HAMMERKIT

Toolset for Memory inspection and automatic Rowhammer detection

MARTIN HECKEL

JANUARY 21, 2021

Martin Heckel: Hammerkit, Toolset for Memory inspection and automatic Rowhammer detection
Academic supervisor: Prof. Dr. Florian Adamsky

Abstract

Kernel Same-page Merging (KSM) deduplicates pages in physical memory by changing the mapping from virtual to physical pages. This brings the benefit that pages with the same content are stored only once. KSM is often used in virtualised environments because there are many duplicates of pages and thereby, lots of memory can be saved.

To inspect this behaviour for security evaluation it is required to analyze the mapping. The Linux kernel provides a user space interface at `/proc/⟨PID⟩/pagemap`. However, there is no such interface for the address translation from inside a Virtual Machine (VM). In this thesis, I will describe the implementation of a kernel module and some user space software to access this address mapping.

Several attack vectors are a combination of multiple techniques. Flip Feng Shui (FFS) [1], a paper published by VUSec [2] uses KSM and the rowhammer bug [3] to change arbitrary bits in arbitrary memory pages.

Since most of the rowhammer Proofs of Concept (PoCs) require manual adjustments and are rather unstable, there is need for an all-in-one tool to check if a system is vulnerable to rowhammer. To simplify the testing, I've created more tools to automatically test some existing PoCs as well.

This practical thesis shows the internal functionality of the tools for memory inspection and rowhammer testing. Additionally, a practical rowhammer exploit will be described.

Zusammenfassung

Kernel Same-page Merging (KSM) dedupliziert Seiten im Arbeitsspeicher durch Verändern der Umsetzung von virtuellen zu physischen Speicherseiten. Dadurch werden Seiten mit identischen Inhalt nur einmal physisch gespeichert, was den Speicherverbrauch eines Systems senkt. KSM wird häufig in virtualisierten Umgebungen verwendet, da in solchen Umgebungen viele duplizierte Speicherseiten vorkommen und entsprechend viel Speicher eingespart werden kann.

Es ist notwendig, das Mapping von virtuellen zu physischen Seiten zu analysieren, um das Verhalten von KSM untersuchen zu können. Der Linux Kernel stellt Dateien unter `/proc/⟨PID⟩/pagemap` zur Verfügung, um dieses Mapping aus dem user-space abfragen zu können. Es gibt allerdings keine Möglichkeit, das Mapping aus einer Virtual Machine (VM) heraus zu betrachten. In dieser Praxisarbeit wird die Implementierung eines Kernelmoduls sowie einiger Software im user space erklärt, die einen Zugriff auf dies Mapping aus einer VM heraus ermöglichen.

Einige Angriffsvektoren bestehen aus einer Kombination mehrerer Techniken. Flip Feng Shui (FFS) [1] ist ein solcher Angriffsvektor, der auf KSM und Rowhammer [3] basiert und es ermöglicht, beliebige Bits einer beliebigen Speicherseite zu verändern.

Da viele der veröffentlichen Proofs of Concept (PoCs) für Rowhammer manuelle Anpassungen

benötigen und nicht besonders stabil laufen, besteht der Bedarf einer Software, die ein System auf Anfälligkeit für Rowhammer testen kann und dabei möglichst stabil und einfach zu bedienen ist. Um das Testen zu vereinfachen, habe ich weitere Tools zum automatischen Ausführen einiger vorhandener PoCs erstellt.

In dieser Praxisarbeit wird die interne Funktionalität der Software zum Untersuchen von Speicherstellen und zum automatisierten Testen auf Anfälligkeit für Rowhammer vorgestellt. Zusätzlich wird die praktische Durchführung eines Rowhammer-Angriffs beschrieben.

We don't demand solid facts!
What we demand is a total absence of solid facts.
I demand that I may or may not be Vroomfondel!

— Vroomfondel [4]

Acknowledgements

During the development of the tools and the writing of this practical thesis I received a lot of support and assistance.

When writing this thesis there were multiple situations in which I got strange results and was not able to explain them. Especially – but not only – in these situations my supervisors, Prof. Dr. Florian Adamsky and Adrian Maertins supported me. Until now, I was able to solve all of these problems with the help of their feedback. Many of the ideas used in the thesis came around in meetings with them. At this point, I want to thank my supervisors for the great support.

After I started to test if my systems were affectable by the rowhammer vulnerability, I did not find any bit flips at first. Because of this, I asked two of the research teams that had already published rowhammer-related papers. I want to thank the research groups `IAIK` [5] and `VUSec` [2], especially Ass.Prof. Dr. Daniel Gruss, for their support at solving that problem.

I want to thank my supervisors as well as Nico Bretschneider, Johanna Heckel and Michelle Madeline Krebs for proof reading this thesis.

Contents

List of Figures

List of Tables

Listings

Abbreviations

BIOS	Basic Input/Output System
BKDG	BIOS and Kernel Developer's Guide
CAS	Column Access Strobe
CPU	Central Processing Unit
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random-Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
FFS	Flip Feng Shui
GFN	Guest Frame Number
GPA	Guest Physical Address
GVA	Guest Virtual Address
HPA	Host Physical Address
HVA	Host Virtual Address
IC	Integrated Circuit
KSM	Kernel Same-page Merging
KVM	Kernel-based Virtual Machine
MMU	Memory Management Unit
OS	Operating System
PFN	Page Frame Number
PoC	Proof of Concept
QEMU	Quick EMUlator
RAM	Random-Access Memory
RAS	Row Access Strobe
SOIC8	Small Outline Integrated Circuit
SPI	Serial Peripheral Interface
SPD	Serial Presence Detect
THP	Transparent Hugepage
UEFI	Unified Extensible Firmware Interface
VM	Virtual Machine

1 Introduction

Moore's law states that the number of transistors in Integrated Circuits (ICs) nearly doubles every two years. This integration density is one of the big limiting factors when ICs are developed. A higher density of transistors makes it possible to build more complex ICs. In the last decades, complex ICs were highly optimized for their use cases. For example, Central Processing Units (CPUs) got more cores and memory chips were built with more capacity. In general, hardware components are optimized to provide the maximal possible performance while behaving like specified.

However, nobody seemed to test what happens when hardware components are used in the specified range but not as intended by the designers. Due to that, many hardware bugs were found in the last years, such as Meltdown [6], Spectre [7], and Plundervolt [8]. Also, a memory level hardware bug called rowhammer was found back in 2014 [3]. Rowhammer gives an attacker the possibility to flip bits in memory (change a "1" to a "0" and vice versa) by frequently reading other data in system memory.

