the accesses must not be cached.

However, this technique does not provide any information about the location of the rows within the bank, which is required for most efficient hammer patterns. When used on contiguous memory, e.g., huge pages, the contiguity also translates to the rows according to the DRAM mapping functions (cf. Section 4.1.2). When no contiguous memory is available, the side channel can be combined with the knowledge of the DRAM mapping

[1] On systems with an *open-row policy* the row is kept open until another row is opened. (cf. Section 2.1.1)

functions to detect contiguous memory blocks and reverse-engineer additional address bits (cf. Section 5.2).

### 4.1.2 Contiguous Memory

Because Rowhammer exploits a weakness in the DRAM it is beneficial to gain knowledge about which locations of the virtual address space are physically adjacent inside the DRAM. Having a block of contiguous memory simplifies this task.

#### Huge Pages

Huge pages are a performance optimization used in all modern operating systems and supported by all major CPU architectures that can speed up applications relying on large amounts of memory [60]. Usually, the physical and virtual memory is divided into 4 kB long chunks, called pages. Virtual pages of different processes are then mapped to physical pages through multiple levels of page tables. When accessing a virtual address, the memory controller of the CPU has to load every level of the page table from the memory to find the corresponding physical address. There is a special cache for virtual to physical memory translations called the TLB that speeds up similar translations. If, however, a program accesses a big memory area that spawns many last-level page tables, there is a noticeable negative impact on performance.

To solve this problem, the last level page table or the last two levels can be skipped. This results in pages that are $4\,\text{kB} \cdot 512 = 2\,\text{MB}$ or $2\,\text{MB} \cdot 512 = 1\,\text{GB}$ large. However, the performance increase is not relevant for Rowhammer, but the knowledge that huge page memory is contiguous and the knowledge of nine additional bits of the physical address per skipped page table level. Therefore, with a 2 MB huge page, an attacker has 512 contiguous 4 kB pages and knows 21 bits of the physical address.

**Huge Page Support.** When mapping memory with the `mmap` syscall the flag `MAP_HUGETLB | MAP_HUGE_2MB` tells the kernel to map a 2 MB huge page. If the kernel does not map a 2 MB page, the syscall fails and no memory is returned. The advantage of this approach is that the memory returned if the call was successful is guaranteed to be a huge page. For this to work, huge page support must be explicitly enabled in the kernel and it has to reserve memory for huge pages at boot time or by a user with root privileges. Because of this inflexibility, huge page support is usually not enabled on systems and it is not enabled by default in many Linux distributions. As a consequence, an attacker cannot rely on this type of huge pages to get contiguous memory.

**Transparent Huge Pages.** The conflict between the performance increase for many applications and the complex and inflexible configuration led to the development of transparent huge pages [15]. With transparent huge pages, the kernel uses 2 MB pages whenever possible without the knowledge of the running program. This increases its performance and does not require any changes by the programmer or add any restrictions.

Additionally, a program can request a huge page with the `madvise(MADV_HUGEPAGE)` syscall, which will always result in a huge page mapping if the address is 2 MB page aligned and there is space in the physical DRAM.

#### Contiguous Memory Detection

Huge pages are solely a performance optimization, because of that they can easily be deactivated should there be a security concern. Therefore, relying on them is not optimal for an exploit. Another way to get contiguous memory is to detect it in a large memory area. The buddy allocator used in Linux tries to keep physically contiguous memory intact as long as possible by always using the smallest available chunks, usually single pages, for new mappings. But when mapping a large enough area, it has to use contiguous memory eventually [30].

Using a side channel to get rows on the same bank and the reverse-engineered DRAM mapping functions, it is possible to detect these contiguous memory areas. For this the unique *distance patterns* given by the DRAM mapping functions are used. We use this technique in our exploit and describe it in more detail in Section 5.2.

#### Row Unscrambling

Row scrambling is a simple Rowhammer countermeasure that applies a function to some row address bits, scrambling the rows. It is used to make it harder to find adjacent rows inside a bank. Row scrambling is described in more detail in Section 3.3.5. To hammer efficiently with most Rowhammer patterns, neighboring rows are required and we therefore have to unscramble them.

In our target device, the memory uses physical address bits 16, 17 and 18 to perform the scrambling. With huge pages, the unscrambling is trivial because they provide the lowest 21 bits of the physical address. If contiguous memory detection is used, higher physical address bits are not available straightforward, but it is still possible to retrieve bit 16 and 17 using the *distance patterns* given by the DRAM mapping functions. We explain the details in Section 5.2.2.

## 4.2 Data Preparation

Random bit flips in the memory can have severe effects on a system like crashing programs, corrupting file systems or modifying user credentials [58]. But most attacks have a specific goal which requires more control over the data that is hammered. During *data preparation* the attack puts the target data into the victim row. It is usually split into two steps. During memory templating the attack searches the memory for a victim row with an exploitable bit flip. After finding one, it puts the victim data in that row by spraying the memory with it [30, 75], deterministically placing the victim data [80] or detecting when the victim data is in that row while moving it around randomly [39].

### 4.2.1 Memory Templating

Memory templating utilizes the fact that Rowhammer bit flips are repeatable most of the time [52]. The reason is that Rowhammer bit flips happen in weak memory cells where due to production variance, the cell transistor has a high subthreshold leakage [82]. The repeatability also involves the bit flip direction. Cells flipping from zero to one do usually not flip from one to zero and vice versa. With this knowledge, the attacker can map a big chunk of memory and check every row for victim cells with bit flips. The search can be even more fine-grained by selecting the victim cells based on their offset within the page. This ensures that later in the attack when the victim data is in the victim page, a bit flip happens in an exploitable location, e.g., the PFN-field of a PTE and not in any other bit that could corrupt the PTE.

### 4.2.2 Blind Hammering

Memory templating is based on the assumption that Rowhammer induced bit flips are repeatable. However, this assumption is not entirely accurate in the presence of ECC memory. Without ECC memory, the success of a bitflip is only dependent on the content of the victim cell. For example, if this cell is prone to flip from 0 to 1, it does not flip if it currently holds a one. With ECC memory, this dependency extends to the content of other cells in the same 64-bit data word. To evade this issue, we developed a technique that does not require templating, which we describe in more detail in Section 5.4.2.

### 4.2.3 Data Spraying

Regardless of whether memory templating or blind hammering is used, the victim data must be placed between aggressors rows. Placing physical pages in the DRAM is done by the kernel and there is no direct way for userspace processes to influence the position of physical pages. One option is to fill as much memory as possible with the victim data increasing the probability that victim data is placed between aggressor by chance [75].

Data spraying is only applicable if there is a mechanism to create many copies or instances of the victim data from an unprivileged process. This is for example the case for page tables and instructions belonging to the attack process. Both types of data were used by Seaborn *et al.* [75] in their first kernel privilege escalation and Google Chrome Native Client (NaCl) sandbox escape. To spray the memory with page tables it is enough to repeatedly map a shared memory file. Every mapping creates multiple page tables depending on the size of the file and the amount of mappings is only limited by the size of the virtual address space and the number of virtual memory areas. We describe the page table exploitation in more detail in Section 4.4.1. For the NaCl exploit, it is enough to hammer instructions belonging to the attack process. To spray these the instructions they are written many times into writable and executable pages. Instruction flipping is explained in Section 4.4.2.

The size limitation of the virtual address space and the maximum amount of virtual memory areas one process can create must be considered. On most systems, the virtual address space has a size of hundreds of terabytes and is therefore large enough, it can however be smaller on some ARM devices due to a performance optimization. This is the case on the target device of our exploit and we explain the solution in Section 5.3.

$$\text{filesize} = \left\lceil \frac{\text{memsize}}{\text{pagesize}} \cdot \frac{1}{\#\text{VMA}_{\text{max}} - \text{buffer}} \right\rceil \cdot \frac{\text{pagesize}}{\text{PTEsize}} \tag{4.1}$$

One virtual memory area (VMA) represents one `mmap`'d memory area and the maximum amount is $2^{16} = 65.536$ VMAs. As a consequence, the size of the shared memory file must be large enough that mapping it less than $2^{16}$ creates enough page tables to fill the whole available physical memory. It should, however, also be as small as possible to not waste memory that page tables could use [75]. Equation 4.1 calculates the minimal possible size for the shared memory file given the size of the memory that should be filled and a buffer to use a bit less than $2^{16}$ VMAs.

### 4.2.4 Phys Feng Shui

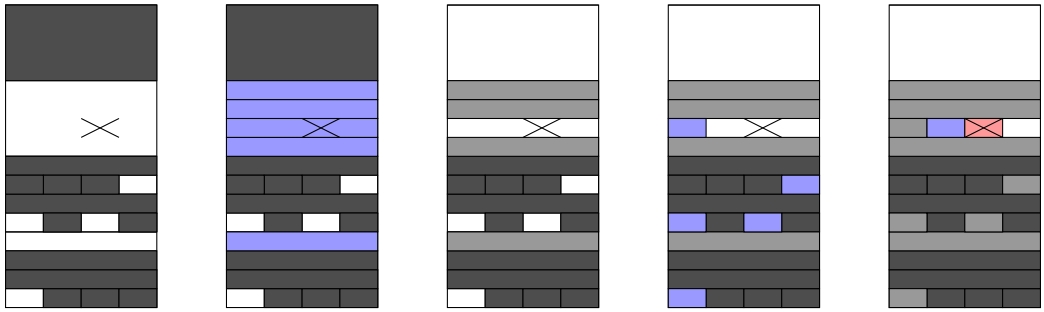
While page table spraying is relatively simple, its behavior is not predictable and the outcome depends a lot on the attacker's luck. There is no control over the locations of the page tables or their contents, thus also, after flipping a bit successfully in a PTE the new PFN target can be anywhere. With Phys Feng Shui, Van der Veen *et al.* [80] utilize the predictable behavior of the memory allocator used in Linux and Rowhammer bit flip repeatability for the first fully deterministic Rowhammer exploit on Android they called Drammer [80]. To hammer double-sidedly, it uses uncached contiguous DMA memory, which is available to all apps on Android. Like the previous exploit, it also tries to hammer a page table to get full access to the memory, which is then used to find and change the UID of the process in the `struct cred` to gain root privileges.

Linux uses the buddy allocator algorithm to manage its physical memory. Its main goal is to keep large contiguous memory areas intact as long as possible by always using the smallest available chunks for new allocations and splitting up bigger chunks only if necessary. It additionally recombines smaller chunks whenever possible. Figure 4.1 shows how to exploit this behavior to allocate a page at an attacker controlled location.

(a) Is the initial memory situation, the dark grey memory is filled, the white is free. The attacker found an exploitable bit flip at the location marked with an `X` during memory templating.

(b) All `M` chunks are allocated. The system does not come close to a out-of-memory situation because there are still plenty `S` chunks available.

(c) The `M` chunk that contains the target page is freed. It is now the only free `M` chunk in the memory.



**(a)** **(b)** **(c)** **(d)** **(e)**

**Figure 4.1:** An example of using memory waylaying to deterministically place a page at an attacker chosen position (marked with an X).

(d) `S` all allocated until the `M` chunk is split into `S` chunks. The attack knows that subsequent allocations with size `S` will be placed were the `M` chunk was.

(e) Additional `S` chunks are allocated as a padding, the attack knows how many are required. Finally the victim page is allocated in the target location.

To do these memory allocations correctly, the attacker needs a way to know about the current state of the buddy allocator. On Linux, it is possible to get the number of free blocks of every size from the file `/proc/buddyinfo` and detailed information about the memory zones from `/proc/zone-info`. The DRAMMER [80] exploit used `/proc/pagetypeinfo`, but this file was since changed to be only readable by the root user.

With the ability to put pages precisely into the physical memory, it is also possible to predict precisely where a PTE will point to after hammering. This can be used to always have it point at itself, which simplifies the exploit. To do this, the offset and flip direction of the bitflip found during templating is important. The offset within the PFN-field of the PTE determines the distance between the page table and the page it maps to, the target page. The direction of the bitflip determines if the page table is in front or the back of the target page. Additionally, a part of the physical address is known because of the alignment properties of the different block sizes.

If a $0 \rightarrow 1$ bit flip was found in the seventh bit of the PFN of the 56th PTE in a page table on a 64-bit system with 4 kB pages during templating, the following conditions must be fulfilled to exploit this bit. A $0 \rightarrow 1$ bitflip increases the PFN which means the page table must be behind the target page. The seventh bit of the page number of the target page must be zero. To have the PTE at index 56 of the page table the virtual address must have the bits 12 to 21 set to 56. If all these conditions are met, hammering the page table will lead to the PTE at index 56 point to its own page.

From there, the exploitation techniques of Section 4.4 can be used. The Drammer exploit changes the UID field of the app's process.

### 4.2.5 Memory Waylaying

Memory Waylaying is a technique by Gruss *et al.* [39] to move a victim binary to an attacker selected location in the memory. It is required for opcode flipping if it is not possible to spray the memory with exploitable instructions like in the NaCl exploit. When opening and reading a file, its content is copied from the hard disk into the DRAM. When the file is not needed anymore, it is not removed from the memory but kept in the page cache, resulting in faster subsequent reads of the same file. The page cache usually fills the whole unused memory because the kernel can easily discard pages if more memory is required by running programs. The same applies to new entries in the page cache. They simply replace older entries. Because of that, every application running in userspace can evict the page cache by accessing enough locations in a file. When a file is evicted from the page cache and reloaded later it is loaded into another location in the memory. The advantage of this technique is that the system never runs out of memory because the page cache only uses the otherwise free memory.

