Lukas Raab,

Nethammer: Rowhammer over Network

Bachelor's Thesis

to achieve the university degree of

BSC

Bachelor's degree programme: Softwareentwicklung und Wirtschaft

submitted to

Graz University of Technology

Supervisor

Michael Schwarz

Institute of Applied Information Processing and Communications
Head: Univ.-Prof. Reinhard Posch

Graz, November 2017

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 bachelor's thesis.

Date	Signature

Abstract

Cloud computing and virtual servers become more important daily. Prices became affordable and even private users might rent a virtual machine. This makes security more and more important. Many researches showed that there are several security issues due to sharing parts of the hardware such as the CPU cache or the memory. However, until now, all of these side-channel attacks require remote access to execute software on the target system.

This paper is going to present a new method to attack a virtual machine (VM) without executing any software on the target system. The user datagram protocol (UDP) is used to flood a VMs network stack and since all of this packages need to be processed by the operating system (OS) kernel, a new Rowhammer attack over network (Nethammer) is possible. This hammering can cause several security issues. First, this can lead to a new type of Denial-of-Service (DoS), due to a kernel crash or crashing, for example, the web service. Second, various other security issues occur due to corrupting userspace memory.

This attack is really fast. We observed bitflips within 350ms and just 450.000 memory accesses. Due to this speed the attack might harm the victim before potential firewalls or other security systems can detect it. Further, we evaluated general statistics for one-location Rowhammer. This includes data about the minimal number of accesses until the first bitflip occurred and diagrams comparing the number of bitflips within different time periods between two accesses.

Contents

List of Figures

List of Tables

1 Introduction

We are living in a world full of computers. Everybody carries around many computers and uses them many times a day. Most people do not consider that inside their credit card, student ID or calculator hides a small intelligent calculation device helping their needs.

Cyber security became a big business, exploiting programming mistakes of others leads to money or big financial damage of others. It gets more and more important everyday. Quite often there are news about companies, governments or researchers mentioning new types of attacks and complaining about security risks. Most of these exploits are based on software mistakes and every computer is vulnerable in the same way. This includes breaking an encryption, phishing private data, infecting users with malware or mounting a Denial-of-Service attack. Cyber attacks can lead to financial damage and should be impossible. The good thing about most software mistakes is, that once the programmers know the issue they will fix it and publish a patch. The user just needs to update the program and the potential threat is closed.

Even less people think about attacks exploiting the law of physics, such as hardware attacks. This work is based on the well-researched Rowhammer attack [1, 2, 3, 4, 5, 6, 7, 8].

Rowhammer attacks are hardware attacks which try to harm the victim by changing the memory without having valid permissions. In a modern operating system, memory is assigned to a certain process to protect it from malicious other processes like a malware. This means every process is just allowed to read, write and execute its own memory and has no possibility to access the memory of an other process except the memory is explicitly shared.

To modify the memory, the attacker frequently accesses memory locations to create electromagnetic coupling effects. There are three Rowhammer access patterns (one-location hammering, single-sided hammering and double-sided hammering), which describe the memory access strategy [1, 2, 3, 4, 5, 6, 7, 8]. These electromagnetic coupling effects might change the stored information without any privileges. This makes Rowhammer attacks really dangerous. First, one changed bit might be enough to take over control of another computer. Second, it is hard do protect against hardware attacks by using software. Until now, there is no reliable method to avoid the Rowhammer attack just by using a protection software [1]. This is why the user might need to change the memory modules to expensive new memory modules, which leads to many users keeping the old memory.

Kim et al. did extensive research which shows how widespread these vulnerable memory modules are. They found out that rather 85% (110 out of 129) of modern memory modules are vulnerable. The tested memory modules were from three major manufacturers. Therefore, many systems might be in risk to get hacked [8].

Usually, frequently used memory addresses get stored in the CPU cache which is about three times faster than the memory. Therefore, whenever a program accesses an address the system looks up the corresponding value in the cache. If this address is not stored in the cache (cache-miss), the DRAM gets accessed. Due to the very fast access in the memory and the related electromagnetic coupling effects it is possible to change bits nearby without accessing them. In order to access the physical memory the needed addresses must not be stored in the cache to access the memory. To accomplish this requirement, known attacks use different ways to flush a stored address out of the cache. *clflush* is an CPU instruction which clears the value of a certain address in the CPU cache. This means the next time this address is accessed, the CPU has to load the value from the memory.

However, until now, existing Rowhammer attacks required access to the target computer or another VM on the same physical system to execute their program on the target CPU. The attacker needs to know the VM provider and most importantly due to many systems of every provider, a large amount of luck to get the same system. To attack a specific server this requirement is hard to satisfy. Moreover, without access, it was not possible to execute the malicious software using the *clflush* instruction and millions of memory accesses per second.

The Nethammer shows a new Rowhammer attack, without any valid access to the target or the host system. It just requires a fast working network connection and two computers to flood the victim with millions of senseless UDP packages. 500Mbit/s is enough uplink to induce bitflip at the target system. This has an huge impact on the security of webservices. Since webservers have to reply to web requests, the underlying OS has to process every single UDP package even if the destination port is closed and the package unrequested. This circumstance leads to the possibility of a remote Rowhammer attack.

There is still the need to disable caching the addresses the attacker is hammering on, without having the chance to use the *clflush* CPU instruction. Therefore, we are going to exploit the Intel CAT technology. As other research shows, using Intel CAT it is possible to create memory errors and evict cached address even faster than using the *clflush* instruction [5].

This work shows a new threat to disturb the unobstructed workflow of a server just by sending millions of network packages. This might result for example in a Denial-of-Service. In contrast to known Distributed-Denial-of-Service attacks, the Nethammer needs just 500MBit/s uplink to perform a successful attack. Moreover, the Nethammer does not freeze the victim due to a overload of packages [9] but the software of the attacked system might crash due to changes in the physical memory.

Many times, virtual machines are used to host a webserver or other services such as a GitLab server. All these services require the three major requirements of secure systems: *availability*, *confidentiality*, *integrity*. These might be violated by the Nethammer. However, to attack for example a VM, with the Nethammer the victim system has to fulfill the following requirements:

1. *Intel Cache Allocation Technology (Intel CAT)*: Due to no access to the CPU the attacker needs a possibility to replace the *clflush* instruction with a technology which is exploitable over network. As other research shows, it is possible to evict cached addresses using Intel CAT. Aga et al. showed a higher bitflip frequency using Intel CAT than by using *clflush*[5].

2. *One-location Bitflips*: this work shows that there is a rather minimal chance of doublesided hammering [2]. This results in the knowledge that all target configurations need to be vulnerable to one-location hammering bitflips [1].
3. *Network access*: this requirement is ubiquitous, most of the virtual machines have access to the internet which makes them a possible victim.

The outcome of the Nethammer is really interesting. It is possible to introduce bitflips just by sending packages. These flips occur either in the userspace, kernelspace or both. This results in a big threat model.

1.1 Outline

The structure of the paper is as follows. Section 2 discusses various types of related work, it explains the required computer vocabulary, technology terms and Linux tools. Section 3 describes the Nethammer idea very precisely, it gives an attack overview and a detailed threat model. Section 4 shows how to find the vulnerable memory accesses and mentions all used tools. Section 5 illustrate the possibilities of a network one-location Rowhammer. Finally, in Section 6, there is a conclusion about the Nethammer, discussing what currently is possible, the limitations and further research work.

2 Background

This chapter provides all necessary background information on the Nethammer attack.

2.1 Operating System

The OS is the software heart of the computer, it gets initially loaded by the bootloader at boottime. The operating system works as a referee by isolating different users and applications. It manages the software and hardware resources and finally handles the communication with the hardware. The OS pretends a large amount of memory and an abstraction of hardware to allow hardware independent programming and make software compatible with any hardware. It helps the programmer to not have to worry about handling resources or scheduling processes. Moreover, it is a layer of protection to defend, for example, hardware from malicious software or a novice programmer [10, 11].

Nowadays there are the three common desktop OS ordered by their marketshare [12]:

1. Windows
2. OSX
3. Linux

For smartphones there are mainly two important OS [13]:

1. Android
2. iOS

Kernel: The kernel is the central software part of every OS. It is privileged to do everything without any software limitations. This includes, for example, handling processes and actually using the hardware devices such as the hard-disk drive or the network. Many drivers like keyboard, mouse or network are included into the kernel using kernel modules. These modules process the input and makes it available to other software using an interface. The OS recognizes generic hardware IDs and tries to load the driver to make use of the hardware. Whenever a new user program gets started the kernel sets up the memory, CPU registers and its threads until it can run on its own. For user programs, the kernel works like an interface that handles hardware access and watches everything what happens. Whenever a user program wants to perform restricted actions, it communicates with the kernel using an interface. This is called a *syscall*. Every action which requires kernel privileges has a unique number (syscall number). Using this syscall number and additional parameters the kernel knows what the user program wants it to do. For example write (syscall number 4) to the hard-disk drive [14, 11].

Currently, most kernels are closed source which means the source code is not published. However, the Linux and Android kernel are open source. This gives us the chance of understanding

what the OS is doing and setting up a test environment with debug features and to modify the kernel at needed positions [15, 16].