In 2016 Flip Feng Shui (FFS) [1], a new rowhammer-related attack vector based on memory massaging was published. This approach gives the possibility to flip bits on chosen Dynamic Random-Access Memory (DRAM) pages leading to a powerful attack. However, the attack presented in the paper only allows to flip one chosen bit per page which reduces the impact of it significantly.

1.1 Objective of this Thesis

The thesis is split into the practical thesis (this document) and the bachelor thesis [9].

I've developed some tools to perform tests for the rowhammer and FFS vulnerabilities. In this practical thesis, the general approaches and usage of most of these tools will be explained. This thesis will also describe the steps that are necessary to do a practical exploitation of the rowhammer vulnerability.

1.2 Structure of this Thesis

In Section 2, some basic background topics are explained. The Sections 3 and 4 describe the tools I've developed and the necessary steps to get bit flips on a system. Afterwards Section 5 with related projects and papers follows. Section 6 concludes the results of this thesis.

If not noted otherwise, the figures, listings and table in this thesis were created by myself.

2 Background

This section provides the required information to understand the rest of this thesis. In the scope of this thesis, GNU/Linux with Kernel version 5.9.14 was used as Operating System (OS) and Kernel-based Virtual Machine (KVM) as hypervisor. The used computers have the x86–64 architecture.

2.1 Basic Functionality of DRAM

In DRAM, data is stored in memory cells consisting of capacitors and transistors as shown in Figure 1. The capacitor of a cell is accessed using the gate of the corresponding transistor (pin labeled 2 in Figure 1). When the capacitor is accessed, the state can be read at pin 1. It is also possible to set a new state by setting pin 1 to the corresponding state and accessing the capacitor using pin 2.

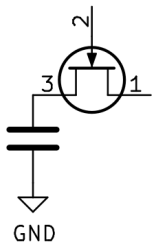


Figure 1: Simplified schematic of a single memory cell used in DRAM.

Because reading the state of a cell is destructive [10], the state of a cell has to be written back after it was accessed. This can be achieved by writing the state into a buffer and setting the state of the cell to the state of the buffer afterwards.

The capacitors in a DRAM cell loose charge over time. They have to be refreshed periodically to avoid data loss. This can be done by reading and restoring the cell. Normally, this happens every 64 ms [11].

Multiple of these DRAM cells are organized in an array with rows and columns as shown in Figure 2. All gates (pin labeled 2) of the cells in the same row are connected to each other, so it is only possible to access all cells in a row at the same time. The accessed row is determined by the selected pin of `Row select`. When a row is accessed, the state of the cells in this row can be read or set at the pins denoted with `Data`.

2.2 Architecture of System DRAM

On current systems, DRAM is often organized in Dual In-line Memory Modules (DIMMs) that can be plugged into slots. On some systems, DRAM is soldered to the main board instead. The following description of the general architecture of DRAM applies to systems with DIMMs and one CPU.

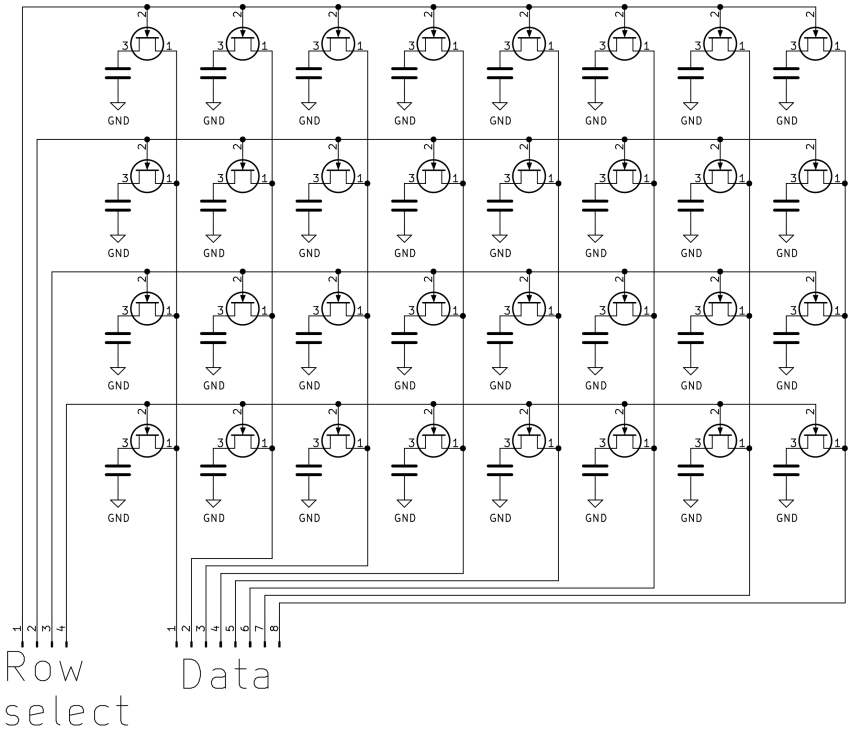


Figure 2: Array of DRAM cells with 4 rows and 8 memory cells per row.

The system DRAM shown in Figure 3a can consist of one or multiple channels [10], [12]. In Figure 3b, a system with two channels is shown. Each channel is a separate bus between the CPU and the DIMMs of the channel. So, DIMMs in different channels can be accessed parallel because they are at different busses.

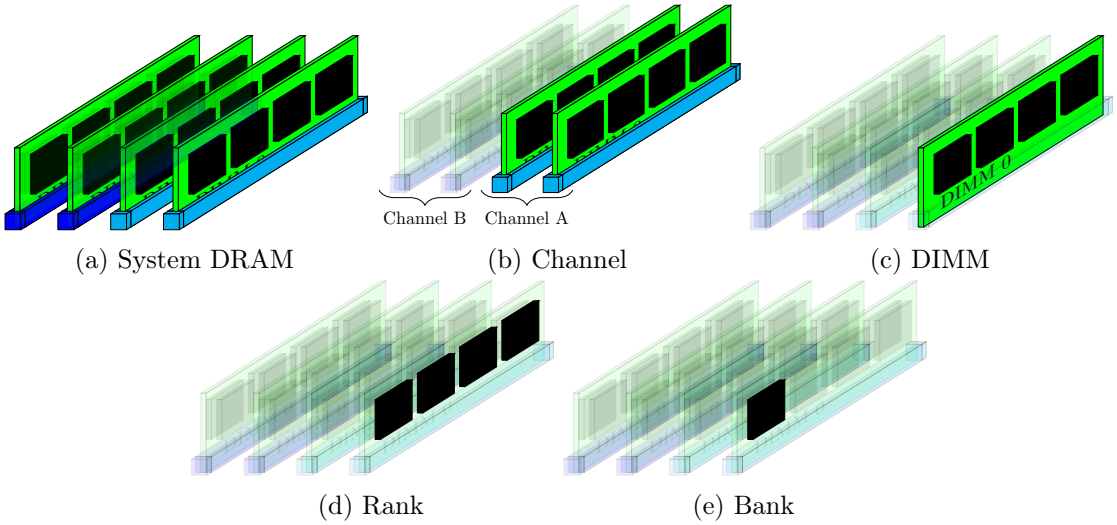
On each channel, there can be multiple DIMMs as shown in Figure 3c. One DIMM has several DRAM ICs called banks. In Figure 3e, one bank is shown. These banks can be logically organized in ranks, e.g. the front and back side of a DIMM. Figure 3d highlights one rank.

