David Bidner, BSc

All The Small Flips

Finding Rowhammer Targets in the Wild

Master's Thesis

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to
Graz University of Technology

Supervisor
Daniel Gruss

Assessor
Stefan Mangard

Institute of Applied Information Processing and Communications

Faculty of Computer Science and Biomedical Engineering

Graz, October 2018

Affidavit

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

Date	Signature

Abstract

Microarchitectural attacks exploit design choices made by hardware vendors. Cache attacks, or attacks like Meltdown and Spectre, target the CPU. Microarchitectural attacks either target design choices or exploit bugs introduced by mistakes in the hardware or the microcode. Rowhammer attacks exploit a design flaw in DRAM chips. Hardware vendors steadily decreased sizes and lowered refresh rates of the cells. With specially crafted memory access routines it is possible to access DRAM cells fast enough to create interference between cells, which will cause them to change their load, hence, changing their logical state.

Rowhammer attacks use this phenomenon to target memory areas where bitflips would benefit the attacker. Researchers have shown that bitflips in page tables can cause privilege escalation and bypass security mechanisms implemented in modern operating systems. It has also been shown that similar attacks work for memory chips used by solid-state disks. Besides flipping bits in page tables, researchers showed that when flipping bits in executables, the program's behaviour can change. With such changes, it is possible for an attacker to abuse programs for privilege escalation, or to bypass authentication-checks.

In this thesis, we present a way of testing binaries for possible execution-path changes introduced by bitflips. In our work, we pre-define an outcome for a program and then search for all single bitflips which cause the program to behave in the desired way. We scan the entire address space of the program, including all dynamically loaded libraries. We show results for searched bitflips in programs used for authentication on Linux-based systems. We bypass user privilege checks, which lead to privilege escalation, or make login possible without knowing the user's password. We also show a bypass of HTTP basic authentication, allowing an attacker to download files which unauthenticated users are not allowed to access. In addition to searching for bitflips in executable files, we also look at possible other attack vectors for Rowhammer. We show that bitflips applied to the runtime of cryptographic calculations can break assumptions made by the communicating parties and can even allow key leakage. We apply bitflips to the implementation of AES-GCM in OpenSSL and show how Rowhammer can be used to cause reusing of nonces.

With our work, we want to increase the awareness of Rowhammer and show how software security is affected by bitflips. We call on to all vendors of hardware to not forget to keep their systems secure and do not put lower prices and higher performance ahead of security, which would harm their users.

Kurzfassung

Angriffe auf die Mikroarchitektur zielen meistens auf Fehler von Hardwareherstellern ab. Attacken auf Caches nutzen hierbei ein gewolltes Verhalten des Systems aus. CPU Lücken wie Meltdown und Spectre machen sich Fehler im Design der Hardware zu Nutzen, welche vom Hersteller entweder durch Mikrocode Updates oder einem Tausch der CPU berichtigt werden müssen. Rowhammer macht sich einen Designfehler in DRAM Chips zu Nutze. Hersteller dieser Speicherchips produzieren immer kleinere Chips mit geringeren Refresh-Zyklen. Spezielle Reihenfolgen von Speicherzugriffen ermöglichen es Interferenzen zu erzeugen welche Ladungsänderungen in benachbarten Speicherzellen verursachen, diese können dadurch ihren logischen Zustand wechseln.

Rowhammer Angriffe machen sich dieses Verhalten zu Nutze und zielen damit auf Speicherbereiche ab, welche durch eine Änderung dem Angreifer einen Vorteil verschaffen. Forscher haben gezeigt, dass es möglich ist mit Bitflips in Page Tables Privilegien einen Superusers zu bekommen. Ebenso wurde gezeigt, dass ähnliche Angriffe auch auf Speicherchips in Solid-State-Disks möglich sind. Neben Flips in Page Tables wurde auch gezeigt, dass Änderungen in ausführbaren Files Folgen auf das Verhalten des Programms haben, sie ändern dieses durch das Wechseln eines einzigen Bits. Dies kann zum Beispiel dazu führen dass Berechtigungsüberprüfungen umgangen werden.

Wir zeigen eine Möglichkeit Programme auf solche Verhaltensänderungen durch Bitflips zu testen. Wir geben hier ein gewünschtes Verhalten vor und suchen danach nach allen möglichen Bitflips welche das Verhalten in den gewünschten Zustand ändern. Im Gegensatz zu früheren Arbeiten haben wir diesen Vorgang automatisiert um eine größere Anzahl von Programmen abzudecken, zusätzlich betrachten wir den gesamten Speicher in ausführbaren Files und dynamisch geladenen Software-Bibliotheken. Wir zeigen die gefundenen Bitflips welche Sicherheitsüberprüfungen umgehen, wie zum Beispiel Passwortabfragen. Auch zeigen wir Bitflips, welche es erlauben HTTP-Authentication Checks in einem Webserver zu umgehen. Zusätzlich zur Untersuchung von statischen Files betrachten wir auch die Auswirkung von Bitflips auf kryptographische Algorithmen. Hier untersuchen wir, wie Rowhammer dazu genutzt werden kann um in der AES-GCM Implementation von OpenSSL eine falsche Verwendung von Noncen zu verursachen.

Unsere Arbeit soll auf die Sicherheitsrisiken die durch Fehler wie Rowhammer entstehen hinweisen und als Aufruf an die Hersteller von Hardware dienen, damit diese nicht billigere Hardware und bessere Performance über die Sicherheit ihrer Anwender stellen.

Contents

1 Introduction

With the latest releases of Microarchitectural attacks like Meltdown [50] and Spectre [67], the topic of flaws in hardware implementations was presented to the general public. Media outlets, like BBC [8] and NBC [53], reported on these issues of modern CPUs. Vendors of x86-architectures, like Intel or AMD, are affected, but also ARM, and with it, most mobile devices have similar flaws. Such issues show that these vendors have set performance above security. The demand for faster hardware is rising all the time, and not only vendors of CPUs are influenced by this demand. Another field of silicon chip design ran into a similar problem in the past, namely DRAM chip vendors.

In 2014, Kim et al. [93] showed how specially crafted memory-access routines can cause bits in DRAM chips to flip, without accessing them directly. This work showed how higher memory densities caused faults where interfering voltages and leaking currents influence storage cells. While at first, this was just seen as a stability issue, Seaborn and Dullien [68] showed how this effect, called Rowhammer, can be used for privilege escalation and sandbox escapes. With reports like this, the interest in researching the field of Rowhammer increased. Gruss et al. [14] showed that it is not only possible to target systems by executing native code, but also that Rowhammer can be triggered by using JavaScript. Van der Veen et al. [89] published work named “Drammer”, where it is shown that not only desktop computers are affected by the Rowhammer flaw, but also mobile devices. In 2019, Gruss et al. [51] released a way to trigger bitflips by sending specially-crafted network requests. Publications like these show that Rowhammer is an active research topic, where still new findings come up.

This thesis builds on work released by Gruss et al. [15], which states that application-binary code can directly be attacked with Rowhammer. They show, for example, that a bitflip applied to `sudo` can result in a bypass of the password check. They report some bitflips causing this bypass. They look at the disassembly of the authentication-check code and find opcodes, which when changed, would result in a different outcome. With our work, we want to automate, and therefore simplify, this process. With this, we predict a finding of a higher number of bitflips in a shorter time. In addition to that, we want to provide a toolset, which allows us to apply similar searches to other applications. This tool should be able to run lots of tests in parallel and verify the outcomes. Therefore, we want to make use of modern testing techniques.

Testing and debugging were always a significant part of software technology, and with rising sizes of projects and an increasing number of old code bases, it is more vital than ever. Not only developers are putting much work into these topics, but also researchers releasing new ways of testing regularly. As the bug reports of “american fuzzy lop” [100] in many open-source software show, modern approaches for testing like fuzzying make bug searching in unknown code more successful. Also, the field of proving software's correctness got much attention. With symbolic execution techniques, the possibility to prove each software state on its own got more practical.

The release of the open-source symbolic-execution framework `angr` [91] made it possible for a wide range of users to apply symbolic execution to programs. This tool mostly gets used in testing, in combination with fuzzing, like Stephens et al. [56] showed in their work "Driller". Also, security researchers use `angr` to find exploitable code segments and execution paths, such as Shoshitaishvili et al. [90] showed with their work "Firmalice", where `angr` was used to detect authentication bypasses in firmware images.

Understanding what programs do, and how they are executed by the CPU, gets harder with every improvement and change in hardware design. Instrumentation is a technique to inject code into programs, providing the possibility to collect runtime information. With tools like Intel Pin [7] it is possible to check changes to processor registers, log accessed memory, and performance measuring at machine code level.

1.1 Goals and Motivation for the Thesis

As we know from previous work done by Gruss et al. [15], there are bitflips in the ELF files loaded by the `sudo` program which allow privilege escalation by providing a bypass of the password check. However, Gruss et al. [15] only look at the code section providing the permission check. They could not claim to find all bitflips, and their approach is very time-consuming. We want to simplify the search by automatic testing of bitflips. Also, we want to make it easier for future applications to be tested for possible bitflip outcomes by providing a framework for such tests.

Common Linux-based operating systems use package management to roll out applications to users. Every instance of the operating system uses the identical binaries for execution of programs. With this in mind, a bitflip found in the `sudo` application distributed by a GNU/Linux distribution, can be used to attack all instances of this operating system. An attacker, therefore, could use our framework to find bitflips in widely distributed binaries.

With our work, we want to present an easy-to-apply framework to search for bitflips providing a pre-defined outcome. To show how this framework works, we apply it to real-world applications and compare our results to the ones reported by Gruss et al. [15]. Also, we want to show how likely it is to introduce exploits in applications by Rowhammer.

1.2 Contributions of our Work

Our contribution to the field of microarchitectural attacks and Rowhammer is providing a practical analysis of real-world applications and how bitflips can affect them. We present a framework which can be used to find bitflips, which change a program's behaviour to a pre-defined outcome. The structure of the framework is designed to be extensible and adaptable for multiple testing purposes.

We apply our tool to real-world applications to show the impact of bitflips on users of personal computers. On the one hand, we show how privilege escalation is made possible by bitflips. We show bits, which when flipped, allow us to skip the password check. On the other hand, we also analyse attacks benefiting remote attackers. We show bitflips, which allow an attacker to

```c
#include <stdio.h>
int main(void)
{
  int x = 0;
  if(x == 1)
    printf("success\n");
  else
    printf("fail\n");
  return 0;
}
```

Listing 1.1: Simple branching code to illustrate how a single bitflip can change the execution path.

```
0x063a  push rbp
0x063b  mov rbp, rsp
0x063e  sub rsp, 0x10
0x0642  mov dword [local_4h], 0
0x0649  cmp dword [local_4h], 1   ; [0x1:4]=0x2464c45
0x064d  jnz 0x65d
0x064f  lea rdi, str.success.     ; 0x6f4 ; "success"
0x0656  call sym.imp.puts         ; int puts(const char *s)
0x065b  jmp 0x669
0x065d  lea rdi, str.fail.        ; 0x6fd ; "fail"
0x0664  call sym.imp.puts         ; int puts(const char *s)
0x0669  mov eax, 0
0x066e  leave
0x066f  ret
```

Listing 1.2: Disassemby of the main function created by the code in Listing 1.1. Shows disassembly at the given address range inside the ELF file, starting at `0x063a`.

bypass HTTP basic authentication. We present results for four applications and if there exist bits, which when flipped, allow us to achieve our set outcome. Besides analysing the bits in the main program's executable, we also examine any dynamically loaded library programs use. By that, we cover possibilities where external functions change the application's outcome.

In addition to that, we look at possible cryptographic vulnerabilities introduced by bitflips. As a basis, we take the work by Böck et al. [29], who showed how it is possible for web servers to misuse nonces when using AES-GCM. We build on their approach to bypass the fixes applied by server software to re-introduce this nonce misusage via bitflips. We look at the current implementation of AES-GCM in the TLS library OpenSSL. We show that nonce misusage can be reintroduced by bitflips, and give a probability for them to happen during a Rowhammer attack.

1.3 Motivational Example

Gruss et al. [15] present a novel showing the impact of a single bitflip in binaries. This thesis is about automating this process. Therefore we look at how a single bitflip can influence the execution path of a program. As an example, see the code in Listing 1.1. We want to change the binary in a way that it will print `success` instead of `fail` by just toggling a single bit in the

Opcode	Hex Value	Describtion	Output
JZ	0x74	Jump short if zero/equal	success.
JNBE	0x77	Jump short if not below or equal/above	success.
JNO	0x71	Jump short if not overflow	fail.
JNL	0x7D	Jump short if less or equal/not greater	success.
GS	0x65	GS segment override prefix	Illegal instruction
PUSH	0x55	$(50 + r)$ Push onto the Stack	Illegal instruction
XOR	0x35	Logical Exclusive OR	Segmentation fault
CMC	0xF5	Complement Carry Flag	Illegal instruction

Table 1.1: Possible opcodes resulting from changing a single bit in `JNZ 0x75` and the output when applied in the assembly showed in Listing 1.2.

binary executing this code. Looking at the disassembly of the code in Listing 1.2, we find that we can start with changing the `1` to a `0` at the `mov` instruction on address `0x0642`. Besides that, the opcode of `jnz` (namely `0x75` or `1110101`), can be changed into a `jz` (`0x74` or `1110100`, at address `0x064d`.

We flip each bit of the instruction opcode `jnz` to illustrate the possible outcomes with just a single flip. We apply all possible eight flips and check the resulting output of the binary. Table 1.1 shows the flips and results. We can see that in three cases the program would print `success`, in one case the output would not change and in four cases the program would crash for various reasons.

From this, we gain the knowledge that at least three bitflips in the opcode give us our desired behaviour. Additionally, we can add the bitflip to the 1, which makes it four. These flips we can find manually quite fast. Now, the question arises how do we automate this searching process?

1.4 Outline of this Work

This thesis is structured as follows: In Section 2, we describe general terms and technologies our work is build on or makes use of. We discuss other microarchitectural attacks, and give an overview of the functionality of programs which our work targets. In Section 3, we discuss our work regarding the automatic bitflip search. We present our testing framework and discuss how bitflips could be introduced to systems. In Section 4, we show our results and present multiple tested applications and what adaptions had to be applied to the framework for successful testing. In Section 5, we discuss our work regarding Rowhammer attacks targeting dynamic memory. We thereby show how the OpenSSL implementation of AES-GCM can be attacked by flipping bits. In Section 6, we discuss countermeasures which could be applied to improve system security. We discuss countermeasures against microarchitectural attacks in general, and discuss what could be done to reduce the impact of our tests. In Section 7, we show possible future works, and an overview of directions the research in the field of microarchitectural attacks could take. In Section 8, we close our thesis with a summary and give a conclusion of our work.

2 Background

In this chapter, we describe the basics, techniques, tools, and programs used in this thesis. It provides a general overview that is required to understand the details of the work in the following chapters. We start by showing how modern computers load programs and execute them. We discuss file formats used to describe programs and how common computing architectures structure machine code. We then look at how programs and source code is tested and what technologies evolved in that matter. As we deal with Linux-based operating systems, we take a look at the permission system used by these systems.

Additionally, we discuss topics in secure communication. We describe the basic principles of network connections, web servers, and the related security. As the thesis provides automation for searching for bits to flip with Rowhammer, we look at this kind of attack, how it works, and discuss microarchitectural attacks in general.

2.1 Executing Programs

Processing units execute machine code. Any device running programs contains at least one processing unit. Usually, the central processing unit (CPU) runs programs and triggers other processing units to run tasks if needed. Often referred to as the main processor, the CPU carries out the instruction given to it by the machine code representing a program. A CPU can only step over single instructions and execute them one by one. This is done by moving the instruction pointer to different memory locations. Simple processors may only execute one single program. On ordinary desktop computers, the operating system is the main program which can load other programs as processes and manage them. CPUs can only execute machine code, but programs are usually written in human-readable programming languages which are then translated to machine language by either a compiler or an interpreter. We refer to files created by a compiler as executables or binaries. These files can directly be loaded into memory by the operating system. Operating systems use defined structures for executables to make this loading possible, for example, GNU/Linux systems use the executable and linkable format (ELF), and Windows systems use the portable executable format (PE).

2.1.1 CPU Instructions

Machine code, in general, is a list of instructions, encoded in some binary format. Usually, an instruction consists of an operation code (opcode), telling the CPU what to do, and parameters for the opcode, which the processor uses for the operation. Instructions can operate on registers or locations in memory directly. Available instructions and their design are depending on the

architecture used. The most common instruction set architectures (ISA) are Intel $x86$, which exists in 32-bit mode (`x86_32`), and 64-bit mode (`x86_64` or `AMD64`), and the ARM ISA which is the architecture used by many mobile processors.

	Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
bytes	1 per prefix	1, 2 or 3	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2 or 4

Table 2.1: Instruction Format for Intel 64 and IA-32 architectures [31], showing how many different instruction lengths are possible in modern architectures and also which parts they may contain.

Table 2.1 shows the instruction format for Intel 64 and IA-32 Architectures. An instruction is divided into six possible parts, which all have different functionality for the CPU executing it. The parts are described in detail by the Intel manual [31] Section 2.1.

Prefix

The instruction prefix can be used to change the behaviour of the CPU for the following opcode. It can be used as a lock (`0xF0`), which ensures shared memory is protected between multiple processing cores. It also can be used as a prefix for repeating an instruction (`0xF2` - repeat-not zero or `0xF3` - repeat-if-zero), which could be used for strings or I/O-operations. Besides repeat, the `0xF2` prefix is also mandatory for some instructions, such as `CPUID`. The prefix can also indicate a larger operand-size by `0x66` or a larger address-size by `0x67`. This address-size prefix provides programs with the ability to switch between addressing modes. There are also branch-hint prefixes (`0x2E` - branch not taken, `0x3E` - branch taken), but according to the Intel manual [31], their usage is deprecated for current architectures.