By repeatedly loading a file into the page cache and evicting it again, its page cache entry is randomly moved around in the memory and will eventually land in the victim row. To exploit this, knowledge about the current location of this file within the page cache respectively physical memory is required. Prior to the memory waylaying, a memory cell prone to flipping is searched that is at one of the required offsets within a page. This page is the target location for the file in the page cache. To get the physical location of the page cache entry and know when it is on the target page, the prefetch address-translation oracle is used [40]. This oracle exploits the fact that prefetch instructions on x86 do not check for access permissions. This makes it possible for userspace processes to prefetch every physical address into the memory through the direct physical mapping in the Linux kernel that maps every physical page. This was possible because all kernel pages were also mapped in the virtual address space of every userspace process to make context switches faster.

The KAISER patch developed by Gruss *et al.* [38] for the Linux kernel removes most of the mappings of the kernel pages in the userspace's virtual memory on x86, rendering this prefetch oracle unusable. Its initial purpose was to defend against side channels used to break KASLR. It proved, however, to be an effective countermeasure against the Meltdown vulnerability found later and KPTI, which is based on KAISER, was quickly implemented in Linux as well as similar functionality in the kernels of Windows and MacOS [59].

Gruss *et al.* [39] combined a new hammer pattern, novel memory waylaying, a new attack target and Intel SGX to defeat all countermeasures. Intel SGX is a secure enclave that protects programs from all other software running on the system, including the operating system and a hypervisor. It does this by, among other things, encrypting all

memory and disabling the performance counters for an enclave application. The disabled performance counters defend against countermeasures that use the performance counters to find processes that cause an unusually high amount of cache misses. The exploit is also secured against countermeasures that use static code analysis because it is encrypted and only decrypted within the enclave. The third improvement is the usage of a novel hammer pattern called *one-location Rowhammer*. It hammers by only accessing a single row repeatedly, which is possible due to an optimized memory controller policy for CPUs with high core counts.

The memory encryption and integrity check of enclave applications can also be exploited by hammering enclave memory. If this integrity check done by the memory controller fails, the CPU halts until a manual power cycle is performed. This enables simple DoS attacks.

## 4.3 Hammering

In the hammering phase the attack tries to flip a bit in the victim data with Rowhammer. Firstly, this requires direct the accesses to the rows in the DRAM and therefore bypassing of the CPU caches. This can be achieved with uncachable memory (most efficient), cache flushing or cache eviction (least efficient). The cache bypass technique usually used depends on which one is available and practical on the system under attack.

Secondly, there are many different hammer patterns that were found by researchers over the years. Every one comes with its own characteristics like efficiency in terms of bit flip frequency, stealthiness and countermeasure resistance. The used pattern again depends highly on the system under attack. Most attacks run their hammer code on the CPU in the attack process, but it was also shown that hammering from the GPU [30] or over the network is feasible [58, 78].

### 4.3.1 Cache Bypassing

Compared to the CPU, the DRAM is really slow. Accessing the DRAM can take hundreds of CPU cycles, limiting the number of memory operations a CPU can do. To counter this problem, CPUs contain multiple levels of caches of different sizes and latencies. They save the result of memory operations and make subsequent operations to the same memory location faster. This hinders the memory operations to actually reach the DRAM which is a necessity for Rowhammer. Consequently, a procedure is required to avoid the cache and access the DRAM as quickly as possible.

#### Uncachable Memory

It is possible for the kernel to mark pages as uncachable with a bit in the PTE on x86 and ARM. This is, for example, used for memory areas that are shared with other devices through DMA. Access to the memory from other devices through DMA bypasses the CPU and therefore its caches. Because of that, cache coherency problems can occur

if the CPU architecture does not actively monitor the memory bus to invalidate its corresponding caches. x86 CPUs have this functionality, ARM CPUs do not and because of that, Android had an API that allowed every app to request uncachable DMA memory. The memory returned by this API had the second advantage that it was physically contiguous. This was exploited in 2016 by Victor van der Veen *et al.* [80] in the first Rowhammer exploit on ARM and Android called Drammer.

Uncachable memory has the advantage that it is the fastest way to access the DRAM in terms of accesses per second. So for some memory chips that are only slightly vulnerable to Rowhammer induced bitflips, hammering with cache flush instructions or cache eviction can be too slow.

### Cache Flush Instruction

All CPU architectures come with instructions to invalidate and write-back cache lines in the main memory. There a various legit reasons for a program having to flush cache lines from specific caches like modifications to page table entries or when the dynamic generation of code requires the synchronization of the instruction- and data-caches [18, 23]. JIT compilation is used in the JavaScript engines of all modern web browsers, in the Java virtual machine and many more applications [28, 61]. This is an example of legit use of the cache maintenance instructions from the userspace, which is why they are available in userspace in x86 and ARM-v8.

The x86 instruction `clflush` was used in the first description of the Rowhammer bug by Kim *et al.* [52] and in the first two exploits by Seaborn *et al.* [75]. These instructions are, however only available from native code. For interpreted languages like JavaScript, a third option, cache eviction is possible.

### Cache Eviction

When no uncached memory or cache flush instruction is available, the cache lines must be evicted by accessing a set of other addresses, a so-called *eviction set*. Because the LLC is inclusive, it is enough to evict a cache line from the LLC, which also evicts the cache line from the L1 and L2 cache.

The LLC on all modern CPUs is divided into slices, sets and ways. The slice and set are calculated from the physical address with usually undocumented functions. The ways are where cache lines with the same slice and set are stored flexibly, meaning that a cache replacement policy decides which cache line to evict. Therefore, the eviction set to remove an address $\mathcal{A}$ from the cache must consist of addresses that map to the same slice and set as $\mathcal{A}$. The number of addresses must be greater than the number of ways and must be accessed in the correct order, the *access pattern*, to trick the cache replacement policy to evict $\mathcal{A}$. For systems where the cache mapping functions and the replacement policy are available, an eviction set can be built manually. However, this information is usually not available when running an attack on an unknown system. To

counter this Gruss *et al.* [41] build *eviction sets* dynamically for their Rowhammer.js exploit.

The significant disadvantage of cache eviction is the reduced hammer rate. The additional DRAM accesses that are necessary to evict the two cache lines of the aggressor rows slow down the hammer loop. Therefore fewer accesses to the aggressor are performed during one refresh interval, which leads to a smaller amount of bit flips.

Rowhammer.js by Gruss *et al.* [41] was the first attack that showed the viability of Rowhammer exploits from more restricted environments and programming languages like JavaScript. The basic concept is the same as in the first kernel privilege escalation. Rowhammer.js tries to hammer a page table to gain full access to the memory. It does this however, without access to `/proc/self/pagemap` for physical address translation and no cache flush instruction. To get contiguous memory and find adjacent rows, the exploit takes advantage of the behavior of Firefox and Google Chrome that use huge pages for large typed arrays. The huge page borders can be found with a timing side channel measuring added delay from the page fault when a new page starts. Rowhammer.js is splitted into an offline and online phase to perform efficient cache eviction. During the offline phase, a tool tries to learn the most efficient *eviction set* and *access pattern* automatically on many different systems. Later in the online phase, when attacking a system, the exploit tries all learned *eviction sets* and *access patterns* and uses a timing side channel to find one that is working and uses that to hammer.

#### Self-Evicting Rowhammer (SMASH)

Self-Evicting Rowhammer is a technique presented by de Ridder *et al.* [29] that combines cache eviction with the dummy accesses required for the many-sided Rowhammer patterns to defeat TRR [31].

SMASH is a Rowhammer exploit that builds on the findings of TRRespass to hammer TRR memory from JavaScript. Some many-sided Rowhammer patterns found by TRRespass access up to 19 rows which is too much for eviction-based hammering. Additionally, huge pages are not large enough to span over all rows required for some many-sided patterns.

To solve the first obstacle, de Ridder *et al.* [29] developed self-evicting Rowhammer. It uses the same accesses to bypass TRR and to evict the accessed cache lines from the cache. In contrast to Frigo *et al.* [31], they did not find any dependency of the row locations within the bank on the number of bitflips. Therefore, the requirement for the eviction set is only to evict the aggressor rows and access the memory an additional $N$ times on rows in the same bank. $N$ being the number of accesses required to break the targeted TRR implementation. A 2 MB huge page is not large enough to build such eviction sets as it contains only four congruent cache lines. To get more congruent cache lines, they use a technique called *huge page coloring* to find huge pages similarly mapping to the LLC.

The downside of a self-evicting pattern is that the number of accesses required to evict the cache is often higher than the accesses required to break TRR. Thus, these additional accesses not required for TRR only slow down the hammering. To counter this, the eviction pattern also includes cache-hit accesses to decrease the perceived associativity of the LLC. Memory controllers have some flexibility in their timing of `REFRESH` commands sent to the memory. To improve performance, they try to send them when there is no memory activity, for example, during cache hits. However, it is important that the `REFRESH` commands are not sent before all accesses of the many-sided Rowhammer are done to trick the sampler. This is achieved by distributing the cache hit accesses evenly over the hammer accesses and the addition of `NOPs` that trigger the memory controller to send the `REFRESH` command at the correct time [29].

With the ability to produce bitflips from JavaScript on TRR memory, de Ridder *et al.* [29] use the same exploit technique presented in GLitch [30] to obtain full access to the memory of 64-bit Firefox.

### 4.3.2 Hammer Pattern

The basic idea behind Rowhammer is to flip bits by creating electrical disturbances inside the DRAM chips that inject electrons into transistors of other rows, and capacitive crosstalk between wordlines. Since Rowhammer was first described in 2014 [52], many access patterns with different properties were found. The most important ones are shown in Figure 4.2. The red rows are the aggressors and the blue rows the victims.

#### Single-Sided

Single-sided Rowhammer (Figure 4.2a) was the first pattern found by Kim *et al.* [52], using two or more aggressors in the same bank with a distance greater than 2. This pattern is easy to achieve because rows on the same bank can be found with a timing side channel and no further physical address information about the aggressors is required.

#### Double-Sided

Double-sided Rowhammer (Figure 4.2b) sandwiches one victim between two aggressors, thus greatly increasing the electrical disturbances and therefore, the bitflip count. It was first used by Searborn *et al.* [75] in their two exploits against the Google Chrome NaCl and the Linux kernel. The increased bitflip probability comes with the added complexity that parts of the physical address and at least basic knowledge of the DRAM mapping functions are required to be able to select the aggressor rows precisely.

#### One-Location

Kim *et al.* [52] explained in 2014 how accesses to at least two aggressors on the same bank are required for Rowhammer. The reason being that the last accessed row is kept in the row buffer and served from there to save time and energy. Four years later,

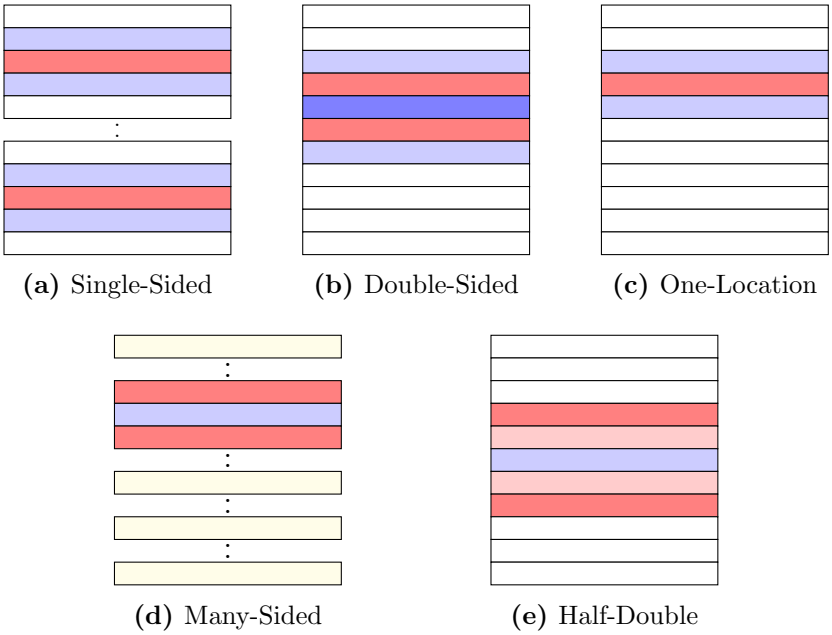



**Figure 4.2:** Depiction of five notable Rowhammer patterns. The red rows are the aggressors, the blue rows the victims and the yellow rows for dummy accesses. In the dark blue row probability of a bit flipping is higher than in the light blue rows.

Gruss *et al.* [39] found one-location Rowhammer that can, on some systems, flip bits with only accesses to a single row in a bank.

Keeping the last accessed row in the row buffer improves performance if multiple accesses to addresses in close proximity happen frequently. If however, another row is accessed, the latency is increased because the row currently in the row buffer must be closed before opening the new one. On CPUs with many cores, this increased latency can outweigh the advantage of keeping a row open, which is the reason why Intel uses a closed-row policy on these CPUs. On systems with a closed-row policy, the row is not kept in the row buffer but written back (closed) after every access. This is what enables one-location Rowhammer.

One-location Rowhammer produces fewer bitflips than double-sided Rowhammer but has the big advantage of being more stealthy. As a result, many proposed Rowhammer countermeasures from that time tried to detect Rowhammer with as little false positives as possible by looking for frequent accesses to at least two rows in a bank which is subverted when accessing only one row [20, 32, 45].

#### Many-Sided

Most modern DDR4 memories come with a Rowhammer countermeasure called TRR. It is described in more detail in Section 4.3.5. TRR tries to count the accesses to every row in a refresh cycle and refresh neighbors of rows that were activated above a given threshold, the MAC. This works well for all previously mentioned hammer patterns but is possible to circumvent with new TRR-aware patterns.

The implementation on most DRAM modules does, however, not count the accesses for every single row because that would be too costly. They usually use a system of samplers and an inhibitor that can be bypassed with more complex access patterns. TRRespass finds patterns automatically for 70% of DRAM modules on the market [31]. The same technique is also used in the Smash exploit from JavaScript together with cache eviction (Section 4.3.1) [29].