As shown in Figure 2, each bank consists of an array of DRAM cells. Additionally, there is a row buffer [10], [12] on each bank. The array of cells is organized in columns. One column contains as many bits as the bus width. On a system with 64 bit memory channels, one column has a size of 64 bit. So, columns are rather a logical than a physical structure. The internal structure of a bank is shown in Figure 4.

Because the capacitors in the DRAM array loose charge, the charge of the accessed capacitors has to be amplified to store the correct state in the corresponding memory cell in the row buffer. This is done by amplification circuits between the data lines and the row buffer.

On the systems I used, one row has a size of 8 KiB. This means, one row of 8 KiB is loaded into the row buffer and the entire row buffer is written back to the corresponding row later.

In Linux, memory is managed in blocks called pages. Each page has a size of 4 KiB, so one row can store two of those pages.



(a) System DRAM (b) Channel (c) DIMM

(d) Rank (e) Bank

Figure 3: General DRAM architecture of a dual-channel system with two DIMMs per channel and two ranks on each DIMM.

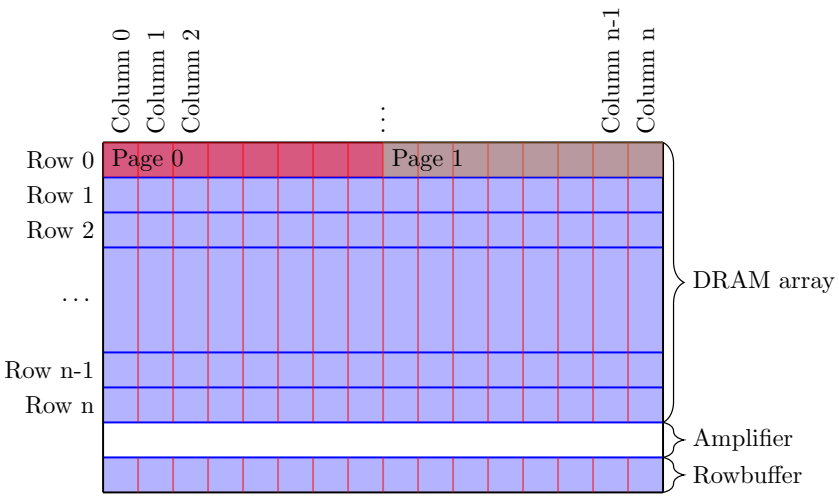


Figure 4: Basic internal architecture of a DRAM bank

In DRAM, only entire rows can be refreshed based on the design. The rows are not refreshed one by one but in 8192 batches [13]. On a system with 32 768 rows, each batch would consist of 4 rows. Because each row has to be refreshed every 64 ms and there are 8192 batches, one batch is refreshed every $\frac{64\,\text{ms}}{8192} \approx 7.8\,\mu\text{s}$.

2.3 Row Buffer

DRAM rows are accessed in a buffered way which means an accessed row has to be loaded into the row buffer first. The data in the row buffer can be accessed by the CPU afterwards. When the access is done, the row buffer has to be written back to the DRAM array at some point of time to restore the data that were destroyed when the row was loaded into the buffer.

In order to optimize the performance of DRAM, the buffer is usually written back when another

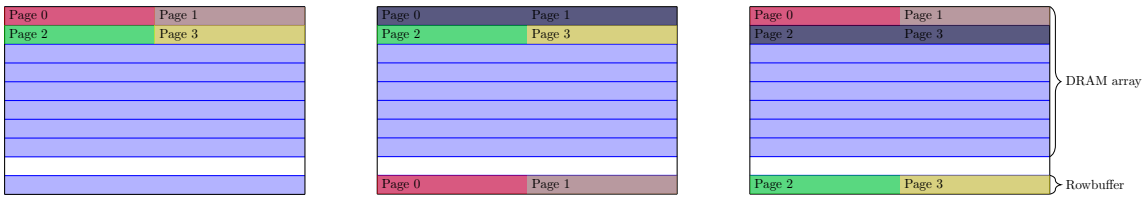
row is accessed. This brings the benefit that another access to the same row can be handled pretty fast because the row does not have to be loaded into the buffer and written back to the DRAM array afterwards.

In general, there are three possibilities when a row is accessed in respect to the row buffer. Figure 5 shows the different cases when the first row (containing pages 0 and 1) should be accessed. Note that the content of the DRAM row that is in the row buffer was destructed when it was read.

The row buffer can be empty (Figure 5a). In this case, the row that should be loaded to the row buffer has to be activated. It takes t_{RCD} (Row Access Strobe (RAS) to Column Access Strobe (CAS) Delay) to load the row into the row buffer. Then, the content of the row can be accessed from the row buffer. This access takes t_{CL} (CAS Latency). So, accessing a row when the row buffer is empty takes a total time of $t_{RCD} + t_{CL}$.

There is also the possibility that the row that should be accessed is already in the row buffer (Figure 5b). In this case, the data can be served directly from the row buffer and no accesses to the DRAM array are required. This is called `row hit`. It takes t_{CL} to access the data from the row buffer.

The third possibility is that there is another row than the requested one in the row buffer (Figure 5c). In this case, the row that is currently in the row buffer has to be written back to the DRAM array first which takes t_{RP} (Row Precharge time). Then, the new row has to be activated and loaded to the row buffer (t_{RCD}). Next, the data can be accessed (t_{CL}). This is called `row conflict` and takes a total time of $t_{RP} + t_{RCD} + t_{CL}$.



(a) empty row (b) `row hit` (c) `row conflict`

Figure 5: Memory access situations in respect to the content of the row buffer. The dark blue memory areas were destroyed while reading and have to be written back before another memory area can be accessed.