User Programs: Every program running outside of the kernel is called a user program. A user program runs in its own virtual address space (Section 2.2.1). It gets executed in a protected environment with limited direct access to hardware. It can use the CPU or memory but whenever a user program needs to interact with restricted hardware or kernel features a syscall is required to communicate with the kernel to dictate jobs [11].

File System - Inode: A file system controls how and where data is stored. It manages a table containing all files and folders in a fast reusable way. Currently, there are several different types of file systems in use, since kernel version 2.6.28 the Linux standard uses the fourth extended filesystem (EXT4) [17]. EXT4 works with Inodes to handle all files and their attributes. Inode stands for index node and represents a single file on the system [18]. It knows the position on the drive and several other attributes such as the size or permissions of the file [18].

2.2 Memory Address Space

The Memory Address Space defines a range of addresses that can be allocated. It depends on the system architecture. This Section is going to describe 64-bit systems.

2.2.1 Virtual Address Space

Virtual memory is a type of memory management. In order to access the actual memory, virtual address have to be translated to physical addresses. Modern 64-bit systems would be able to address 64-bit of memory. This gives an virtual address space form `0x0` to `0xffff ffff ffff ffff`. Currently, just 48-bit out of 64-bit are used. The OS working as an illusionist pretends the program to have full 48-bit memory even if the system has less physical memory. The program does not need to worry about its memory the kernel is going to manage everything in the background.

On Linux systems, the whole address space is separated into two ranges as follows: [16]
The **userspace** for x86_64 architecture starts at `0x0` and goes up to `0xfff7 ffff ffff ffff` (this is called the pageoffset).
The **kernelspace** consume the other half that means it reaches from `0xfff8 0000 0000 0000` to `0xffff ffff ffff ffff`.

Both kernel and userspace work with contiguous virtual addresses. Whenever the CPU accesses a virtual address, it gets **translated** by the Memory Management Unit (MMU) into a physical address (Section 2.2.2). In the userspace the translation from virtual to physical addresses is not a direct mapping meaning that there is no simple offset calculation to translate virtual addresses to physical addresses. The smallest physical contiguous section is called a *page*. The size of a page is system dependent but most configurations use a page size of 12 bit (= 4kB). In contrast to userspace addresses, kernelspace addresses are offset-based which means they

can be translated easily to physical addresses just by subtracting the pageoffset from the virtual address [19, 20].

Since every program can use this huge virtual address space, the OS might run out of physical memory. Whenever this happens the kernel will copy physical memory to the hard disk drive in order to get free physical memory. This process is called **swapping** [21].

2.2.2 Physical Address Space

A 64-bit architecture could theoretically address 2^{64} byte but currently only 48-bit of 64-bit are used. This means the system can address up to

$$2^{48} = 256\,\text{Terabyte (TB)} \tag{2.1}$$

of memory. Some brand new systems already support 57-bit virtual addresses [16].

The translation from virtual to physical addresses is done by the MMU using paging structures like the pagetable and its entries. [22]

2.2.3 Memory Protection

The following content is related to the Linux system, other OS might handle memory protection differently. In general, the MMU is able to control the access rights of the computer memory on a per-page basis. There are three different permissions which can be combined with each other: [23]

1. *Read:* the program is allowed to read from this location
2. *Write:* the application is allowed to read and write to this address
3. *Execute:* finally the program is allowed to read and execute commands from this address

Often software enforces executable pages to be not writable to avoid security issues. This is called "W ^ X" which stands for Write XOR Execute policy. It means a page can either be writeable or executable but not both. Any violation to this access rights leads to a segmentation fault, resulting in the OS killing the process to avoid other problems like memory corruption [23].

2.3 Virtualisation

Virtualisation describes the approach of creating virtual versions of existing systems. The virtualisation software pretends virtualised resources like hardware components to a virtual machine. The virtualised computer or application does not know about its own virtualisation and acts like a real computer. Most servers are running in idle for a big part of their lifetime. As a result, especially server systems are virtualised, this saves energy and hardware costs [24, 25]. Sometimes the user wants to run a second OS simultaneously on the same computer: for example, host system Linux and a virtualised Windows to use incompatible programs.

The downsides of virtualisation are performance costs due to a overhead and the computer security gets more complex [24]. Typically, there are two types of attacks. First, VM-escape: the attacker tries to escape from the virtualised environment. Second, side channel attacks to leak information from an other VM running on the same physical hardware. For example there is a Flush+Reload attack which lead to AES key recovering [26, 27].

Basically, a VM consists of a Host Virtual Machine and a Guest Virtual Machine [25].

2.3.1 Host Virtual Machine

The host machine is the real computer, it has full control over the hardware to provide the required physical computing resources. Since it is running a virtual machine monitor (VMM) or hypervisor, it has full control over the guest machines and has to manage them. The VMM separates all VMs enabling running multiple different OS on the host system. [24]

2.3.2 Guest Virtual Machine

The guest VM is a virtualised operating system or application that is running in an isolated environment. Due to the layer of abstraction and illusion the VM does not know that it is emulated in an virtual environment and acts like every other computer. [24]

2.3.3 Intel Cache Allocation Technology

The Intel CAT gets used by the Xeon Processor E5 v4 Family. These CPUs are usually used for server systems running VMs. Software helps to control the last-level cache (LLC) to prioritize applications. All VMs, applications and threads are separated into different classes of service (CLOS) to control the different priority groups. The cache and CLOSs are bitmasked to control the amount, ways and position of usable cache. [28]

This approach helps to get a fair-use policy of the CPU cache on virtualised server systems. Often low cost VM provider assign just very little cache to the VM, enabling Rowhammer attacks. [5] As recent work shows, the Intel CAT functionality can increase the performance of Rowhammer attacks without even having access to the *clflush* instruction [5].

Our research tries to exploit this security issue over network, as described in Section 3.2.

2.4 RAM

When people talk about the memory of a computer, they usually think about the Random-Access Memory (RAM). On 64-bit systems 64-bits (data bus width) of the RAM can be addressed directly using the address bus. Whenever the processor requests data from memory, it is sent from the memory through the cache (Section 2.5.1) to the CPU.

2.4.1 DDR-SDRAM

The Dynamic Random-Access memory (DRAM) is a subtype of RAM. It is a volatile memory meaning that whenever it loses the electric supply, after some time, every stored information is lost and unrecoverable. Without modifying the physical environment the data is lost within seconds. This property can be exploited by cold boot attacks [29]. A running system periodically refreshes its data (refresh rate) to prevent data loss. PCs and laptops mostly use Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM). The name Double Data Rate (DDR) describe the fact of using the falling and rising triggers of the clock frequency to increase the speed. Modern systems use DDR4 modules. It is the successor of DDR3 SDRAM modules. Previous versions are deprecated. Together they hold a marketshare of about 3% [30, 31].

2.4.2 DRAM Organization

DRAM is organized in a hierarchy of channels, DIMMs, ranks and banks. A computer setup normally has one or two channels, which are used to distribute memory traffic. The actual physical memory modules are called Multiple Dual Inline Memory Modules (DIMMs) which are placed on the motherboard. Each DIMM has one or two ranks, which often match to the front and back of the module. Finally, every rank consists of multiple banks which contain the actual memory. The number of banks depends on the type of DDR. Every bank is organized like a grid with columns and rows. These columns and rows get addressed by the memory bus to read the saved data [2].

2.4.3 Reverse engineering the DRAM addressing

Pessl et al. describe how to reverse the DRAM addressing to increase the efficiency of double-sided Rowhammer attacks. Summarized, they found out that the memory controller translates physical addresses with simple linear functions. In general, there are multiple simple calculations: bits of the physical address are XORed with other bits of the physical address. The result determines the channel, the DIMM, the rank, the bank and the row. Inside a row the memory gets address contiguously. DRAM address differ with different CPUs, owing to a full automated tool the linear addressing function can be calculated easily on most modern CPUs. This helps to reverse the DRAM addressing on remote systems. Moreover, it improves Rowhammer attacks enormously and enables practical double-sided hammering on DDR4 SDRAM. [2]

2.5 Rowhammer

Rowhammer is a well-researched hardware security issue [1, 2, 3, 4, 5, 6, 7, 8]. This attack modifies physical adjacent storage location due to electromagnetic coupling effects created by frequent activation of rows in the DRAM. The attacker tries to influence an independent location by frequently accessing memory locations. As a result of the DRAM hierarchy,

Rowhammer attacks are limited to target memory addresses in the same bank. Normally, frequently accessed addresses are served from the CPU cache due to the principle of locality. Therefore, no row activation takes place and no memory errors occur. In order to exploit the Rowhammer bug, the attacker needs to flush cached addresses to access the physical memory, as described in Section 2.5.1. [8, 2]

2.5.1 Cache eviction

The cache is located in the dataflow between the CPU and the RAM. It is more than three times faster than the memory. To increase the performance it stores the requested data temporarily due to the principle of locality which says that applications trend to reuse data they have already used recently. Whenever the processor requests data, first it tries to find the data within the cache. If the data is still in the cache, this is called a cache-hit and the CPU receives the data faster because it does not have to access the memory. Otherwise, it is called a cache-miss, and the data needs to be loaded from the memory which takes about three times more time [32].

Most CPU caches have three levels, the lower the cache level the faster is the speed and the smaller the capacity. The cache size depends on the processor but is small compared to the memory. Currently, modern systems use multiple GB memory but level 3 cache (L3-cache) has a capacity in the low megabyte range, for example, the Intel Xeon D-1541 has 12MB L3-cache [5].