#### Half-Double

*Half-double* is a novel Rowhammer pattern, discovered by Qazi *et al.* [71] in 2021, that is not only able to circumvent TRR on some DDR4 modules but is enabled by the additional refreshes done by TRR [71].

The *half-double* pattern consists of two pairs of aggressor rows, two near aggressors directly adjacent to the victim row and two far aggressors in the next rows. To hammer, only the two far aggressors are repeatedly accessed by the attacker. This causes TRR to refresh the near aggressors, which amplifies the electrical disturbances and causes bit flips in the victim. To test their hypothesis, that the additional accesses to the near aggressors done by TRR are required and it not being a simple distance-two Rowhammer, Qazi *et al.* [71] used a FPGA platform to have full control over the commands sent to the DRAM module. With it, they deactivated TRR by not sending any refresh commands and did not observe any bit flips.

We tried to increase this effect further by additionally accessing the near aggressors when hammering, but this did not increase the amount of bitflips. When the dilution became to small the bitflips decreased because TRR started to refresh the victim row (Section 5.4.3).

### 4.3.3 Hammering with the GPU

GLitch by Frigo *et al.* [30], is the first Rowhammer exploit from JavaScript on mobile devices, which utilizes the integrated GPU to build high precision timers and hammer the memory. Previously shown Rowhammer exploits from Javascript [21,41] had to use CPU cache eviction, which is too slow on ARM devices to induce any bitflips. Additionally, the timer precision was decreased in all major browsers due to found side channel and Rowhammer attacks, making the detection of page boundaries and contiguous memory areas more difficult. GLitch tackles both obstacles with the help of the integrated GPU in the mobile SoC which any website can access with JavaScript and the WebGL API.

With high precision timers not being available anymore in modern web browsers [1, 26, 84], GLitch identifies four new high precision timing sources based on WebGL. Two are based on OpenGL extensions to measure the performance of applications and their availability is dependent on the GPU driver. The other two utilize functions used for synchronization between the CPU and GPU that cannot be deactivated because they are part of the WebGL2 standard.

Similarly to CPU-based Rowhammer attacks, cache eviction is required to access the DRAM repeatedly from the GPU. Frigo *et al.* [30] reverse-engineered the cache architecture with the help of the GPU's performance counters of an Adreno 330 GPU used in most high-end smartphones of the time. The cache consists of two non-inclusive levels with a simple FIFO replacement policy.

The Adreno 330 GPU operates on virtual memory, which requires a way to get physically contiguous memory to be able to use the double-sided Rowhammer pattern. Because the GPU does not use huge pages, Frigo *et al.* [30] utilizes their GPU timers to build a side channel to detect contiguous memory allocated by the Linux buddy allocator.

With a cache-eviction strategy and physically contiguous memory, they are able to flip bits with Rowhammer on all three mobile devices they tested. On the Nexus 5, they achieved 23.7 1-to-0 flips/min and 5 0-to-1 flips/min and used these to build an exploit to escape the Firefox Javascript sandbox. The exploit is based on the way different data types are saved in JavaScript, called NaN-boxing. Because of NaN-boxing, references to objects can be transformed into doubles and back with only one bitflip each. This is used to build fake typed arrays and get access to the whole process's memory. The technique is described in more detail in Section 4.4.3 [30].

### 4.3.4 Hammering over the Network

Nethammer by Lipp *et al.* [58] showed at the same time as Throwhammer [78] that flipping bits on a remote machine with Rowhammer over the network is possible. Nethammer uses the package handling code to hammer, Throwhammer hammers directly on the target through remote DMA accesses.

The network driver executes code for every received network packet, which must be loaded from the DRAM every time. This code is however cached after the first execution and cache flushing or eviction is not easily possible without code execution on the target. Intel Cache Allocation Technology (CAT) is a quality of service feature for server platforms that allows limiting the number of LLC cache ways every CPU core can use. If this is enabled on the target system and the network driver code runs on a core with only one LLC cache way, this code can already be enough to evict the cache, making Rowhammer possible. The difference to other exploits is that bitflips cannot be targeted. Random bitflips however, can do a lot of harm. They can modify kernel data or code, leading to crashes or data in file system drivers which can corrupt data on disks. Lipp *et al.* [58] also noticed the crashing of various user space programs and the permanent modification of user credentials which broke the ssh login for a user.

### 4.3.5 Target Row Refresh (TRR) Mitigation

ECC memory was not designed as a countermeasure against Rowhammer and is therefore not effective as a mitigation. TRR on the other hand, was developed solely as a reaction to the increasing threat coming from Rowhammer. The concept is relatively simple, but as shown recently, the way it is implemented on most DRAM modules weakens the protection, does not prevent Rowhammer attacks [31] and can even support hammering [71]. Details about the behavior and implementation of TRR can be found in Section 3.3.4.

TRRespass is a black box fuzzer developed by Frigo *et al.*, that finds access patterns automatically that defeat TRR on 1/3 of all DRAM modules tested by them. The fuzzer accesses patterns with different cardinality and location. The cardinality is the number of aggressor rows in the pattern. To reach 50k activations each the maximum cardinality is 28, with a $t_{RC} \approx 45$ ns (time between `REFRESH` commands) and a refresh interval of 64 ms. Some modules they tested responded differently to patterns with the same cardinality but different aggressor locations, probably due to internal optimizations. The fuzzer is therefore also randomizing the location of the aggressors.

TRRespass was able to find Rowhammer access patterns on 13 of the 42 modules they tested, with a cardinality ranging from 3 to 19 on modules of all three vendors after running it for 6 hours on every module. Frigo *et al.* [31] note that this does not mean that the other modules are not vulnerable, but that it can only be a matter of more prolonged testing or a better fuzzing strategy to find patterns for these modules.

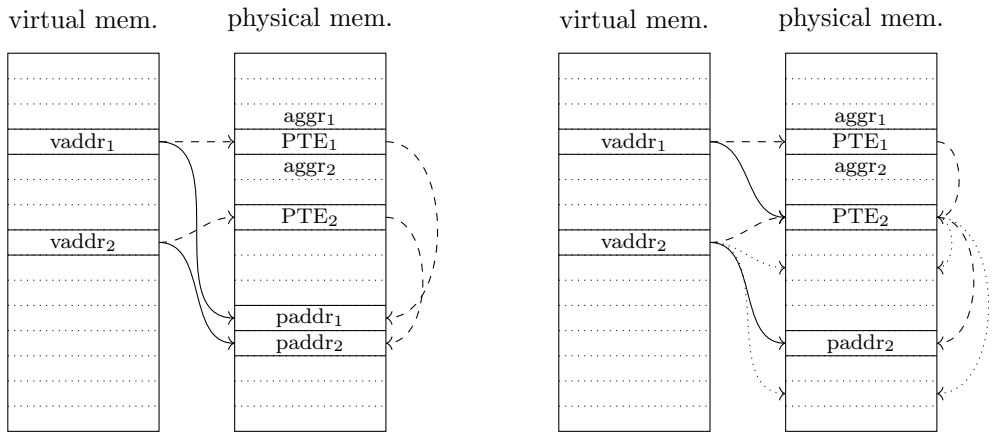
Frigo *et al.* [31] were also able to find patterns on 5 out of 13 smartphones with LPDDR4x memory. For this test they did not have control over the location of the aggressors due to limitations imposed by the systems. [31]

A novel hammer pattern called *half-double* Rowhammer by Qazi *et al.* [71], that flips bit with the help of TRR is described in Section 4.3.2.

## 4.4 Exploitation

Until here, we discussed concepts and techniques required to flip bits in the DRAM with Rowhammer. This in itself, however, is not enough to compromise an operating system or sandbox. To do so, specific bits, depending on the kind of exploit, must be flipped.

Most known attacks aim for privilege escalation. These include gaining kernel privileges from an unprivileged process [39, 75, 80], escaping from a JavaScript sandbox to have access to the web browser's memory [29, 30, 75] or escape from a virtual machine to have access to the memory of the host and other virtual machines [83]. The different techniques used by exploits can be categorized into three groups *page table flipping*, *instruction flipping* and *type flipping*. Additionally to these privilege escalation exploits an exploit was presented that can read arbitrary memory [55] and exploits that aim for a denial of service by hammering random memory [39, 58].




**(a)** Correct Mapping **(b)** Successful Exploit

**Figure 4.3:** Simplified example of page table flipping with only one level of pages tables shown. After successful exploitation (b), page table 2 is writeable through virtual address 1.

### 4.4.1 Page Table Flipping

The goal of page table flipping is to flip a bit in the page frame number (PFN) of a page table entry (PTE) making this PTE point to a page table [75]. This can be another page table belonging to the process or the page table containing the PTE itself. If that is the case, the attacker has read and write access to the second page table through the first one. With this setup, any page in the physical memory can be accessed by pointing the second page table to it.

Page table flipping is illustrated in Figure 4.3. It is simplified by showing only one level of page tables in contrast to the three or four levels that are typical and every page table is only mapping one page. At the beginning (Figure 4.3a) in a correct mapping $vaddr_1$ maps to $paddr_1$ (*solid line*) through $PT_1$ (*dashed lines*), same for $vaddr_2$. To be able to hammer a page table entry, the attacker needs access to rows adjacent to a page table. Ways to achieve this are discussed in sections 4.2.1 to 4.2.4. In this example the attacker controls the two rows $aggr_1$ and $aggr_2$ below and above of $PT_1$. By hammering with these two aggressors, a bit is flipped in the PFN-field of the $PT_1$, making it point to another page in the physical memory. If the whole memory contains many page tables, the chances are fairly high that the attacked PTE will point to a page containing another page table. This is illustrated in Figure 4.3b. The $PT_1$, which we call the *address page table* is pointing to the page table containing $PT_2$, which we call the *access page table* giving the attacker read and write access to page table 2 through $vaddr_1$. This gives the attacker the ability to choose where $vaddr_2$ is mapped to by writing any PFN into the $PT_2$, resulting in full access to the whole physical memory. There are multiple ways to obtain a page table that is in a row with bitflips und between to aggressor rows

that are described in Sections 4.2.1 to 4.2.4. When the hammering was successful, the access to the whole physical memory can then be exploited in different ways detailed in Section 4.4.

A very similar technique named *page table replacement* can be used to escape paravirtualized Xen virtual machines [83].

A successful first step of the page table flipping exploit gives the attacker the means to read and write the whole physical memory. From there, the possibilities are almost endless. Some techniques are presented here ordered by the universality of the attack.

### Dump Memory

The least complicated attack is to read and dump the contents of the whole physical memory. It can contain many things interesting for an attacker like cryptographic keys, passwords or other private information. The data gathered can then be, for example, uploaded to a remote server for further investigation. To read the memory efficiently, all 512 PTEs can be set to 512 consecutive physical pages. This enables reading the memory in chunks of 2 MB.

### Modify SUID-root Binary

If the attacker wants to gain root privileges, one possibility is to modify a binary in the memory that has the set-user-ID (SUID) bit set and is owned by root. If the (SUID) bit is set, a binary executes with the permissions of the owner of the binary and not the user that called it. This is, for example, the case for `sudo`, which needs to run as the root user to be able to grant root permissions or `mount`, which must be able to mount user-mountable disks. To modify the binary, its physical address in the memory is required. Then a malicious shell code can either be inserted directly to be executed, or as a less invasive way, the checks done in `sudo` can be removed to allow every caller to gain root privileges.

Seaborn *et al.* [75] used this technique to gain root privileges. To get the physical address of their target, they simply mapped it into their memory and used `/proc/self/pagemap` to translate the virtual address. Because Linux is using shared pages for binaries, the modified code is then executed on the next call. This is, however, not possible anymore. Another method is to search the binary in the memory. The target binary can still be mapped and read, giving the attacker a clear picture of the contents it is looking for in the physical memory. In a second step, the page to look for can be found by iterating over the memory.

Modifying a SUID-root binary is not possible if the attacker process is running in a container that does not give access to any (SUID)-root binaries. This is the case for Android apps running on Chrome OS.

#### Edit UID Field

Another way to gain root privileges is to edit the user-ID field of the attacker process inside the kernel data. Van der Veen *et al.* [80] used this technique to gain root privileges on Android with Drammer . What makes this attack more challenging than the previous one is that it is harder to find the structure in the memory because pages can not be simply compared. The kernel stores multiple credential-related fields, including among others, its real, effective, and saved user and group IDs [80]. Android gives every app unique UIDs summing up to 24 known bytes and also the alignment of the `struct` is fixed. This made it possible for Drammer to find the `struct` and modify it in around 22 seconds on a Nexus 5. Samsung is especially protecting the `struct` cred on their Android phones with a hypervisor. This makes the attack a bit more difficult but does not protect against it [43].

#### Modify Kernel Code

The most invasive way to exploit the full read and write access is to inject shell code directly into the kernel by modifying it. What made this easier than modifying the UID field to gain root privileges is that the location of some parts of the kernel could be obtained with unprivileged instructions. The interrupt descriptor table on x86 contains the physical address of all interrupt handler registers by the operating system and its location could be read with the `SIDT` (Store Interrupt-Descriptor-Table) instruction from user space. Since Intel Cannon Lake and AMD Zen 2, the `SIDT` instructions can be disabled in user space by setting the `UMIP` (User Mode Instruction Prevention) bit, which is done in Linux [14].

### 4.4.2 Instruction Flipping

Another way to exploit Rowhammer is to flip a bit in an instruction of another process or a sandbox which changes the victim's execution in an exploitable way. This can be the change of a register argument to jump to another unconstrained address or the skipping of the password check in the sudo binary. The former was used by Seaborn *et al.* [75] to escape the NaCl sandbox of the Google Chrome Browser. The latter, named opcode flipping, was first used by Gruss *et al.* [39].