Streaming SIMD Extensions (SSE)

"Single instruction multiple data"-instructions use the `0x66` prefix to indicate a particular behaviour of the CPU. The instruction can perform the same operation on multiple data pointers at the same time. These instructions are often used in signal processing or graphics processing. Registers used with SIMD contain mostly with single precision floating point data. According to the Intel manual [31] Section 5.5, CPUs supporting this feature provide at least eight SIMD registers, namely `XMM0` to `XMM7`. Newer generations of Intel CPUs might also provide double-precision floating point 64-bit registers.

Opcode

Opcodes describe what the instruction does, if it needs parameters, like registers or addresses to physical memory. An opcode is usually one byte long, there exist two-byte opcodes, which would be declared by a prefix. The primary opcode also defines so-called fields, which are used to declare the direction of operation, size of displacements, register encoding, or sign extension. The ModR/M byte, therefore, also holds a 3-bit opcode field.

ModR/M and SIB Bytes

The ModR/M byte is split into three parts. First is the Mod field, it is combined with the R/M field to provide eight registers and twenty-four addressing modes. Second is the reg/opcode field. It is used to provide a register number or three additional bits for the opcode. The R/M field is either used in combination with the mod field to name addressing modes, or it specifies a register as an operand for the opcode.

Some addressing modes given by the ModR/M byte require a second byte of information. This information is given by the Scale Index Base (SIB) byte. The CPU uses this byte as follows: 3 bits give the base, 3 bits give the index, and the last 2 bits give the scale. To get the address the value stored in the index register is multiplied with the scale and added to the value stored in the base register.

Displacement and Immediate Data

Besides using the ModR/M and SIB bytes for addressing, the mod field can also state to use a so-called displacement as an additional address offset. The calculated address from the SIB byte is then added to the value in the displacement. The displacement can have a size between zero and four bytes.

Some operands may use data directly provided by the instruction instead of loading it from memory or a register. Such data is given in the immediate field of an instruction. Most arithmetic instructions allow immediate data to be passed, but also instructions like `mov` allow such parameters. The usage of immediate data is declared by the ModR/M byte.

2.1.2 Virtual Memory

To prevent direct memory accesses between processes, CPUs provide the possibility of memory translation. Every process holds its virtual memory, and the CPU translates the virtual address to an address in physical memory on memory accesses. As described in the Intel Manuel [31] Section 3.3.2, the address space is divided into pages which are mapped to virtual memory. As this paging is transparent to programs running, processes still work on a linear virtual address space. Pages are usually 4 kB, but larger sizes are also supported. In both address spaces, addresses of pages are aligned to their used page size.

The translation between page addresses is required to be as fast as possible. Storing a direct lookup table per process might be possible but slower than for example memory array accesses. The usual address size on 64-bit processors is 48 bit. To address single bytes on a 4 kB page 12 bit are required. This leaves 36 bit to address the pages. Storing an array for a direct lookup of page numbers would require multiple gigabytes of data, which is not practical. Therefore, a multi-level translation with so-called tables is provided, which keeps the memory overhead in the area of some pages.

Figure 2.1 shows the 4 levels of tables for translating a virtual address to the position in physical memory used by Intel [31] on `x86_64` systems. The highest level is the page map level 4 (PML4),

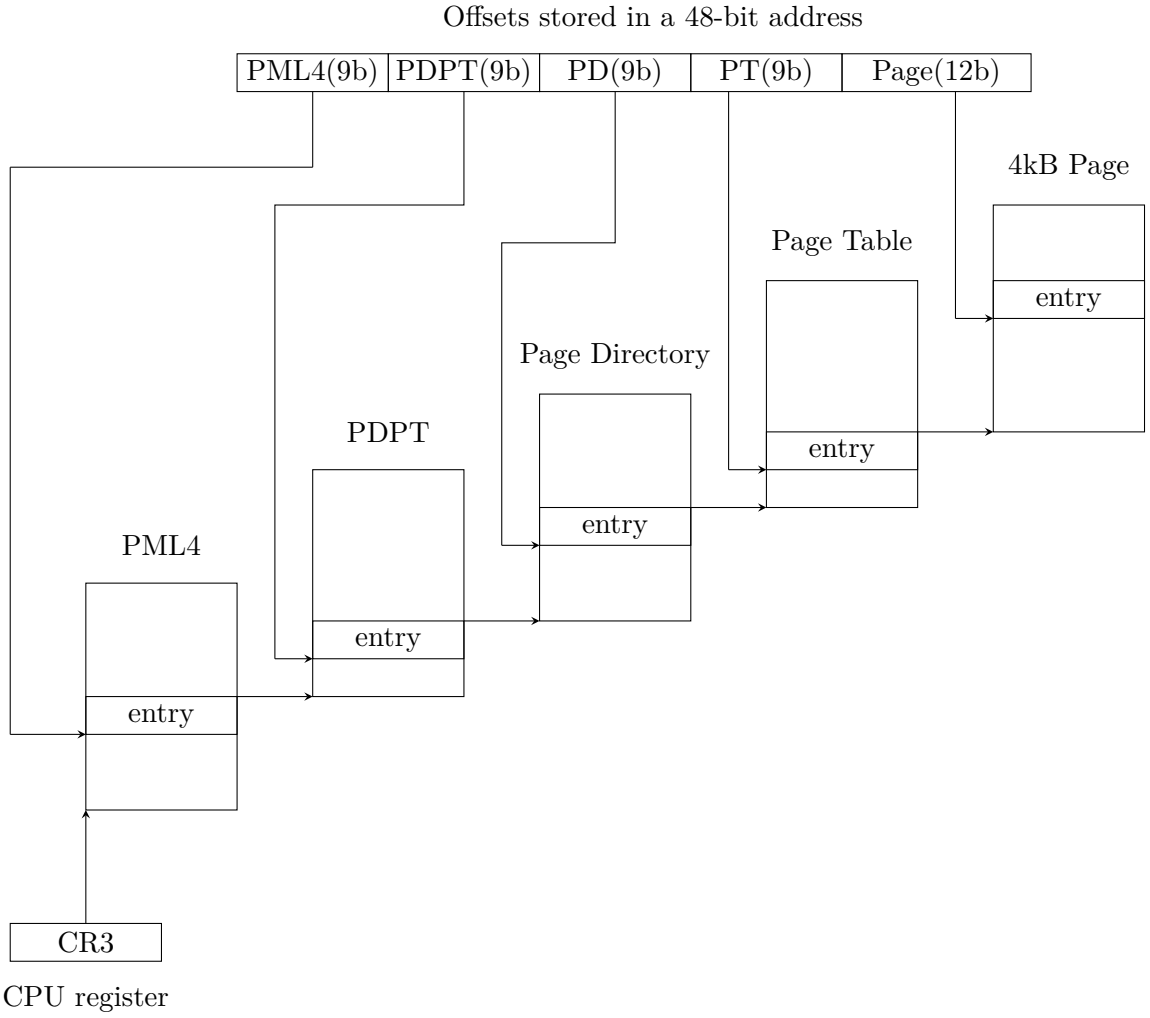

Figure 2.1: Address translation for 4 kB pages for `x86_64` CPUs. The 48-bit address is split into multiple offset parts which are added for different tables. The lookup starts with the PML4, which location is stored in the CR3 CPU register and ends with the byte entry in the addressed page.

the position of the PML4 is stored in the `CR3` CPU register for the running process. The PML4 holds 512 entries of 8 byte each, which point to so-called page directory pointer tables (PDPT). The PDPT again has 512 entries, where every entry either points to a page directory (PD) or a memory page of the size of 1 GB. The PD entry either maps to a 2 MB page or a page table (PT), which then holds entries to the 4 kB pages. For every level of paging, 9 bit of the address are used to the define the offset used in the tables.

An entry in tables is 8 byte large, but not all of them are required to address pages. Therefore, the entries also defines properties of the addressed memory. For example, the entries hold bits giving information about access rights, cache behaviour, or tell if the addressed memory is present. These properties are checked by the memory management unit on memory access. If accesses fail, for example because of writing read-only memory or, accessing memory which is not present, a so-called page fault interrupt is thrown, and the running operating system kernel has to deal with it, by either signaling the running process, killing it, or resolving the issue by for example mapping a missing memory page.

2.1.3 Executable and Linkable Format (ELF)

ELF is an object file format. It is used to describe a program and gives instructions on how to load it into memory to make it ready for execution by the processor without applying changes to the binary-content itself. A linker creates ELF files after an assembler turned the program's source code into machine code. Most modern Linux-based operating systems follow the ELF specification released by the Tool Interface Standard Committee [87]. Therefore, we need to understand the layout of such executable files before introducing bitflips in them.

Structure of an ELF file

There are two views of an ELF file, the linking view and the execution view. Those are also called section and segment view respectively. They share the same ELF header but serve different purposes for the operating system. Figure 2.2 show the connection between sections and segments via the two different headers. Sections and segments can be seen as follows.

Sections

Sections describe the binary for the linking view. A section can contain instructions, data, a symbol table and relocation information. Sections reserved for the system, start with a dot. There can be additional sections defined by the user. Sections are created and managed by the linker. Sections that are directly loaded into the program's memory image are initialized data (`.data, .data1`), read-only data (`.rodata, .rodata1`), and executable instructions (`.text`).

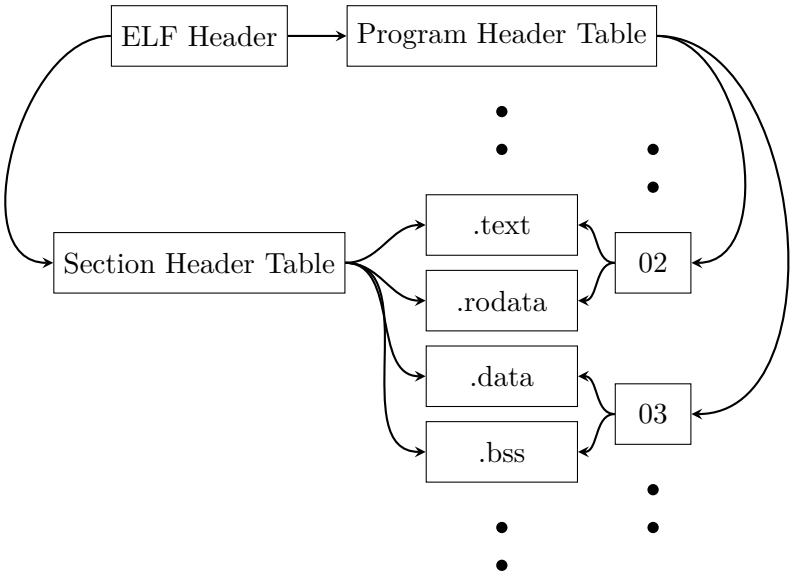

Figure 2.2: Schematics of the structure of an ELF file showing the four main sections inside an executable. Sections and segments are connected via the program header table and the section header table. Segments usually are just numbers and can refer to multiple sections inside the section header table.

Segments

Segments describe the virtual memory layout of a loaded binary. Figure 2.2 shows how segments are used in an ELF file. Usually, the linker splits segments into different behaviours the loader has to apply. For example, all read-only data sections are in the same segment. When loading the image into memory, references inside the ELF file need to be resolved and loaded into the memory too. After successfully loading the image and its dependencies the program can be executed.

Loading ELF files into Memory

The ELF specification [87] was released by the Tool Interface Standard (TIS) in 1995 for UNIX System-V Release 4-based operating systems. Many operating systems, such as GNU/Linux-based or BSD-based ones, adapted this standard to execute programs. As we focus on GNU/Linux operating systems, we look at how to handle ELF files in order to create running programs. Most operating systems used on computers or mobile phones do not use physical addresses during the execution of programs, and the operating system is free to change the position of sections in the virtual address space. Therefore, in an ELF file, a section only contains a base address and offsets to that address. For certain compilation settings, the operating system is free to change the base address in the programs virtual address space.

Dynamic Linking

In modern systems, it is common to have multiple functions used by different programs. Shared libraries can be used to provide such general functions. When deploying programs, developers

usually make sure all required shared libraries exist on the target platform or ship their software with those included. When depending on such system libraries, the loader will insert the shared-library functions into memory at the given entry point. This technique is called dynamic linking. The loader fetches the information about the required library from the ELF file and then searches the library paths provided by the operating system environment to find the library files. If they cannot be found in the given paths, an error occurs, and the program cannot be executed.

2.2 Analysis and Testing of Executables

Testing is a significant part of software development. There are many different approaches and styles for testing. One approach to test a program is to provide input to it and check if the code operates as expected by validating the output. With an increasing amount of code and an increasing number of bugs found, the style of testing changed over the years. On the one hand, developers nowadays ideally provide unit tests inside their code to test single functions and make sure they work as intended.

On the other hand, sometimes the source code is not available to a tester, or it is way too much work to read the entire source code to create such tests. Therefore, developers came up with techniques to test binaries without knowing the entire source code. Black-box testing is an approach where a program is tested without knowing its source code.

2.2.1 Fuzzing

Fuzzing is a form of black-box testing, where knowledge about the program itself is just minimal. Duran et al. [37] already showed in the 80s that trying random inputs on programs provides a way to rate its robustness. Tools applying automatically-generated, pseudo-randomised, inputs to programs are called fuzzers. Fuzzers attempt to reach corner cases in programs without knowing which exist and detect undefined behaviour or crashes. Generally, fuzzers give an overall view of the robustness of a program. They are relatively easy to apply to programs and can even find bugs usual white-box tests would miss.

If a fuzzer would only apply random inputs, testing would take a very long time, as many unnecessary tests would run. Because of this, the technology improved over time and different fuzzers evolved. For simple programs, just testing common input mistakes, like negative numbers, newlines, end of file characters, format strings, or just very long strings might be enough. More advanced software requires better ways of fuzzing, for example, network protocols, file systems, or image formats have complex structures which make it less likely to crash by just random data input. Fuzzers for such kind of software generate inputs from given examples and apply small changes to them. Xmlfuzzer [41] is a tool crafted specially to test XML schemes in programs. Besides special format fuzzers, there are more advanced ones which directly interact with the target, allowing to track execution paths, and therefore, create even more test cases automatically. “American Fuzzy Lop” (AFL), is a fuzzer which is used to find bugs in open source software, their website [100] presents a list of bugs found with it. AFL contains algorithms which take usual test inputs which would cause successful runs of a program, parses those inputs and applies random changes to it. These algorithms allow better code coverage than just generating random input. Another more

advanced fuzzer is Steelix by Yuekang et al. [96], which uses program-state-based binary fuzzing. Steelix applies static analysis and binary instrumentation to provide more information about the fuzzing process, and therefore yields better test cases and code coverage.

Shoshitaishvili et al. [90] showed that fuzzing can find authentication bypasses in binary firmware with their tool Firmalice. Wang et al. [79] showed how in-memory fuzzing can be used to detect similarities in binary code. This is done by fuzzing every function available and comparing traces of program behaviours with machine learning models. They show that they can find similarities in binaries even when different compilers, and optimisations are used, or when obfuscation is applied.

2.2.2 Symbolic Execution

Fuzzing provides a pseudo-random input to a program to test it. Therefore, it might not cover all possible execution paths of a program. Simplifying a program to cover all possible execution paths is done in symbolic execution. James C. King was the first to mentioned this technique in 1976 [34]. Symbolic execution simplifies possible inputs for programs by replacing them with a logic symbol which could hold fewer possible states than usual data types. For example, a number can be represented by the following states: maximal value, minimum value, positive, negative, and zero. The possible symbolic-state inputs then run through the program to report possible violations of the program's specification. King [34] used a decision tree to cover all possible execution paths in a binary. Modern frameworks, like `angr` [91], use different techniques to analyse a binary.

On the one hand `angr` can still use a tree structure to cover program paths, on the other hand, it also provides a fuzzing assisted symbolic execution technique. AFL is used to provide a crash report, and the symbolic execution path is used to identify what code sections are reached by the mutation of the input. By this, the user running the test gets better knowledge about where in the code the program crashes and why it crashes. Besides the symbolic execution of binaries, `angr` also allows various other analysis methods such as control-flow graph recovery, automatic exploit generation or automatic binary hardening.

Stephens et al. [56] showed how symbolic execution could be beneficial to fuzzing approaches. Symbolic execution reduces the number of execution paths to test while covering much code as possible. Stephens et al. [56] use concolic execution for the path exploring. Concolic execution is the combination of symbolic and concrete execution of a program.

2.2.3 Instrumentation

Instrumentation comes down to adding new code to already existing one in an automated manner. Developers use instrumentation as a tool for debugging, or performance measurements by adding calls to timing functions, or information logging to the code. Information from instrumentation can be something like what functions are called and how often, what execution branches are taken, how long a subroutine takes, what memory is accessed, and many more. It also allows the developer to change the runtime environment of a program, like change return values or skip instruction calls.

Two different styles of instrumentation exist, one is source instrumentation, and one is binary instrumentation, so-called instrumenter calls are either added to the source of the program before compilation or later added to the machine code of the program. The first style can be easier to apply. Furthermore, the structure of the program is known to the instrumenter. Calls for performance checks over functions can be done by adding timer calls at the entry and exit points. However, binary instrumentation also has advantages. For example, users do not require the source of the program, which might not be available for proprietary programs. Also, recompilation of the program is not required if the code for the instrumentation changes. Given enough debug information in the binary the instrumentation information might not differ much from the source-based one. With binary instrumentation, it is also possible to change and read register values before and after specific CPU instructions are executed, which might not be possible at the source code level, as used registers and memory areas are unknown to the instrumenter at this point.