A Rowhammer attack needs a method to evict the last level cache(LLC) addresses in order to access the memory and therefore arise electromagnetic coupling effects. State of the art are the following three methods:

1. *clflush* instruction [8]
2. Cache eviction [4]
3. Exploiting Intel CAT [5]

2.5.2 History and Related Work

There are some hints pointing out that manufacturer already knew about bitflips in memory but they thought this is just a stability issue [33]. Already in 2003, researchers found out that memory errors could lead to take over control of a virtual machine [34]. In 2014 researchers studied DRAM disturbance errors [8]. Since 2015 the security issue gets researched extensively. There are several exploits, for example, sandbox escaping [3], getting root permissions [3] a JavaScript attack [4] or RSA public key flipping [6]. The Rowhammer bug is a serious security issue without any possibility to protect yourself [1].

The following Sections are going to discuss some exploits in more detail.

Linux - Kernel privileges

As mentioned in Section 2.2.1, the MMU translates virtual addresses to physical addresses. Therefore, OS use a data structure called page table (PT). The final page table entry (PTE) gets addressed using the CR3 register of the CPU and the logical memory address fields of an address. The PTE has an attribute which stores the actual physical page number (PPN) of a virtual page. [22] As Seaborn and Dullien showed, it is possible to get kernel privileges by manipulating a PTE with bitflips. They got write access to the process's own page table and therefore they could change every physical memory location in the system [3].

Android - Deterministic Root exploit

Android is the most popular mobile OS [13]. This makes cyber attacks very interesting. Prior hardware attacks like Prime+Probe, Flush+Reload, Evict+Reload and Flush+Flush where shown on non-rooted devices [35]. Extensive research shows that also mobiles device are vulnerable to bitflips [7]. Van der Veen et al. showed a deterministic way to get root access on an Android smartphone. They tested 17 Nexus 5, 2 OnePlus One and many other ARMv7 and ARMv8 phones. Summarized, they found out that bitflips occur almost on every ARMv7 phone. Finally, they even implemented an end-to-end privileges escalation attack using Stagefright and Rowhammering to get root permissions on an Android smartphone [7].

Rohammer.js

Based on the idea of escaping a virtual environment, Rowhammer.js generates bitflips with pure JavaScript. The attack takes advantage of the speed of modern computers and therefore JavaScript. Gruss et al. developed a program which causes bitflips in the Firefox webbrowser and possible a similar program will work in other browsers. In native environments, Rowhammer attacks use the *clflush* instruction to flush cached addresses. This is not possible with pure JavaScript, therefore they developed new cache eviction strategies to simulate the *clflush* behavior. Finally, this exploit works like all the other Rowhammer attacks: due to these bitflips the attacker can get access to physical memory and therefore the computer. [4]

Flip Feng Shui

Razavi et al. presented a Rowhammer security issue which breaks RSA encryption (Section 2.6.4) for a flipped public key. Their research shows that it is possible to calculate a private key for a public RSA key with a flipped one bit. They were able to successfully attack 84.1% of their victims. Within 50 hours they were able to factorize more than 80% of 4096 bit keys.

2.5.3 Solutions

Until now there are no really reasonable solution [1]. Main issue of Rowhammer attacks is the high speed of modern computers combined with the small architecture. Kim et al. suggests six potential solution [8]:

1. make better chips
2. correct Errors (ECC: Section 2.5.3)
3. refresh all rows frequently
4. Retire cells (manufacturer)
5. Retire cells (end-user)
6. Identify "hot" rows and refresh neighbors

There are several software based solutions which could detect a malicious program using the following methods: [1]

1. static code analysis
2. performance counters
3. memory access patterns
4. physical proximity
5. memory footprint

However, Gruss et al. defeated all known software Rowhammer defenses therefore there is no effective software solution [1].

ECC

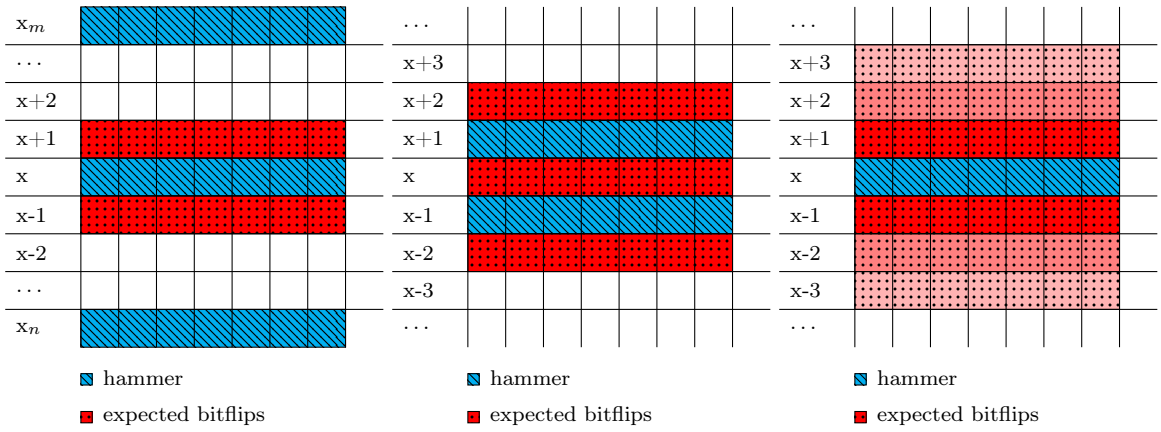
Error-correcting code memory (ECC) is a special type of memory which is able to detect and correct memory errors. ECC are doing a good mitigation by correcting single-bit and detecting double-bit errors. However, whenever more than three error bits occur, the system will not be able to correct or detected the bitflips. However, more than two bitflips are not very likely [3, 8].

2.5.4 Rowhammer Strategies

At the moment, there exist mainly three strategies to cause bitflips:

1. *Single-sided hammering*: The first Rowhammer attacks started with single-sided hammering. The malicious binary hammers eight different addresses. Therefore there is a chance of 61.4% to hammer two or more rows in the same bank. The hammered rows in the target bank are not necessarily neighbors of the target row. This strategy is easy because no DRAM reverse engineering is needed. Unfortunately it is less effective and fewer systems are affected. Figure 2.1a illustrates the principle of single-sided hammering [8].

2. *Double-sided hammering*: Pessl et al. developed methods to reverse engineer the DRAM addressing function. This helped to introduce double-sided hammering. As shown in Figure 2.1b, doublesided hammering tries to create electromagnetic effects by alternating accessing row *x-1* and *x+1* in order to change an adjacent row. Especially on DDR4 memory modules the addressing function helped to improve the number of bitflip [2].
3. *One-location hammering*: as shown in Figure 2.1c the attacker hammers just a single location in the memory and keeps this row open. Therefore, there are less electromagnetic coupling effects what results in less bitflips [1].



(a) Single-sided hammering (b) Double-sided hammering (c) One-location hammering

Figure 2.1: Common Rowhammer Strategies: dashed rows indicate memory cells accessed with maximum frequency. In dotted rows the attacker expects bitflips.

2.6 Network

A computer network is a large distributed system with many computer which can to talk to each other. Modern technologies like cloud systems or webservices use network connections in order to preform their tasks. Worldwide many networks are connected together, this is called the Internet. This section describes some basic things about computer networks and how they work.

2.6.1 Network Stack

The network stack is the mechanism how the Linux kernel processes arriving network packages. Due to different drivers, this varies for each network card but the following will give a short overview: [36]

1. At boottime the network driver gets loaded and initialized
2. The network interface controller (NIC) copies every arriving package to a ring buffer and a hardware interrupt informs the system about the new package.
3. Every CPU thread runs a ksoftirqd process. A poll loop gets started by a ksoftirqd process which loads packets from the ring buffer

4. The polled data gets passed to the next protocol layer using the skb-datastructure
5. In every layer the system performs security checks for every package. For example, the check of the checksum. Finally, a package gets passed up to the last protocol layer which merges the received packages and makes them accessible to applications.

For this process, the content of a package does not matter. Whenever the NIC receives a package which destination MAC address is the MAC address of the NIC, the NIC starts to process the package and passes it up the network stack.

2.6.2 OSI-Model

OSI-Model stands for Open Systems Interconnection Model and is a guideline how to set up network connections. It has seven layers[1]: [37, 38]

1. Physical Layer - physical transportation (cable,...) of bits
2. Data Link Layer - the way data (packages) gets linked and received form the actual hardware by using a Media Access Control (MAC) Address
3. Network Layer - helps to route packages Internet-Protocol(IP) Messages
4. Transport Layer - used to define the actual transport system to avoid bottlenecks (UDP-Protocol)
5. Session Layer - handles the communication between two systems.
6. Presentation Layer - defines how data are presented (e.g. ASCII encoding)
7. Application Layer - The final application often uses a customized protocol on the top of others.