#### Operand flipping

Native Client (NaCl) was a sandbox in Google Chrome that was able to run compiled native code in websites and was since deprecated [34]. One of its sandboxing features, among others, was the restriction of all indirect jumps to 32-byte-aligned addresses. Hammering the code that is enclosing every jump to enforce the alignment can change register numbers and therefore enable unrestricted jumps. It is then possible to jump to instructions hidden inside the operands of other instructions, for example, syscall inside a `mov` [75].

```
1 andl $~31, %eax  // Truncate address to 32 bits and mask to
2                  // be 32-byte-aligned.
3 addq %r15, %rax  // Add %r15, the sandbox base address.
4 jmp *%rax        // Indirect jump.
```

**Listing 4.1:** The code of an aligned indirect jump in NaCl.

```
1 48b80f05eb0cf4f4f4f4  movabs $0xf4f4f4f40ceb050f,%rax
```

**Listing 4.2:** This `mov` instruction hides a `syscall` instruction `0f 05` at offset 2.

Operand flipping was used by Seaborn *et al.* [75] to escape the NaCl sandbox of the Google Chrome browser in the first description of two exploits based on Rowhammer in 2015 [75]. NaCl was deprecated in 2020 and is not supported anymore since Juni 2021 [34]. However, the exploit is still interesting because similar techniques could still be used for novel exploits. NaCl was a sandbox to run native code inside the Chrome browser at near-native speed. To sandbox native code from the user space, it used various tricks, including the restriction that all jumps have to be 32-byte aligned. This is ensured by the following instruction sequence used for every sandboxed indirect jump.

By, for example, flipping a bit in the register operand of the `jmp` instruction to change `rax` to `rcx` jumping to an unaligned address is possible. NaCl programs can use a special API to modify code dynamically. This is used to fill the memory with 250 MB of sandboxed indirect jump instructions. This memory area is then hammered to flip any of the operands of the instruction. NaCl programs can also read their own code, thus after hammering the code, it can be checked for successful bitflips. If the exploit finds an exploitable bit flip, it jumps to an unsafe instruction hidden inside another safe one.

### Opcode flipping

Opcode flipping was first presented by Gruss *et al.* [39] to flip bits in an SUID-root binary which changes the program flow in an attacker's intended way. Until then, all exploits used some form of victim spraying where bug portions of the memory are filled with exploitable data like a page table or jump instructions. However, this is not possible with binaries because the kernel puts them in the memory once and maps all read and execute accesses to this one physical location. To counter this problem, Gruss *et al.* [39] presented memory waylaying, a novel technique to put a victim binary in an attacker-controlled location.

To exploit opcode flipping, the first step is to find an exploitable bit flip in an SUID-root binary. Flipping a bit in an instruction often leads to a similar instruction with slightly changed or inverted behavior. We are taking the example from Gruss *et al.* [39] to show in Table 4.1 that many bits in an instruction lead to other valid and valuable instructions. Out of the 255 possibilities for the following byte, only 21 are illegal

| Instruction | Opcode | Binary | Description |
|---|---|---|---|
| JE | 0x74 | 01110100 | jump if equal |
| JNE | 0x75 | 01110101 | jump if not equal |
| JBE | 0x76 | 01110110 | jump if below or equal |
| JO | 0x70 | 01110000 | jump if overflow |
| JL | 0x7C | 01111100 | jump if lower |
| PUSHQ | 0x54 | 01010100 | push quad word |
| XORB | 0x34 | 00110100 | XOR byte |
| HLT | 0xF4 | 11110100 | halt |
| | 0x64 | 01100100 | prefix |

**Table 4.1:** Instruction outcome when flipping one bit of the x86-64 `JE` instruction [39].

instructions. Gruss *et al.* [39] located 29 instructions in the `sudo` binary where a bit flip causes either the skip of the check if the calling user is in the sudoers file or the password check.

### 4.4.3 Type Flipping

A third technique to exploit Rowhammer bit flips applicable in type-safe programming languages like Java or JavaScript is type flipping.

Multiple exploits used this technique to escape the JavaScript sandbox of web browsers to get read and write access to the virtual memory address space of Microsoft Edge or Mozilla Firefox [21,29,30]. When using the analogon that the operating system kernel is to a userland process, what the JavaScript engine is to a JavaScript program, the exploit is actually reasonably similar to page table flipping (Section 4.4.1).

Firefox uses NaN-boxing to store double, integer and pointer values in a 64-bit word without explicit type information. IEEE-754 double-precision floating-point values (doubles) are 64-bit wide and use one sign-bit, 11 bits for the exponent and 52 bits for the fraction. If all 11 exponents bits are 1 the value of the double is *Not-a-Number* (NaN). Additionally, the highest bit of the fraction defines if the NaN is signaling (0) or quiet (1). The content of all other bits is not prescribed by the standard and can have any value. This means that 51 bits of a double can be used to store any arbitrary information if its raw value is higher or equal to the mask value `0xfff80000 << 32` [64].

By flipping one of these 13 highest bits from $1 \rightarrow 0$ a pointer value can be transformed into a double and by flipping a 0 bit from $0 \rightarrow 1$ back into a pointer, respectively. This mechanism is used three times to gain arbitrary read and write access within the Firefox process.

Javascript has typed arrays that are used to store binary information. The goal of the exploit is to build a fake typed array inside a typed array and have a pointer pointing to

it. With that, it is possible to write the header of the fake typed array to have it point at an arbitrary location within the web browser's memory.

To build this fake typed array, the attacker first needs the header of a real typed array to copy it. To get it, a JSString is used, its header contains only the address to the string data.

1. Use memory templating to find one $1 \rightarrow 0$ and one $0 \rightarrow 1$ bit flip.
2. Create an inlined typed array. An inlined typed array has its header next to the data.
3. Use the $1 \rightarrow 0$ bitflip to change the reference of the typed array into a double. This breaks address space layout randomization and reveals the address of the header of the typed array. Because it is inlined, the addresses of data inside the typed array can simply be calculated with an offset.
4. Build a fake JSString inside the typed array. A JSString has a simple header with only an address pointing to the data, which is set to the typed array.
5. Use the $0 \rightarrow 1$ bitflip to transform a double to a reference pointing to the fake JSString. The address of the JSString can be derived from the address of the typed array.
6. Read the header of the typed array through the JSString. At this point, the JSString could be used to read the whole memory, but it is a read-only primitive.
7. Build a fake non-inlined typed array inside the typed array pointing to whatever the attacker wants to access.
8. Use the $0 \rightarrow 1$ bitflip one last time to create a reference to the fake typed array.

Now the pointer of the fake array buffer can be edited through the first array buffer and put everywhere to read and write memory. Because the garbage collector could crash when encountering these fake objects, the fake array buffer is only used to corrupt the header of other valid objects.

### 4.4.4 Read Flipping

RAMBleed by Kwong *et al.* [55] was the first attack that exploited Rowhammer to read the memory of other processes running on the same hardware. It uses the fact that bits flip depending on the content of the aggressor rows. This data dependency means that a bit usually flips from $0 \rightarrow 1$ when the aggressor bit is 1 and from $1 \rightarrow 0$ when the aggressor bit is 0. RAMBleed can read memory with 3-4 bits per second. Kwong *et al.* [55] demonstrated the successful recovery of an RSA 2048-bit private key from an OpenSSH server.

The attack is working by first templating the memory to find as many weak cells as required for the target. Then for every bit, the following memory arrangement is built

| Row Activation Page | Secret |
|---|---|
| Unused | Sampling Page |
| Row Activation Page | Secret |

**Figure 4.4:** The page structure used for RAMBleed. The row activation pages are used to hammer the sampling page with the secret.

and the victim page is put into the two secret pages with *Frame Feng Shui*.

This arrangement is possible because multiple pages or parts of them are usually in the same row, depending on the dram mapping functions. The sampling page is then hammered by accessing the row activation pages and checked if bits flipped in the sampling page. From these bits the secret can be derived. This does also work with ECC memory because the flips are actually not required to persist on the sampling page. ECC bitflip correction adds additional latency to a read which can be measured and used as a side channel to detect if a bit in the sampling page flipped without reading it.

# Chapter 5

# The Half-Double Exploit

In this chapter, we will present our end-to-end half-double Rowhammer exploit targeting Chrome OS. The exploit combines novel attack techniques to compromise the device in under 12 hours from an unprivileged Android app. In Section 5.1, we give an overview of Chrome OS, its security design and the Android runtime. We discuss possible attack scenarios with their different characteristics and define the threat model. Finally, we describe the attack challenges.

Our exploit has to overcome four challenges that are detailed and the respective solutions explained in the following sections. With *side-channel-assisted memory allocation* (cf. Section 5.2) we are able to obtain physically contiguous memory and unscramble the DRAM rows without the need of huge pages. In Section 5.3, we describe how we use *spray children* to bypass virtual address space limitations that come from a performance optimization on ARM. Our target device uses LPDDR4x ECC memory. ECC memory makes bit flips partially dependent on the data in the victim row. In Section 5.4, we propose an alternative to bit-flip templating, *blind hammering*, to solve this problem. A disadvantage of *blind hammering* is that bit flips are less predictable, and therefore, the chance of corrupting a page table entry is relatively high. With *robust bit flip verification using speculative execution* (cf. Section 5.5), we are able to protect our exploit process from being killed in a case of a data-abort interrupt.

In Section 5.6, we discuss factors slowing down our exploit and evaluate the performance and real-world threat. We show how it can be used to dump the device's memory after a successful compromisation. Finally, we discuss non-Rowhammer related countermeasures already in place that reduce the threat of our exploit and possible countermeasures that could further decrease its impact.

## 5.1 Attack Scenario and Threat Model

The main target of the exploit is the Kukui Chromebook. Kukui is the name of the mainboard that is used in multiple Chromebooks, the ASUS Chromebook Detachable CM3, Lenovo Chromebook 10e and Lenovo Chromebook Duet [11]. They are powered by a MediaTek Kompanio 500 (MT8183) CPU with four Arm Cortex-A73 and four Arm Cortex-A53 cores and 4 GB of LPDDR4x. We tested the exploit on two Lenovo Chromebook Duets.

### 5.1.1 Chrome OS

Chrome OS is an operating system by Google, announced in 2009. It was first conceived as a web-only operating system that mainly runs web applications for low-power, low-cost netbook-like devices [67]. Since its introduction, its market share recorded steady growth and in 2020, it took over Apple's Mac OS as the most popular desktop operating system [70].

#### OS Design and Security

Chrome OS was initially based on Ubuntu and developed with the help of Canonical. The base was changed ahead of the first official release to Gentoo. The reason for this step was Gentoo's package manager Portage that is used to build the whole operating system. Nowadays, Chrome OS uses a Linux kernel close to upstream but kept Portage [81]. The Chrome OS version 90.0.443.218 running on our Kukui uses Linux version 4.19. Parts of Chrome OS are open source under the name Chromium OS.

Google is focusing on security in Chrome OS since its creation, with the goal of building the most secure consumer operating system [62]. To achieve that, Chrome OS uses operating system hardening techniques like process sandboxing, toolchain hardening, forced auto-updating and an integrity-checked read-only root partition. Chrome OS was also the first operating system to enforce a fully verified boot path in its first release [2, 62]. And these measures take effect, Chrome OS was affected by only 45 security flaws in its life time [3]. This number is small when compared to over 2700 vulnerabilities in OS X since 2001 [6] and over 2200 in Windows 10 alone [7].

Chrome OS also comes with a developer mode, allowing the user to remove some security features like the verified boot and read-only root partition to be more flexible [4].

#### Android Compatibility

Since 2016, Chrome OS supports running Android apps, with the ability for users to directly install them from the Google Play store [79]. Chrome OS uses a Linux container that contains the Android runtime and is called Android Runtime for Chrome++ (ARC++). To comply with the strict sandboxing rules of Google in Chrome OS, this container is isolated using namespace, seccomp, alt syscalls, SELinux, etc. to keep the attack surface as small as possible [33].

### 5.1.2 Attack Runtimes

Chrome OS was initially designed as a web-browser-only operating system, but it comes with various ways to run applications in the current version. Apart from web apps, Google added support for Android apps and the Google Play store in 2016 [79] and the official support to run Linux applications in virtual machines was added in 2018 [24]. In the following paragraphs, we will detail these different application runtimes and describe their characteristics and features.

**Websites** can contain complex applications developed in JavaScript and it has been shown before that Rowhammer exploits can be built in JavaScript to attack the operating system [41] or the browser [30]. When we started planning the exploit, no exploit written in JavaScript defeating TRR was known and the first tests revealed that half-double Rowhammer leads to almost no bitflips with cache flushing let alone cache eviction.

**Chrome Apps (NaCl)** were an integral part of Chrome OS and also part of the Google Chrome browser on other operating systems. Packaged Chrome Apps, downloaded from the Google Chrome Web Store, have more permissions than normal websites, including the use of the NaCl runtime, which allows running native code inside the Chrome browser. Seaborn *et al.* [75] showed a Rowhammer exploit for NaCl in 2015. Chrome Apps will, however, not be supported anymore from June 2022 in Chrome OS and are already unsupported since June 2021 on all other platforms [56]. This makes NaCl apps a bad choice as an attack runtime.

**Android Apps.** Google started to support Android Apps on Chrome OS in 2014 with the android runtime for chrome (ARC) which was a NaCl application and therefore fairly limited. In 2016 Google presented ARC++, which runs Android applications in a Linux container [33, 79]. The goal of ARC++ was to run every app from the Play Store and it therefore comes with no limitations, including the possibility to run native code with the NDK. The NDK can be used to mount Rowhammer exploits like shown by Van der Veen *et al.* [80] in 2016. More modern hardware and the Android runtime running in a container do however, still require a new development of the exploit.