The systems I used have the following timings specified in there Serial Presence Detect (SPD) records [14], [15]: $t_{RP} = t_{RCD} = t_{CL} = 13.125\,\text{ns}$. So, a `row conflict` ($t_{RP} + t_{RCD} + t_{CL}$) takes three times as long as a `row hit` (t_{CL}).

2.4 Rowhammer

The rowhammer bug happens when rows are accessed (read, and thereby loaded to and restored from the row buffer) at high frequencies due to physical effects leading to bits flipping in adjacent rows.

The accessed rows are called `aggressor rows` and the rows that are likely to have bit flips are called `victim rows`.

Two memory rows are physically adjacent when they are next to each other on the same bank, which means they are on the same channel, the same DIMM, the same rank and the same bank. This is described in detail in Section 3.4.

Bit flips can only occur between the refreshes of the rows because the capacitors of the cells are completely reloaded at a refresh. One early mitigation for rowhammer was to double the refresh rate, so a cell was refreshed after 32 ms instead of 64 ms [3]. With a doubled refresh rate, one refresh happens every $\frac{32\,\text{ms}}{8192} \approx 3.9\,\mu\text{s}$.

2.5 Virtual Addressing

On most OS, processes do not work directly with physical addresses but with virtual addresses. The Memory Management Unit (MMU) maps the virtual addresses from processes to physical addresses using page tables [16, p. 37 – 50]. These physical addresses are mapped to actual memory locations by the MMU [12].

Most of the recent systems have the MMU included in the CPU. For Intel CPUs, the mapping function between physical addresses and memory locations is not published. For AMD CPUs, the mappings are documented in the BIOS and Kernel Developer's Guide (BKDG) of the corresponding microarchitecture. There is no BKDG for modern AMD systems (starting with microarchitecture `17h`) anymore [17]. Until now, AMD did not reply to a request about the exact mapping behaviour on their more recent systems.

Usually, the mapping from virtual to physical addresses is not done for single addresses but for blocks of addresses called "pages". A Physical address can be interpreted as concatenation of a page number and an offset in the page referenced by the page number [16, p. 20] (Figure 6).

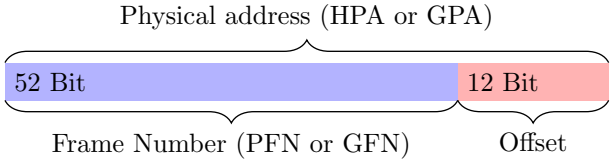


Figure 6: General composition of physical addresses

Assumed, the Host Virtual Address (HVA) that should be mapped is `0x7f77b8581337`. Then, the part of the HVA that is mapped to a Page Frame Number (PFN) is `0x7f77b8581`. Assumed,

the PFN would be `0x20c1e2`, the physical address of the original HVA (`0x7f77b8581337`) is `0x20c1e2337`.

When running a virtualised infrastructure, the guest operating system in the Virtual Machine (VM) has virtualised physical addresses. This results in more address translations: Guest Virtual Address (GVA) to Guest Physical Address (GPA) to HVA to Host Physical Address (HPA) to physical location (Figure 7).

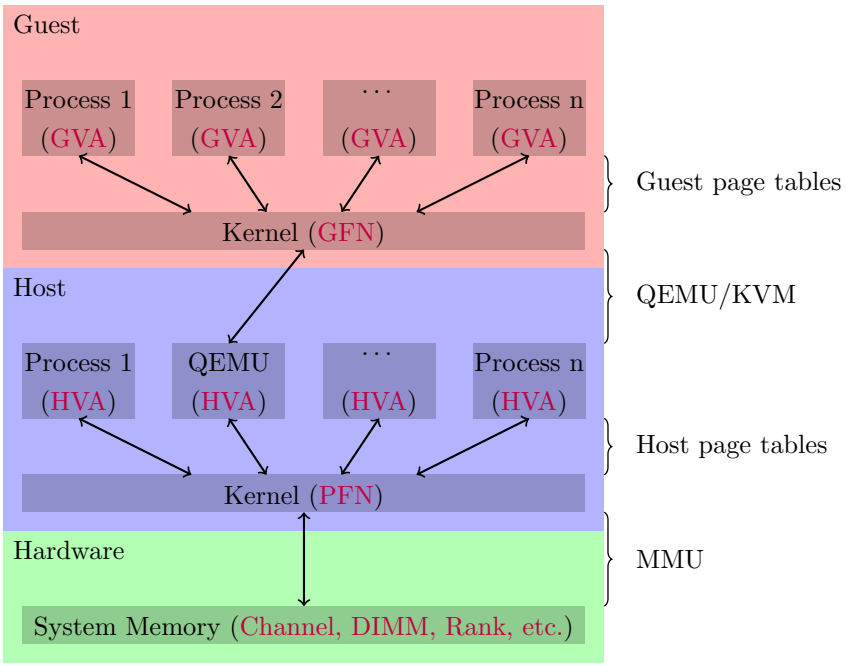


Figure 7: Different levels of address translations in a virtualised environment

The page number of the GPA is called Guest Frame Number (GFN) and the page number of the HPA is called PFN.

2.6 KSM

Kernel Same-page Merging (KSM) is a component of the Linux kernel which detects duplicate pages in physical memory and merges them. When this is done, multiple virtual pages are mapped to the same physical page and thereby, only one physical page has to be stored in memory. This is in particular useful in virtualised environments with multiple VMs with the same operating system because there are many pages that can be deduplicated. Red Hat tested a setup with 52 VMs running Windows XP with 1 GiB of virtual memory each on a host with only 16 GiB memory [18].

3 Tools to support FFS research

FFS depends on two concepts: KSM and rowhammer. I have developed some components to be able to inspect memory mappings and test for bit flips induced by rowhammer in the scope of this practical thesis. Figure 8 gives an overview of the software components described in this section.

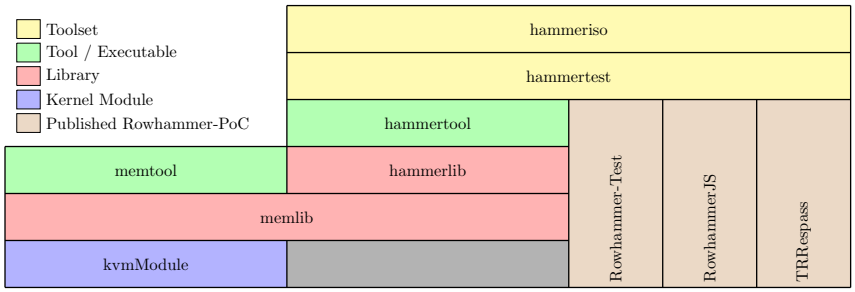


Figure 8: Overview of the software and tool sets created in the scope of this thesis

In the following parts, the concepts behind the different components and the usage of them will be explained.