Every transmission uses all levels packed on top of each other. For example, the Physical Layer contains information of the Data Link Layer which contains information about the Network Layer and so on. In this project, mainly Layer 1 to 4 (transport orientated layers) are interesting. The route of one package is as follows: We are flooding our victim with UDP - package, these get addressed to a certain IP-Address. The network routing devices route these packages to the right target device. Finally, the NIC recognizes its packages due to the MAC address and passes them up the network stack. [39, 37, 38]

2.6.3 UDP

UDP is the abbreviation for User Datagram Protocol which is a minimal network protocol to transmit data (Transport Layer). In contrast to the Transmission Control Protocol (TCP), UDP is a connectionless communication protocol. It has nothing like a handshake or other communication mechanisms. UDP packages are *unreliable*, *unordered* and *lightweight*. There is no congestion control or other principles to mind transmission problems.[40, 41]

A UDP package is build up as follows: [40]

1. 16-bit source port
2. 16-bit destination port

[1] Protocol examples taken from computer communication

3. 16-bit length
4. 16-bit checksum

and gets packed into an IP package.

In general, UDP is used when speed is important, there is no need to check or correct errors and when the application does not have to guarantee the transmission of information. Due to this minimality, UDP is very fast what makes it perfect for our attack.

2.6.4 Encryption - RSA

Many applications use RSA-encryption to hide data from other people. RSA is an asymmetric cryptographic technique to encrypt data. The encryption is using the public key which can be known by everyone. The public key is a multiplication of two big prime numbers. [42]

$$\text{Public key} = p * q \tag{2.2}$$

Messages get decrypted using the private key which should just be known by the valid receiver. The private key gets computed with the two prime numbers p and q.[42]

This means theoretically everybody could calculate the private key out of factorizing the public key but since p and q are very big primes it is not possible to find out these numbers in a reasonable time. [42]

2.6.5 Denial-of-Service

Denial-of-Service (DoS) attacks are truly common [9]. Often they work in form of a Distributed-Denial-of-Service attack. A system gets paralyzed due to overloading it. The network and its transport system cannot handle all requests anymore and randomly drop packages. There are several methods to defend against these attacks but attackers modify their tools and continuously bypass these security mechanisms [9].

2.7 Debugging

Debugging is the processes of finding software bugs in a program. There are many Linux tools like GDB or ftrace helping programmers to find different kind of mistakes.

2.7.1 GNU Debugger - (GDB)

GDB is a simple powerful commandline debugging tool to debug software written in C or C++. It is free, protected by the GNU Public License (GPL). Programs compiled with debug information can be executed with the gdb. It provides many different commands. For example, stepping through instruction for instruction or set breakpoints at code addresses. It can display the current values of variables, memory locations or CPU registers [43].

Another powerful tool is debugging with GDB using its Python API or the GDB server which provides opportunities to debug software which is running on remote host such as a VM [43].

2.7.2 Function Tracer

The Function Tracer (ftrace) is a tool to trace function calls. It is build directly into the Linux kernel and helps to see what is happening in the kernel. [44]

To use ftrace the kernel has to be compiled with the following kernel configuration parameters set to enabled: [44]

```
CONFIG_FUNCTION_TRACER
CONFIG_FUNCTION_GRAPH_TRACER
CONFIG_STACK_TRACER
CONFIG_DYNAMIC_FTRACE
```

Now we can trace for example all the calls and its parameters of any arbitrary function without any modification of the kernel.

The output is stored to a ring buffer. We can access this information by reading the file */sys/kernel/debug/tracing/trace*. It shows the TASK-PID, a timestamp, the function and more.

3 Design

This chapter shows a detailed plan of the attack, how we are going to mount the attack and possible threats to the victim. We present a full proof-of-concept attack and ideas for modifications.

3.1 Attack Overview

This section is a summary of the whole attack idea. It will describe all steps and handle all requirements to perform a successful attack.

In the first task we need to determine the IP address of our target computer. The next task aims to find out whether the required system specifications are fulfilled:

1. we need a *fast internet connection* to the target server to be able to send enough UDP-packages to cause a bitflip. In order to perform a successful attack, the network stack of the victim needs to process 500 Mbit/s.
2. the target needs a CPU with *Intel CAT* to have a way to evict the cache. Especially virtualised server systems use this type of CPU

The third task depends on the attackers goal. We need to keep the data which our attack should modify inside the memory to have the chance to modify it with Rowhammer. This could happen by requesting a lot of data which is publicly available.

Afterwards, we need to find a method to evict addresses from the cache without having access to the CPU. Aga et al. describe two ways to do this. If our victim VM is owned by a cloud provider with a low quality of service, the provider might limit the available amout of LLC what makes the attack possible. If this is not the case, we could initiate a denial-of-service attack to a different VM of the guest server to reduce the ways of LLC of our victim [5]. However, this might limit our uplink and therefore avoid the real attack.

After successfully completing all these tasks we can start flooding the server with millions of packages per second. A simple UDP-flooder should be enough, the uplink of most internet connections might be the bottleneck of this operation. Multiple flooding distributed computer can solve this problem.

Every single package will be processed by the server and therefore hammer the DRAM. Since Rowhammer attacks exploit DRAM hardware errors there is no deterministic attack process. However, hammering a certain physical address of DIMM mostly results in the same bitflips on one DIMM. On a different DIMM it leads to other bitflips for the same physical address.

Finally, we can start some of our UDP flooder. After about 350ms or less we should be able to observe a bitflip.

3.2 Case Study

Renting a VM server to use it as a webservice is quite common. In this example scenario, we assume a target server with a highspeed internet connection, provided by a cheap service provider who limits the last level of cache by default. Moreover, the target server is vulnerable to one-location hammering. The attacker has a small set of attacking computers. Combined they have a fast uplink to flood the victim with simple network packages. The victim is running an instance of a GitLab server. This GitLab server has thousands of members, all listed publicly.

First we are going to find out the IP-address of the server and list all users using the GitLab API: [45]

```
GET /users
```

This GET-request returns a JSON-encoded message with an array of all users. We can iterate over all users, take their id and use a provided call to fetch all public keys of a user: [45]

```
GET /users/:id/keys
```

We are going to store all these public keys on our local computer. Afterwards, still a lot of public keys should be in the memory, for now it is time to start the actual attack.

Next we (the attacker) own enough computers to send about 600Mbit of UDP packages to the victim in order to hammer in the DRAM. Every package received from the NIC gets processed and passed up till the OS recognize that this package is not wanted and gets thrown away. Due to processing every single package, the CPU needs to access memory addresses. These memory addresses are mostly code memory but also some data pages. The good thing about code pages is that the code does not change its memory position. Therefore, we can abuse frequently accessed code memory as one-location hammer addresses. Whenever we send a package to the server, we are going to hammer some memory. After several packages a bitflip will occur and the attack might have been successful. Optionally this bitflip has to occur within seconds to bypass a potential firewall which might protect the servers from denial-of-service attacks [9].

Whenever we think that a bitflip happened, we iterate over all users again to get their public key. We compare the current public keys with the keys stored on our disk. If the attack was successful, we get a difference. Now the system is going to use the modified public key to encrypt data. We take the different key and try to reconstruct the private key. Normally it is not possible to get the private key out of a public key due to the two big prime numbers which got multiplicated. Since a bitflip changed our current public key it is not a multiplication out of two prime numbers anymore [6]. Therefore, we can factorize the public key way faster and consequently crack the encryption or masquerade as the owner of this key [6]. We are having full access to the user's git projects. We can clone repositories, add new commits and push

them to the server. Moreover, we can easily inject some malware to existing projects and take over other computers because the user might not notice the attack and execute the trusted modified code.

3.3 Threat Model

In general, memory corruption brings up a lot of different threats. Our research shows that Rowhammer attacks are deterministic for a specific physical address in the DRAM. Therefore, the result of the attack is always the same. However, we need to keep in mind that the attacker does not know the content of the physical memory which prevents the attacker from knowing the outcome of the attack.

3.3.1 Denial-of-Service

The Nethammer might end in an DoS caused by a flipped bit, for example flipping a bit in a code page. However, the Nethammer does not know where the bitflip occurs but there might be two different situations:

1. Bitflip in the kernel space. Very likely this will result in a kernel error (kernel panic). A kernel module or the whole kernel will stop working. Depending on the crashed part just a single driver might fail or the whole system will be unresponsive.
2. Another outcome might be the violation of userspace memory. Related to DoS this violation might cause a segmentation fault and the OS will kill the program. This service is not reachable anymore.

DoS is a big thing. Based on other purposes DoS attacks should make damage to the victim. This can be a financial damage but also black mailing or revenge [9]. Therefore, this new type of attack leads to serious new problems.

3.3.2 Public key Bitflips

Bitflips can be used to crack RSA encryption [6]. Whenever our attack causes a bitflip within a public key, probably the new public key will not be a multiplication out of two prime numbers anymore [6]. Therefore, factorizing becomes easier and might be solved within hours with lots of computing resources [6]. Now the attacker has the same privileges and opportunities as the valid owner of the key. However, this change is temporary. Therefore, reloading the public key to the memory might lock out the attacker again.

3.3.3 Inode flipping

Many server systems have automated backup systems. These backups might become important due to this special threat. The Nethammer might flip bits in an Inode. This could cause persistent loss of data. As described in Section 2.1, an Inode is part of the filesystem data structure. It contains information like the position of files on the hard-drive-disk (HDD). A bitflip could change the position variable of an Inode. Whenever the system wants to read from or write to this file it will not access the right valid data. A write operation might overwrite other existing data and therefore connote the loss of data.

Concluding, flipping an Inode does not happen often but it is still a dangerous problem which can be undone by restoring backups.