Intel Pin

Intel Pin [7] is a proprietary, dynamic, binary instrumentation framework developed and released by Intel, who supply it for free. The framework provides a large number of API calls which abstracts most of the work done by Pin and gives easy access to binary analysis. As Pin is a binary instrumentation framework, the source of the program to be analysed is not needed, and the program does it need to be recompiled if the instrumentation tool changes. The framework allows to log and modify the runtime of a program, whereas it is no problem to log register values or change those before or after desired instruction calls or even inject code to be run before given calls. Intel Pin provides a C++ API to write so-called pintools to analyse programs. Besides the C++ bindings for Pin, also a wrapper for Python exists, in `python-pin` [2].

Luk et al. [7] describe the system of Pin and how the written pintool is combined with an application in more detail. When instrumenting a program, three binaries are used. One is the application to instrument, one is the compiled pintool, and one is Pin itself. Pin can be seen as a userland virtual machine. When the instrumentation starts, Pin is loaded into the address space of the application. Then the `ptrace` system call is used to capture the program's behaviour. When Pin has initialised itself the tool overtakes the entry point of the program, or the current program counter if it was attached to a running application. It then uses its just-in-time (JIT) compiler to continue the execution. Luk et al. [7] state that their approach of loading Pin has various advantages over other instrumentation tools, which mostly use `LD_PRELOAD`, which would not work on statically linked binaries.

The JIT compiler inside Pin is used to inject the instrumentation API into the application to allow the pintool to use it. The code is loaded into a preserved code-cache and then executed. Pin always tries to execute as many instructions as possible without interrupting it for debugging. Luk et al. [7] state, execution of the code happens one trace at a time. A trace runs either until a control change happens, such as a branch, a call, or a return instruction, or a Pin-API call is reached. With this approach, they gain better performance than other instrumentation techniques.

Luk et al. [7], also state that the JIT is not the bottleneck but the instrumenting code inside the application. That is the reason why Pin optimises the injected code as much as possible. Pin

inlines all analysis functions. Luk et al. [7] state that this approach is faster as no calls happen, which would cause registers to be used for the arguments. Luk et al. [7] also state that most other debugging tools use the `eflags` CPU registers, which is not ideal, as storing and restoring these flags is time-consuming, and an extra stack has to be used, as the original stack is not allowed to be modified. Pin detects in the JIT if the `eflag` content is needed later on, and only if it is, saving routines are called. Another optimisation for their framework is the API call `IPOINT_ANYWHERE`, which specifies that the analysis call can happen anywhere in the instrumentation context, which allows better optimisation of the instruction order. Luk et al. [7] claim, that Pin performs better than tools like `valgrind` [55] and DynamoRIO [17], two open source binary runtime analysis frameworks.

2.3 Permission Model in Linux-based Systems

The POSIX standard [85] defines interfaces to provide compatibility between operating systems. It includes a description of permissions for users and their groups and how they are set. For convenient reasons, a user can be part of multiple groups. A unique integer ID per user and group provides identification to the system. The POSIX-defined permission structure applies to files, which can have properties like readable, writable, executable. On creation, the user can set these properties separately for the owner, the group of the owner and others. POSIX also allows users with special privileges to ignore such permission checks, or change them. Such users are called superusers.

Besides permissions for files, most Linux-based systems also provide permission checks for processes and their owners. For example, only processes owned by a superuser might be allowed to use certain syscalls. In most systems, only one superuser exists, namely `root`. Usually, users do not want to login another time if they need permissions of a superuser for a task. Operating systems, therefore, often ship tools such as `su` or `sudo`. `su` stands for switch user and allows a user to change the current context to another user or execute a command with the other user's permissions, providing the correct password of the other user. `sudo` includes the same functionality as `su`. Additionally, it has a more advanced management system in the background and given the correct permissions the user does not need to know the other user's password to execute commands as the target user.

2.3.1 `setuid` Binary Property

`setuid` [86] stands for “set user identity”. It allows an executable to change its user ID during execution. An additional bit in the file permissions represents the `setuid` property, namely `S_ISUID`. This property allows a user to run a program with the permissions of the file's owner. Tools like `su` and `sudo` require this permission in order to change the current context. It is also convenient to allow normal users to use tools like `ping` or `ip`, without requiring them to authenticate as `root`. We refer to the GNU libc documentation [25] for more details on this file property.

While the `setuid` property might seem convenient, it also brings security risks. A bug allowing code execution in such a program allows an attacker to execute that code with the permission of

another user. Therefore, the number of `setuid` binaries should be low and programs having that property must be well audited and tested to not jeopardise the system's security and integrity.

2.3.2 `chroot`

`chroot` [84] stands for "change root directory". One can either use it directly via a system call or via a wrapper program. A `chroot` call changes the root directory for the running process, resulting in a change of the environment the program runs in. It is not possible to access files from outside the new environment nor to interact with processes running in the outside environment, without any measures applied from the outside environment. `chroot` is either used for testing purposes, to create packages in clean environments, to check compatibility, as the chrooted environment could provide a different operating system userland, or for sandboxing, and privilege separation.

2.3.3 Pluggable Authentication Module (PAM)

PAM can be used by Linux-based operating systems to authenticate users. PAM is a collection of authentication schemes which should provide a general API to allow authentication with abstracting the actual functionality implemented from developers. According to the PAM manpage [19], modules are split into four groups:

- The account group verifies if the user is permitted access and if the password is not expired.
- The authentication group verifies the user's credentials. These could be given in various ways, like a common password input, biometric data, or as a combination of multiple ways.
- The password groups is used to update and manage authentication mechanisms. Modules from this group are usually connected with those from the authentication group.
- The session group provides ways to manage permissions for users after authentication. It ensures the user stays authorised and also modules could implement ways to log out users or check session termination from other sources.

Usually, Linux-based operating systems use the default `pam_unix.so` for all groups. Additionally, modules as `pam_deny.so` and `pam_permit.so` are used to provide access right management for single services per default.

2.4 Connecting Computers

People are exchanging data between devices for several decades now. The Internet started when scientists found a way for universities to share research results between computers over the back then used Advanced Research Projects Agency Network (ARPANET). Clark [10] released a paper in 1988 about the protocols used by the ARPANET and gives arguments on why stated protocols were used. The network already used the TCP/IP protocol and showed requirements for future interconnection networks. Furthermore, Clark [10] also explains the evolution of the Internet by stating the plans of connecting other networks with the ARPANET.

With easier access to computers, more universities, companies and private households wanted to be connected and exchange data. The Internet provided the possibility to do this. The standard for Hyper-Text Transfer Protocol (HTTP) [70] was introduced to provide files from servers to users. The standard for network mail [1] provides a uniform way to send messages between users over the Internet. With the growing amount of users, security in the network and the connected devices became more relevant. Therefore, protocols to ensure security for the participants were introduced as standards.

2.4.1 Web Server

Web servers provide users with content on the Internet. There are many different software implementations for such servers available, some of them being open source and free, such as Apache [23] and `nginx` [54], which are according to Netcraft the two most used open source web servers [52].

Web servers use the Hyper-Text Transfer Protocol (HTTP) [70] to handle requests and deliver the content to the client. Web servers deliver websites encoded in Hypertext Markup Language (HTML) but can also deliver all kinds of files including JavaScript (JS) and Content Style Sheet (CSS), which are rendered together with HTML by web browsers on the client.

A client requests content from the server by sending a Uniform Resource Locator (URL) [81]. The server then looks up the given path in the URL and answers the request by sending the requested file or an error, if one occurs during the handling of the request. As not all requests and answers should be publicly visible, web servers adopted TLS for security and privacy reasons to provide content over HTTPS. The client, most of the time a web browser, can thereby verify the server's identity and apply encryption to the connection if required.

HTTP - Basic Access Authentication

A standard for access control was added to HTTP, as described in RFC 7617 [72], to allow only authorised users to access specified paths on web servers. This technique makes use of a header field in HTTP, namely `WWW-Authenticate`. Basic Access Authentication is an access-control technique which does not protect the transmitted credentials. Therefore, it should be used together with HTTPS to protect the transmitted data. The client has to transmit the credentials with every request, and there is no logout function as in other techniques using cookies, or session identifiers.

Access control in Apache and `nginx`

Apache makes use of so-called `ht` files, which are an extension to the usual server configuration for users with limited permissions [82]. Apache introduced `htpasswd`-files to make use of Basic Authentication, these files store user credentials. The developers of `nginx` adapted these files for their web server implementation. The `htpasswd`-files keep the data either in plaintext or hashed. To create these files Apache provides the tool `htpasswd` [83]. This tool uses MD5 hashing per default (as of version 2.2.18), but it is possible to change the hashing algorithm to something more secure, like bcrypt by Provos et al. [57]. On a request including Basic Authentication, the

web server checks the `htpasswd` file to verify the given credentials, and then returns the requested content, or a permission error according to RFC 7231 [70].

2.4.2 Transport Layer Security

Transport Layer Security (TLS) is a protocol proposed by the Internet Engineering Task Force (IETF). TLS was released to provide secure communication on the Internet. The current version of TLS, 1.2, as in RFC 5246 [18], will be obsolete with the upcoming release of TLS 1.3, as described in RFC 8446 [73].

TLS is used by web servers to provide secure connections with their clients. Also, applications like Virtual Private Networks (VPNs) and Secure Shell (SSH) make use of methods and cyphers described in the TLS standard. OpenSSL [63] is an open source library providing the functionality described by the standard.

TLS provides a secure connection between various devices, usually clients and a server. The connection holds the following properties.

Authenticated Identities

Parties using TLS want to ensure that the other party is the one they claim to be. To protect users from phishing or accidentally accessing wrong servers, TLS provides certificates to authenticate identities. The client requests the certificate from the server, which contains its public key and possible key exchange methods it supports. The certificates may be signed by a certificate authority which the client already trusts.

Private Connections

According to the TLS handshake protocol, described in RFC 5246 [18] Section 7. The server and the client agree on a cypher to encrypt data sent between them. TLS 1.2 states that 128-bit AES needs to be supported, but parties may agree on another cypher during the handshake. As AES uses a symmetric key to encrypt data, this key has to be exchanged between the two parties. Therefore the parties agree on a key exchange protocol during the handshake. TLS 1.2 states that RSA [76] needs to be supported, but parties can choose a different method, such as Diffie-Hellman key exchange [44], or use a pre-shared key.

Reliability

The TLS standard describes message authentication codes (MAC) to allow parties to detect alteration of sent messages. These codes are cryptographic hashes, which use the data contained in a message plus some secret data. Methods to protect message integrity in such a way are described in RFC 2104 [28]

2.4.3 Cryptographic Nonces

In the early days of encrypted communication, replay-attacks were a common problem, as Syverson showed in his taxonomy of replay attacks in 1994 [80]. Rogaway [75, 74], published a solution to this problem. In his work, he proposes to use one-time, pseudo-random data to protect symmetric cryptography from replay-attacks. He calls such data nonce. Rogeway [74] states, that to ensure that nonces will not be reused, they should be used as a counter with a random start value.

However, even with this idea of replay-attack protection, current protocols and technologies still face problems with nonces, like Vanhoef et al. [42] showed in 2017, that it is possible to force nonce reuse in WPA2.

2.4.4 Advanced Encryption Standard (AES)

The Advanced Encryption Standard is a block cypher proposed to the US National Institute of Standards and Technology (NIST) by Rijmen and Daemen in 1999 [13]. In 2001, NIST released AES as a specification for encryption of electronic data. It is the successor of the Data Encryption Standard (DES). AES is also part of ISO/IEC 18033-3 [33], an ISO standard related to security techniques and encryption algorithms.

As Rijmen and Daemen et al. [13] proposed, AES is a block cypher which operates on data blocks. The blocks are stored in a memory array, represented as a matrix of bytes. The algorithm consists of four steps, which are reused in multiple rounds, the steps operate on the memory matrix by shifting entries or replacing them with using a substitution box (S-BOX), this provides non-linearity to the algorithm. The algorithm works very similar for encryption and decryption. Most steps can be reused, this makes it very convenient to provide these functionalities directly in hardware. Intel processors already provide instructions, which can directly be used for AES [27].

Attacks on AES

Researchers and attackers targeted AES since it was proposed for cryptographic usage. Most of these attacks targeted a smaller number of rounds than the algorithm describes. In 2011, Bogdanov et al. [3] showed an attack against AES which targets all of the rounds. Their attack weakens the security of AES slightly, where for 128-bit key recovery the computational complexity drops to $2^{126.1}$ and for 256-bit keys to $2^{254.4}$. As these are still very high complexities, the attack does not affect real-world usages of AES.

Way more common than cryptoanalysis attacks on AES are attacks targeting software and hardware implementations. For example, O'Flynn et al. [11] showed an attack against an AES implementation used to decrypt a firmware image during the bootup process. Whereas most attacks target 128 bits, they showed one against a 256 bit implementation. O'Flynn et al. [11] use a correlation power analysis on the software run by the bootloader which loads an AES-256-CBC encrypted image. Besides recovering the full key, also the initialisation vector is recovered by the attack.

Lo et al. [65] published another hardware side-channel attack. The attack targets the implementation of the S-BOX used in AES-128. Besides using correlation power analysis, they also use differential power analysis. They compare both methods and recover cypher keys from function calls accessing the S-BOX.

Neve et al. [45], show that increasing the key-size in AES from 128 to 256 bits does not increase the complexity of side-channel attacks by the same factor. Neve et al. [45] show that for cache-based side-channel attacks an upgrade from 128 bits to 256 only increases the complexity of an attack by a factor between 6 and 7.

2.4.5 AES-GCM

Galois Counter Mode (GCM) [12] was proposed as a standard by NIST [21] to combine counting nonces with block cyphers, such as AES. This combination allows the creation of a stream cypher where each block is encrypted with a different nonce. With the addition of GCM, also a MAC is generated, in this algorithm called an authentication tag. Figure 2.3 shows the schematic of the proposed AES-GCM. For the description of the algorithm, a similar notation is used as Böck et al. [29] use in their paper about attacks against GCM.

CNT_i The i-th counter block, computed using the concatenation of the initialisation vector (IV), with a size of 96 bits, and the counter value cnt, $cnt = (i + 1) \mod 2^{32}$, to gain a 128-bit value. $CNT_0 = IV \parallel 0^{31} \parallel 1$.
E_k AES encryption with symmetric key k.
$a \oplus b$ XOR operation between a and b.
P_i The i-th plaintext block.
C_i The i-th ciphertext block.
$mult_H$ Multiplication $H \cdot X$ in Galois Field GF(2^{128}), with the irreducible polynomial $g = g(x) = x^{128} + x^7 + x^2 + x + 1$.
A Authenticated data.
$len(X)$ Bit-length of X.
$a \parallel b$ Concatenation of a and b.
TAG Authentication tag.

The following steps describe how AES-GCM is used to encrypt data:

1. An initialisation vector IV of 96 bits is generated.
2. The counters CNT_i of 128 bits are computed with $CNT_i = IV \parallel cnt$, where $cnt = (i + 1) \mod 2^{32}$, for $i \in \{0..n\}$, n represents the number of plaintext blocks.
3. The ciphertexts C_i are generated by encrypting the counter values CNT_i and XORing them to the plaintext P_i, $C_i = P_i \oplus E_k(CNT_i)$.

To get the authentication tag TAG, the $GHASH$ function, $GHASH(H, A, C) = X_{m+n+1}$, is applied. Where, H is the hash key, computed by encrypting 128 zero bits with the AES block cypher, C is the ciphertext, and A is non-encrypted but authenticated data. Figure 2.3 shows

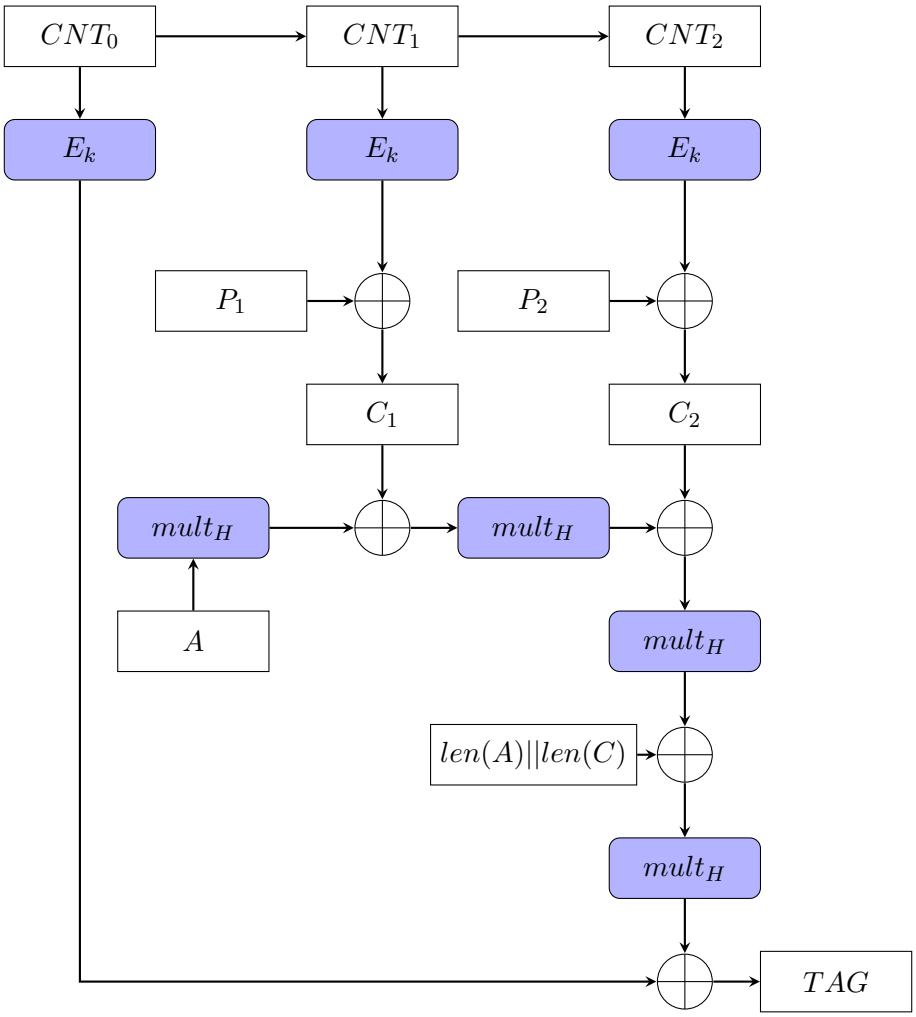

Figure 2.3: Schematic of AES-GCM. The E_k blocks apply AES-encryption with the key k. The $mult_H$ blocks apply a multiplication in the Galois-field. A represents a block of authenticated data, which is added to the MAC TAG but not encrypted.

how the $GHASH$ is computed for one block of authenticated data and two blocks of plaintext. The following steps describe the computation of the TAG:

1. The Galois field multiplication is applied to the block of authenticated data. $R_0 = mult_H(A_1)$.
2. The result of this is XORed to the first ciphertext. The output of this XOR is again multiplied in the Galois field. $R_1 = mult_H(R_0 \oplus C_1)$.
3. This output is then XORed to the second ciphertext and again multiplied. $R_2 = mult_H(R_1 \oplus C_2)$
4. Then the length of the authenticated data and the ciphertext is added. $R_3 = mult_H(R_2 \oplus (len(A) \parallel len(C)))$
5. The hash key is added in the end to form the tag TAG. $TAG = H \oplus mult_H(R_3)$.