**Linux applications in virtual machines.** Chrome OS allows users to start a virtual machine with an image provided by Chrome OS called Termina. Termina is a stripped-down Chrome OS Linux kernel with basic userland tools [33]. Users can then run software in LXC inside the Termina virtual macine. Users have full privileges inside these virtual machines, but Chrome OS does not allow the modification of the kernel. For example, the loading of kernel modules is not possible. Although a Rowhammer attack is probably fairly easy to mount, it stays restricted to the virtual machine. Hypervisor escapes were achieved with Rowhammer, but they are beyond the scope of this thesis [72].

**Linux applications in Chrome OS.** With Chrome OS being a Gentoo based Linux distribution, it is possible to run custom applications directly on Chrome OS. However, the execution of binaries is disabled for all partitions except `/`, but this root partition is read-only and integrity checked on all Chromebooks. To make it writable, the developer mode of the Chromebook must be entered to disable secure boot, which deletes all user data. This makes Linux applications running directly on Chrome OS not practical for a Rowhammer exploit targeting normal users. It is, however, useful during development because it allows access to the whole system, including the loading of kernel modules.

An exploit in JavaScript would have had the biggest impact because it is the easiest way to expose a victim to it. However, we came to the conclusion that it was not feasible for us at that point in time. With the Android NDK it is possible to run native code in a container on Chrome OS, which is a great basis for an exploit. It could be hidden in an Android App that is likely to be installed by many users. For example, a tricky puzzle game, an app that does funny AI face transformations or something useful like a ruler to reach hundreds of millions of users [5].

### 5.1.3 Threat Model

Our threat model consists of an up-to-date Lenovo Chromebook Duet also called Kukui running Chrome OS version 90.0.443.218 without any known software vulnerabilities. The user downloads an inconspicuous app from the official Google Play Store and runs it for some time while solving a tricky puzzle game. The app uses Rowhammer to gain full access to the physical memory of the device and can use that to gain root privileges or extract user passwords and other confidential information.

### 5.1.4 Attack Challenges

The exploit attacks Chrome OS from native code running in an Android app with the NDK. We had to solve four challenges that we describe and discuss in the following sections. The first requirement is the access to contiguous memory regions. To not depend on huge pages, we developed a method to detect contiguous memory and reverse-engineer physical address bits to break row scrambling, which we describe in Sections 5.2. The reduced virtual address space on the Kukui requires the help of child processes to fill the memory with page tables. Section 5.3 describes the operation of these *spray children*. The LPDDR4x memory of the Kukui includes ECC. In Section 5.4, we outline the problem this imposes on memory templating and propose a solution, which we call *blind hammering*. The exploit uses the newly discovered half-double pattern by Qazi *et al.* [71] to hammer the TRR memory. Section 5.4.3 details this approach and provides bit flip measurements from two devices. The *blind hammering* technique bears the risk of page table corruption that can cause interrupts that terminate our attack process. To prevent this, we introduce two novel ways to robustly verify the integrity of page table entries in Section 5.5. Finally, we describe the further exploitation with our access primitive we got through Rowhammer in Section 5.6.1.

Interestingly, running the exploit inside the Android NDK did not add any obstacles. The only difference in the code for the NDK versus the native one lies in obtaining the shared memory file for page table spraying. Android does not support the `/dev/shm` file system. It has its own interface called `ASharedMemory` that returns, similarly to opening a shared memory file, a file descriptor that can be mapped with `mmap` [12].

During development, we used a helper process running with root privileges that simplified debugging and prototyping of the exploit app. It uses `setns` to put itself into the namespace of the ARC++ container and can then communicate with the NDK process through pipes to provide access to `/proc/pid/pagemap` and the PTEditor [74].

## 5.2 Side-Channel-Assisted Memory Allocation

The exploit uses the half-double Rowhammer pattern [71]. This pattern requires access to at least five physically adjacent rows with the center one being the victim. Many exploits utilize huge pages to get physically contiguous memory and access to the lowest 21 bits of the physical address [29, 31, 41]. Chrome OS has transparent huge pages through `madvise` activated by default and they can also be used inside NDK applications. But relying on transparent huge pages has the big disadvantage that it is simple to deactivate them with a small negative performance impact. To make our exploit more robust, we developed a technique to detect contiguous memory chunks in a large mapped memory area. Additionally, we can reverse-engineer some physical address bits that help us to almost completely defeat the row scrambling inside the DRAM.

### 5.2.1 Contiguous Memory Detection

To get physically contiguous memory we allocate a large block of memory and search it for contiguously mapped pages. For this, we combine the timing side channel to find rows in the same bank described in Section 4.1.1 together with knowledge from the reverse-engineer DRAM mapping functions. Equation 5.1 shows the DRAM mapping functions we reverse-engineered with the help of the DRAMA tool from Pessl *et al.* [66]. The functions are also shown in Figure 2.4 in the Background section.

$$\begin{aligned} \mathbf{X}_0 &= b_8 \\ \mathbf{X}_1 &= b_{12} \oplus b_{16} \\ \mathbf{X}_2 &= b_{13} \oplus b_{17} \\ \mathbf{X}_3 &= b_{14} \oplus b_{18} \end{aligned} \tag{5.1}$$

The mapping function has four output bits $\mathbf{X}_0$ to $\mathbf{X}_3$, which means that the Kukui's DRAM has 16 banks $B_0$ to $B_{15}$. $\mathbf{X}_0$ is only controlled by address bit 8, of which we know the value from the virtual address and can therefore ignore it for the contiguous memory detection. Calculating the output from the mapping functions $\mathbf{X}_1$ to $\mathbf{X}_3$ for

| $P$ | $d_1$ | $d_2$ | $d_3$ | $d_4$ | $d_5$ | $d_6$ | $d_7$ | $d_8$ | $d_9$ | $d_{10}$ | $d_{11}$ | $d_{12}$ | $d_{13}$ | $d_{14}$ | $d_{15}$ | $d_{16}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_0$ | 8 | 9 | 8 | 9 | 8 | 9 | 8 | 9 | 8 | 9 | 8 | 9 | 8 | 9 | 8 | **1** | ... |
| $P_1$ | 8 | 7 | 8 | 11 | 8 | 7 | 8 | 11 | 8 | 7 | 8 | 11 | 8 | 7 | 8 | **3** | ... |
| $P_2$ | 8 | 9 | 8 | 5 | 8 | 9 | 8 | **13** | 8 | 9 | 8 | 5 | 8 | 9 | 8 | 5 | ... |
| $P_3$ | 8 | 7 | 8 | 7 | 8 | 7 | 8 | **15** | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 7 | ... |
| $P_4$ | 8 | 9 | 8 | 9 | 8 | 9 | 8 | **1** | 8 | 9 | 8 | 9 | 8 | 9 | 8 | 9 | ... |
| $P_5$ | 8 | 7 | 8 | 11 | 8 | 7 | 8 | **3** | 8 | 7 | 8 | 11 | 8 | 7 | 8 | 11 | ... |
| $P_6$ | 8 | 9 | 8 | 5 | 8 | 9 | 8 | 5 | 8 | 9 | 8 | 5 | 8 | 9 | 8 | **13** | ... |
| $P_7$ | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 7 | 8 | **15** | ... |

**Table 5.1:** The eight *page distance* patterns on the Kukui. There are four unique *page distances* highlighted in blue.

increasing values of $b_{12}$ to $b_{18}$ gives the following bank mapping:

$$B_0, B_3, B_2, B_5, B_4, B_7, B_6, B_1, B_0, B_3, B_2, B_5, B_4, B_7, B_6, B_2, B_3, B_0, B_1, B_6, \ldots \quad (5.2)$$

Calculating the distance between equal banks for all eight banks gives the patterns $\mathbf{P}_0$ to $\mathbf{P}_7$ shown in table 5.1, which we call *page distance* patterns.

The *page distance* patterns $\mathbf{P}_4$ - $\mathbf{P}_7$ are equal to $\mathbf{P}_0$ - $\mathbf{P}_3$ shifted by eight. As we only need the pattern and their unique *page distance* we can ignore patterns $\mathbf{P}_4$ - $\mathbf{P}_7$. This is, however, the reason why we cannot reverse-engineer address bit $b_{18}$ for the row unscrambling in Section 5.2.2.

With these *page distance* patterns and a timing side channel to find rows in the same bank available, it is easy to detect contiguous memory. $b_{12}$ is the lowest bit changing the bank, which is also the size of one page. So we loop over our memory with one page distance and use the timing side channel to check if the page is in the same bank as the first page in our memory. If we found a row in the same bank, we put its position in a list and continue. While we loop over non-contiguous memory, the distances between these pages will look random, but when memory mappings become physically contiguous, the distances will match one of the four patterns $\mathbf{P}_0$ - $\mathbf{P}_3$ in table 5.1. Because the patterns are short, pattern-matching can be done after every row found on the same bank without a sophisticated pattern-matching algorithm. The loop continues until a found row breaks the pattern and signals the end of the contiguous memory area.

The timing side channel compares the currently checked page with the first page in the memory area. The random location of this first page determines the bank on which the adjacent rows are found. In a second step, all rows for the remaining seven banks are searched. For this, every already found row is marked by writing a known value into the page. Then to find the next bank, we search the next page that is not marked and use this page as the base for the timing side channel. From this page we loop over the memory again, one page at a time, to find all the rows on the bank of this page. After doing this seven times, all rows of all banks in the contiguous memory area are found.

The speed with which we can scan through memory is mainly determined by the side channel used to determine if two rows are in the same bank. On our target device we can scan $19.05\,\mathrm{MB\,s^{-1}}$ ($\sigma_n = 0.02$) of memory for contiguity. Because we are scanning contiguous areas eight times to find the rows on all banks the time it takes to scan the whole memory is dependent on how many contiguous memory we find. It takes us on average 24.7 s ($n = 20$, $\sigma_{\bar{x}} = 2.7$) to scan 1 GB of memory which finds 173 MB ($n = 20$, $\sigma_{\bar{x}} = 29$) of contiguous memory on a freshly booted system.

### 5.2.2 Row Unscrambling

The memory in the Kukui uses a technique called row scrambling to complicate Rowhammer attacks. It works by applying a function to the row index bits to remap rows and make it harder to find adjacent ones. The row scrambling function used on the Kukui is shown in Equation 5.3.

$$r_i \oplus (r_i[3] \ll 2) \oplus (r_i[3] \ll 1) \tag{5.3}$$

Bit 4 of the row index is XORed with bit 2 and 3. The row index starts at bit 15 of the physical address, therefore the row scrambling function uses the physical address bits 16, 17 and 18. With the knowledge of these bits, the rows can be unscrambled.

In case of the availability of (transparent) huge pages, the unscrambling is a trivial task because we have the 21 lowest physical address bits as the page offset. It gets more difficult when huge pages are disabled and we have to use contiguous memory detection. In that case, we have no knowledge about the address bits 12 and upwards available right away. By using the DRAM mapping functions and the *page distance* patterns of the different banks from the previous section, two of the three bits used in the row remapping can be recovered leaving us with only two different possibilities for the correct row mapping.

Bits 16, 17 and 18 that are already used in the DRAM mapping are also used in the row remapping. We use these bits already indirectly for the calculation and matching of the *page distance* patterns, and can therefore determine these three bits with a 50 percent chance. This 50 percent chance comes from the fact that we cannot determine address bit 18 because the patterns $\mathbf{P}_x$ and $\mathbf{P}_{x+4}$ are equal and just shifted by eight.

To put this into practice, we assume we have a contiguous memory area in bank $B_0$. Like any other of the four distance patterns, it has one distinct distance that only exists once per period, shown in blue in Table 5.1. In the case of $B_0$ that is the distance 1 happening at the end. Two consecutive rows on bank $B_0$ means that the values of $\mathbf{X}_1$ to $\mathbf{X}_3$ in Equation 5.1 stay 0 with a 1 added to $b_{12}$. This can only happen if bits 12 to 18 are 1 and jump to 0. Table 5.2 shows the change of the bits for all four *page distance* patterns at their unique distance.

In other words, when there are two consecutive rows on bank $B_0$, the second row has

| Page Distance | From $b_{18}$ | $b_{17}$ | $b_{16}$ | $b_{15}$ | $b_{14}$ | $b_{13}$ | $b_{12}$ | To $b_{18}$ | $b_{17}$ | $b_{16}$ | $b_{15}$ | $b_{14}$ | $b_{13}$ | $b_{12}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | B | 1 | 1 | 1 | 1 | 1 | 1 | B | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | B | 1 | 1 | 1 | 1 | 1 | 0 | B | 0 | 0 | 0 | 0 | 0 | 1 |
| 13 | B | 1 | 1 | 1 | 0 | 0 | 1 | B | 0 | 0 | 0 | 1 | 1 | 0 |
| 15 | B | 1 | 1 | 1 | 0 | 0 | 0 | B | 0 | 0 | 0 | 1 | 1 | 1 |

**Table 5.2:** Reconstruction of physical address bits via the unique *page distances* of all four patterns.

the address bits $b_{12}$ to $b_{18}$ all set to 0 and from there, these bits are easy to calculate for all offsets from that address. If two consecutive rows are found on bank $B_4$ the second row has bit $b_{18}$ set to 1 instead of zero, inverting the row scrambling result.

## 5.3 Bypassing Virtual Address Space Limitations with Spray Children

In case of a TLB-miss, the MMU has to retrieve up to four levels of page tables from the DRAM which slows down a memory access considerably. To improve performance, it is possible on ARM to reduce the page table hierarchy to three levels. This optimization is used on the Kukui and the current version of Chrome OS. The consequence is a smaller virtual address space of only 512 GB. This is enough for every website and Android app running on Chrome OS, but it imposes a restriction on the maximum amount of page tables that can be created. Equation 5.4 gives the maximum size of physical memory that can be filled with page tables.

$$\frac{\text{VAS}}{\#\text{PTEs} \cdot \text{page size}} \cdot \text{page size} = \frac{512\,\text{GB}}{512 \cdot 4\,\text{kB}} \cdot 4\,\text{kB} = 262\,144 \cdot 4\,\text{kB} = 1\,\text{GB} \tag{5.4}$$

In our testing, we discovered that our victim pages are often filled early in the page table spraying and in these cases 1 GB of page tables is enough. However, having a page table on the victim page is only the first requirement for a successful exploit. After hammering a page table, at least one of its PTEs must reference another page table of the attacker. The chance of this happening increases linearly with more page tables in the memory. To solve this problem, we spawn child processes (*spray children*) that map the same shared memory file as the parent process, filling the memory with additional page tables.