4 Implementation

This chapter provides information to find vulnerable addresses and the needed tools to find them. Rowhammer tests showed that our test system has bitflips whenever we can do two accesses within 300ms. Knowing this fact, we tested the Linux kernel and tracked accessed addresses. With this information we were able to successfully reproduce bitflips over the network without any hardware or software modification. The detailed outcome is described in Chapter 5.

4.1 Debugging

This section describes our debugging tools. We developed plugins for existing debuggers and reused Linux tools to get knowledge about the accessed memory for each processed package.

4.1.1 MemAccess

MemAccess (Memory Access) is a script to trace the accessed memory locations. It uses Python and GDB which attaches itself to a GDB server running on localhost.

MemAccess takes configurable parameters: a start/stop address or line and the path to the binary. We start memaccess using the following command:

```
gdb -q -x memaccess.py
```

It is recording all memory accesses between the start and stop parameters.

For code pages, we use the Linux tool *addr2line* which returns the line for a certain code-address in a binary.

The output of MemAccess is providing important information such as *accessed addresses*, *number of accesses*, the *function name* and more.

4.1.2 Dmesg Using Serial Port

Dmesg stands for "display message" and is a terminal command to print the ring buffer of kernel messages. It helps to determine what faults happened during runtime or boot. Due to the nethammer attack, often the graphical user interface (GUI) of the victim stuck, resulting in no information about what went wrong. We used a feature of the Linux kernel to receive the ring buffer over serial connection. With this method all kernel panics and logs were still accessible.

4.1.3 Perf-Tools

Perf-Tools is a collection of performance analysis tools for Linux. It uses the core Linux tracing tools like ftrace and perf_events to observe performance critical Linux parts [46].

For this project we used the *funccount* script which is part of the Perf-Tools framework. The bash scrip *funccount* wraps the standard ftrace functionalities. After starting *funccount* as super user it counts function calls where the function name matches a user defined pattern. Combined with a package sender this is a easy and fast way to find frequently called functions.

```
sudo ./funccount -t 10 "*udp*" &
./udp-flooder <num_of_packages>
```

This listing counts the functions calls for 10 seconds. The functionname has to contain *udp*.

FUNC	*COUNT*
udp4_gro_receive	*3760878*
udp4_lib_lookup_skb	*3760878*
__udp4_lib_rcv	*3760878*
udp_error	*3760878*
udp_gro_receive	*3760878*
udp_packet	*3760878*
udp_rcv	*3760878*
udp_v4_early_demux	*3760878*
udp_get_timeouts	*3760888*
udp_pkt_to_tuple	*3760890*
__udp4_lib_lookup	*7521756*

Table 4.1: Truncated output of *sudo ./funccount "*udp*"*

Figure 4.1 shows an example output when flooding the computer with UDP packages.

4.2 FileScan

FileScan is a C program to spot and verify bitflips in userspace memory. FileScan uses the function *mmap* to map the data of a file, bigger than the physical memory, into virtual memory. This data is generated randomly but it is still recognizable. All pages get accessed once to allocate the physical memory. Finally, the program uses a function called *mincore* to determine the memory status of all page. We need this function because some pages might have been swapped out due to exceeding the physical memory capacity. Using this information the binary only accesses pages which are still in the memory. Without using *mincore* we might overwrite potential bitflips because pages might get swapped in. Now the whole memory should be used by the kernel or the FileScan application.

The actual job of FileScan is done in an endless loop which scans all the pages in the physical memory. The data is random but still recognizable. Therefore, it can detect bitflips. FileScan corrects these errors to observe the frequency. Finally, it logs all errors to a file to reuse this data for statistics.

4.3 MemoryScan

Higher DRAM refresh rates lead to more bitflips and therefore a higher chance of crashing the kernel. A kernel panic often results in a system freeze. However, we wanted to find out whether there occurred bitflips in the remaining memory. Therefore, we needed to scan the other memory for bitflips. This scan got triggered by a kernel panic and logs all messages to the kernel log. Using dmesg (Section 4.1.2) we could still read the log containing messages about bitflips to reconstruct what went wrong.

MemoryScan is a kernel module and therefore does not get swapped (Linux does not swap kernel processes). Compared to the FileScan application there are some small differences:

1. the memory is allocated with *vmalloc*
2. it is a kernel module and therefore running in kernel space
3. it does not check continuously, a scan gets triggered by reading the */proc/memscan* pseudofile or whenever a kernel panic occurs.

Kernel modules do not get swapped out, therefore the MemoryScan module cannot allocate all the memory. This results in the risk of missing bitflips because some physical memory is not scanned.

4.4 Nethammer Kernel Module

The Nethammer Kernel Module was used at the beginning of our research to find out the requested frequency of arriving packages to flip bits. It is a Rowhammer program translated into a kernel module. It allocates 90% of the RAM and then hammers a randomly picked address. It uses the kernel function *dev_add_pack* to add a protocol handler to the networking stack. This handler gets called for every network package processed on the network stack.

The Nethammer kernel module hammers a specific number of times per package. After a predefined number of accesses it scans for bitflips. We tried one-location hammering and double-sided hammering. To increase the chance of double-sided bitflips we used the DRAM addressing function [2].

Out of this information we could conclude that six accesses are necessary to cause a bitflip with one-location hammering.

4.5 Simulating Intel CAT

The target of our exploit is a VM with Intel CAT. Our test system did not have a CPU which supports Intel CAT technology. Therefore, we had to find an other method to simulate the cache eviction of Intel CAT.

First we were trying to disable the cache using the CR0 register of the CPU. Bit 30 of the CR0 register is used to disable the CPU cache. We wrote a kernel module to set this bit to 1. Unfortunately, the system speed decreased dramatically to a speed were no bitflip occurred anymore.

We had to modify the kernel at one position to simulate cache eviction:

Filepath:
net/netfilter/core.c
Function:
*int nf_hook_slow(struct sk_buff *skb, struct nf_hook_state *state, struct *nf_hook_entry entry)*

```
348   static size_t count = 0;
349   count++;
350   if(count % 10000000 == 0)
351   {
352     trace_printk("AccessCount: %ld\n", count);
353   }
354   size_t rip;
355   asm("leaq (%%rip), %0;": "=r"(rip));
356   asm volatile("clflush (%0)" : : "r"((void*)rip) : "memory");
```

Line 348-353 is just a counter to measure the effective number of accesses. Line 355 is an inline assembler instruction which saves the Instruction Pointer (RIP) to the variable *rip* defined in Line 354. Finally, in Line 356, we flush the address of *rip* out of the cache to make the attack possible.

The instruction pointer is a register of the CPU always showing to the address of the next instruction. Therefore, after flushing this address, the next time the CPU executes this instruction it will hammer once. The procedure of flushing the cache using *clflush* should not be necessary and is not possible in a real attack over network. Therefore, we use the possibility to evict cached memory with the Intel CAT.

4.6 UDP Flooding

Finally, we used a simple UDP flooder to attack our test system. The UDP flooder is written in C and uses the standard POSIX commands to send a UDP package.

Every single package gets processed by the network stack. In our testing setup (Section 5.1), every package hammered six times while being processed. This means we send as many packages as fast as possible to hammer with the highest frequency. Therefore, we used the minimal package content of one character (1 byte) which was always the same. Our measurements showed about 460Mbit/s uplink on a local network and even 2Gbit/s uplink on the loopback interface. 460Mbit/s with a package size of 43 byte (Ethernet header (14 byte) + IP header (20 byte) + UDP header (8 byte) + one character payload (1 byte)) are equivalent to about 1.350.000 packages per second.

As an even faster alternative, we tried to write our own implementation of UDP package sending using a *rowsocket*. This method should reduce the overhead of calculating the UDP checksum and therefore send more packages within the same time. Unfortunately, our measurements showed the same effective average uplink to the target computer.

5 Evaluation

In this section we give a detailed overview of our results. We will discuss the used environment hardware and software and finally the test results.

5.1 Test Environment

This is a small overview about the used hardware and software. Due to the widespread Rowhammer vulnerable DRAM DIMMs other hardware and software might work too. The most important thing about our setup is the fact that we did not have a CPU with Intel CAT therefore this is just a proof of concept where the cache eviction is based on the hardcoded *clflush* instruction.

5.1.1 Hardware

The used hardware of the testing computer is shown in Table 5.1.

Hardware	*Model*
CPU	*Intel® Core™ i7-6700K CPU 4x@ 4.0*
DRAM	*2x8GB DDR4 Crucial @ 2133MHz*
Refresh Rate	*8000 (default)*
Mainboard	*ASUSTeK COMPUTER INC. Z170-WS*
Network Card	*Intel Ethernet Controller 10G X550T*

Table 5.1: Hardware setup of the test environment

This hardware is known to be vulnerable to Rowhammer attacks. For statistical data the test environment BIOS settings were set to default. In general, we did not modify the hardware configuration at all.

5.1.2 Software

The victim was running a standard Linux 4.12.2 x86 64-bit kernel which was built with a config file based on the old config file of the running kernel. The current attack uses a function within the netfilter framework. Therefore, we had to make sure that some options were set the following way[1]:

[1] Other configurations might work too.

```
CONFIG_IP_NF_IPTABLES=y
CONFIG_IP_NF_FILTER=y
CONFIG_IP_NF_TARGET_REJECT=y
CONFIG_IP_NF_NAT=m
CONFIG_IP_NF_TARGET_MASQUERADE=m
CONFIG_IP_NF_MANGLE=y
# CONFIG_IP_NF_RAW is not set
```

5.2 Bitflips

This section is about the frequency and number of bitflips on the test system. Using this numbers we calculate the theoretical limits of the Nethammer.