The authenticated tag TAG can therefore be found by solving the polynomial $g(X) = A_1X^{m+n+1} + \cdots + A_mX^{n+2} + C_1X^{n+1} + \cdots + C_nX^2 + LX + S$, as $g(H) = TAG$. L is the combined length of A and C, and S is the nonce plus the counter. We refer to the NIST standard about AES-GCM [21] for more details.

OpenSSL source code

The libraries souce code is structured in a way that each part of a cypher is implemented on its own. For our thesis and analysis we want to refer to version `1.1.0g` of OpenSSL [64]. The AES-GCM code part is located at `crypto/evp/e_aes.c`. The structures used are defined in `crypto/evp/evp_locl.h`, `crypto/include/internal/evp_int.h` and `ssl/ssl_locl.h`. The file `crypto/evp/evp_lib.c` shows the implementation for the creation of the cypher data.

In the OpenSSL library, the IV for AES-GCM is computed by using 12 bytes, where 4 bytes are a salt computed during the TLS handshake, and additional 8 bytes represent the used nonce.

AES-GCM Nonce Reuse Attack

Joux [5] calls his attack against GCM a "forbidden attack" because in cryptography the uniqueness of nonces is seen as given. Böck et al. [29] describe how relevant nonce reuse still is in real-world applications and why clear definitions for nonces in protocols are required for software development.

Böck et al. [29] present an attack based on the simplified attack by Joux, which he presented in his comment on the NIST proposal [5]. In those attacks, it is assumed that the same nonce is used for two messages. Each of those messages consists of a single ciphertext block and no additional authenticated data. It is also noted, that all values beside S, the secret nonce value, are known to the attacker.

$$g_1(X) = C_1X^2 \oplus L_1X \oplus S$$
$$g_2(X) = C_2X^2 \oplus L_2X \oplus S$$

We add the corresponding tag TAG to each polynomial to get $g_i'(X)$ as follows.

$$g'_1(X) = C_1X^2 \oplus L_1X \oplus S \oplus TAG_1$$
$$g'_2(X) = C_2X^2 \oplus L_2X \oplus S \oplus TAG_2$$

Knowing that $g'_x(H) = 0$, because $g_x(H) = T_x$, we combine those two formulas to the following polynomial.

$$g'_{1+2}(X) = (C_1 + C_2)X^2 + (L_1 + L_2)X + TAG_1 + TAG_2$$

This polynomial should be 0 for the point H, $g'_{1+2}(H) = 0$. Therefore, an attacker can calculate a list of values for H which solve this equation. The number of possible solutions for the polynomial is reduced with an increasing number of nonce reuses. Therefore, by using the same nonce more often it gets easier for an attacker to find H and therefore break the used cryptography.

2.5 Software-based Microarchitectural Attacks

Computers can leak information by behaving differently during the computation of different data. Aciicmez et al. [60, 62, 61] released work on timing leaks which leak information about used key or data. Aciicmez et al. [62] show how RSA implementations were attackable with microarchitectural analysis. For the implementation in OpenSSL, it was possible to detect called reduction steps in the Montgomery multiplication, which lead to a total break of the RSA implementation. Aciicmez et al. [61], also show further microarchitectural attacks against OpenSSL, which break other functions implemented in the open source library.

2.5.1 Low-Level Performance Measuring

Many side-channel attacks are based on timings. Like, predicting what a machine currently does because of the time it takes to compute something. Researchers and attackers using time measurement as an information source, therefore usually try to find a way to measure time with a high resolution.

Native Timing Measurement

In native code, the highly precise timestamp counter (TSC), provided on x86 architectures since the first Pentium CPU [31], can be used for timing measurement. The Intel manual [31] Section 4, states that the current timestamp is stored in a model-specific 64-bit register (MSR). It also states that the value in the MSR is increased with every clock cycle and reset to zero on every processor reset.

Calling the instruction `RDTSC` will load the current value of the MSR into the `EDX:EAX` registers. The `EDX` register is loaded with the higher-order 32 bits, and the `EAX` register with the lower-order 32 bits. On architectures using Intel 64 the higher-order 32 bits of `RAX` and `RDX` are cleared. `RDTSC` is not a serialising instruction, that means the counter MSR could be read before the

previous instruction is finished. This behaviour is often not wanted, as results may not be accurate enough. In the Intel manual [31] it is therefore recommended to use either the serialising `RDTSCP` instruction or use the sequence `LFENCE;RDTSC`, if it is required for `RDTSC` to be called after previous instructions finished.

JavaScript Timing Measurement

Calls to the `RDTSC` instruction are generally only possible in native code. Van Goethem et al. [88] showed how modern web browsers allow new side-channels by providing accurate timing measurements. They show how the JavaScript timestamp function `performance.now()` can be used for timing measurements. Mozilla reacted to the released timing attack by lowering the resolution of the timing API to five microseconds [101]. Schwarz et al. [46] show how this countermeasure can be bypassed and give an evaluation of new sources of timing information in JavaScript. Lipp et al. [49, 48] show how a counting thread can be used as an accurate timing source in JavaScript and also state that the time measurement is accurate enough to detect keystroke interrupts. Schwarz et al. [46] state that with this accuracy of timings it is possible to perform cache-attacks and Rowhammer inside web browsers.

2.5.2 Caches

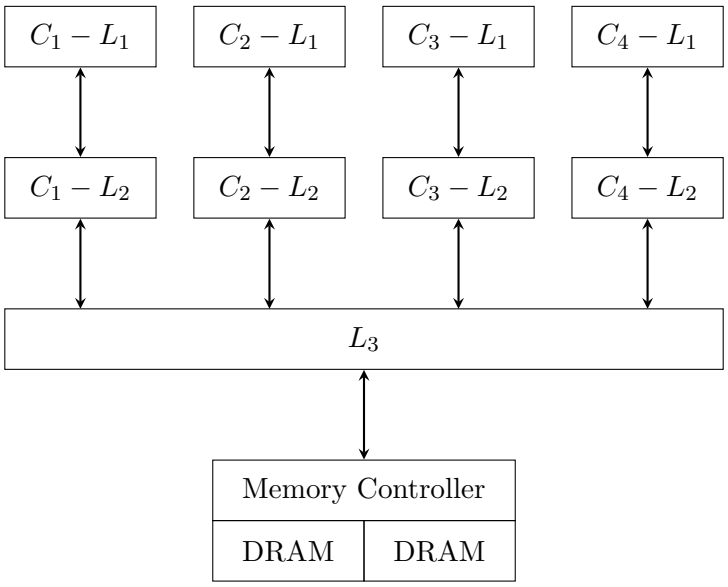

Figure 2.4: Schematic of Intel caching architecture. C_x represents each processing core and L_x the level of the cache. Intel [32] states that cache content and memory requests are handled by a memory controller inside the processor.

Vendors of processors introduced the principle of caching to memory management to speed up access times for memory. According to the Intel architecture white paper [32], modern CPUs implement three or four levels of caching. As shown in Figure 2.4, the design in Intel Kaby Lake and Intel Haswell assigns caches for level one and two (L1, L2) to each core and the cores share the

level three cache (L3) between each other. L1 caches can be split up in a data and an instruction cache. There might also be an L0 cache for micro-operations, as it is also implemented in the two described architectures.

Caches are managed and updated by the CPU's memory controller. Inclusive caching is a form of managing the caches in a way where data stored on one level of caching is also placed in lower cache levels. The microarchitecture defines if the CPU uses inclusive caches and which ones are inclusive to others. The memory controller of the processor needs to keep caches up to date to make sure memory accesses do not get wrong data from other cache levels or the DRAM.

We refer to the Intel manual [31] chapter 11.1 and the Intel white paper about the architecture [32] for more details about cache management in modern CPUs.

Cache Attacks

Cache attacks are part of microarchitectural attacks where scientists have shown that caching also introduces security risks to systems. Dan Page [66] showed that caches can be used for side channels in 2002. Tsunoo et al. [97] published an attack on Data Encryption Standard implementations in 2003, using timing-side-channel information gathered from the CPU delays caused by cache accesses. Knowledge gained from cache attacks is reused in the exploitation of Rowhammer, and other attacks, such as Meltdown [50] and Spectre [67].

FLUSH+RELOAD

This cache attack is a low noise, L3 cache side-channel attack, discovered by Yarom et al. [98]. The attack allows leaking information about memory accesses on pages shared between two processes. As the attack targets the last-level cache, it does not require the two processes to run on the same core. Pages are either shared when actively mapped with such property or mapped as shared with copy-on-write when a process is forked. One process is referred to as the spy and the other as the victim. The attack is build of rounds of attack, where one round is split into three phases:

1. The spy uses the instruction `clflush` to remove the monitored memory line from all CPU caches.
2. The spy waits a pre-defined time to allow the victim to access the memory line.
3. The spy measures how long it takes to read the flushed memory line. If the victim accessed the line, the memory would be back in the cache, which would take a significantly shorter time than reloading the memory from the DRAM.

With this attack, information can be leaked to the spy processes. For example, in cryptographic algorithms, accessed code lines could leak the private key to the attacker.

2.5.3 Other Microarchitectural Attacks

With growing knowledge on the CPU and caching level, researchers found major security vulnerabilities in modern CPUs. Attacks like Meltdown [50], Spectre [67] and Foreshadow [36].

Lipp et al. [50] target the memory isolation between the operating system's kernel and the userland memory and shows how the CPU's out-of-order execution can lead to a leak of information from kernel memory to a user program. Meltdown uses cache-accesses timings to transmit data from the out-of-order execution to the userland program. Spectre [67] uses a similar technique to leak information, by exploiting the branch-prediction unit of modern CPUs. The attack is used to leak memory between processes, where Meltdown is used to leak kernel memory. Kocher et al. [67] state that this attack is also possible in JavaScript. Bulck et al. [36] showed that it is possible to use another similar approach to attack the Guard eXtension (SGX) in Intel CPUs. In the so-called Foreshadow attack memory from inside this secure environment is leaked to an attacker. This attack bypasses security layers inside Intel's microcode and cannot be fixed inside the operating system's kernel. Gruss et al. [16] proposed a countermeasure against Meltdown, where memory is separated between the userland and the kernel. By applying this current Linux-based operating systems might be secured against Meltdown but remain attackable against some Spectre variants.

2.6 Rowhammer

Technology is steadily changing and improving, with vendors of computer parts being forced to produce cheaper and better hardware every year. For DRAM this resulted in faster memory chips, with less space required for the same amount of memory and less energy required to store electric load, to represent data. After years of decreasing memory densities, researchers like Kim et al. [93] found a flaw in the DRAM chip design. With the little space used and energy stored it was possible to use changes of charges in memory rows to interfere with data of nearby cells. This interference resulted in a change of bits in other memory rows.

This behaviour caused researchers to dig deeper into the possibilities, and several attacks were found based on Rowhammer. For example, Seaborn and Dullien [68] show that it is possible to use Rowhammer for privilege escalation and sandbox escapes. Van der Veen et al. [89] state that this attack not only affects desktop computers but also memory in mobile devices. Gruss et al. [14] published work showing that it is possible to trigger bitflips from JavaScript. Gruss et al. [15] also show that flipping bits in memory has further risks than previously thought and that Rowhammer defences need a general overhaul. They state that the only solution is to fix the hardware.

2.6.1 Design of Dynamic Random-Access Memory (DRAM)

Random-access memory is designed to store information as bits. A simplified view of RAM is a circuit containing amplifiers, one-bit storage cell matrix, an address decoder, and buffers for input and output of the memory. Figure 2.5 shows a simple schematic of a DRAM module. A DRAM chip contains multiples of such modules, stored in so-called banks. The storage matrix is a two-dimensional array of 1-bit storage cells. Accesses to DRAM do not happen per bit but per

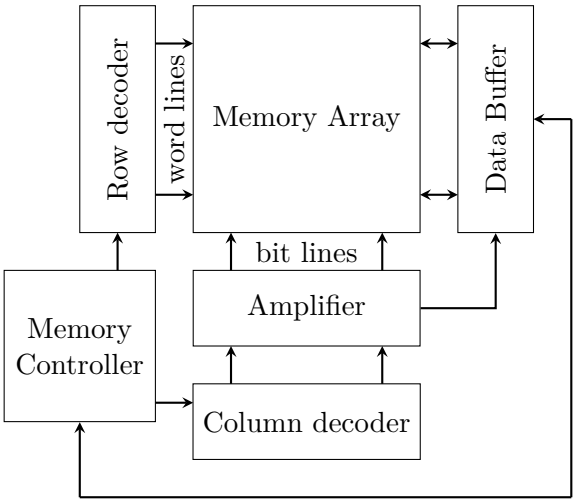

Figure 2.5: Schematic of a single DRAM module. The memory controller is managing the memory array and has a Bus connection to other memory controllers and the CPU. The decoders are used to access bits in the memory array according to the sent address. The amplifier is used because the bit voltage differences are usually too small to change the memory in the buffer directly.

row, which has a size of 4096 bit. The memory controller of the CPU sends a request, which then is translated for the DRAM chip to access the corresponding bank, holding the memory array which then loads the columns needed from the memory row. A memory array usually holds 2 MB. Therefore 8 GB of DRAM memory are stored in 4096 memory arrays.

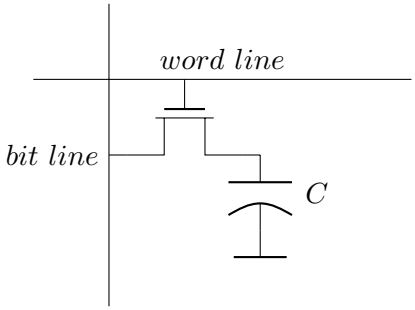

Figure 2.6: Schematic of a 1-T memory storage element. The capacitor C is used to store information. The transistor makes it possible to access the storage by applying charge on the connected lines. Depending on the voltage used it is a read or a write access.

A one-transistor (1-T) memory cell, as seen in Figure 2.6, is used to store a single bit. To access memory, first, the desired bit lines are charged with $\frac{V_{CC}}{2}$, then the word line is also charged to switch on the transistors. An amplifier is then used to detect changes of charge on the bit lines. A discharged capacitor would cause the voltage on the bit line to drop, whereas a charged capacitor would raise the voltage to its load. If a charged or discharged capacitor represents true or false for the bit state, depends on the design of the DRAM chip. Writing the bits works in a similar way, where the bit lines of a row are either set to V_{CC} or 0, and then the word line is used to switch on the capacitors on the row, making the capacitors change their charge to the level of the connected bit line.

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

Listing 2.1: Code to trigger the Rowhammer bug, as of Kim et al. [93]. The repeating accesses to the aggressor locations cause bitflips in a third victim row. The `CFLUSH` is required to make sure the DRAM rows are accessed directly. According to Seaborn and Dullien [68] the `mfence` is not needed and even lowered the number of flips.

The capacitor used for the 1-T cell will lose load over time and on read accesses. Therefore, these capacitors need to be periodically refreshed. The JEDEC standard [35] sets the refresh cycle to a 64 ms interval. For a usual chip with 8192 rows, this means a refresh happens all 7.8 µs. The easiest way to do this is by reading a bit and writing the response back to it. So-called Row Access Strobes (RAS) are sent to switch on the transistors in the addressed row. Column Access Strobes (CAS) are then used to read the values of the capacitors holding the load storing one bit. As refresh cycles need to be applied, DRAM chips usually cycle over all rows, read them into their data buffer and write them back into memory. This refreshing is done to keep the capacitors on a certain charge level.

Jung et al. [43] present a study about the exact neighboorhood relation of single DRAM cells. In the presented work, the location of memory bits in the DRAM banks is found by analysing temperature changes of the device.

2.6.2 Introducing Bitflips to DRAM

Kim et al. [93] show how it is possible to cause bits to flip inside memory by certain access patterns. Two memory rows are used as so-called aggressors, and one as a victim row. As the code in Listing 2.1 shows, the two aggressor rows are read repeatedly. The `clflush` instructions are used to empty the memory in the used cache lines, causing the System to access the DRAM for each read instead of getting the value out of the cache. The changes in electrical charge caused by the read accesses will by chance interfere with the loads in the target row and change content there. These accesses happen so fast that even with correct refresh cycles applied, the DRAM controller is unable to keep the load of the capacitors on their level.