Spraying page tables runs at $79.39\,\mathrm{MB\,s^{-1}}$ ($\sigma_n = 0.24$) on average when using two or more child processes on our Chromebooks. The memory is, therefore, filled with page tables within less than one minute.

### 5.3.1 Child Functions

The *spray children* do not only spray page tables but also have additional functionality that is required later in the exploit. To communicate with the parent process, which controls the operation of the exploit, the children are connected with two pipes for reading and writing. The children wait for new commands in a loop and return information about the success of the operation after its completion.

**Spray page tables** The main functionality of the children is to create page tables to fill the memory. The parent specifies how often every child should map the shared memory file.

**Check mappings** After hammering one or multiple page tables, every child and the parent have to go through all mappings of the shared memory file and compare it with its expected content. If the hammering flipped a bit in a page table, the new page it points to is found in this step.

**Find access page table** If a page table was found in the previous step, the corresponding *access pages* must be found. These are the virtual pages that are mapped through the accessible page table (*address page table*).

**Set address page table random** This sets the value of the first PFN in the *address page table* to a random value within the range of the physical memory. It is required for page table untying.

**Dump memory** The final exploit step which dumps the content of the whole physical memory into a file. This must be implemented by every child because it can end up with the *address-* and *access page table* after page table untying.

**Clean up** Tries to clean up as good as possible by restoring page table entries and unmapping pages.

**Exit** Exits the child.

### 5.3.2 Page Table Untying

After successfully hammering a page table, the use of spray children entails that the *address page table* and *access page table* can belong to different processes. But to access the memory efficiently, both page tables should belong to the same processes. To enable

that, some additional steps are required, which we call *page table untying*. The easiest way to do that is to always give the *address page table* to the process that is already in possession of the *access page table*.

In the first step, the process with the *address page table* ($p_2$) sets the PFN to a random value and the process with the *access page table* ($p_1$) checks if it is pointing to a page table. If it does, the *access page table* of $p_1$ becomes its *address page table* and it uses the same procedure used after successfully hammering to find the new corresponding *access page table*. If it succeeds process $p_1$ has an *address* and *access page table*. It can, however, also fail if the PFN of the initial *address page table* was not pointing to a page table of $p_1$, but of another process. In that case, the untying is tried again and started from the beginning with another random value for the PFN. This process is controlled by the parent by sending commands to the children for every step.

## 5.4 An Alternative to Bit Flip Templating: Blind Hammering

Kim *et al.* [52] showed that Rowhammer bit flips are repeatable because some cells are more vulnerable than others. Many attacks use this property to split themselves up into a templating phase that finds bit flips and an exploitation phase that exploits them [29, 30, 41, 55, 75, 80]. This repeatability is reduced when ECC memory is used on the system under attack.

Figure 5.1 visualizes the reduced repeatability, with eight bits that are error corrected with single error correction, double error detection. The second cell from the right ($c_1$) flips only from zero to one, $c_5$ flips only from one to zero. In the first example, both cells can flip because they contain the vulnerable value and because two flips cannot be corrected, the flips are persistent. In the second and third example, one cell does not contain the vulnerable value resulting in only one bit flip that is corrected and therefore, no flip is persistent.

This effect of ECC memory makes classical memory templating like it was used by many previously shown exploits less effective. We developed two approaches to tackle this problem, *better templates* and *blind hammering*.

### 5.4.1 Better Templates

We know that we want to hammer page table entries and have, therefore, an approximate idea of what the content of our victims will look like. More than half of the bits of the page table entries for our shared file mapping are always the same. We can use this knowledge to make the victim content during templating as similar as possible to the content we want to attack. To do this our two templates are `0x68000555555FD3` and `0x68000AAAAAAFD3`. The bytes filled with `0x55` and `0xAA` are where the PFN is located.

| | | | |
|---|---|---|---|
| Initial | 01**1**010**0** | 01**0**010**0** | 01**1**010**1**0 |
| | ↓ | ↓ | ↓ |
| Flipped | 01**0**010**1**0 | 01**0**010**1**0 | 01**0**010**1**0 |
| | ↓ | ↓ | ↓ |
| Corrected | 01**0**010**1**0 | 01**0**010**0** | 01**1**010**1**0 |
| | **(a)** $c_1$ and $c_5$ flip and are not corrected | **(b)** $c_1$ flips but is corrected | **(c)** $c_5$ flips but is corrected |

**Figure 5.1:** Rowhammer bit flip correction of eight memory cells with ECC. Bits only flip visibly if at least two bits in the victim have the correct value.

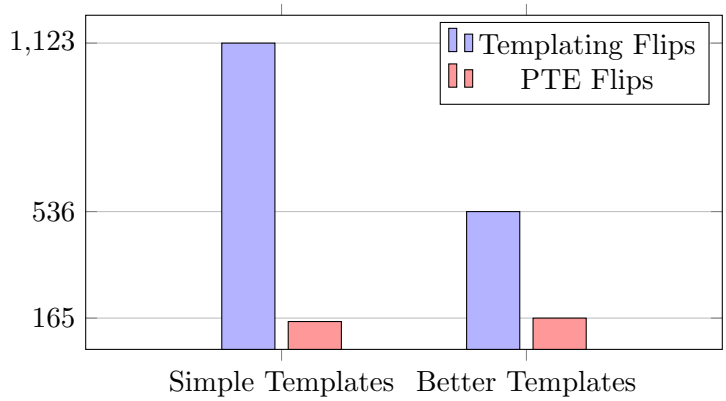



**Figure 5.2:** With better templates, more rows are found that lead to a bit flip in a PTE. On our target device, templating with simple templates finds 1123 flips in the PFN bits after 10.000 hammers. Of these 1123 flips, 153 flip with PTEs in the victim row. Templating with better templates finds 536 flips of which 165 flip, when filled with PTEs.

If we hammer these templates with the opposite value (aggressors = `0x68000555555FD3`, victim = `0x68000AAAAAAFD3` or vice versa) we can make sure that bits only flip in the PFN, because Rowhammer requires the bits in the victim and aggressors to be inverted. However, this also reduces the number of bit flips we get.

If we hammer them with `0xAAAAAAAAAAAAAAAA` and `0x5555555555555555` we increase the bit flips, but there is a high risk of corrupting a page table entry by flipping bits outside the PFN. This is however not a problem, because we have robust bit-flip verification to detect and avoid corrupted PTEs (cf. Section 5.5).

Figure 5.2 shows the results after 10.000 hammers with simple and better templates. The chance of getting a bit flip in a PTE, after finding one during templating, is over two times higher when using better templates.

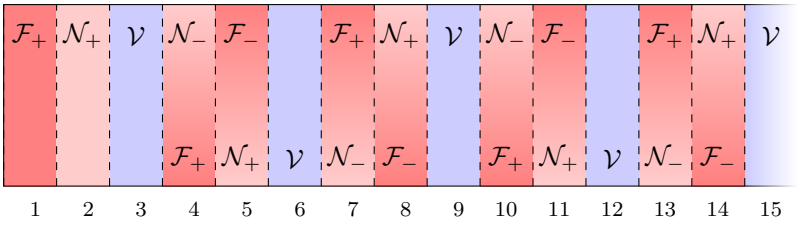



**Figure 5.3:** Zebra pattern for a non-templating page table flipping exploit.

### 5.4.2 Blind Hammering

Our novel concept that does not depend on memory templating at all is *blind hammering*. The idea is to create a zebra pattern of attacker-owned aggressor rows and victim page tables spawning as much memory as possible. This zebra pattern is shown in Figure 5.3. The victim rows are in blue and the aggressors in red. The half-double pattern requires two aggressors on each side of the victim ($\mathcal{V}$). Two near aggressors ($\mathcal{N}_\pm$) in light red and two far aggressors ($\mathcal{F}_\pm$) in dark red that are swapped depending on the hammered victim.

To build this pattern, we first have to be able to get contiguous memory areas through huge pages or contiguous memory detection. In a second step, we unmap the victim rows. Here it is crucial to not unmap every third page but to consider the DRAM-mapping to unmap every third row. When the victims are unmapped, page table spraying with spray children is used to fill as many victim rows as possible with page tables.

The attacker does not know which victims contain page tables and which victims contain weak cells that flip, therefore the bit flips can happen after every Rowhammer and the page tables must be checked periodically.

With the templating method, it is possible to control where in a PTE a bit flip will likely happen. As a result, the chance of corrupting PTE is small and precautions to detect corrupted PTEs are not strictly required. This changes with *blind hammering*. Because the bit flips are not predictable, it is very likely that bits in PTEs flip that corrupt them and would force the kernel to kill the attacker process upon an access. We present different methods to detect corrupted PTEs in Section 5.5.

### 5.4.3 Half-Double Rowhammer

Our exploit uses the half-double pattern that was recently discovered by Qazi *et al.* [71] and is able to induce bit flips in TRR memory. It uses a set of four aggressors, two *near aggressors* directly adjacent to the victim row and two *far aggressors* in the next two rows. The Rowhammer code only accesses the *far aggressors*, this triggers TRR to access the *near aggressors* which in turn leads to bit flips in the victim [71]. For this to work, the contents of the *near aggressors* and the *far aggressors* must be equal.

| System | $N_{Hammers}$ | $UC_{0\to1}$ | $UC_{1\to0}$ | $Flush_{0\to1}$ | $Flush_{1\to0}$ |
|---|---|---|---|---|---|
| $Chromebook_1$ | 23 274 | 27 | 40 | 2 | 5 |
| $Chromebook_2$ | 23 586 | 235 | 2379 | 12 | 101 |

**Table 5.3:** Performance of the half-double pattern with uncacheable memory and the flush instruction on the Chromebooks.

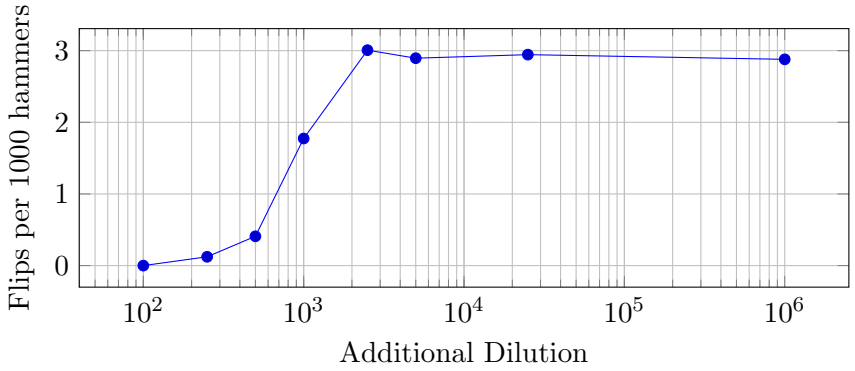



**Figure 5.4:** Bit flips vs additional dilution on $Chromebook_1$. A higher dilution means less accesses to the near aggressors (Equation5.5). If the dilution is too low, TRR starts to refresh the victim row, which leads to less flips. Above the TRR threshold, the number of flips does not change.

The ARMv8 instruction set contains unprivileged cache maintenance instructions. We are using `dc civac` in our exploit to flush the cache lines of the far aggressors. During development, we also used uncacheable memory due to the higher frequency of bit flips. Table 5.3 shows the number of bit flips on our two Chromebooks after hammering for eight hours. The additional delay added by the cache flush instruction can explain the lower number of bit flips when compared to uncacheable memory. With uncacheable memory, the hammer loop does approximately 1.6 as many accesses per refresh interval as the hammer loop with the flush instruction.

The significant discrepancy between the numbers of the two devices makes it hard to evaluate the broader impact of Rowhammer on Kukui Chromebooks. To make any predictions about the Rowhammer vulnerability, a bigger data set would be required.

The odd numbers of flips come from three flips happening at the same time. We mostly saw two flips, but never a single flip per row, due to the error correction in the Kukui's memory.

$$\texttt{RoundRobin(A, A + 4)}^{\texttt{dilution}}\texttt{, A + 1, A + 3} \tag{5.5}$$

Contrary to our expectations, adding additional accesses to the near aggressors apart from the ones done by TRR did not increase the number of bit flips. The number stays

constant and when the *dilution* becomes too small, TRR starts to refresh our victim and the flips decrease. This is shown in Figure 5.4. The *dilution* is defined like shown in Equation 5.5 [71], it reflects additional accesses to the near aggressor, not counting the ones done by TRR. The rightmost point at a *dilution* of $10^6$ corresponds to half-double Rowhammer without any additional accesses to the near aggressors.

## 5.5 Robust Bit-Flip Verification using Speculative Execution

On a system without ECC DRAM the predictability of bit flips, their location and direction is very high [52]. This predictability is useful when selecting a victim row for page table hammering. It is possible to exclude any bit flips that would corrupt a PTE in a way that forces the kernel to kill the process referencing this page table. This happens for example on the MT8183 CPU when the PFN bits 40-47 are set to 1. In this case, the process does not receive a segmentation fault exception that could be caught and ignored. Instead, the kernel kills the process with a `KILL`, which cannot be caught and always leads to the termination of the process.

We present two possible approaches to detect corrupted PTEs without triggering an interrupt in our process. The usage of `vfork()` is faster and more straightforward but not available on every system and can be deactivated easily. The second approach uses speculative execution within the CPU to access the address-under-test without leaving any architectural traces and, therefore, no interrupt.