5.2.1 Double-sided hammering

At the beginning we tried to find a chance to perform double-sided hammering. We used the feature of MemAccess to find frequently accessed memory locations with the same page offset to hammer adjacent locations. Unfortunately, this method showed that there are only few accessed addresses with the same page-offset. Therefore, the chance to get two adjacent physical addresses with the same offset is rather small. Moreover, the access frequency might be too slow.

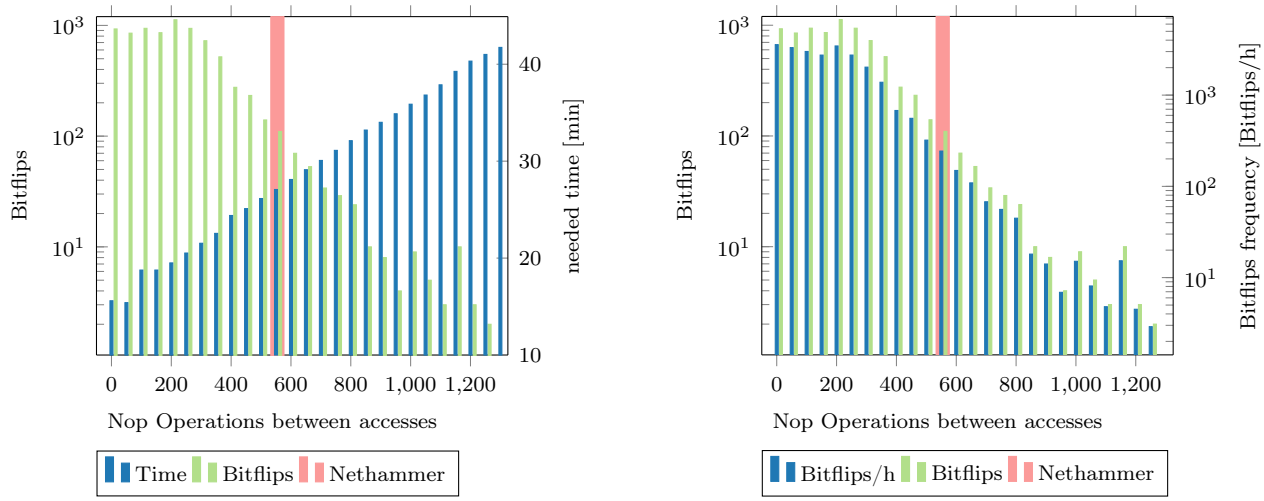
5.2.2 One-location hammering

Rowhammer test showed that one-location hammering causes bitflips on our test system. To find out the minimal access frequency we started some timing tests.

Figure 5.1 shows the impact of the number of no-operation CPU instructions (nops) between the memory accesses and the number of bitflips that occurred. Additionally, we measured the total time taken by ten million accesses to 20 different addresses to see the linear growing time consumption (Figure 5.1b).

The test set for a specific number of nops was as follows:

1. start the time measure
2. we pick 20 random addresses out of 10 GB allocated memory.
3. now each of this addresses gets accessed 10.000.000 in a row
4. between two accesses there is a loop executing the number of nops
5. after accessing an address 10.000.000, times we check for bitflips
6. the procedure from 2-5 gets executed 25 times
7. stop the time measure
8. finally, all test results are summed up.



(a) Bitflips and the needed Time

(b) Bitflips and their average frequency

Figure 5.1: The number of bitflip/frequency for 25 tests: each test is hammering 20 different addresses where every address gets accessed for 10.000.000 times in a row. The required time and bitflips refer to the sum over all 25 tests. The Nethammer bar indicates the area where we are hammering using UDP packages.

Out of this tests, we calculated the average bitflip frequency, shown in Figure 5.1b.

For every test set we execute:

$$10.000.000\frac{\text{accesses}}{\text{address}} \cdot 20\frac{\text{addresses}}{\text{test}} \cdot 25\frac{\text{tests}}{\text{testset}} = 5.000.000.000\frac{\text{accesses}}{\text{testset}} \tag{5.1}$$

Out of the time consumption bars we can see: the test took us about 30min with 700 nops between two accesses. For 950 nops, the test took 35min. Assuming linear growth, we can say 50 nops between two accesses are equal to 1 minute in a whole test set. Therefore:

$$5.000.000.000\frac{\text{accesses}}{\text{testset}} \cdot 50\frac{\text{nops}}{\text{access}} = 250.000.000.000\frac{\text{nops}}{\text{testset}} \equiv 1\frac{\text{min}}{\text{testset}} \tag{5.2}$$

1min has $60 \cdot 10^9$ ns, which gives us the final result:

$$25 \cdot 10^{10}\frac{\text{nops}}{\text{testset}} \equiv 6 \cdot 10^{10}\frac{\text{ns}}{\text{testset}} \tag{5.3}$$

$$1 \text{ nop} = \frac{6}{25} \text{ ns} \tag{5.4}$$

$$1 \text{ ns} \approx 4 \text{ nops} \tag{5.5}$$

Now we are going to apply this result to the knowledge of the maximum number of nops between two accesses. Whenever there are more than 1250 nops between two accesses, bitflips do not occur anymore.

$$1250 \text{ nops} \cdot \frac{6}{25}\frac{\text{ns}}{\text{nop}} = 300 \text{ ns} = \frac{3}{10}\mu\text{s} \tag{5.6}$$

Another important property is described by how many accesses are necessary to trigger a bitflip. This might be important for example to bypass a firewall. As Figure 5.2 shows, we need at least 300.000 accesses to an address to trigger a bitflip.

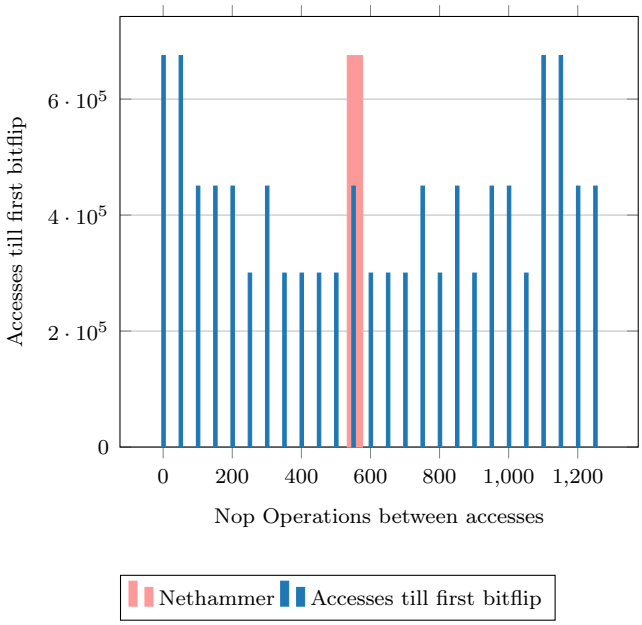


Figure 5.2: Minimal access count until the first bitflip occurs.

Moreover, we need more accesses when we are accessing with less than 250 nops between two accesses. This might be because the row of the DRAM is not closed between these accesses.

Figure 5.3 shows the correlation between bitflips and the duration for accessing 20 random addresses 10.000.000 times.

5.3 Network

For testing purposes, we setup a local area network connected to the internet. As shown in Figure 5.4 the victim and two attacking computers got attached to the same 1Git-Switch.

The victim kernel was modified (Section 5.1.2) with one *clflush* instruction to simulate a VM using a CPU supporting Intel CAT.

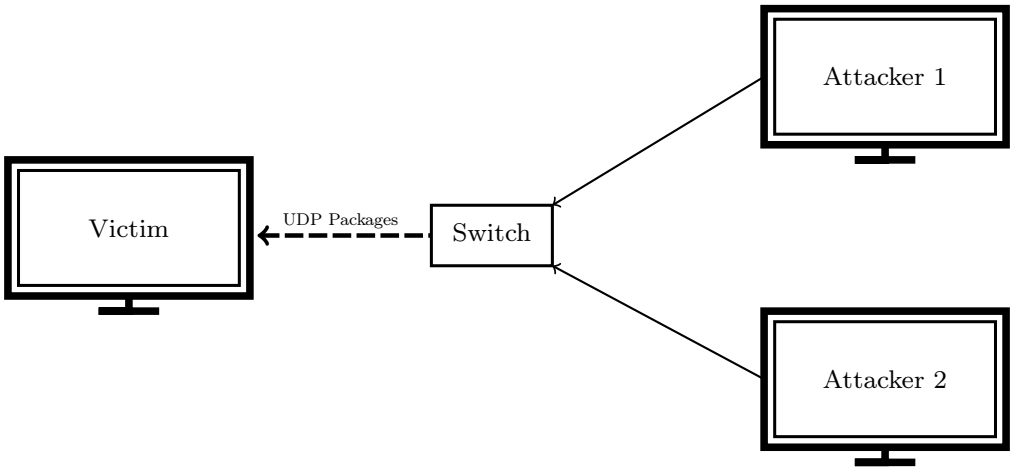


Figure 5.4: Nethammer test environment. One or two attacking computers were able to cause bitflips in the victims DRAM

Due to hardware specific reasons, Attacker 1 was able to send about 500Mbit/s and Attacker 2 could just send 100Mbit/s. Together they reached 600Mbit/s. Due to causalities the victim NIC had to try to process about 460Mbit/s. One package has 43 bytes (344bit).

$$\frac{460 \cdot 10^6 \frac{\text{bit}}{\text{s}}}{344 \frac{\text{bit}}{\text{Packages}}} \approx 1.350.000 \frac{\text{Packages}}{\text{s}} \tag{5.7}$$

In our victim kernel, every package was hammering 6 times.

$$1.350.000 \frac{\text{Packages}}{\text{s}} \cdot 6 \frac{\text{Accesses}}{\text{Packages}} = 8.100.000 \frac{\text{Accesses}}{\text{s}} \tag{5.8}$$

This result was cross-checked by incrementing a counter in the modified kernel and print it every 10.000.000th access with *trace_printk*.