The memory addresses to be accessed to gain the rows next to each other are not obvious to find, as operating systems use virtual addresses instead of direct physical addresses. Linux provides a file, `/proc/<PID>/pagemap`, where the mapping between virtual and physical addresses is stored. Some systems allow huge pages, where 2 MB of memory are used continuously. These 2 MB will use more than one row, other than normal pages with the size of 4 kB. In the paper, Kim et al. [93] use addresses by the scheme of $Y = X + 8M$, as previous work reverse engineered this address mapping for Intel CPUs [39, 58]. By using these addresses, they get accesses resulting in the same bank of DRAM but different rows. Kim et al. [93] state in their paper that they use two memory locations as by using just a single one, no bitflips occurred in their tests. Seaborn and Dullien [40]

describe single-sided Rowhammer as a technique where only one row is used as the aggressor, but multiple memory accesses are applied to this row. The advantage of the single-sided hammering is that no knowledge about the DRAM address mapping is needed.

Yim [92] presents a framework to identify, validate and evaluate Rowhammer attacks. The framework can be used to find defective DRAM chips being attackable with Rowhammer. Also, the framework provides automatic exploit generation to be applied via Rowhammer attacks.

2.6.3 Exploits using Rowhammer

Seaborn and Dullien [68] show how it is possible to use Rowhammer to escape the NaCl sandbox and how to gain privilege escalation on `x86_64` based systems by flipping bits in page table entries. As Kim et al. [93] state, bitflips might be very likely to reoccur in the same position, but it is still a challenge to target an exact memory location when just a virtual address is known. To provide higher reliable attacks memory spraying [68, 14, 95] is used. Memory spraying fills the memory of a system with as much targetable memory as possible. For example, Seaborn and Dullien [68] would spray the memory with page tables.

Bhattacharya et al. [78] show how Rowhammer can be combined with timing analysis to break implementations of RSA. In the described attack, the last-level cache is monitored by a spy process to determine the position of the targeted memory and then use timing measurements to determine the DRAM bank which holds this memory to apply a Rowhammer attack to this bank. Razavi et al. [38] present their work “Flip Feng Shui”, which allows introducing bitflips in a fully controlled way. In their work, it is shown how to target specific memory areas by using Rowhammer and memory deduplication. Razavi et al. [38] present an attack which allows breaking of OpenSSH public-key authentication and thereby successfully login to a co-hosted VM on the same hypervisor.

Qiao et al. [77] present new approaches of Rowhammer attacks, where they use non-temporal instruction of Intel CPUs to trigger Rowhammer instead of using a `clflush` approach. Also, it is shown how it is possible to trigger Rowhammer by using malicious inputs on programs instead of running malicious code. The input on running programs causes calls to `memset` or `memcpy` which can be abused to trigger Rowhammer.

2.6.4 Flipping bits in Binaries

Targeting files with Rowhammer is an improved attack on previously shown usages of the vulnerability. Gruss et al. [15] showed, that besides targeting page table entries, it is also possible to target shared libraries or ELF files directly. Thereby, the possible attack vector for Rowhammer grew. To flip the bit in the binary using Rowhammer an attacker needs to make sure the file is loaded to DRAM.

Most operating systems use a disk cache. Meaning the OS will keep file content in DRAM as long as the memory is used. When a file is opened the system loads the content to the memory, which in case of loading an ELF would put the entire file into the DRAM, making it possible to apply

bitflips. Therefore, a later call to the program would execute the flipped version already cached in DRAM.

Another idea would be to use Rowhammer on storage devices. As Kurmus et al. [4] showed it is possible to find a similar behaviour as Rowhammer for solid-state drives (SSD). They refer to a problem in MLC NAND chips, showing the possibility of a similar attack. They show that corrupting a block inside the SSD is enough to gain privilege escalation. A similar attack could also be used to flip bits inside the stored ELF files.

3 Bitflip Attacks on ELF Files

In this chapter, we present a novel to automate finding bitflips in ELF which yield exploitable execution path changes. Gruss et al. [15] already have shown a similar approach by disassembling binaries and manually looking for exploitable bitflips. We start by discussing this approach and point out the downsides of this mainly manual searching.

We go on with describing our used approach and the framework we created to find exploitable bitflips in binaries to change the behaviour to result in a pre-defined outcome. We start with a general overview of the framework design and go on with discussing the details on how we find bits to check, how the definition for searching works, how we test the bitflips and how results are captured and verified.

3.1 Analysing the Manual Search Approach

Gruss et al. [15] introduce so-called opcode flipping to bypass the Rowhammer countermeasure of physical kernel memory isolation, which was proposed to prevent bitflips in page tables. With opcode flipping, it is possible to gain privilege escalation by flipping bits in pages storing executable memory. Gruss et al. [15] describe their approach as follows. First, a memory area in the binary is defined, which holds bits to test with flipping. Secondly, all bits in the defined area are flipped, resulting binaries are executed, and grouped by the outcoming results. In the end, the results are analysed manually, and exploitable behaviours are noted and then attacked with Rowhammer. Gruss et al. [15] show a case study of this approach where the `sudo` program is analysed and report 29 bitflips which give an attacker superuser permissions without knowing the calling user's password.

As this approach seems feasible, we claim that it is not practical. Gruss et al. [15] analysed the program's binary, and the dynamically loaded libraries to identify targetable memory areas, by this choice, areas might be missed. Also, each bit in a defined area is tested, which could include memory which is never accessed during the execution. Manually analysing the results may take much time and might not be feasible for a larger number of bitflips. Besides flipping opcodes, also prefixes and parameters of instructions should be tested for bitflips. Also, not only specific memory areas holding executable code should be tested but all memory in binaries accessed during the runtime of the program. As Gruss et al. [15] only check a specific version of a program, the reported bitflips might not work in a newer version, and the same work has to be done again.

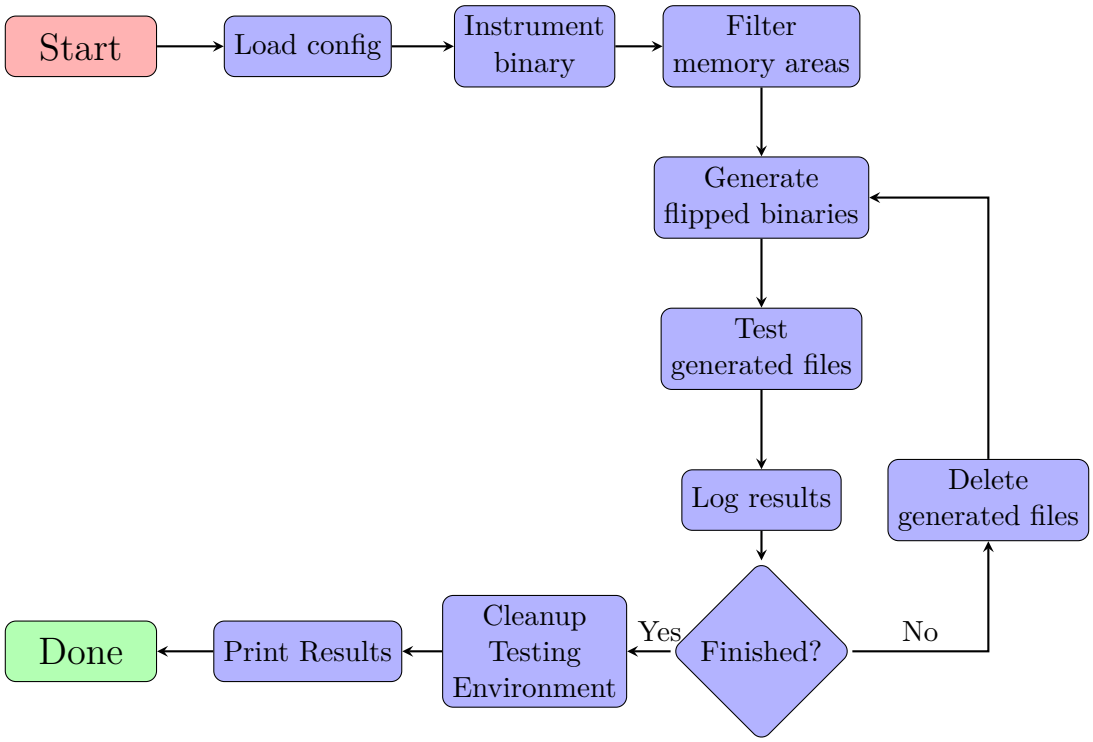

Figure 3.1: Flowchart showing each part of the framework and their connections.

3.2 Automating the Finding of Feasible Bitflips

Flipping all bits inside a binary, reporting each output and check for a successful outcome might be a way to generate a list of all exploitable bitflips. Looking at the compiled binary from Listing 1.1, in Section 1.3, it would result in a binary size of 8288 byte. Following the approach of testing all possible bits, would result in 66 304 bitflips to test. With an execution time of approximately 0.003 s per run, a complete test would take close to 200 s.

Looking at `sudo`, the program's main binary alone has 149 080 byte, without dynamic libraries. The execution time is around 0.03 s. The same approach applied to `sudo` would, therefore, take around 35 780 s, which is approximately 10 h. Such a long time is not feasible, as dynamic libraries used also need to be tested. Including those libraries, would result in 3 219 208 byte. Testing all these bitflips would require approximately 215 h or 9 days.

For our automation of the exploitable bitflip search, we want to require way less time than with testing all bits, as we target larger real-world applications.

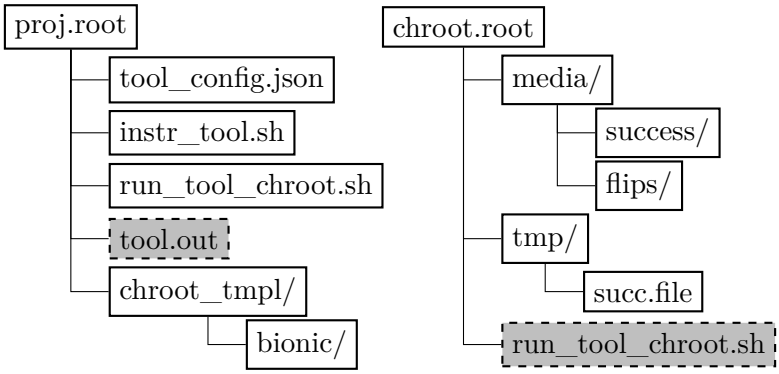

Figure 3.2: File system structure of the framework and the chroot loaded for the test runs, whereas the chroot is created from a templated loaded from the `proj.root` and the `run_tool_chroot.sh` is copied over to the created chroot.

```
{
  "instrumenter_call": "./instr_tool.sh",
  "instrumenter_outfile": "tool.out",
  "chroot_template": "chroot_tmpl/bionic",
  "tmp_chroot_folder": "/media/ramdisk/chroot/",
  "folder_with_flips": "/media/ramdisk/flips/",
  "num_of_parallel_checks": 1,
  "CR_exec_file": "./run_tool_chroot.sh",
  "CR_flip_folder": "/media/flips",
  "CR_success_folder": "/media/success/",
  "CR_log_file": "/tmp/succ.file"
}
```

Listing 3.1: JSON style config file for the framework, showing all parameters used to tweak each part of the framework. Entries starting with `CR_` are used inside the testing `chroot`.

3.2.1 Design of the Testing Framework

We have implemented a framework to search for exploitable bitflips in programs. Figure 3.1 shows a diagram describing the framework's design. We split it into multiple parts:

1. Accessed memory areas are logged from all the ELF files the program uses.
2. Pre-defined filters are applied to the reported memory areas.
3. ELF files for each of the bitflips are generated.
4. Each ELF file gets executed, and the behaviour is checked against a pre-defined success state

Each part of the framework is designed to work on its own and has a clearly defined in- and output-interface. Each part can be adopted or easily used for other testing purposes. The configuration file for the framework defines how each step is applied to the application, as instrumentation and verification may differ for each program. The file in Listing 3.1 shows an example configuration file for the framework. Figure 3.2 shows the layout of the framework in the file system. The configuration file defines where to find each of the required files. The template chroot describes the system which is used for testing.

```
ADDRINT ins_addr = INS_Address(ins);
IMG img = IMG_FindByAddress(ins_addr);
ADDRINT base = IMG_LowAddress(img);
img_offsets[base] = IMG_LoadOffset(img);

for(size_t i = 0; i < INS_Size(ins); i++)
  accesses.insert(std::make_pair(ins_addr + i, base));

UINT32 memOperands = INS_MemoryOperandCount(ins);
for (UINT32 memOp = 0; memOp < memOperands; memOp++)
{
  if(INS_MemoryOperandIsRead(ins, memOp))
  {
    INS_InsertPredicatedCall(
      ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
      IARG_INST_PTR,
      IARG_MEMORYREAD_EA,
      IARG_ADDRINT, base,
      IARG_MEMORYREAD_SIZE,
      IARG_END);
  }
  if(INS_MemoryOperandIsWritten(ins, memOp))
  {
    INS_InsertPredicatedCall(
      ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
      IARG_INST_PTR,
      IARG_MEMORYWRITE_EA,
      IARG_ADDRINT, base,
      IARG_MEMORYWRITE_SIZE,
      IARG_END);
  }
}
```

Listing 3.2: Example C++ code for a pintool logging memory accesses. The tool stores the locations of instructions and parameters if they point to memory. The tool stores accesses for reading and writing seperately.

The framework starts with executing `instr_tool.sh` to generate a report showing all bitflips to test. Additionally, it can also filter them. After the script's execution, the `tool.out`-file contains a list of all byte positions to test. Afterwards, the defined number of chroots are created by copying the template from `chroot_tmpl`. The flipped binaries are copied to each `/media/flips` folder in the testing chroot. The `run_tool_chroot.sh` file is copied to each testing environment and executed there. This script runs in each of the `chroot`, replaces the original binary with a flipped version, runs the configured program and checks it's outcome. If the outcome matches a pre-defined success state, the flipped binary is stored and reported by the framework.

Instrumenting Programs

We use Intel Pin [7] for instrumentation. Every memory access during execution of the program is logged. We want to log all accesses which happen inside of ELF files, which means the program's binary and all libraries it uses during runtime. The pintool we use to achieve this, adds instrumenter calls to all memory accesses. The tool logs all accesses per binary and the offsets of the address

in the corresponding ELF file. This information is required to determine which accessed area is mapped to which file later on.

Listing 3.2 shows a code snippet of the used pintool which logs all memory accesses as it instruments each instruction and logs the opcode position and its parameters if they address memory. For Pin, the image `img` refers to an ELF file, the images represent the program's executable and the dynamic libraries loaded during execution. The `INS_InsertPredicatedCall` function is used to insert a function call on the machine code layer. The function can have an arbitrary number of parameters, as the user has to define the function with each parameter and type. Pin then inserts a call to the given function pointer and manages the handling of the parameters and return values in a way that the instrumented program's execution is not affected.

We instrument the program with a run resulting in a failed state, after the run, we filter the accessed memory areas for each file and check if areas were written before reading. If so, we can also skip testing those bits, as the program would overwrite the flips again.

Instrumentation is done via the configured `instrumenter_call` in the config file shown in Listing 3.1. The `instrumenter_call` generates the configured `instrumenter_outfile`. The output file contains the memory accesses formated in the following manner:

`<hex:position_in_file> - str:filename`.

Generating and Testing the Flips

Next, we create a new ELF file for each bit, for each of the reported bytes. Depending on the configured number of parallel checks, chroots are created. Afterwards, the framework makes sure the number of created ELF files does not require more than the memory available. If there are more files to test than the available memory allows, the framework takes multiple runs, as shown in Figure 3.1. A test run starts by running the configured `CR_exec_file` which will copy a flipped file from the `CR_flip_folder` to the original position, then run it and check if the output matches a defined success state. If the flipped file reaches the success state, the framework writes the flipped address to the `CR_log_file` and copies the flipped ELF file to the `CR_success_folder`. The framework passes the file to replace to the `CR_exec_file` when it calls it. To keep track of which bitflip is tested the filename for generated ELF files is structured as follows:

`<original_filename>_<address_in_file>_<bit_number>`.

3.2.2 Tweaks added to the Framework

As our framework reports all accessed memory areas, we apply some tweaks to these reported memory areas generated with Pin. The additions help us to either filter unnecessary flips and also to add memory areas which would be missed by instrumentation as they are only accessed during the setup phase of the program.

```
1  extern const char *__progname;
2
3  __attribute__((constructor)) void init(void)
4  {
5    if(!strcmp(__progname, "instrumented_program"))
6    {
7      size_t i = 0;
8      while(i++ < 600000000);
9    }
10 }
```

Listing 3.3: Code of the preloaded library to keep the process waiting for some milliseconds, which gives enough time for Pin to attach to the process.

Filtering Memory Accesses

As seen in Listing 3.2 we log accesses separated by reading and writing. This separation is done to apply a simple filter on the reported memory areas. It is not practical to test a memory area which is written before being read, as the program will overwrite the memory area before a bitflip would influence the execution path. We, therefore, test memory areas which are either only read or read before write access.

Testing flips in the ELF structure

As described in Section 2.1.3, the ELF structure contains information for the linker on where to load different segments of the binary in the virtual address space. Applying flips in this section could lead to misplaced libraries or different offsets for segment mappings. A call of a function might result in executing different code as intended and lead to a different outcome of the program. Therefore, we also test each bit which is used for representing ELF structures.

Instrumenting Binaries Running as Superuser

Some binaries behave differently when being run by a superuser. For example `sudo` is a `setuid` binary owned by `root`, therefore the active user running the binary is `root`. If Pin is running as a regular user, it is not allowed to access the memory of a superuser's process, it, therefore, cannot instrument `sudo`. Running Pin as superuser is possible, but starting `sudo` as `root` does not trigger any authentication checks, as the calling user already got superuser permissions, therefore a change of the current user is not required.