Note that the components not drawn in brown in Figure 8: `kvmModule` (Section 3.1), `memlib` (Section 3.2), `memtool` (Section 3.3), `hammerlib` (Section 3.4), `hammertool` (Section 3.5), `hammertest` (Section 3.6) and `hammeriso` (Section 3.7) were developed in the scope of this thesis. The other components: `Rowhammer-Test` [19], `RowhammerJS` [20] and `TRRespass` [21] are publicly available Proofs of Concept (PoCs) for the rowhammer vulnerability and were not developed by me.

The left part of the tool set shown in Figure 8 (`kvmModule`, `memlib` and `memtool`) can be used to inspect the multi-stage memory mappings from GVA to HPA. Thereby it is possible to find out which pages are merged by KSM and which physical address is used for the merged page.

The right part (`memlib`, `hammerlib`, `hammertool`, `hammertest` and `hammeriso`) can be used to automatically test a system for the rowhammer vulnerability.

3.1 Kernel Module kvmModule

In a virtual environment, there are multiple layers of address translation: GVA to GFN is done using the page tables of the guest operating system, GFN to HVA is carried out by the Quick EMUlator (QEMU) process of the guest VM and the KVM kernel component running on the host. The translation of HVA to PFN is done using the page tables of the host operating system.

After this, the physical location of the data in memory is calculated by the MMU based on the PFN.

We assume a process running on a KVM-based VM. That process works with GVAs which are accessed by the process. In C, these addresses are equal to pointers.

The guest kernel translates GVAs to GFNs through page tables [16, p. 37 – 50]. This mapping

can be read from user space using the file `/proc/⟨PID⟩/pagemap`. This file is part of the proc file system (procfs) [22] provided by the kernel. To get the GFN of a known GVA, the corresponding pagemap file can be read at the offset of the GVA. Practically, this results in a call of `lseek` [23] to specify the offset required and a call of `read` [24] to read the data into a buffer.

Because the guest operating system has no information that it is virtualised, it is impossible to get the other mappings from inside the guest operating system directly. This problem is solved with a kernel module on the host system which creates two files in the procfs that behave like the pagemap files. There is one file to return the HVA for a specified GFN and another file to return the PFN for a specified GFN.

3.1.1 Concepts

Kernel modules can be used to extend the kernel functionality by running additional code inside the kernel space without the necessity to modify and rebuild the kernel itself. A kernel module consists at least of two functions: one is called when the module is loaded, another one is called when the module is unloaded [25].

The functions of a kernel module are called in an asynchronous way (like the calls to a library). For this reason, the module does not implement a complete control flow but rather single functions that can be called. The control flow has to be implemented by the program using the functions of the kernel module. In the perspective of the module it is not guaranteed that the functions are called in a specific order or even from the same process.

Normally, the communication between user space programs and the kernel space is done with system calls. The definition of own system calls requires a rebuild of the kernel [26]. There are other ways to communicate with kernel modules as well. For `kvmModule`, I decided to create some files in the proc filesystem for communication with user space processes.

The proc filesystem is a virtual file system which means there are no files stored physically (e.g. on a disk, USB thumb drive, etc.). The execution of an operation on such a file (e.g. `open`, `read`, `write`, etc.) calls a function with the corresponding parameters.

When a file is created in procfs, these callback functions have to be specified. This is done with the `struct proc_ops` shown in Listing 1.

```
1  struct proc_ops {
2      unsigned int proc_flags;
3      int (*proc_open)(struct inode *, struct file *);
4      ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
5      ssize_t (*proc_write)(struct file *, const char __user *, size_t,
           loff_t *);
6      loff_t  (*proc_lseek)(struct file *, loff_t, int);
7      int (*proc_release)(struct inode *, struct file *);
8      __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
9      long    (*proc_ioctl)(struct file *, unsigned int, unsigned long);
10 #ifdef CONFIG_COMPAT
```

```
11      long    (*proc_compat_ioctl)(struct file *, unsigned int, unsigned
            long);
12  #endif
13      int (*proc_mmap)(struct file *, struct vm_area_struct *);
14      unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long,
            unsigned long, unsigned long, unsigned long);
15  } __randomize_layout;
```

Listing 1: `struct proc_ops` from `include/linux/proc_fs.h`

After the procfs file is created using `proc_create_data`, the kernel calls the corresponding functions specified in the `struct proc_ops` when the file is accessed.

When, for example, the file is opened, the `proc_open` function specified in the according `struct proc_ops` is called.

When a kernel module creates multiple files with the same callback functions assigned, it is required to know which file was accessed from inside the callback function. Because this information is not submitted as parameter when the specified functions are called, it has to be accessed otherwise. This is done using the `private_data` field in the `struct file` submitted to each of the callback functions.

If `proc_create_data` is used rather than `proc_create`, the `i_private` field of the `struct inode` for the created file is set to a value specified when calling the function. Usually, a pointer to a data structure is used for this. When the file is opened, the `private_data` field of the `struct file` can be set to the `i_private` value of the inode.

The KVM kernel component provides some information of running VMs in the kernel debugfs [27]. Basically, this works just like the procfs described above. For KVM, there is `struct kvm` that stores information about a VM. The `i_private` field of the debugfs files is set to a pointer to a data structure containing a pointer to such a `struct kvm`.

Basically, the kernel module is running in three stages: Initialization, Runtime and Cleanup. The module interactions in the first two stages are shown in Figure 9. Because the last stage does only free the allocated data structures and remove created files, it is omitted in the figure.

When the module is loaded, it scans for running VMs in the KVM debugfs. For each VM, a new folder is created in the procfs. In each folder, one file is created for the mapping between GFN and HVA and one for the mapping between GFN and PFN.

To get the PFN from the HVA, the function `follow_page` from the Linux kernel can be used. However, this function is not exported so it cannot be called from a kernel module when running a vanilla kernel. To solve that problem, the module reads the content of the file `/proc/kallsyms` when it is loaded. This file contains the addresses of the kernel symbols. The content of the file is parsed to get the address of the `follow_page` function. Then, this address is cast to a function pointer and can be called to get the PFN from the HVA.