Using the result of 5.4:

$$\frac{\frac{25}{6} \frac{\text{nop}}{\text{ns}}}{\frac{8.100.000}{10^9} \frac{\text{Accesses}}{\text{ns}}} \approx 515 \frac{\text{nop}}{\text{Access}} \tag{5.9}$$

Looking at Figure 5.1 this should be enough to cause bitflips. According to Figure 5.2 we should require less than 450.000 accesses until the first bitflip occurs.

$$\frac{450.000\text{Accesses}}{8.100.000 \frac{\text{Accesses}}{\text{s}}} = \frac{1}{3}\text{s} \rightarrow \text{theoretically 1 Bitflip in} \leq 333\text{ms} \tag{5.10}$$

The running FileScan program (Section 4.2) found a bitflip every searching cycle. Scanning the whole memory of the test system took about 5 seconds. To find the bitflip frequency we scanned only memory locations where we expected bitflips. Therefore, the program found a bitflip every 350ms, proving result 5.10. This would be enough to bypass a firewall which might protect the potential victim of a DDoS attack [9].

5.4 Insight

We have been able to trigger bitflips just by sending network packages. These bitflip occur every 350ms which is really fast. Unfortunately, these bitflips are not deterministic for unknown hardware. Therefore, we cannot say how an attack might end. Overall, we have found mainly five different kind of bitflips:

1. *Inode bitflip:* After some minutes of an apparently unsuccessful attack, we rebooted the computer to place the kernel on other physical addresses due to address space layout randomization. However, we have not been able to boot anymore. After backing up the old kernel and replacing it with a new unmodified, we have been able to boot again. To find the result of the crash, we compared the old modified kernel and the new one. There were many binary differences so we conclude to flipping an Inode. A program wanted to modify its own file but the Inode showed to the kernel image and therefore destroyed it.
2. *Library bitflip:* This bug showed as follows: whenever we wanted to start a simple flawless binary it crashed at a specific position. We have not been able to recompile it until rebooting the system.
3. *Password bitflip:* The system was still running without any problems. Network connection was working and no service crashed. However no user was able to login. Remote connection software like SSH showed "Permission denied" and after three tries the connection got closed.
4. *Kernelspace bitflip:* these bitflips were observed using *dmesg* and the serial connection. Concluding we can say that usually the operating system freeze and we could not interact with it anymore. Sometimes, just a single module crashed and therefore we sill could use the computer, but a certain functionality such as the mouse or the keyboard was not working anymore.
5. *Userspace bitflip:* Basically we could observe two different behaviors. Either the process crashed (Segmentation fault) or we modified data of the process which were processed later on. This might be a major security issue, for example, flipping public keys of applications as described in Section 3.2.

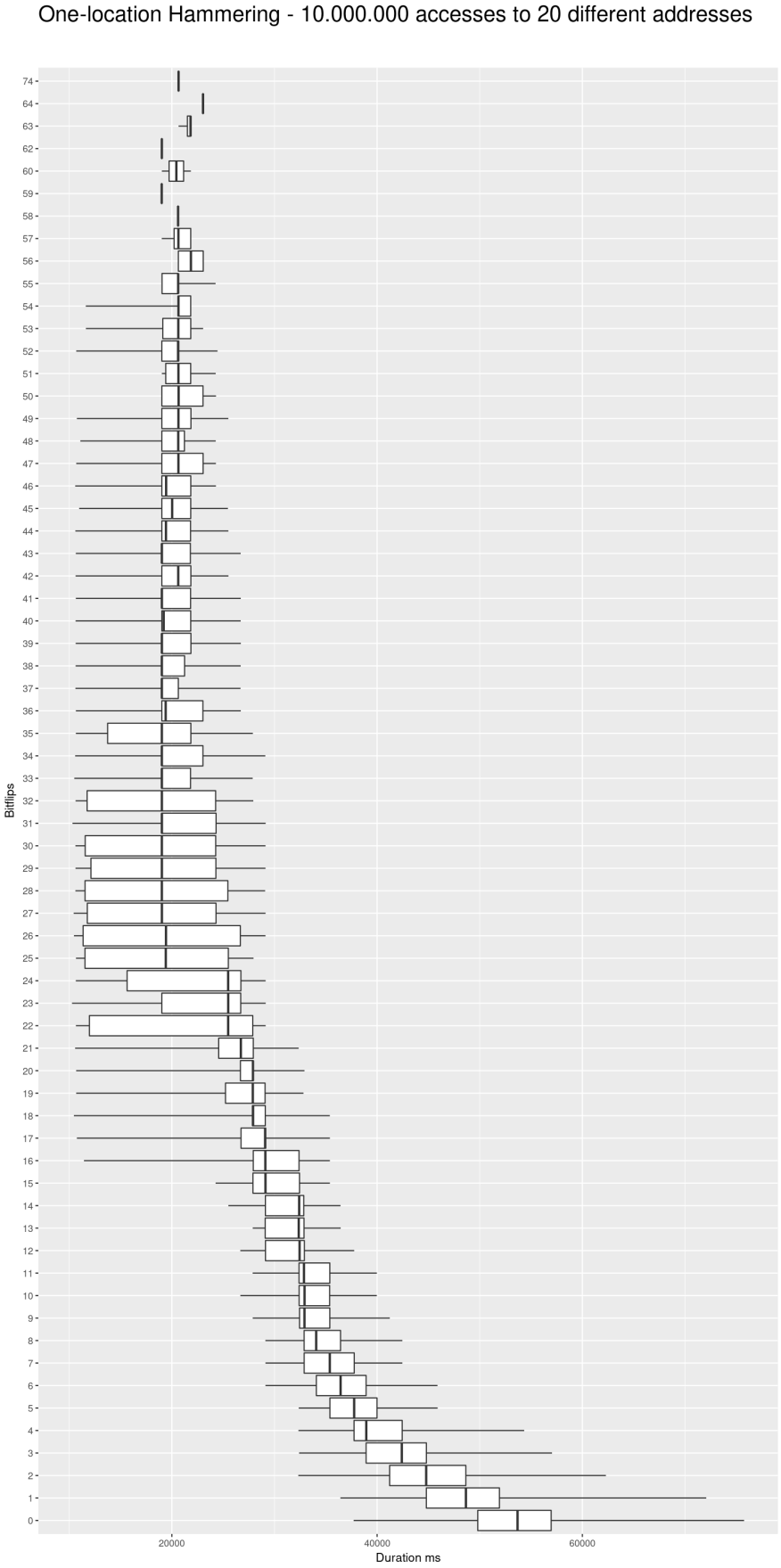


Figure 5.3: Bitflips and the require time for accessing 20 random address 10.000.000 times.

6 Conclusion

We presented a method for one-location hammering over network just by flooding the target with millions of senseless UDP-packages. Based on some semi-automatic code analysis program, we found multiple vulnerable addresses to exploit code accesses of the network traffic and proved that the speed of two simple UDP-flooder is enough to cause bitflips in under 350ms. These bitflips can harm the victim in various ways. Unfortunately, we could not observe any form of deterministic bitflips because they are hardware dependent. This makes this the type of attack sneaky. There are many different ways an attack could end. Summarized, we observed six different types of faults :

1. Denial of service due to a bitflip in the kernel binary the OS crashed. The server was unreachable until a reboot.
2. Killing a kernel module for example just the keyboard is not working anymore but the server is still reachable.
3. In consequence of flipping an Inode the victim was unable to boot. The OS overwrote some parts of the kernel and other files.
4. A bitflip occurred in the userspace, creating a new large threat. The attacker needs to be lucky but there is a chance to change public keys or to kill an application.
5. Flipping information of the Linux password file makes it impossible to login. The server is still working as usually.
6. Destroying library files. This kills many userspace processes whenever they are calling the library function.

In general, bitflip changes are temporary. After a reboot the system works again as expected. The only exception is mentioned in point 3. Whenever the system writes memory back to the disk, bitflip changes might become persistent.

Limitation: Unfortunately there are some limitations of this attack. First, the target needs to be a virtual server running using a CPU supporting Intel CAT. This is required to have an opportunity to evict addresses from the cache without having access to the *clflush* CPU instruction. Second, the used memory has to be vulnerable to one-location hammering bitflips. Third, the attack is not deterministic, meaning it is not sure how the attack will end.