Therefore, we use the attach-to-process functionality of Pin to bypass this behaviour. In that way, we can start `sudo` as a regular user and run Pin as superuser. We would start `sudo` and then attach to the resulting process ID. The problem here is that getting the process ID takes time and during that time `sudo` could already perform accesses which then would be missed by instrumentation. Therefore, we preload a shared library which keeps the `sudo` program in a defined state for some time to give the framework enough time to look up the process ID and instrument all relevant code parts.

Listing 3.3 shows the code used to generate such a preloaded library. We use the counting while-loop instead of a call to the sleep system call because attaching Pin to a process in sleeping state results in a state where the process cannot wake up again. We also experienced the same issue with debugging tools on our Linux-based environment, such as `gdb`. The number of increments in the loop was chosen to have enough time to search the process ID and attach to it. The compiler attribute `constructor` makes sure the program executes this code on loading the library before any other code. We add the path to our library to the `/etc/ld.so.preload` file, which makes sure the library is preloaded by any executed program. The name of the executable is checked in Line 5 of Listing 3.3 to only execute the while-loop when the binary is our instrumentation target. Otherwise, all executed programs would execute the time-consuming while-loop during startup.

3.2.3 Comparing our Approach to other Techniques

Our described approach for finding exploitable bitflips is built on a simplified fuzzing-like approach. We want to look at other ideas we tried to adapt or use for the searching and argument on why we stuck with our described framework.

Symbolic Execution

`angr` [91] offers many tools and functions for binary analysis and symbolic execution. For this, it can load a binary, search all possible states and provides locations in the executable where a state has more than one possible successor. `angr` allows searching for inputs and arguments which would end up in a state defined as successful. This state could either be defined by an address reached during execution or on the output of the program. The approach `angr` seems useful for our task of searching bitflips to change to a success state. However, `angr` is very limited when changing the executed machine code to find new states. It works well for finding inputs for the program to result in a success state. Testing bitflips with `angr` results in a limited application of the tool's possibilities, where the binary is loaded, a bitflip is applied, and a single run is made which is then checked if it ends up in the defined success state. Changing inputs or arguments will not be tested. `angr` could be used to test bitflips, but the time needed would be about 10 to 15 times longer than our approach, because of the time it takes `angr` to load the program and its libraries.

Using AFL

Fuzzers like AFL [100] test programs by applying different forms of input and generate those applying a form of edge coverage to learn from changes from the control flow. AFL uses instrumentation to document the program's control flow and adapt its test cases based on changes of it. AFL offers source code based instrumentation, which might not be suitable for testing bitflips in binaries as with that approach, the tested executables would change. According to their documentation [99], AFL also offers instrumentation of binary-only applications. Here they use the QEMU processor emulator [6] as a userland virtual machine. They state in Section 4 of the documentation [99], that in this mode execution is 2-5 times slower than for the compile-time instrumentation approach.

For testing bitflips, the adaption of test cases can be ignored, as we always stick to the same environment and external inputs. The AFL techniques to learn from control flow changes could be adapted but as we do not want to build on source-based instrumentation because of binary changes the additional execution time for their binary instrumentation made the AFL adaption not practical.

Using an AFL-like Approach

The described instrumentation based fuzzing of AFL can be adapted for instrumentation based bitflip testing. We used Intel Pin [7] for this approach, as it is the best performing tool for binary instrumentation. The implementation would instrument every instruction of the program, fork multiple times. One time for each bit in the instruction and the addressed memory by the instruction. After the forks, individual bitflips are applied to the memory of the new processes, and the new processes continue execution without instrumentation. The outputs of the new processes are then checked for resulting in a successful state.

This approach turned out to be practical and performed well enough for simple programs. However, we ran into problems with applications which had the `setuid` property and the solution we described in Section 3.2.2 did not work for this implementation. Looking at the execution times for simple programs they were just a little lower, about 5 % as for running pre-generated binary files. However, the time needed for the generation would be saved, and the memory footprint would be lower as the processes memory would be copy-on-write. We decided to not go with this approach, as we wanted to test all kind of binaries, also `setuid` ones, such as `sudo`.

For our framework, we generated the binaries containing the bitflip still in DRAM memory, to speed up the procedure of copying and applying a bitflip. This generation phase of the framework showed to take less than 1 % of the whole testing process time. Therefore, running multiple generation phases, because of the increased memory usage, was accepted as a downside of our chosen testing approach.

4 Case Study: Rowhammer targets in real-world Applications

In this thesis, we want to apply the framework to larger, real-world applications, with realistic bitflip targets. As Gruss et al. [15] already showed, there are bits in the `sudoers` library which, when toggled, allow using `sudo` without or with a wrong password to still gain the privileges of a superuser. We show that our framework can find a larger number of exploitable bitflips. We present the results of the framework applied to real-world applications, which are commonly used in GNU/Linux operating systems. In this section, we present all the applications we tested, the pre-defined outcomes, and the bitflips leading to the desired behaviour. For the results, we discuss how many exploitable bitflips were found, in what memory section they were and if in an instruction, in which part. In the end, we want to give some statistic about the exploitable bitflips and also point out the ones which were exploitable in multiple applications.

For testing, we used a `Debian GNU/Linux 9.5` operating system. The testing environment inside the `chroot` used the same. The CPU used was a `Intel(R) Core(TM) i7-960` and a total of 16 GB DRAM memory was installed.

4.1 `sudo` - Privilege Escalation

`sudo` is a tool used for privilege management on Linux-based operating systems. Users can use it to execute commands as another user, if the tool's configuration allows it, also as the superuser `root`. For the tool to run the given command as another user, the user has to provide their password.

We want to find all exploitable flips which allow an attacker to run `sudo` without knowing the password. This exploit results in a privilege escalation as the user can become superuser with a bypass of the credential check.

For testing we call `sudo` and provide a wrong password for the credential check. As command we provide reading a file which is owned by `root` and can not be read by the calling user. We define the success state as the call of `sudo` must print the content of the file owned by the superuser. We tested version 1.8.19p1 of `sudo`.

4.1.1 Results

We present 425 exploitable bitflips in total. Table 4.1 shows the ELF files where the flips were found and also shows how many bitflips were exploitable in different ELF sections. Most bitflips

ELF File	.text	.rodata	.got.plt	.plt	Sum of flips:
ld-linux-x86-64.so.2	1	0	0	0	1
libpthread.so.0	2	0	0	0	2
sudo	6	0	0	0	6
libnns_compat.so.2	6	0	1	0	7
libpam.so.0	64	0	0	0	64
pam_unix.so	85	0	0	1	86
sudoers.so	219	35	5	0	259
Sum of flips:	383	35	6	1	425

Table 4.1: List of ELF files used by the `sudo(1.8.19p1)` program and the number of flips causing privilege escalation listed per section.

```
1   351b:   0f 84 90 00 00 00   je      35b1
2   3521:   48 89 ee            mov     %rbp,%rsi
3 - 3524:   e8 47 dc ff ff      callq   1170 <strcmp@plt>
4 + 3524:   e8 c7 dc ff ff      callq   11f0 <malloc@plt>
5   3529:   85 c0               test    %eax,%eax
6   352b:   0f 85 3e ff ff ff   jne     346f
```

Listing 4.1: Diff for a bitflip applied to `libnss_compat.so.2` in order to bypass a user privilege check. The call to `strcmp` is exchanged because the offset used for the lookuptable is flipped.

were found in the `sudoers` library, but also bitflips in the linker (`ld-linux-x86-64.so.2`) and the threading library (`libpthread.so.0`) were exploitable.

We can see that most exploitable bitflips are in the `.text` sections of binaries, which is the section holding the executed machine code. The `.plt` section refers to the Procedure Linkage Table, which is used to resolve mappings between position-independent functions and their absolute addresses. Close to the same is the `.got.plt` section, which represents the global offset table, which resolves mappings for position-independent code parts.

ELF File	opcode	parameter	Sum of flips:
ld-linux-x86-64.so.2	0	1	1
libpthread.so.0	0	2	2
sudo	2	4	6
libnns_compat.so.2	2	5	7
libpam.so.0	17	47	64
pam_unix.so	36	50	86
sudoers.so	87	172	259
Sum of flips:	144	281	425

Table 4.2: List of bitflips causing privilege escalation in `sudo(1.8.19p1)` and what they change in instructions.

From the 425 exploitable flips 144 were placed in the opcodes of instructions and 281 were in parameters. Table 4.2 shows how the flips are divided in that two sections for the different ELF files. It is shown for all files, that about one third of the flips are applied in the opcodes and the other two in the parameters.

Listing 4.1 shows a diff of the `libnss_compat.so.2` ELF file, where a call to `strcmp` is replaced with a call to `malloc` by a single bitflip, causing the later `jne` instruction to behave differently.

Testing the `sudo` program for exploitable bitflips took about 8 hours of CPU time. The testing was parallelised and split into 24 `chroot` environments. Testing all bits in the `sudo` binary itself took about two to three times longer with an equal configuration.

4.2 `nginx` - HTTP Basic Authentication Bypass

`nginx` is a web server, available on for GNU/Linux operating systems [54]. The tool can be used to serve websites, and it can also be configured to support basic authentication schemes.

We want to find all exploitable bitflips which allow an attacker to access websites protected with HTTP basic authentication. Such an exploit bypasses the credential check of the configured web server, which uses `htpasswd` for basic authentication, as described in Section 2.4.1.

For testing, we start `nginx`, wait for the server to go into an initialised state. Then we request a website protected with HTTP basic authentication, providing credentials containing the correct username but a wrong password. If the response contains the content of the requested website, we add the flipped bit to the list of exploitable bitflips. We do not care if the response is malformed. We rate the response as successful as long as the protected site is leaked by it. We tested version 1.10.4 of `nginx`.

4.2.1 Results

For the authentication bypass in `nginx` we can report that only bitflips in the program's binary itself were exploitable. We can show 298 exploitable bitflips in the executable, where all of them were found in the `.text` section of the ELF file.

65 bitflips changed the opcode of an instruction 231 changed parameters. 2 bitflips changed the execution context of the executable memory area because the instruction length was changed by the flip and therefore the context was changed. In Listing 4.2 we can see how changing the lenght of the `add` instruction can result in a different outcome for the execution.

Testing the `nginx` ELF files took a total of about 18 h CPU time. Most of this time was spent during the startup of `nginx`. For testing, we used 8 testing-`chroot` instances.

4.3 `sshd` - Secure Shell Server Login Bypass

`sshd` is a secure shell server implementation which runs on GNU/Linux operating systems. It is used together with the `ssh` client, which allows to log in on remote hosts via the `ssh` protocol. According to the documentation [20], not only log in via username and password credentials is supported, but also public key cryptography can be used for authentication.

```
  2f1365:  00 00                   add    %al,(%rax)
  2f1367:  00 91 7a 0c 00 00       add    %dl,0xc7a(%rcx)
  2f136d:  00 00                   add    %al,(%rax)
  2f136f:  00 10                   add    %dl,(%rax)
- 2f1371:  00 00                   add    %al,(%rax)
- 2f1373:  00 00                   add    %al,(%rax)
- 2f1375:  00 00                   add    %al,(%rax)
- 2f1377:  00 a1 7a 0c 00 00       add    %ah,0xc7a(%rcx)
- 2f137d:  00 00                   add    %al,(%rax)
- 2f137f:  00 14 00                add    %dl,(%rax,%rax,1)
- 2f1382:  00 00                   add    %al,(%rax)
- 2f1384:  00 00                   add    %al,(%rax)
- 2f1386:  00 00                   add    %al,(%rax)
- 2f1388:  b2 7a                   mov    $0x7a,%dl
- 2f138a:  0c 00                   or     $0x0,%al
- 2f138c:  00 00                   add    %al,(%rax)
- 2f138e:  00 00                   add    %al,(%rax)
- 2f1390:  0d 00 00 00 00          or     $0x0,%eax
+ 2f1371:  40 00 00                add    %al,(%rax)
+ 2f1374:  00 00                   add    %al,(%rax)
+ 2f1376:  00 00                   add    %al,(%rax)
+ 2f1378:  a1 7a 0c 00 00 00 00    movabs 0x1400000000000c7a,%eax
+ 2f137f:  00 14
+ 2f1381:  00 00                   add    %al,(%rax)
+ 2f1383:  00 00                   add    %al,(%rax)
+ 2f1385:  00 00                   add    %al,(%rax)
+ 2f1387:  00 b2 7a 0c 00 00       add    %dh,0xc7a(%rdx)
+ 2f138d:  00 00                   add    %al,(%rax)
+ 2f138f:  00 0d 00 00 00 00       add    %cl,0x0(%rip)  <ngx_http_headers_out>
  2f1395:  00 00                   add    %al,(%rax)
  2f1397:  00 c7                   add    %al,%bh
  2f1399:  7a 0c                   jp     2f13a7  <ngx_http_headers_out>
  2f139b:  00 00                   add    %al,(%rax)
```

Listing 4.2: Diff of the disassembly for a bitflip applied to the `nginx` binary in order to bypass the credential check. The `add` instruction is prefixed with a `0x40`, which is the Intel REX prefix used to change addressing modes. This prefix changes the computation before the jump to `ngx_http_headers_out` at address `0x2f1399`.

```
   1341c5:  74 ad                   je     134174 <EVP_MD_CTX_cleanup>
   1341c7:  be 04 00 00 00          mov    $0x4,%esi
-  1341cc:  49 89 df                mov    %rbx,%rdi
+  1341cc:  48 89 df                mov    %rbx,%r15
   1341cf:  e8 dc e5 f3 ff          callq  727b0 <EVP_MD_CTX_test_flags@plt>
   1341d4:  85 c0                   test   %eax,%eax
```

Listing 4.3: Diff for a bitflip applied to the `libcrypto.so.1.0.2` binary in order to bypass a credential check. The move from `rbx` to `rdi` is exchanged with a move to `r15`, this changes the parameter for `EVP_MD_CTX_test_flags`, which is highly likely to result in a different outcome.

We want to find all exploitable bitflips which would allow an attacker to log into a remote machine without knowing the correct password of the user. The `ssh` server is configured to allow accesses from users using password authentication.

For testing, we start `sshd`, wait for the server to start and be ready to accept connections. We then use a `ssh` client to execute commands in the testing environment via the secure shell connection. We defined a success state when the output of the client showed a successful login. We verified the login by executing `whoami`, which returns the name of the logged in user. We tested version 7.4*p*1 of the `ssh` server.

4.3.1 Results

For the used `ssh` server we can only report working bitflips in the loaded libraries. Additional, all of the 41 reported exploitable bitflips were found in the `.text` section of the ELF files. Most bitflips were found in the `libc` library. Also, the cryptographic library `libcrypto.so.1.0.2` and the linker contain exploitable bits.

ELF File	opcode	parameter	Sum of flips:
`ld-linux-x86-64.so.2`	0	1	1
`libcrypto.so.1.0.2`	0	6	6
`libc.so.6`	4	30	34
Sum of flips:	4	37	41

Table 4.3: List of bitflips causing a bypass of the credential check in the `ssh` server and what part of the instruction they change.

Table 4.3 lists the bitflips exploiting the remote login and also lists what parts of the instructions are flipped to gain the credential check bypass. We can see that only about 10 % of the bitflips are in the opcode of instructions, the rest is in the parameters.

In Listing 4.3, we show a diff for the `libcrypto` library used by `ssh`. We can see how changing the function parameters (`rdi`) for `EVP_MD_CTX_test_flags@plt` changes the outcome of the libary call in a way ssh will think it is working with correct credentials.

Testing the `ssh` server took about 12 h CPU time. A large time of the testing was waiting for the startup of the server. We ran the tests in 12 parallel test environments.

4.4 Local Login Bypass

When a local login is attempted on GNU/Linux operating systems, it can be handled via the `login` program. The program provides different authentication methods via PAM modules, as described in Section 2.3.3. The `/bin/login` binary is checking credentials via a PAM module and if the given credentials are correct, the login shell for the given user is started with the user as process owner.

We want to find all bitflips which allow bypassing of a setup where login credentials consist of username and password. Also, we assume the attacker knows the username but not the password.

For testing, we run the `login` binary and give credentials containing the correct username and a wrong password. If the login succeeds, a shell will be started by the program. We run `whoami` to check if the login as the provided worked. If so, the bitflip provides a successful state for our testing. We tested the version 4.4 of `shadow-utils`, which provides `/bin/login`.

4.4.1 Results

We show exploitable bitflips in the `login` binary and loaded libraries. All the 178 exploitable bitflips were in the `.text` section of the ELF files. Most flips were found in libraries related to PAM, which is a major part of the user management in Linux-based operating systems.

ELF File	opcode	parameter	Sum of flips:
`pam_nologin.so`	0	2	2
`pam_securetty.so`	0	5	5
`libpam.so.0`	11	39	50
`login`	14	42	57∗
`pam_unix.so`	30	34	64
Sum of flips:	55	122	178∗

Table 4.4: List of bitflips causing a bypass of the credential check in the local login service and what part of the instruction they change. (∗) One bitflip in the `login` binary changed the execution context.

Table 4.4 shows the bitflips which result in a successful bypass of the `login` credential check, allowing the login of a user without providing the correct password. We can see that about one third of the bitflips are in the opcodes of instructions and the others are in the parameters.

In Listing 4.4, we show a diff for the disassembly of the `login` binary, where the execution context has been changed by flipping a bit in the `mov` instruction opcode, resulting in a longer `add` instruction and some other pointless instruction calls before the call of `test`. These changes provide other values which cause a successful login.