### 5.5.1 Architectural Method: vfork()

To check if a PTE is corrupted, the corresponding virtual address must be accessed somehow without triggering an interrupt in the attack process. One approach to achieve this is an access from another process that uses the same page tables as the attack process. A process like this can be created with the `vfork()` system call.

`vfork()` creates, similarly to `fork()`, an exact copy of the calling process with the only difference that the page tables are not copied. Its primary purpose is to provide a faster version of `fork` for child processes that immediately execute another process by calling an `exec` function. In that case, copying all page tables of the parent process is unnecessary. But working with the same page tables imposes some restrictions on what the child process is allowed to do which do not matter for its intended purpose. The child must not modify any data except the variable which contains the process id that is returned by `fork()`. It furthermore must not call any functions except `_exit()` or any of the `exec()` functions. As our child accesses an address that could potentially cause a signal, it is important to ensure that a potential signal handler catching this signal does not change any memory. If that is the case, the behavior of the child does comply with the POSIX standard.

```
1  int check_address(volatile uint64_t *address)
2  {
3    int status;
4    int pid = vfork();
5    if (!pid) {
6      // Child process
7      flush((void*) address);
8      *address;
9      _exit(0);
10   }
11   // Parent process resumes
12   wait(&status);
13
14   if (WIFSIGNALED(status)) {
15     // Address is not safe
16     return 1;
17   }
18   // Address is safe
19   return 0;
20 }
```

**Listing 5.1:** Using vfork() to check if `address` is safe to access.

Listing 5.1 shows the minimal code necessary to check an address with `vfork()`. `vfork()` returns, like `fork()`, the PID of the child in the parent and 0 in the child. The child then flushes the address, which is important because the page table entry is not read if the address is in the L1 cache. The address is then accessed, triggering an interrupt if it is ivalid. If valid, the child exits with the status 0. When forking with `vfork()`, the parent sleeps until the child exits or executes another binary. The `wait(&status)` call is required to get the exit status of the child process, which is checked in the following `if`. If the child was killed by the operating system, the function returns 1 otherwise 0.

A single address verification takes 0.057 ms ($n = 10\,000$, $\sigma_{\bar{x}} = 0.005$). The access of the address takes only a small fraction of that. The overhead of the two system calls takes most of the time. To reduce this overhead, the child process can also check multiple addresses inside a loop. Doing that takes 0.206 ms ($n = 10\,000$, $\sigma_{\bar{x}} = 0.009$) for 4 MB of memory or 1000 PTEs. In case this range check fails, every address must be checked individually.

Under its intended use, the behavior of the `fork()` system calls is indistinguishable from `vfork()` for the program calling it. Therefore the POSIX standard allows to implement `vfork()` as an alias for `fork()`, rendering this check non-functioning. In the early days of Unix operating systems, `fork()` was copying all page tables on its invocation, which is why BSD introduced `vfork()`. Linux, however, is copying the page tables using copy-on-write, reducing the added overhead drastically. Thus `vfork()` could be changed to a an alias of `fork()` without any consequences as a countermeasure against this check.

```
1 if (misprediction)
2   access(probe + (*pointer1 & 1) + ... + (*pointer5 & 1));
3 if (flush_reload(probe) == CACHE_HIT)
4   // Report valid address
```

**Listing 5.2:** Using speculative execution to check if addresses are safe to access

### 5.5.2 Novel Speculative Oracle

Depending on an optional implementation of a system call that can easily be changed is a big disadvantage. We are therefore using exception suppression by mistraining the branch predictor similarly to Lipp *et al.* [59] for bit flip verification. With that technique, we are able to access potentially corrupted addresses during speculative execution in which the CPU must not change the architectural state and can, therefore, not raise an exception. Finally, we use a cache side channel to detect if the access was successful, which means that the PTE is not corrupted.

Listing 5.2 shows the heart of the code, used to check if five addresses are accessible safely or not.

The branch predictor learns from previous branch outcomes to decide which path to execute speculatively in the future when the required branch condition is not available. We use this behavior to mistrain it by executing the branch 20 times, with the input to execute our probing code. Then we set our pointers to the values we want to verify and flush `probe` and the variables that are part of the branch condition. Now when reaching the branch, variables must be fetched from the DRAM which takes time, so the CPU recalls the previous branch outcomes and executes our probing code speculatively. To access probe, it first has to read from `pointer1` to `pointer5`, which can have two outcomes:

- All pointers are **valid**. The CPU successfully calculates the offset of `probe` and accesses it. This caches the contents of `probe` to make future accesses faster.
- One or more pointers are **invalid**. The MMU raises an exception upon access because it can not translate the virtual address. The CPU cannot calculate the offset of `probe` and therefore not access it.

After speculatively executing our verification code, we check if `probe` is cached or not and therefore, if the pointers are valid or not. We do this by measuring the time it takes to read `probe` with `clock_gettime(CLOCK_MONOTONIC, &tp);`. For this measurement, an exact timing source is required, `clock_gettime` is implemented as a virtual system call and has therefore a small enough overhead to give reliable timing.

Our speculative execution is different from the one from a normal Spectre attack in that we only need to know if an address is cached and not which one of an array of addresses is cached. Therefore, we can optimize our probing code by limiting the values read from

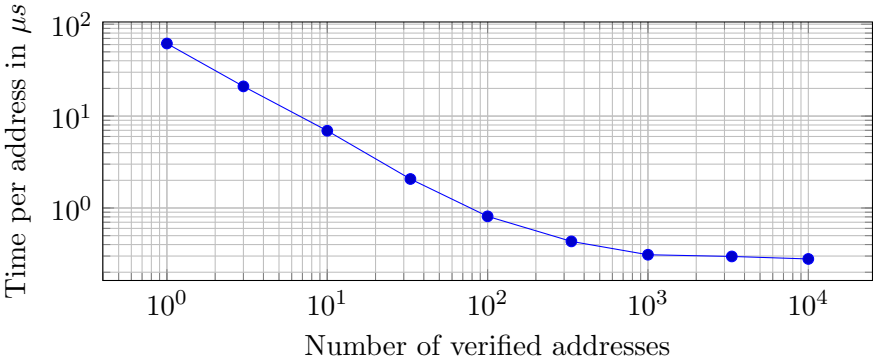



**Figure 5.5:** The time it takes to verify one address when checking multiple addresses in the child with `vfork()`. The overhead of the `vfork()` system call becomes negligible when checking roughly 1000 addresses.

the pointers to 1 or 0 with the `& 1`. This keeps the offset always smaller than 64 and we only have to check one cache line.

If the access time to `probe` is below the threshold for a cached access, the tested addresses are valid with almost 100% certainty. One thing we have to ensure is that we prevent prefetching of `probe` by putting `probe` at the beginning of a page. So if `probe` is cached it was accessed during the speculative execution, which means that all accesses to the pointers were successful. This translates to a false positive rate (detecting an unsafe address as safe) of practically 0.

Failing to mistrain the branch predictor, a too-quick correction of the branch outcome or delays in the *flush+reload* measurement can lead to `probe` being classified as uncached, even if all pointers are valid. To reduce this false-negative rate, we repeat the measurement multiple times for all pointers. If `probe` is cached, all pointers are valid and the verification can be concluded early.

$$\frac{1\,\text{GB} \cdot 512}{4\,\text{kB}} \cdot \frac{0.008\,\text{ms}}{5\,\text{pointers}} \approx 215\,\text{s} = 3\,\text{min}\ 35\,\text{s} \tag{5.6}$$

We tested our oracle by passing it five valid pointers or one corrupted and four valid pointers. The success rate of a correct classification was already 99.01 % after only one run of the oracle. The average runtime to test 5 pointers is 0.008 ms ($n = 10000$, $\sigma_{\bar{x}} = 0.002$) for both cases. We are always checking five addresses at once to improve performance. In the case that the verification fails, we check every address individually. Checking 1 GB of page tables takes approximately 3 min 35 s as shown in Equation 5.6.

| Step | Method | Duration | Comment |
|---|---|---|---|
| 1 | Huge Pages | $< 1s$ | deactivatable |
| | Contiguous Mem. Detection | 24.7 s | per 1 GB |
| 2 | Spray Children | 12.6 s | per 1 GB |
| 3 | Blind Hammering | 1 h | highly fluctuating |
| 4 | Speculative Oracle | 3 min 35 s | per 1 GB |
| | `vfork()` | 27 s | deactivatable |
| 5 | Dump Memory | 8.5 s | per 1 GB |

**Table 5.4:** Average duration of the exploit steps on the Chromebook$_2$. The duration of all steps except the hammering is steady between runs and devices. Only step 3 can range from minutes to hours.

### 5.5.3 Robust Bit-Flip Verification Evaluation

The performance results of the two methods show that the speculative oracle is faster than our `vfork()` approach when checking a small number of addresses. The reason for that is that the overhead of the `vfork()` system call is a lot higher than the measurement itself. The impact of this overhead is reduced when multiple addresses are checked within one run of the child. When checking 1000 addresses the `vfork()` oracle is a lot faster than the speculative oracle (0.2 ms vs 1.6 ms). Checking many addresses at once is a viable approach because corrupted PTEs are rare and only in these cases a recheck of every individual address is required.

The disadvantage of the `vfork()` oracle is that it can be easily deactivated in the kernel by making `vfork()` an alias of the `fork()` system call without a big performance penalty. However, it is possible to check if `vfork()` is implemented as intended by changing a variable in the parent from the child and checking for the change in the parent. If the change is visible in the parent, the `vfork()` does work and if not, the speculative oracle is used.

## 5.6 Exploit Evaluation

Our exploit is able to attack and compromise Chrome OS, from an unprivileged Android app using Rowhammer. To achieve that, we use the novel hammer pattern half-double Rowhammer to defeat TRR protected memory. The exploit does not require huge pages, which can be deactivated easily and hammers without templating to be independent of ECC correction. Finally, its immunity to corrupted data due to our novel bit-flip verification, makes it very reliable. Table 5.4 summarizes the duration of the attack steps.

The run time can be split into the different steps of the exploit. Searching 1 GB of memory for contiguous blocks takes less than 30 seconds and finds on average 173 MB ($n = 20$, $\sigma_{\bar{x}} = 29$) on a freshly booted system. The found contiguous memory becomes

less when the system is running for longer. When huge pages are available, this step takes less than a second. Spraying the memory with page tables takes less than a minute when multiple spray children are used.

The hammering has the biggest impact on the run time because it is highly dependent on the Rohammer vulnerability of the DRAM and luck to find a bit-flip quickly. Due to the fact that we are not able to reconstruct physical address bit 18, our row unscrambling is only correct for 50 % of the found contiguous memory areas. This doubles the theoretical minimum hammer time. The big discrepancy in the bit flip frequency on our two tested Chromebooks makes a prediction on the broader vulnerability of these devices impossible. The hammering is additionally prolonged because not every victim row is filled with a page table after spraying. Because of these factors, the hammering phase can take a few minutes to many hours in the worst case for the attacker, on average it takes approximately 1 h.

When employing the non-templating technique, all shared memory mappings must be checked regularly to not miss successful flips in page table entries. This takes less than 30 seconds when using `vfork()`. With the speculative oracle, it takes approximately 3.5 minutes to check 1 GB of page tables.

### 5.6.1 Dumping Memory

Our goal was to develop a simple proof of concept exploit to show that Rowhammer exploits are still possible on a modern, fully patched secure operating system with TRR and ECC DRAM using the half-double hammer pattern. After gaining the physical memory read and write primitive, the final phase of the exploit is kept easy and only serves as a simple demonstration of the access capabilities. The memory of a running system contains many interesting secrets, be it decryption keys of encrypted drives, private keys of asymmetric cryptography systems, or text entries by the user like a password or credit card information. Our exploit dumps the physical memory's entire content into a file that is saved and could be uploaded to a remote server for later analysis.

To read as fast as possible, we set all 512 page table entries of our address page tables to individual page frame numbers. This allows us to read 2 MB of memory at once. Before that, we must however, flush the TLB to force the MMU to read the translations from the memory. To do this, we read a chunk of our mappings used during page table spraying. The biggest performance bottleneck in this phase is the write-speed of the drive we are dumping the data on. It takes our exploit an average of 33.8 s ($n = 5$, $\sigma_{\bar{x}} = 0.6$) to dump the whole memory, which translates to a write-speed of approximately $121\,\mathrm{MB\,s^{-1}}$ which we confirmed with `dd`.

### 5.6.2 Real-World Threat

Compared to the hammering, all other phases of the exploit run quickly. Therefore, the overall attack time does not change significantly if the attack process is stopped during hammering and has to start from the beginning repeatedly. This makes the attack more dangerous because it is enough to run the malicious app only for an hour per day which is, for example, realistic for a puzzle game.

We conclude that the app is a real-world threat to the security of Chrome OS users that run Android apps on their devices.

Our app does not require any Chrome OS specific functionality and is running in an Android app on ARM. The chance of this exploit also working on Android phones is therefore high. We did, however, not investigate this further due to time constraints.

### 5.6.3 Existing and Potential Countermeasures

Existing *detection* based countermeasures can be circumvented by our exploit or are ineffective. The Android NDK allows writable and executable pages. This enables the encryption of the hammer code to evade the detection through static code analysis. Software that tries to detect Rowhammer attacks through run time traces like ANVIL [20] have the same effect as TRR when protecting the victim page. This does not prevent half-double Rowhammer but actually enable it. *Neutralization* countermeasures like ZebRAM [54] do also not consider half-double Rowhammer and are therefore not effective. *Elimination* based countermeasures like disabled huge pages, ECC, TRR and row scrambling are evaded by our exploit.

Our attack exploits the design of the ARC++ that is only running inside a Linux container and therefore not properly separated from the host kernel. Running the Android runtime in a virtual machine would add another separation layer between Android apps and the host. This would still allow the current exploit to gain access to the memory of the virtual machine and it could in the worst-case, compromise other apps. However, an attack on the Chrome OS host would require an exploit to escape the virtual machine, which adds a lot of difficulties.