Outlook: at the moment we modified the kernel with a *clflush* CPU instruction. Since the final target is a VM which runs on a CPU using Intel CAT, the attacker could evict address without having access. As other research shows, the attack should still be possible and even more effective than using *clflush* [5].

Another idea would be to execute the GitLab-attack described in Section 3.2. In practice big servers like github.com or gitlab.com have many public users. Therefore, the attacker could load many public keys to the memory. If these servers use a CPU supporting Intel CAT and the memory is vulnerable to one-location bitflips an end-to-end attack should be possible.

Bibliography

(1) Gruss, D.; Lipp, M.; Schwarz, M.; Genkin, D.; Juffinger, J.; O'Connell, S.; Schoechl, W.; Yarom, Y. *CoRR* **2017**, *abs/1710.00551* (cit. on pp. 1, 3, 8, 9, 11, 12).

(2) Pessl, P.; Gruss, D.; Maurice, C.; Mangard, S. *CoRR* **2015**, *abs/1511.08756* (cit. on pp. 1, 3, 8, 9, 12, 23).

(3) Seaborn, M.; Dullien, T. *Black Hat* **2015** (cit. on pp. 1, 8–11).

(4) Gruss, D.; Maurice, C.; Mangard, S. In *Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: 2016, pp 300–321 (cit. on pp. 1, 8–10).

(5) Aga, M. T.; Aweke, Z. B.; Austin, T. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, 2017, pp 8–13 (cit. on pp. 1, 2, 7–9, 16, 32).

(6) Razavi, K.; Gras, B.; Bosman, E.; Preneel, B.; Giuffrida, C.; Bos, H. In *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association: Austin, TX, 2016, pp 1–18 (cit. on pp. 1, 8–10, 17, 18).

(7) Van der Veen, V.; Fratantonio, Y.; Lindorfer, M.; Gruss, D.; Maurice, C.; Vigna, G.; Bos, H.; Razavi, K.; Giuffrida, C. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp 1675–1689 (cit. on pp. 1, 8, 10).

(8) Kim, Y.; Daly, R.; Kim, J.; Fallin, C.; Lee, J. H.; Lee, D.; Wilkerson, C.; Lai, K.; Mutlu, O. In *ACM SIGARCH Computer Architecture News*, 2014; Vol. 42, pp 361–372 (cit. on pp. 1, 8, 9, 11).

(9) Mirkovic, J.; Reiher, P. *SIGCOMM Comput. Commun. Rev.* **2004**, *34*, 39–53 (cit. on pp. 2, 14, 17, 18, 29).

(10) Palmer, M.; Walters, M., *Guide to Operating Systems*, 4th ed., 2012, p 720 (cit. on p. 4).

(11) Rusling, D. The Linux Tutorial - The place where you learn linux. `http://www.linux-tutorial.info/` (accessed 10/27/2017) (cit. on pp. 4, 5).

(12) NETMARKETSHARE Desktop Operating System Market Share. `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=0` (accessed 10/27/2017) (cit. on p. 4).

(13) NETMARKETSHARE Mobile/Tablet Operating System Market Share. `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1` (accessed 10/27/2017) (cit. on pp. 4, 10).

(14) operating-system.org Basic knowledge about Operating Systems. `http://www.operating-system.org/betriebssystem/_english/w-wissen.htm` (accessed 10/27/2017) (cit. on p. 4).

(15) Droidwiki Android Kernel. https://www.droidwiki.org/wiki/Kernel (accessed 10/27/2017) (cit. on p. 5).

(16) kernel.org Linux Kernel. https://www.kernel.org/ (cit. on pp. 5, 6).

(17) kernel.org Ext4 Howto. https://ext4.wiki.kernel.org/index.php/Ext4_Howto (accessed 10/28/2017) (cit. on p. 5).

(18) nixCraft Understanding UNIX / Linux filesystem Inodes. https://www.cyberciti.biz/tips/understanding-unixlinux-filesystem-inodes.html (accessed 10/28/2017) (cit. on p. 5).

(19) Torvalds, L. Translating Addresses in Kernel Space. http://www.tldp.org/LDP/khg/HyperNews/get/devices/addrxlate.html (cit. on p. 6).

(20) Jones, M. User space memory access from the Linux kernel. https://www.ibm.com/developerworks/library/l-kernel-memory-access/index.html (cit. on p. 6).

(21) Hat, R. What is Swap Space? https://www.centos.org/docs/5/html/5.2/Deployment_Guide/s1-swap-what-is.html (accessed 10/27/2017) (cit. on p. 6).

(22) kernel.org Page Table Management. https://www.kernel.org/doc/gorman/html/understand/understand006.html (accessed 10/29/2017) (cit. on pp. 6, 10).

(23) Corbet, J. Memory protection keys. https://lwn.net/Articles/643797/ (accessed 10/27/2017) (cit. on p. 6).

(24) TechTarget Server Virtualization Definitions. http://searchservervirtualization.techtarget.com/definitions (accessed 10/28/2017) (cit. on pp. 6, 7).

(25) Bigelow, S. J.; Haletky, E. L. Virtualization Explained: Definitions You Need to Know. http://cdn.ttgtmedia.com/searchServerVirtualization/downloads/Virtualization_Explained.pdf (cit. on pp. 6, 7).

(26) Irazoqui, G.; Inci, M. S.; Eisenbarth, T.; Sunar, B. In *International Workshop on Recent Advances in Intrusion Detection*, 2014, pp 299–319 (cit. on p. 7).

(27) Zhang, S. *Whitepaper, provided by Kansas State University, TechRepublic/US2012, available online: http://www. techrepublic. com/resourcelibrary/whitepapers/deep-diving-into-an-easilyoverlooked-threat-inter-vm-attacks* **2013** (cit. on p. 7).

(28) Nguyen, K. T. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology (accessed 10/28/2017) (cit. on p. 7).

(29) Halderman, J. A.; Schoen, S. D.; Heninger, N.; Clarkson, W.; Paul, W.; Calandrino, J. A.; Feldman, A. J.; Appelbaum, J.; Felten, E. W. *Communications of the ACM* **2009**, *52*, 91–98 (cit. on p. 8).

(30) Elektronik-Kompendium DDR-SDRAM - Double Data Rate SDRAM. https://www.elektronik-kompendium.de/sites/com/1312291.htm (accessed 10/28/2017) (cit. on p. 8).

(31) Matas, B. DDR4 Set to Account for Largest Share of DRAM Market by Architecture. http://www.icinsights.com/news/bulletins/DDR4-Set-To-Account-For-Largest-Share-Of-DRAM-Market-By-Architecture/ (accessed 10/28/2017) (cit. on p. 8).

(32) Fussell, D. S. Locality and Caching. www.cs.utexas.edu/users/fussell/courses/cs429h/lectures/Lecture_18-429h.pdf (cit. on p. 9).

(33) Boeck, H. Angriff der Bitschubser. http://www.zeit.de/digital/datenschutz/2015-03/rowhammer-angriff-pc-speicher-linux (cit. on p. 9).

(34) Govindavajhala, S.; Appel, A. W. In *2003 Symposium on Security and Privacy, 2003.* 2003, pp 154–165 (cit. on p. 9).

(35) Lipp, M.; Gruss, D.; Spreitzer, R.; Maurice, C.; Mangard, S. In *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association: Austin, TX, 2016, pp 549–564 (cit. on p. 10).

(36) Research, P. Linux networking stack from the ground up. (accessed 10/31/2017) (cit. on p. 12).

(37) Day, J. D.; Zimmermann, H. *Proceedings of the IEEE* **1983**, *71*, 1334–1340 (cit. on p. 13).

(38) Zimmermann, H. *IEEE Transactions on Communications* **1980**, *28*, 425–432 (cit. on p. 13).

(39) Krieger, U. Multimedia-Kommunikation in Hochgeschwindigkeitsnetzen. https://www.uni-bamberg.de/fileadmin/uni/fakultaeten/wiai_lehrstuehle/informatik_ktr/Dateien/MMK-SS08/MMKHGN08-Kap01-Einfuehrung-V040408.pdf (accessed 11/01/2017) (cit. on p. 13).

(40) Postel, J. *User datagram protocol*; tech. rep.; 1980 (cit. on p. 13).

(41) Handley, M.; Floyd, S.; Padhye, J.; Widmer, J. *TCP friendly rate control (TFRC): Protocol specification*; tech. rep.; 2002 (cit. on p. 13).

(42) Zhou, X.; Tang, X. In *Proceedings of 2011 6th International Forum on Strategic Technology*, 2011; Vol. 2, pp 1118–1121 (cit. on p. 14).

(43) Stallman, R. M.; Pesch, R. H., *Debugging with Gdb: The Gnu Source-level Debugger Fifth Edition, for Gdb Version, April 1998*; iUniverse Com: 2000 (cit. on pp. 14, 15).

(44) Rostedt, S. Debugging the kernel using Ftrace. https://lwn.net/Articles/365835/ (accessed 11/02/2017) (cit. on p. 15).

(45) Documentation, G. GitLab - Users API. https://docs.gitlab.com/ce/api/users.html (accessed 11/01/2017) (cit. on p. 17).

(46) Gregg, B. Performance analysis tools based on Linux perf_events (aka perf) and ftrace. https://github.com/brendangregg/perf-tools (accessed 10/26/2017) (cit. on p. 21).