The testing of the local login service took about 35 h of CPU time. This long time is due to the longer response time by the program itself, which is used as a countermeasure against local brute-forcing.

```
  4c09:  48 89 e2                 mov    %rsp,%rdx
  4c0c:  be 02 00 00 00           mov    $0x2,%esi
  4c11:  e8 da e3 ff ff           callq  2ff0 <pam_get_item@plt>
  4c16:  85 c0                    test   %eax,%eax
- 4c18:  89 c5                    mov    %eax,%ebp
- 4c1a:  75 45                    jne    4c61 <freeaddrinfo@plt+0x1b99>
- 4c1c:  48 8b 3b                 mov    (%rbx),%rdi
- 4c1f:  48 85 ff                 test   %rdi,%rdi
- 4c22:  74 05                    je     4c29 <freeaddrinfo@plt+0x1b61>
- 4c24:  e8 0f e1 ff ff           callq  2d38 <free@plt>
- 4c29:  48 8b 3c 24              mov    (%rsp),%rdi
+ 4c18:  81 c5 75 45 48 8b        add    $0x8b484575,%ebp
+ 4c1e:  3b 48 85                 cmp    -0x7b(%rax),%ecx
+ 4c21:  ff 74 05 e8              pushq  -0x18(%rbp,%rax,1)
+ 4c25:  0f e1 ff                 psraw  %mm7,%mm7
+ 4c28:  ff 48 8b                 decl   -0x75(%rax)
+ 4c2b:  3c 24                    cmp    $0x24,%al
  4c2d:  48 85 ff                test   %rdi,%rdi
  4c30:  74 26                    je     4c58 <freeaddrinfo@plt+0x1b90>
  4c32:  e8 d9 28 00 00           callq  7510 <freeaddrinfo@plt+0x4448>
```

Listing 4.4: Diff for the disassembly after an exploitable bitflip was applied to the `login` binary in order to bypass the credential check. Changing the instruction for `mov` to an `add` with addressing modes sets changes the execution calls until the `test` instruction is called. These changes affect register values and the outcome of the program.

4.5 Summary of the Results

We report that all tested programs can be exploited via bitflips. We show that Rowhammer can be used for privilege escalation and bypasses of security measures in applications provided for GNU/Linux operating systems. Most exploitable bitflips were in the `.text` section of ELF files. Also, we show that not only the program's main binaries contain exploitable bitflips, but also dynamically loaded libraries are exploitable via bitflips.

Program name	opcode	parameter	context	Sum of flips:
`sudo`	144	281	0	425
`nginx`	65	231	2	298
`sshd`	4	37	0	41
`login`	55	122	1	178
Sum of flips:	268	671	3	942

Table 4.5: Summary of all exploitable bitflips and what part of instructions is changed by them.

As Table 4.5 shows, most of the exploitable bitflips were found in the parameters of instructions. From our results, only 28.45 % of the exploitable bitflips were in opcodes. What also can be concluded from our research is, that only a few bitflips exploit the program by changing more than a single instruction in the ELF file. From our tests only 3 happen to exploit programs in this way.

For `sudo`, we found the same exploitable 29 bitflips as Gruss et al. [15] in the `sudoers` library.

ELF File	shared flips
pam_unix.so	52
libpam.so.0	18
Sum of flips:	70

Table 4.6: Number of flips which work for a login bypass and a sudo privilege escalation.

```
  fc64:  41 57                 push   %r15
  fc66:  41 56                 push   %r14
  fc68:  41 55                 push   %r13
  fc6a:  41 54                 push   %r12
- fc6c:  53                    push   %rbx
+ fc6c:  57                    push   %rdi
  fc6d:  48 83 ec 38           sub    $0x38,%rsp
  fc71:  48 8b 05 c8 4c 21 00  mov    0x214cc8(%rip),%rax  <_rtld_global@GLIBC>
  fc78:  48 83 e8 01           sub    $0x1,%rax
  fc7c:  48 89 45 c8           mov    %rax,-0x38(%rbp)
```

Listing 4.5: Diff for the disassembly of the linker binary with a bitflip exploiting the `sudo` and `ssh` program. The bitflip changes the register pushed on the stack.

Besides these bitflips, we show 230 additional exploitable bitflips in the library and also report multiple other bitflips to achieve the same exploit as Gruss et al. [15] with our automated bitflip search.

Besides that, we can report that some bitflips can exploit multiple applications. For example, many flips in PAM can exploit `login` and `sudo` as both check against the user management. An attacker, therefore, could introduce a flip in the `libpam.so.0` file, log in as the user and gain superuser access if the user is allowed to use `sudo`. Table 4.6 show the shared bitflips between these two applications.

Also, for `sudo` and `sshd`, there was one shared bitflip found, one in the `ld-linux-x86-64.so.2` file. The change introduced the by bitlfip is shown in Listing 4.5. The bitflip changes the behaviour of the linker library and because of this change the outcome of the two programs lead to a successful state.

5 Bitflip Attacks on Dynamic Data

In this chapter, we take a look at data created at the runtime of a program and how bitflips applied to that memory could benefit an attacker. We will go into details about OpenSSL and how bitflips can enable for instance nonce-misuse attacks. We want to apply a similar attack as Böck et al. describe in their work about "practical nonce misusage attacks" [29]. By this, we want to show how Rowhammer can introduce new attack vectors to applications.

5.1 Analysis of OpenSSL for possible Nonce Misuse Flips

We look at the implementation of AES-GCM inside OpenSSL. We see the general AES-GCM context struct in Listing 5.1. Additional to that, we take a look at the cypher-context struct in Listing 5.2. The connection between those two is the initialisation vector `iv` which is updated on each call of the GCM function.

To apply the described attack by Böck et al. [29], we require the `iv` to be reused by multiple GCM calls. OpenSSL uses a counter for the `iv` for AES-GCM instead of a random value, flipping a bit to zero in the `iv` causes a decrement of the counter which makes reuse likely. The counter starts at a random value and is increased after. Therefore, a flip in the lower bits is more likely to result in reuse of the nonce. Böck et al. [29] state that this reuse constitutes breakage of the cryptography for future messages and the attacker can craft valid ciphertexts and overtake sessions. For the attack with Rowhammer, we target the `iv` array in the `EVP_CIPHER_CTX` struct, as seen in Listing 5.2.

5.1.1 Likelihood of a Nonce-Misuse introduced by Rowhammer

Lipp et al. [51] show that it is possible to send network requests which cause memory accesses that induce Rowhammer faults. With this knowledge and the possibility of nonce-misuse attacks in OpenSSL via bitflips, an attacker could overtake TLS sessions with only remote access.

Looking at the OpenSSL source code, we can see that for each TLS connection at least these two context structs are generated. There is one general SSL struct needed, one AES-GCM context struct and two cypher contexts, as one is used for sending and one for receiving. The working IVs make 32 byte of this memory. The sum of the structs for one TLS connection is 1960 byte. If we fill the DRAM with just these structs, Only 1.5 % of the memory would hold IVs. Therefore, the attack is already quite unlikely to succeed. As also, only lower bits should be flipped, to make a nonce counter reuse likely. We cannot just hold these structs in memory, and operating systems usually limit the number of parallel TLS connections.

```
typedef struct {
  union {
    double align;
    AES_KEY ks;
  } ks;                         /* AES key schedule to use */
  int key_set;                  /* Set if key initialised */
  int iv_set;                   /* Set if an iv is set */
  GCM128_CONTEXT gcm;
  unsigned char *iv;            /* Temporary IV store */
  int ivlen;                    /* IV length */
  int taglen;
  int iv_gen;                   /* It is OK to generate IVs */
  int tls_aad_len;              /* TLS AAD length */
  ctr128_f ctr;
} EVP_AES_GCM_CTX;
```

Listing 5.1: Struct used by OpenSSL to describe the AES-GCM context. The IV used is stored in the memory pointed to by `iv`. Source is taken from OpenSSL version 1.1.0*g*

```
struct evp_cipher_ctx_st {
  const EVP_CIPHER *cipher;
  ENGINE *engine;     /* functional reference if
                       * 'cipher' is ENGINE-provided */
  int encrypt;        /* encrypt or decrypt */
  int buf_len;        /* number we have left */
  unsigned char oiv[EVP_MAX_IV_LENGTH]; /* original iv */
  unsigned char iv[EVP_MAX_IV_LENGTH]; /* working iv */
  unsigned char buf[EVP_MAX_BLOCK_LENGTH]; /* saved partial
                                            * block */
  int num;           /* used by cfb/ofb/ctr mode */
  /* FIXME: Should this even exist? It appears unused */
  void *app_data;    /* application stuff */
  int key_len;       /* May change for variable length cipher */
  unsigned long flags; /* Various flags */
  void *cipher_data; /* per EVP data */
  int final_used;
  int block_mask;
  unsigned char final[EVP_MAX_BLOCK_LENGTH]; /* possible final
                                              * block */
} /* EVP_CIPHER_CTX */ ;
```

Listing 5.2: Context struct describing the Cipher used in TLS. This struct is used as the SSL context inside OpenSSL. Source is taken from OpenSSL version 1.1.0*g*

```
if (SSL_accept(ssl) <= 0) {
    ERR_print_errors_fp(stderr);
}
else {
  int run = 1;
  while(run)
  {
    if(SSL_write(ssl, reply, strlen(reply)) < 0)
      run = 0;
    sleep(1);
  }
}
SSL_free(ssl);
close(client);
```

Listing 5.3: Code showing an example for a simple TLS server, keeping sending a reply until the client disconnects.

5.1.2 Analysis of Practical Nonce-Reuse caused by Rowhammer

We set up two versions for the practical analysis of the nonce reuse, where one uses multiple processes, and one uses multithreading. In both cases, we implemented a simple endless sending loop which will keep the TLS connection to clients open and sent a string every second. Listing 5.3 shows the code parts used in both cases. The code makes sure the connection will send the reply every second, as long as the client accepts the write.

For our tests, we set the cypher suite to `ECDHE-RSA-AES256-GCM-SHA384` during the TLS handshake, by this, we make sure the described GCM structs are used by OpenSSL.

For our experiments, we had a memory increase of about 5440 byte for each forked process. With threading, the increase was slightly less. We never achieved filling a larger of the memory with TLS structs so that we came near the 1.5 %. Thus, the probability in practical implementations is therefore even lower. If it is possible to get 1000 parallel connections, with 5440 bytes each, with 32 bytes in initialisation vectors, it will result in 5.19 MB of memory representing TLS connections, with 0.59 % of this being IVs. In a server setup with 4 GB of DRAM, the IVs would only occupy 74.5×10^{-6} % of memory. Even if an attacker could gain knowledge about the 4 kB pages used to store TLS structs, it would be difficult to hit IVs with Rowhammer. Given this knowledge, we can conclude that a remote attack on TLS nonces with Nethammer is very unlikely to succeed.

6 Countermeasures

In this chapter, we discuss countermeasures. First, we look at countermeasures against microarchitectural attacks in general. Secondly, we look at countermeasures against Rowhammer. In the end we discuss measures that make our testing less practical, and our results less useful for an attacker.

6.1 Microarchitectural Attacks

Fixes for microarchitectural attacks can be applied to different layers of a system. The KAISER patch for kernel page-table isolation by Gruss et al. [16] is applied to the kernel to prevent leakage of kernel memory by Meltdown [50]. For other attacks, such as Spectre, Kocher et al. [67] state that lower layers, like the CPU's microcode, have to be fixed. Kocher et al. [67] also state that a long-term solution to prevent microarchitectural attacks is a fundamental change of instruction set architectures.

6.1.1 Rowhammer

Besides fixing systems to prevent microarchitectural attacks, it is also possible to prevent the exploitation of microarchitectural flaws by checking programs for malicious behaviour. Irazoqui et al. [26] present a microarchitectural-side-channel-attack trapper (MASCAT), which uses static analysis of binaries to detect microarchitectural attacks. Irazoqui et al. [26] state characteristics of attacks and what code parts indicate a microarchitectural attack. To detect a possible use of Rowhammer, MASCAT checks for cache evictions in binaries. Irazoqui et al. [26] show that using `clflush` in a loop is likely part of a Rowhammer attack. Also, `monvnti` and `movntdq` are listed as suspicious, as these instructions allow direct access to the DRAM and bypass caching mechanisms. Besides looking for these instructions, Irazoqui et al. [26] also state, that a program using self-map translation to resolve the mapping between physical and virtual memory is likely to be a threat to the system as knowledge about this mapping is part of Rowhammer attacks.

Besides statically checking binaries, checking the behaviour of a program also can detect an ongoing attack. Aweke et al. [102] present a software-based solution to detect ongoing Rowhammer attacks, namely ANVIL. ANVIL uses hardware performance monitoring to monitor DRAM row accesses. If a row is accessed repeatedly, ANVIL forces refreshing of neighbouring rows. Aweke et al. [102] implemented ANVIL as a kernel module for Linux and state that their detection and prevention system leads to an average slowdown of 1 % and a worst-case slowdown of 3.2 % of the system.

Brasser et al. [22] show another software mitigation against Rowhammer attacks. Their countermeasure prevents an attacker to corrupt kernel memory from user mode. Brasser et al. [22]

extend the physical memory allocator of the Linux kernel to isolate the memory of the kernel and the userspace on DRAM storage. The mitigation by Brasser et al. [22], CATT, does not prevent bitflips from happening but removes the possibility to exploit bitflips in kernel memory. CATT splits the memory layout into security domains and makes sure that memory from different domains is not placed in neighbouring rows on the DRAM chip. This placing prevents an attacker from using kernel memory as a target for Rowhammer.

Gruss et al. [15] show that it is still possible to mount privilege-escalation attacks with such countermeasures present. For the approach by Brasser et al. [22], they show that the isolation from kernel memory does not prevent privilege escalation as they introduce bitflips to ELF files, which then allow an attacker to gain superuser privileges. For defeating static analysis and performance counter based mitigations, Gruss et al. [15] present attacks based on the SGX enclave. Gruss et al. [15] state that this defeats performance counter monitoring countermeasures as executions inside the enclaves are not monitored by the CPU's performance counters.

To prevent exploiting ELF files with bitflips introduces to the storage devices operating systems can store a hash of the executables and check the checksums before execution. Bitflips attacks on storage devices are described by Kurmus et al. [4]. Cai et al. [94] show how MLC NAND flash memory devices can be targeted to introduce exploitable memory corruptions to solid-state drives. Checking the hash of the executable before execution would detect such a change of the ELF file and can prevent the execution of a binary containing an exploitable bitflip.

6.2 Making Bitflip Testing Less Practical

With our testing framework, we can search for exploitable bitflips in ELF files which are identical for every user of the software distribution. For example, the same exploitable bitflips have the same position in all installations of the current version of the Debian operating system. Distributing randomised ELF files would make the search for exploitable bitflips harder as an attacker would need to run the search framework for each target.

However, randomised binaries would not comply with the current trend of research in the field of reproducible builds. The Debian-close organisation reproducible-builds [9] works on making building packages for Debian reproducible. With their techniques, it is possible to produce the same binary using the same compiler, build configuration and same source code. Reproducible builds make binary changes detectable and allow a bit-by-bit verification of the full build chain. Consequently, changes or backdoors introduced by a malformed compiler or linker are detected.

Reproducible builds can also help to introduce other countermeasures against Rowhammer. The operating system can verify ELF files by using stored checksums. Suh et al. [24] show how memory integrity verification can bring a similar check to programs at runtime. Gelbart et al. [59] present a secure program execution environment (SPEE) providing static verification before execution and code block verification at runtime. During compilation, SPEE adds a signature and hashes to each ELF section of the executable. The operating system checks these additions before performing the execution. To allow checking during runtime, SPEE adds a security module to the Linux kernel which performs additional checks during execution. Andre Rein [71] present a software component for dynamic runtime integrity verification and evaluation (DRIVE). DRIVE adds monitoring for

memory changes during the runtime of a program. It uses data gained from the ELF file to check the memory of the process during the runtime.

Encryption of binaries can be seen as a countermeasure against exploitable bitflips, if decryption happens on each access to the file content based in DRAM. If a bit is flipped inside an encrypted ELF file, decryption is likely to produce garbage machine code for the CPU to execute, meaning the program would crash in case of an applied bitflip.

7 Future Work

In this chapter, we present an outlook for attacks in the field of Rowhammer. We derive possible future attack vectors from current trends. We also look at combinations of attacks, where Rowhammer could reintroduce exploits, especially regarding attacks in cryptography. Finally, we find possibilities to extend our testing framework in future works.

7.1 Future Work regarding Rowhammer

Kim et al. [93] state that it is not possible to make definitive claims on how word lines interact in DRAM chips and how the bitflips happen in detail. Kim et al. [93] also state that there are possible ways of interaction based on their studies and findings but research of DRAM chips at the device-level, regarding Rowhammer, is still missing.

While vendors could fix the exploitation in DRAM chips, by increasing refresh rates or built non-load-leaking chips, there is other hardware which already faces similar issues, as Cai et al. [94] show for MLC NAND flash memory used by solid-state drives.

We show how nonce misusage can be introduced by single bitflips to target an AES-GCM implementation. Attackers could use this idea to target other cryptographic implementations as well, especially those using counter nonces.

We show all exploitable bitflips in binaries to achieve a defined outcome of a program. We aim to find all exploitable bitflips in a time way slower than testing all bits in a binary. Searching for a single bitflip to achieve the same outcome in a short time is another research task for the future.

7.2 Improving and Reusing our Framework

For attacks based on memory accesses, we can take away that tools such as Pin [7] or `angr` [91] will be a significant factor for analysis. The work of Chabbi et al. [47] could also be taken to improve our testing framework as they provide an even more in-depth view of execution paths in binaries. By using such detailed graphs, more bytes-to-test could be filtered, speeding up the framework.

On the other side, we can view our framework as a base implementation for other test environments for parallel testing use cases. The environment is easy to adapt, and developers can exchange single components without much effort. Testing inside `chroot` could be changed to container environments like Docker [30], to provide more safety or better abstraction of the host operating system. Also, virtual machines could be used, if the software has to be tested on other kernels

or operating systems. The testing scripts could be replaced to work with multiple input files for a program instead of different binaries. This would make it possible to use the framework as a basis for a fuzzer. Even a combination with fuzzers would be possible, as the framework could be combined with `afl` [100] to provide a parallel fuzzing framework, which would either allow fuzzing in multiple environments or as a general way to parallelise `afl`.