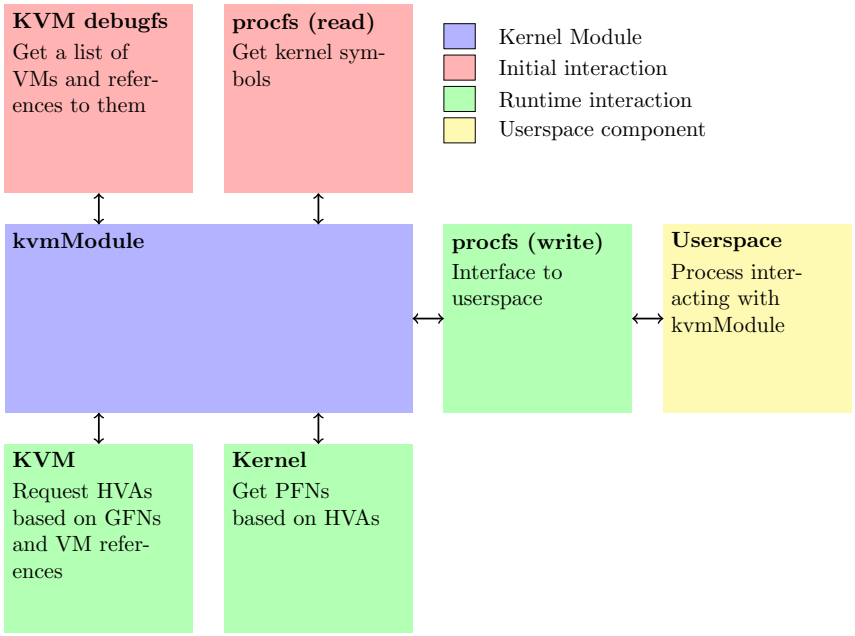


Figure 9: Interactions of `kvmModule` with kernel space components

For each of those files, a `kvmModuleFileInfo` data structure (Listing 2) is created using `kmalloc`. This is necessary because the data structure should be accessed from other functions as well so it has to be stored outside the scope of the functions. The `i_private` field of the `struct inode` is set to the address of the `kvmModuleFileInfo` data structure. When a function is called, it can parse the `private_data` field of the submitted `struct file`. This field was set when the file was opened. Then, the information of this specific virtual file can be accessed and modified.

```
1  typedef struct {
2      off_t offset;
3      int count;
4      int pid;
5      int fd;
6      struct kvm *kvm;
7      struct proc_dir_entry *parent;
8      struct proc_dir_entry *proc;
9      int wantHva;
10 } kvmModuleFileInfo;
```

Listing 2: `kvmModuleFileInfo` data structure used in `kvmModule`

To solve the problem that the procfs file could be accessed by multiple processes in parallel, the module allows each file to be opened only once. When a process tries to open a file while it is still open, this will be refused.

KVM provides functions to get the HVA from the GFN and the data structure with the VM information. When one of the files created by the module is read, the corresponding functions of the KVM kernel component are called to get the HVA. After this, the `follow_page` function is called to get the PFN (only if the PFN file is accessed).

Because the module scans only for VMs when it is loaded, it is not able to detect changes at runtime. Practically, this means that a VM that is started after the module was loaded is not listed and that a VM that was turned off after the module was loaded is still listed. It is possible to automatically detect changes at the running VMs, but this would require a new scan at every access to any of the modules functions. This would make the module more complex and much slower.

3.1.2 Usage

When the module is loaded, it creates the folder `/proc/kvmModule/` and one subfolder for each running VM named by the PID of the userland QEMU process on the host. In each folder, a file for the mapping of GFN to HVA (`hvamap`) and another file for the mapping of GFN to PFN (`pfnmap`) are created. The directory structure created by the module is shown in Figure 10.

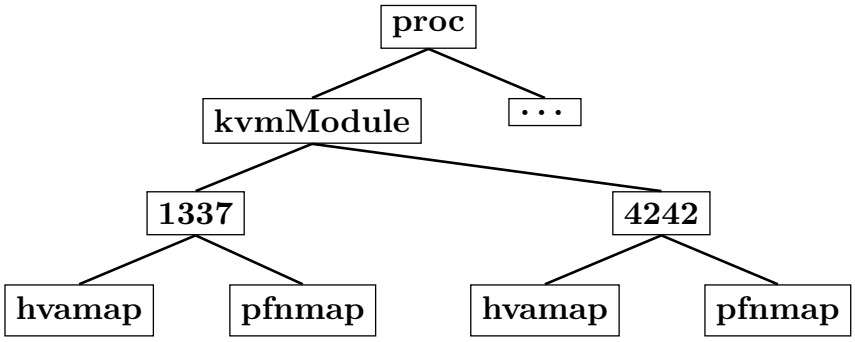


Figure 10: Directory structure created by `kvmModule`

All of these files can be opened only once a time. When a file is opened, `lseek` can be used to specify the GFN that should be translated. When reading from the file, the corresponding HVA or PFN is returned.

If the module is unloaded, it removes the `/proc/kvmModule` folder from the procfs.

Likely, this function will be used from a VM. In order to get access to the procfs on the host, sshfs [28] can be used to mount the procfs of the host system to the guest system.

This approach requires root privileges on the guest system to read the GFNs from `/proc/⟨PID⟩/pagemap` and on the host system to load the kernel module. Also, the guest has to be able to reach the host via the `ssh` protocol to mount the procfs of the host with `sshfs` [28].

When the running state of VMs changes, the module has to be reloaded (unloaded and loaded again) in order to detect this change because it scans only for running VMs when it is loaded.

3.2 Library Memlib

With `kvmModule` it is possible to get the mapping of GFN to HVA and GFN to PFN. When the files created by `kvmModule` are accessible from a guest system in a VM, this functionality can be

used from a VM as well.

Additionally, there are the pagemap files in procfs provided by the kernel (`/proc/⟨PID⟩/pagemap`).

With these files it is possible to get all addresses that are more hardware-related from a submitted address: A GVA can for example be translated to a GFN, a HVA and a PFN.

`Memlib` provides a set of functions to easily access these information. It also provides some data structures to store the fetched data so it can be accessed in the calling tools in an easy and uniform way.

3.2.1 Concepts

Processes in user space can access the mapping from virtual to physical addresses using the file `/proc/⟨PID⟩/pagemap`. Additionally, `kvmModule` creates two files to map GFNs to HVAs or PFNs.

Using this functionality, virtual addresses on a host or in a VM can be translated down to physical addresses on a host. Furthermore, the kernel provides the files `/proc/kpagecount` and `/proc/kpageflags` containing the information how often a physical page is mapped and which flags the page has. Just like in the pagemap files, the address is specified with the offset. For these files, the PFN is used instead of the HVA or GVA.