Google's steps to harden Chrome OS as strongly as possible also protect against worse impacts of our exploit. The fully verified boot path and integrity checked root partition prevent the exploit from keeping elevated permissions across device reboots or even exploit restarts without additional software vulnerabilities, which are not the scope of this work.

Walker *et al.* [82] proposed changes to the physical layout of the DRAM cells that could eliminate the Rowhammer problem altogether. The two effects causing Rowhammer are electrons wandering through the active region to other cells and capacitive crosstalk between word lines (cf. Section 2.4.1). A MOSFET transistor with a vertical channel where the active region is not shared with other transistors would prevent the wandering

of electrons. To block capacitive crosstalk metal shielding can be used between the word lines like in almost all cables for high transmission speeds like USB or Ethernet.

Another option is a more advanced memory error correction scheme that would be able to detect and potentially correct any amount of memory errors. This would not only protect against Rowhammer induced bit flips but also against bit flips from other sources like cosmic rays. This would very likely also require a hardware change that is however smaller, than the complete redesign of DRAM cells, when done right.

# Chapter 6

# Conclusion

In this thesis, we examined if Rowhammer exploits are still possible by developing a novel exploit to attack Chrome OS using the half-double pattern. By studying previously shown exploits and defenses and Chrome OS's security architecture and possible attack surfaces, we identified different challenges and developed reliable and future-proof solutions for our exploit.

To run native code on Chrome OS, we used the ability of Chrome OS to run Android apps with ARC++. Because ARC++ runs only in a container, we can use the Android NDK to run the same Rowhammer exploit technique as Seaborn *et al.* [75]. With a novel way to automatically detect contiguous memory blocks that can be used simultaneously to unscramble DRAM rows, we made our exploit independent of huge pages,. Removing the templating step made our exploit more effective on ECC memory which is used in an increasing number of devices. We introduce the idea of spray children to allow the spraying of page tables on devices with reduced virtual address space and double the spraying speed. Our final contribution, robust bit-flip verification using speculative execution makes the exploit more reliable. Since all these techniques do not rely on Chrome OS specifics, there is a high chance that this attack works on Android phones as well.

Therefore, the exploit is a real threat to Chrome OS users that occasionally run Android apps on their Chromebooks and potentially also Android users. We conclude that Rowhammer exploits are still possible and practical and, therefore, remain a security issue for computer systems worldwide. Furthermore, current countermeasures to the problem, like TRR, are not sufficiently protecting the DRAM from Rowhammer, and manufacturers must continue searching for reliable solutions. It becomes, however, increasingly difficult to develop novel exploits that evade all countermeasures when compared to the first kernel privilege escalation by Seaborn *et al.* [75]. This gives hope that Rowhammer will be mitigated in the near future.

# Bibliography


[1] Actions required to mitigate speculative side-channel attack techniques. https://www.chromium.org/Home/chromium-security/ssca. accessed: June 2021.

[2] Chrome OS Security Overview. https://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/security-overview. Accessed: June 2021.

[3] Chrome OS Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-20320/Google-Chrome-Os.html. Accessed: June 2021.

[4] Chromium OS Developer Guide. https://chromium.googlesource.com/chromiumos/docs/+/HEAD/developer_guide.md. Accessed: June 2021.

[5] FaceApp. https://www.faceapp.com/. Accessed: June 2021.

[6] Mac OS X Vulnerability Statistics. https://www.cvedetails.com/product/156/Apple-Mac-Os-X.html?vendor_id=49. Accessed: June 2021.

[7] Windows 10 Vulnerability Statistics. https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html?vendor_id=26. Accessed: June 2021.

[8] Mitigations Available for the DRAM Row Hammer Vulnerability. http://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability, Mar. 2015.

[9] Row hammer privilege escalation. https://support.lenovo.com/at/en/product_security/row_hammer, Sept. 2015.

[10] About the security content of Mac EFI Security Update 2015-001. https://support.apple.com/en-us/HT204934, Jan. 2017.

[11] Developer Information for Chrome OS Devices. https://www.chromium.org/chromium-os/developer-information-for-chrome-os-devices, 2021. Accessed: June 2021.

[12] NDK Shared Memory. https://developer.android.com/ndk/reference/group/memory, June 2021.

[13] *proc(5) Linux User's Manual*, 2021.

[14] AMD Corporation. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, March 2021. Revision: 3.37.

[15] Arcangeli, A. Linux: Commit 71e3aac thp: transparent hugepage core. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=71e3aac0724ffe8918992d76acfe3aad7d8724a5`, Jan. 2011.

[16] ARM. *Arm Architecture Reference Manual Armv8, for A-profile architecture*, 2013.

[17] ARM. *Arm Cortex-A53 MPCore Processor, Technical Reference Manual*, 2013.

[18] ARM. ARM Cortex-A Series Programmer's Guide for ARMv8-A. `https://documentation-service.arm.com/static/5fbd26f271eff94ef49c7020`, Mar. 2015.

[19] ARM. *Arm Cortex-A73 MPCore Processor, Technical Reference Manual*, 2015.

[20] Aweke, Z. B., Yitbarek, S. F., Qiao, R., Das, R., Hicks, M., Oren, Y., and Austin, T. ANVIL: Software-based protection against next-generation Rowhammer attacks. *ACM SIGPLAN Notices* (2016).

[21] Bosman, E., Razavi, K., Bos, H., and Giuffrida, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P* (2016).

[22] Bovet, D., and Cesati, M. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[23] Bramley, J. Caches and Self-Modifying Code. `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/caches-and-self-modifying-code`, Sept. 2013.

[24] Brangers, G. Linux Apps Land On Beta Channel For A Lot Of Chromebooks. `https://chromeunboxed.com/news/chrome-os-beta-channel-linux-apps-update`, Aug. 2018.

[25] Brasser, F., Davi, L., Gens, D., Liebchen, C., and Sadeghi, A.-R. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium* (2017).

[26] Christensen, A. Reduce resolution of performance.now. `https://bugs.webkit.org/showbug.cgi?id=146531`, July 2015.

[27] Cojocar, L., Razavi, K., Giuffrida, C., and Bos, H. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P* (May 2019).

[28] de Mooij, J. Warp: Improved JS performance in Firefox 83. `https://hacks.mozilla.org/2020/11/warp-improved-js-performance-in-firefox-83/`, Nov. 2020.

[29] de Ridder, F., Frigo, P., Vannacci, E., Bos, H., Giuffrida, C., and Razavi, K. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security Symposium* (2021).

[30] Frigo, P., Giuffrida, C., Bos, H., and Razavi, K. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P* (2018).

[31] Frigo, P., Vannacci, E., Hassan, H., van der Veen, V., Mutlu, O., Giuffrida, C., Bos, H., and Razavi, K. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P* (2020).

[32] Ghasempour, M., Lujan, M., and Garside, J. ARMOR: A Run-time Memory Hot-Row Detector. `http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer`, 2015.

[33] Google. Running Custom Containers Under Chrome OS. `https://chromium.googlesource.com/chromiumos/docs/+/8c8ac04aed5d45bb6a14605c422dbbd01eeadf15/containers_and_vms.md`. last accessed: May 2021.

[34] Google. Native Client Documentation. `https://developer.chrome.com/docs/native-client/`, June 2021.

[35] Google. Start building apps for Chrome OS. `https://developer.android.com/topic/arc`, Feb. 2021.

[36] Govindavajhala, S., and Appel, A. W. Using memory errors to attack a virtual machine. In *S&P* (2003).

[37] Greenberg, M. Row Hammering: What it is, and how hackers could use it to gain access to your system. `https://blogs.synopsys.com/committedtomemory/2015/03/09/row-hammering-what-it-is-and-how-hackers-could-use-it-to-gain-access-to-your-system/`, Mar. 2015.

[38] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *ESSoS* (2017).

[39] Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., and Yarom, Y. Another Flip in the Wall of Rowhammer Defenses. In *S&P* (2018).

[40] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[41] Gruss, D., Maurice, C., and Mangard, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA* (2016).

[42] Herath, N., and Fogh, A. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings* (Aug. 2015).

[43] Horn, J. Mitigations are attack surface, too. https://googleprojectzero.blogspot.com/2020/02/mitigations-are-attack-surface-too.html, Feb. 2020.

[44] Hwang, A. A., Stefanovici, I. A., and Schroeder, B. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *ASPLOS* (2012).

[45] Irazoqui, G., Eisenbarth, T., and Sunar, B. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY* (2018).

[46] Jacob, B., Ng, S., and Wang, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[47] Low Power Double Date Rate 4. Standard, JEDEC, Feb. 2017.

[48] DDR3 SDRAM. Standard, JEDEC, Nov. 2008.

[49] DDR4 SDRAM. Standard, JEDEC, June 2017.

[50] Jung, M., Rheinländer, C. C., Weis, C., and Wehn, N. Reverse engineering of DRAMs: Row hammer with crosshair. In *International Symposium on Memory Systems* (2016).

[51] Kim, D.-H., Nair, P. J., and Qureshi, M. K. Architectural support for mitigating row hammering in DRAM memories. *IEEE Computer Architecture Letters 14* (2015).

[52] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA* (2014).

[53] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *S&P* (2019).

[54] Konoth, R. K., Oliverio, M., Tatar, A., Andriesse, D., Bos, H., Giuffrida, C., and Razavi, K. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *USENIX OSDI* (2018).

[55] Kwong, A., Genkin, D., Gruss, D., and Yarom, Y. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P* (2020).

[56] Laforge, A. Changes to the chrome app support timeline. https://blog.chromium.org/2020/08/changes-to-chrome-app-support-timeline.html, Aug. 2020.

[57] Lanteigne, M. How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware. http://www.thirdio.com/rowhammer.pdf, 2016.

[58] Lipp, M., Aga, M. T., Schwarz, M., Gruss, D., Maurice, C., Raab, L., and Lamster, L. Nethammer: Inducing Rowhammer Faults through Network Requests. In *SILM Workshop* (2020).

[59] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[60] Luo, T., Wang, X., Hu, J., Luo, Y., and Wang, Z. Improving TLB Performance by Increasing Hugepage Ratio. In *IEEE/ACM CCGRID* (May 2015).

[61] McIlroy, R. Firing up the Ignition interpreter. `https://v8.dev/blog/ignition-interpreter`, Aug. 2016.

[62] Messmer, E. Google sheds light on Chrome OS Netbook security. `https://www.computerworld.com/article/2762946/google-sheds-light-on-chrome-os-netbook-security.html`, Mar. 2010.

[63] Micron. ECC Brings Reliability and Power Efficiency to Mobile Devices. `https://media-www.micron.com/-/media/client/global/documents/products/white-paper/ecc_for_mobile_devices_white_paper.pdf?la=en&rev=b892d7cdf84a495d83b2560b2fe523ef`, Feb. 2017.

[64] Mozilla. Firefox source code /js/public/value.h. `https://github.com/mozilla/gecko-dev/blob/master/js/public/Value.h`, may 2021.

[65] Mutlu, O., and Kim, J. S. RowHammer: A Retrospective. *IEEE TCAD* (2020).

[66] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[67] Pichai, S. Introducing the Google Chrome OS. `https://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html`, Sept. 2017.

[68] Poddebniak, D., Somorovsky, J., Schinzel, S., Lochter, M., and Rösler, P. Attacking deterministic signature schemes using fault attacks. In *EuroS&P* (2018).

[69] Potash, H. Branch predicting computer, 1981.

[70] Protalinski, E. Chromebooks outsold Macs worldwide in 2020, cutting into Windows market share. `https://www.geekwire.com/2021/chromebooks-outsold-macs-worldwide-2020-cutting-windows-market-share/`, Feb. 2021.

[71] Qazi, S., Kim, Y., Boichat, N., Shiu, E., and Nissler, M. Half-Double: Next-Row-Over Assisted Rowhammer. `https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html`, May 2021.

[72] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., and Bos, H. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium* (2016).

[73] Schroeder, B., Pinheiro, E., and Weber, W.-D. DRAM Errors in the Wild: A Large-Scale Field Study. *Commun. ACM* (2011).

[74] Schwarz, M., Lipp, M., and Canella, C. misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8. `https://github.com/misc0110/PTEditor`, 2018.

[75] Seaborn, M., and Dullien, T. Exploiting the DRAM Rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, Mar. 2015.

[76] Shin, C., Kim, K., Kim, J., Ko, W., Yang, Y., Lee, S., Jun, C., and Kim, Y. Fast, exact, and non-destructive diagnoses of contact failures in nano-scale semiconductor device using conductive afm. *Scientific Reports 3* (06 2013), 2088.

[77] Spessot, A., and Oh, H. 1t-1c dynamic random access memory status, challenges, and prospects. *IEEE Transactions on Electron Devices 67*, 4 (2020), 1382–1393.

[78] Tatar, A., Krishnan, R., Athanasopoulos, E., Giuffrida, C., Bos, H., and Razavi, K. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX ATC* (2018).

[79] Taylor, E., and Reid, D. The Google Play store, coming to a Chromebook near you. `https://blog.google/products/chromebooks/the-google-play-store-coming-to/`, May 2016.

[80] van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS* (2016).

[81] Vaughan-Nichols, S. J. The secret origins of Google's Chrome OS. `https://archive.is/TODn1`, Mar. 2013.

[82] Walker, A. J., Lee, S., and Beery, D. On DRAM Rowhammer and the Physics of Insecurity. *IEEE Transactions on Electron Devices* (2021).

[83] Xiao, Y., Zhang, X., Zhang, Y., and Teodorescu, R. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium* (2016).

[84] Zbarsky, B. Clamp the resolution of performance.now() calls to 5us. `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab`, July 2015.