Instead of instrumenting binaries with Intel Pin, replacements or other sources for analysis could be used. For instance, the QBDI [69] tool for dynamic instrumentation could be used instead, which supports a wider range of software architectures, such as `x86_32`, and AArch64. Also, QBDI supports instrumentation on the Android platform used by mobile phones. Additionally, a complete open source solution for instrumentation is available in DynamoRIO [17], which could also replace Pin.

8 Conclusion

We have shown that the Rowhammer bug is still relevant to system engineers and that users and vendors need to be aware of this issue.

We present a framework to test any binary to find bitflips which change the execution path to a given, pre-defined outcome. We apply this framework to programs available for GNU/Linux operating systems. We gain privilege escalation by exploiting `sudo`, bypass the credential check for local and remote login measures, and bypass HTTP basic authentication implemented by the `nginx` web server.

For `sudo`, Gruss et al. [15] show 29 bitflips which yield privilege escalation, which they find by manual analysis of the ELF files used by the program. We show that our framework finds over 10 times more bitflips gaining the same outcome in a few hours of testing time.

Besides searching for exploitable bitflips in ELF files, we also look at the consequences of bitflips applied to memory generated and used at the runtime of programs. As an example, we show how bitflips can introduce nonce reuse in the AES-GCM implementation of OpenSSL.

We present a summary of countermeasures already applied to systems against Rowhammer and how developers need to extend them. Also, we discuss measures which make systems more secure by checking the ELF files before execution, or check the memory of the process during the runtime.

We close with a recommendation for vendors of hardware, such as CPUs or DRAM chips, that besides increasing performance and space reduction, also the security of products should be improved. Hardware vendors and researchers should work together more often to find new microarchitectural flaws in hardware, and thereby improve the security for users.

Bibliography

[1] Abhay Bhushan, Ken Pogran, Ray Tomlinson, Jim White. *Standardizing Network Mail Headers*. RFC 561. Sept. 1973. URL: `https://tools.ietf.org/html/rfc561` (cit. on p. 16).

[2] Ancat. *python-pin*. (visited 08.08.2018). URL: `https://github.com/ancat/python-pin` (cit. on p. 13).

[3] Andrey Bogdanov, Dmitry Khovratovich, Christian Rechberger. "Biclique Cryptanalysis of the Full AES." In: *ASIACRYPT'11*. 2011 (cit. on p. 18).

[4] Anil Kurmus, Nikolas Ioannou, Nikolaos Papandreou, Thomas P. Parnell. "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks." In: *USENIX'17*. 2017 (cit. on pp. 29, 50).

[5] J. Antoine. "Authentication failures in NIST version of GCM." In: (2006) (cit. on p. 21).

[6] F. Bellard. *QEMU - the FAST! processor emulator*. (visited 28.09.2018). URL: `https://www.qemu.org/` (cit. on p. 36).

[7] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, Kim M. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *SIGPLAN'05 Conference on Programming Language Design and Implementation*. 2005 (cit. on pp. 2, 13, 14, 33, 37, 52).

[8] M. W. Chris Baraniuk. *Meltdown and Spectre: How chip hacks work*. (visited 06.09.2018). URL: `https://www.bbc.com/news/technology-42564461` (cit. on p. 1).

[9] Chris Lamb, Lunar, h01ger, Vagrant Cascadian, Ximin Luo. *reproducible-builds - a verifiable path from source code to binary*. (visited 20.08.2018). URL: `https://reproducible-builds.org/` (cit. on p. 50).

[10] D. D. Clark. "The design philosophy of the DARPA internet protocols." In: *SIGCOMM'88*. 1988 (cit. on p. 15).

[11] Colin O'Flynn, Zhizhang (David) Chen. "Side channel power analysis of an AES-256 bootloader." In: *CCECE'15*. 2015 (cit. on p. 18).

[12] J. V. D. McGrew. "The Galois/Counter Mode of Operation (GCM)." In: (2005) (cit. on p. 19).

[13] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. 1999 (cit. on p. 18).

[14] Daniel Gruss, Clémentine Maurice, Stefan Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *DIMVA'16*. 2016 (cit. on pp. 1, 25, 28).

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, Yuval Yarom. "Another Flip in the Wall of Rowhammer Defenses." In: *CoRR'17*. 2017 (cit. on pp. 1–3, 25, 28, 30, 38, 44, 45, 50, 54).

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, Stefan Mangard. "KASLR is Dead: Long Live KASLR." In: *ESSoS'17*. 2017 (cit. on pp. 25, 49).

[17] Derek Bruening. *DynamoRIO - Dynamic Instrumentation Tool Platform*. (visited 22.08.2018). URL: `https://www.dynamorio.org/` (cit. on pp. 14, 53).

[18] T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. URL: `https://tools.ietf.org/html/rfc5246` (cit. on p. 17).

[19] L. documentation. *pam.d(8) - Linux man page*. (visited 17.09.2018). URL: `https://linux.die.net/man/8/pam.d` (cit. on p. 15).

[20] L. documentation. *ssh(1) - Linux man page*. (visited 29.09.2018). URL: `https://linux.die.net/man/1/ssh` (cit. on p. 40).

[21] M. Dworkin. "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC." In: (2007) (cit. on pp. 19, 21).

[22] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, Ahmad-Reza Sadeghi. "CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory." In: *USENIX'17*. 2017 (cit. on pp. 49, 50).

[23] T. A. S. Foundation. *Apache - HTTP server*. (visited 08.08.2018). URL: `https://httpd.apache.org/` (cit. on p. 16).

[24] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, Srinivas Devadas. "Efficient memory integrity verification and encryption for secure processors." In: *ACM'03 Symposium on Microarchitecture*. 2003 (cit. on p. 50).

[25] GNU. *Permission Bits (The GNU C Library)*. (visited 08.08.2018). URL: `https://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html` (cit. on p. 14).

[26] Gorka Irazoqui, Thomas Eisenbarth, Berk Sunar. "MASCAT: Preventing Microarchitectural Attacks Before Distribution." In: *CODASPY'18*. 2018 (cit. on p. 49).

[27] S. Gueron. "Intel® advanced encryption standard (AES) new instructions set." In: (2010) (cit. on p. 18).

[28] H. Krawczyk, M. Bellare, R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. Feb. 1997. URL: `https://tools.ietf.org/html/rfc2104` (cit. on p. 17).

[29] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, Philipp Jovanovic. "Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS." In: *USENIX'16*. 2016 (cit. on pp. 3, 19, 21, 46).

[30] D. Inc. *Build, Manage and Secure Your Apps Anywhere. Your Way.* (visited 22.08.2018). URL: `https://www.docker.com/` (cit. on p. 52).

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 253669-033US. 2009 (cit. on pp. 6, 7, 22–24).

[32] Intel Corporation. *White Paper - Introduction to Intel® Architecture*. 330119-001US. 2014 (cit. on pp. 23, 24).

[33] *Information technology – Security techniques – Encryption algorithms – Part 3: Block ciphers*. Standard. International Organisation for Standardisation, 2010 (cit. on p. 18).

[34] James C. King. “Symbolic Execution and Program Testing.” In: *Commun. ACM* 19.7 (1976), pp. 385–394 (cit. on p. 12).

[35] JEDEC Solid State Technology Association. *Double Data Rate (DDR) SDRAM Specification*. 2003 (cit. on p. 27).

[36] Jo Van Bulck, Marina Minkin, Ofir Weisse, Danien Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, Raoul Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution.” In: *USENIX'18*. 2018 (cit. on p. 25).

[37] Joe W. Duran, Simeon Ntafos. “A Report on Random Testing.” In: *ICSE'81*. 1981 (cit. on p. 11).

[38] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel Cristiano Giuffrida, Herbert Bos. “Flip Feng Shui: Hammering a Needle in the Software Stack.” In: *USENIX'16*. 2016 (cit. on p. 28).

[39] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, Chengyong Wu. “A software memory partition approach for eliminating bank-level interference in multicore systems.” In: *PACT'12*. 2012 (cit. on p. 27).

[40] Mark Seaborn, Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges.” In: *Blackhat Briefings* (2015) (cit. on p. 27).

[41] A. Markov. *xmlfuzzer*. (visited 08.08.2018). URL: `http://komar.bitcheese.net/en/code/xmlfuzzer` (cit. on p. 11).

[42] F. P. Mathy Vanhoef. “Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2.” In: *SIGSAC'17 Conference on Computer and Communications Security*. 2017 (cit. on p. 18).

[43] Matthias Jung, Carl Christian Rheinländer, Christian Weis, Norbert Wehn. “Reverse Engineering of DRAMs: Row Hammer with Crosshair.” In: *MEMSYS'16*. 2016 (cit. on p. 27).

[44] R. C. Merkle. “Secure Communications Over Insecure Channels.” In: (1978) (cit. on p. 17).

[45] Michael Neve, Kris Tiri. “On the complexity of side-channel attacks on AES-256 - methodology and quantitative results on cache attacks.” In: *IACR Cryptology ePrint Archive* 2007 (2007), p. 318 (cit. on p. 19).

[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, Stefan Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript.” In: *FC'17*. 2017 (cit. on p. 23).

[47] Milind Chabbi, Xu Liu, John M. Mellor-Crummey. “Call Paths for Pin Tools.” In: *CGO'14 Symposium on Code Generation and Optimization*. 2014 (cit. on p. 52).

[48] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, Stefan Mangard. “Practical Keystroke Timing Attacks in Sandboxed JavaScript.” In: *ESORICS'17*. 2017 (cit. on p. 23).

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer and Clémentine Maurice, Stefan Mangard. “ARMageddon: Cache Attacks on Mobile Devices.” In: *USENIX'16*. 2016 (cit. on p. 23).

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. "Meltdown." In: 2018 (cit. on pp. 1, 24, 25, 49).

[51] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, Lukas Lamster. "Nethammer: Inducing Rowhammer Faults through Network Requests." In: *CoRR'18*. 2018 (cit. on pp. 1, 46).

[52] Netcraft. *January 2018 Web Server Survey*. (visited 19.09.2018). URL: `https://news.netcraft.com/archives/2018/01/19/january-2018-web-server-survey.htm%20l` (cit. on p. 16).

[53] A. Newcomb. *Meltdown, Spectre chip flaws raise questions about hardware security*. (visited 06.09.2018). URL: `https://www.nbcnews.com/tech/security/meltdown-spectre-chip-flaws-raise-questions-about-hardware-security-n834701` (cit. on p. 1).

[54] I. Nginx. *nginx - HTTP and reverse proxy server*. (visited 08.08.2018). URL: `https://nginx.org/en/` (cit. on pp. 16, 40).

[55] Nicholas Nethercote, Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *SIGPLAN'07*. 2007 (cit. on p. 14).

[56] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS'16*. 2016 (cit. on pp. 2, 12).

[57] Niels Provos, David Mazières. "A Future-Adaptable Password Scheme." In: *USENIX'99*. 1999 (cit. on p. 16).

[58] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Björn Andersson, Lutz Wrage, Mark H. Klein, Ragunathan Rajkumar. "Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses." In: *CSE'13*. 2013 (cit. on p. 27).

[59] Olga Gelbart, Bhagirat Narahari, Rahul Simha. "SPEE: A Secure Program Execution Environment tool using code integrity checking." In: *Journal of High Speed Networks* 15.1 (2006), pp. 21–32 (cit. on p. 50).

[60] Onur Aciiçmez, Çetin Kaya Koç, Jean-Pierre Seifert. "On the Power of Simple Branch Prediction Analysis." In: *IACR'06*. 2006 (cit. on p. 22).

[61] Onur Aciiçmez, Shay Gueron, Jean-Pierre Seifert. "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures." In: 2007 (cit. on p. 22).

[62] Onur Aciiçmez, Werner Schindler. "A Major Vulnerability in RSA Implementations due to MicroArchitectural Analysis Threat." In: 2007 (cit. on p. 22).

[63] OpenSSL Software Foundation. *OpenSSL - Cryptography and SSL/TLS Toolkit*. (visited 08.08.2018). URL: `https://www.openssl.org/` (cit. on p. 17).

[64] OpenSSL Software Foundation. *OpenSSL - Cryptography and SSL/TLS Toolkit*. (visited 08.08.2018). URL: `https://www.openssl.org/source/old/1.1.0/openssl-1.1.0g.tar.gz` (cit. on p. 21).

[65] Owen Lo, William J. Buchanan, Douglas Carson. "Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)." In: *Journal of Cyber Security Technology* 1.2 (2017), pp. 88–107 (cit. on p. 19).

[66] D. Page. “Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.” In: *IACR'02* 2002 (2002), p. 169 (cit. on p. 24).

[67] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution.” In: 2018 (cit. on pp. 1, 24, 25, 49).

[68] Project Zero. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. (visited 08.08.2018). URL: `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-%20gain.html` (cit. on pp. 1, 25, 27, 28).

[69] Quarkslab. *QBDI - QuarkslaB Dynamic binary Instrumentation*. (visited 22.08.2018). URL: `https://qbdi.quarkslab.com/` (cit. on p. 53).

[70] J. R. R. Fielding. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. URL: `https://tools.ietf.org/html/rfc7231` (cit. on pp. 16, 17).

[71] A. Rein. “DRIVE: Dynamic Runtime Integrity Verification and Evaluation.” In: *ACM'017 Conference on Computer and Communications Security*. 2017 (cit. on p. 50).

[72] J. Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Sept. 2015. URL: `https://tools.ietf.org/html/rfc7617` (cit. on p. 16).

[73] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. URL: `https://tools.ietf.org/html/rfc8446` (cit. on p. 17).

[74] P. Rogaway. “Authenticated-encryption with associated-data.” In: *CCS'02*. 2002 (cit. on p. 18).

[75] P. Rogaway. “Nonce-Based Symmetric Encryption.” In: *FSE'04 Workshop - Fast Software Encryption*. 2004 (cit. on p. 18).

[76] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” In: (1978) (cit. on p. 17).

[77] Rui Qiao, Mark Seaborn. “A new approach for rowhammer attacks.” In: *HOST'16*. 2016 (cit. on p. 28).

[78] Sarani Bhattacharya, Debdeep Mukhopadhyay. “Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis.” In: *CHES'16*. 2016 (cit. on p. 28).

[79] Shuai Wang, Dinghao Wu. “In-memory fuzzing for binary code similarity analysis.” In: *ASE'17Conference on Automated Software Engineering*. 2017 (cit. on p. 12).

[80] P. F. Syverson. “A Taxonomy of Replay Attacks.” In: *CSFW'94 Computer Security Foundations Workshop*. 1994 (cit. on p. 18).

[81] L. M. T. Berners-Lee R Fielding. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. URL: `https://tools.ietf.org/html/rfc3986` (cit. on p. 16).

[82] The Apache Software Foundation. *Authentication and Authorization*. (visited 08.08.2018). URL: `https://httpd.apache.org/docs/2.4/howto/auth.html` (cit. on p. 16).

[83] The Apache Software Foundation. *htpasswd - Manage user files for basic authentication*. (visited 08.08.2018). URL: `https://httpd.apache.org/docs/2.4/programs/htpasswd.html` (cit. on p. 16).

[84] The Open Group. *chroot - change root directory*. (visited 13.08.2018). URL: `http://pubs.opengroup.org/onlinepubs/7908799/xsh/chroot.html` (cit. on p. 15).

[85] The Open Group. *IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008) - POSIX.1-2017*. (visited 08.08.2018). URL: `http://pubs.opengroup.org/onlinepubs/9699919799/` (cit. on p. 14).

[86] The Open Group. *setuid - set user ID*. (visited 08.08.2018). URL: `http://pubs.opengroup.org/onlinepubs/9699919799/` (cit. on p. 14).

[87] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. 1995 (cit. on pp. 9, 10).

[88] Tom van Goethem, Wouter Joosen, Nick Nikiforakis. "The Clock is Still Ticking: Timing Attacks in the Modern Web." In: *SIGSAC'15*. 2015 (cit. on p. 23).

[89] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms." In: *SIGSAC'16*. 2016 (cit. on pp. 1, 25).

[90] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, Giovanni Vigna. "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." In: *NDSS'15*. 2015 (cit. on pp. 2, 12).

[91] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *SP'16 Symposium on Security and Privacy*. 2016 (cit. on pp. 2, 12, 36, 52).

[92] K. S. Yim. "The Rowhammer Attack Injection Methodology." In: *SRDS'16*. 2016 (cit. on p. 28).

[93] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." In: *ISCA'14*. 2014 (cit. on pp. 1, 25, 27, 28, 52).

[94] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, Erich F. Haratsch. "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques." In: *HPCA'17*. 2017 (cit. on pp. 50, 52).

[95] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, Radu Teodorescu. "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation." In: *USENIX'16*. 2016 (cit. on p. 28).

[96] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, Alwen Tiu. "Steelix: program-state based binary fuzzing." In: *FSE'17 Foundations of Software*. 2017 (cit. on p. 12).

[97] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi. "Cryptanalysis of DES Implemented on Computers with Cache." In: *CHES'03*. 2003 (cit. on p. 24).

[98] Yuval Yarom, Katrina E. Falkner. "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 448 (cit. on p. 24).

[99] M. Zalewski. *afl documentation*. (visited 28.09.2018). URL: `http://lcamtuf.coredump.cx/afl/README.txt` (cit. on p. 36).

[100] M. Zalewski. *american fuzzy lop*. (visited 08.08.2018). URL: `http://lcamtuf.coredump.cx/afl/` (cit. on pp. 1, 11, 36, 53).

[101] B. Zbarsky. *Clamp the resolution of performance.now() calls to 5us*. (visited 19.09.2018). URL: `https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab` (cit. on p. 23).

[102] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, Todd M. Austin. "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks." In: *ASPLOS'16*. 2016 (cit. on p. 49).