3.2.2 Usage

- **Address translation on a host**

  When running on a host system (not inside a VM), the address mapping of a process can be inspected as shown in Listing 3.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<unistd.h>
4  #include<malloc.h>
5  #include<sys/mman.h>
6
7  #include "memlib/pfnInspect.h"
8  #include "memlib/memoryInspect.h"
9  #include "memlib/util.h"
10
11 int main(int argc, const char **argv) {
12     int64_t hvas[512];
13
14     //Allocate 512 pages aligned on a 2MiB border
15     //(this will allocate a 2MiB Transparent Hugepage).
16     char *ptr = NULL;
17     posix_memalign((void**)&ptr, sysconf(_SC_PAGESIZE) * 512, sysconf
           (_SC_PAGESIZE) * sizeof(char) * 512);
```

```
18      madvise(ptr, 512 * sysconf(_SC_PAGESIZE) * sizeof(char),
            MADV_HUGEPAGE);
19
20      //Write to each page so the pages are allocated. Add the
21      //first address of each page to the list of hvas that
22      //should be translated
23      for(int i = 0; i < 512; i++) {
24          ptr[i*sysconf(_SC_PAGESIZE)] = 0x2a;
25          hvas[i] = (int64_t)&ptr[i*sysconf(_SC_PAGESIZE)];
26      }
27
28      //Create a addrInfo array based on the submitted hvas
29      addrInfo **aInfo = getAddrInfoFromHva(hvas, 512);
30
31      //Add the pfns to the data structure. Print an error message
32      //when this fails
33      if(addHpaToHva("/proc/self/pagemap", aInfo, 512) != 0) {
34          printf("Unable to get physical addresses.\n");
35          return EXIT_FAILURE;
36      }
37
38      //Print the found addresses to stdout
39      for(int i = 0; i < 512; i++) {
40          printf("HVA: 0x%lx PFN: 0x%lx\n", aInfo[i]->hva, aInfo[i]->
                pfn);
41      }
42
43      //Exit
44      return EXIT_SUCCESS;
45  }
```

Listing 3: Usage example of `memlib` on a host system

- **Address translation on a VM**

  When running on a VM, the address mapping inside the VM can be inspected like described above. The complete mappings including the host addresses can be analyzed as shown in Listing 4.

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<unistd.h>
4   #include<malloc.h>
5   #include<sys/mman.h>
6
7   #include "memlib/pfnInspect.h"
8   #include "memlib/memoryInspect.h"
9   #include "memlib/util.h"
10
11  int main(int argc, const char **argv) {
12      int64_t hvas[512];
```

```
13
14      //Allocate 512 pages aligned on a 2MiB border
15      //(this will allocate a 2MiB Transparent Hugepage).
16      char *ptr = NULL;
17      posix_memalign((void**)&ptr, sysconf(_SC_PAGESIZE) * 512, sysconf
            (_SC_PAGESIZE) * sizeof(char) * 512);
18      madvise(ptr, 512 * sysconf(_SC_PAGESIZE) * sizeof(char),
            MADV_HUGEPAGE);
19
20      //Write to each page so the pages are allocated. Add the
21      //first address of each page to the list of hvas that
22      //should be translated
23      for(int i = 0; i < 512; i++) {
24          ptr[i*sysconf(_SC_PAGESIZE)] = 0x2a;
25          hvas[i] = (int64_t)&ptr[i*sysconf(_SC_PAGESIZE)];
26      }
27
28      //Create a addrInfo array based on the submitted hvas
29      addrInfo **aInfo = getAddrInfoFromHva(hvas, 512);
30
31      //Add the pfns to the data structure. Print an error message
32      //when this fails
33      if(addHpaToHva("/proc/self/pagemap", aInfo, 512) != 0) {
34          printf("Unable to get physical addresses.\n");
35          return EXIT_FAILURE;
36      }
37
38      //This has to be set to the name of the VM. In this
39      //environment, the hostname equals the name of the
40      //VM so it has not to be specified additionally.
41      char hostname[256];
42      hostname[0] = 0;
43      if(gethostname(hostname, 255) < 0) {
44          printf("Unable to get hostname\n");
45          return EXIT_FAILURE;
46      }
47
48      //Path to the kvmModule folder of the host
49      //((/proc/kvmModule)
50      char PROCPATH[] = "/mnt/hostproc/kvmModule/";
51
52      //Path to the libvirt folder of the host
53      //((/var/lib/libvirt/qemu)
54      char VIRTPATH[] = "/mnt/libvirt/qemu/";
55
56      char modulePath[256];
57      snprintf(modulePath, 255, "%s%ld", PROCPATH, getGuestPID(VIRTPATH
            , hostname));
58
59      //Translate the found PFNs to GFNs and add host
```

```
60      //addressing information
61      if(addHostAddresses(modulePath, aInfo, 512) != 0) {
62          printf("Unable to get host addresses.\n");
63          return EXIT_FAILURE;
64      }
65
66      //Print the found addresses to stdout
67      for(int i = 0; i < 512; i++) {
68          printf("HVA: 0x%lx PFN: 0x%lx\n", aInfo[i]->hva, aInfo[i]->
                pfn);
69      }
70
71      //Exit
72      return EXIT_SUCCESS;
73  }
```

Listing 4: Usage example of `memlib` on a guest system

As stated before, it is required to mount some directories of the host systems to the guest in order to access the needed information (Listing 5).

```
root@host:/proc on /mnt/hostproc type fuse.sshfs (rw,nosuid,nodev,noexec,
    relatime,user_id=0,group_id=0,default_permissions,allow_other,
    _netdev,user,direct_io)
root@host:/var/run/libvirt on /mnt/libvirt type fuse.sshfs (rw,nosuid,
    nodev,noexec,relatime,user_id=0,group_id=0,default_permissions,
    allow_other,_netdev,user)
```

Listing 5: Mount points of the host system in the guest (using `sshfs`)

It is necessary to access the `libvirt` folder on the host system in order to translate the name of the VM to the PID of the QEMU user space process on the host. This information is used to identify the VMs by `kvmModule`.

3.3 Command-line Tool Memtool

`Memtool` inspects address mappings of processes using `memlib`. With `memtool`, the calls to `memlib` do not have to be added to every program whose address mapping should be inspected but can be analyzed from the command line by submitting the PID of the process. The next section contains some examples on how to use `memtool`.

3.3.1 Concepts

When analyzing the memory layout, it is required to specify the process which needs to be inspected. The reason for that is that the mappings from HVAs to PFNs are in the scope of a process. Another process may have the same HVA mapped to another PFN.

When the process is specified, the maps file of the process in the procfs (`/proc/⟨PID⟩/maps`) is

used to get a list of virtual memory regions and their content. These virtual memory regions are parsed to single HVAs. After this, the mapping of the HVAs is requested from the pagemap file of the process in the procfs. Then, the PFNs are inspected and grouped by ascending numbers.

3.3.2 Usage

By default, only groups of at least two PFNs are shown. This can be changed with the command line option `-c ⟨NUM⟩`. The command shown in Listing 6 prints the physical mappings of all HVAs of the process with the submitted PID. Mind that there is no space between the `-m` and the PID. This is the case because the command-line argument is optional and `getopt()` is unable to match the argument when there is a space between.

```
./memtool -m<PID> -c1
```

Listing 6: Command to use `memtool` to get a list of HVAs and PFNs they are mapped to

`Memtool` can use `kpagecount` and `kpageflags` in procfs to get additional information about PFNs. This functionality can be enabled with the command line flag `-p`. Listing 7 shows how to get the physical mappings of all HVAs of a process with additional PFN information.

```
./memtool -m<PID> -c1 -p
```

Listing 7: Command to use `memtool` to get a list of HVAs and PFNs with additional information

When running in a VM, it is possible to access the host address mappings using `kvmModule`. This functionality can be enabled using the command line flag `-i`. To be able to access the required files, the file systems should be mounted with the parameters stated in Section 3.2 in Listing 5.

The host name of the VM should be the same as the name of the VM in libvirt. If this is not the case, the command line option `-n ⟨NAME⟩` can be used to specify the name of the VM manually. The command shown in Listing 8 prints the mappings for the VM `majikthise`:

```
./memtool -m<PID> -c1 -i -n majikthise
```

Listing 8: Command to use `memtool` to get mapping information for a VM

3.4 Library Hammerlib

To exploit the rowhammer bug successfully, there are multiple tasks that have to be done. `Hammerlib` provides functions to solve these tasks. It implements the scanning for the address function, the calculation of aggressor and victim rows, and the execution of the actual attack.

Additionally, there are some functions to measure different timings that can be used to get a better understanding of the system on which the library is running.

3.4.1 Concepts

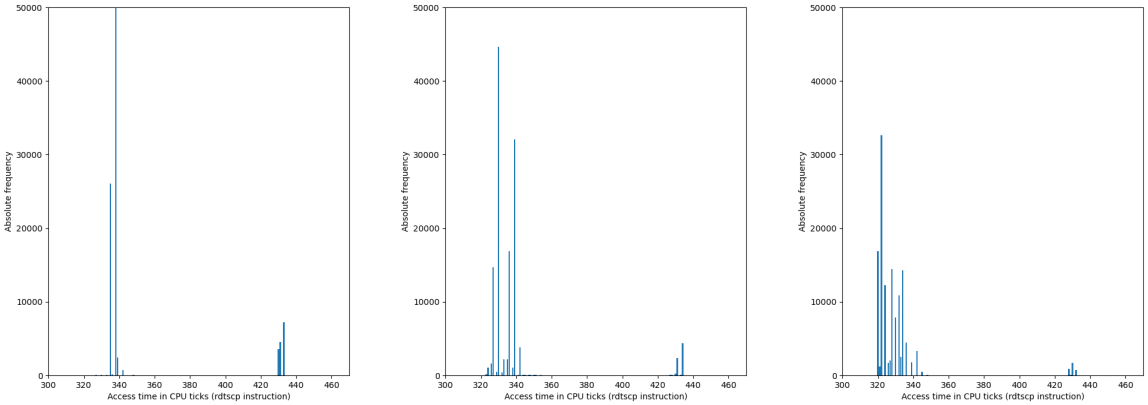
To get bit flips, an attacker has to choose addresses which are in different rows at the same bank. This is required because the rows have to be loaded into the row buffer and written back to the DRAM cells frequently. To do this, the attacker has to know how data is written to physical memory. This mapping is done by the MMU as described in Section 2.5

For this reason it is necessary to reverse-engineer the function used to map the addresses. I chose the approach to group addresses based on the time required for alternating accesses as described by Pessl et al. [12]. If the access is slow, there is a row conflict and the addresses are at the same bank and in different rows. Otherwise, there is a row hit and the addresses are either at different banks or at the same bank in the same row. To measure the actual memory access time rather than the access time to the cache, the data are removed from the cache with the `CLFLUSH` instruction before measuring.

To get continuous physical addresses, I use Transparent Hugepages (THPs). On the Systems I tested, a THP is 512 pages with a size of 4 KiB each. With this approach it is not necessary to translate the used virtual addresses to PFNs because the allocated memory is continuous and aligned on a 2 MiB border in physical and in virtual address space.

Hammerlib implements the following general approach: At first, addresses are allocated and grouped based on the access time (addresses with slow access time against each other are added to the same group). Then, the mapping function is calculated, so it matches the grouped addresses. Next, the addresses are grouped for hammering so the aggressor and victim rows match a specified pattern. At last, the hammer candidates are hammered in a way that the aggressors are accessed and the victims are checked for bit flips afterwards.

In order to group addresses based on the access time, it is required to define which accesses are fast and which are slow. To get this information, one THP is allocated. Then, the first addresses of each page in the THP are accessed in an alternating way and the access times are measured. On a system with n banks, a fraction of $\frac{1}{n}$ accesses can expected to be slow. This can be plotted as a histogram (access time and frequency of the occurrence of that access time) as shown in Figure 11.



(a) 8 Banks (b) 16 Banks (c) 32 Banks

Figure 11: Histograms from the access time on different systems measured when accessing addresses in a THP alternatingly

The threshold value t is calculated in a way that it is between the end of the first peak of fast accesses and the start of the smaller peak of the slow accesses.

There are multiple peaks sometimes. Because of this, a minimum amount of fast accesses is calculated using the number of banks. The end of the peak of fast accesses is detected only if there were at least as many fast accesses as calculated before. Otherwise, the next peak is interpreted as fast as well. The amount of banks is not known but that is no problem because it can be guessed (v).

When the addresses are grouped into v groups based on the calculated t, the addresses that can not be grouped are counted (e). If e falls below a defined threshold, the guessed number of banks is assumed to be correct and is used as banks ($n = v$). Theoretically all addresses should be grouped correctly when the assumed number of banks is correct. However, sometimes there are errors when grouping the addresses. After multiple experiments, I determined the threshold to 50, since this gave the best results.

After the number of banks is known, some THPs can be allocated and the addresses of the THPs can be grouped. If this was successful, there are n groups with a nearly equal amount of addresses in each of them.

The address bits of the banks are calculated by using a XOR function to the bits of the physical address determined by a bit mask [12]. The goal is to find the masks that are responsible for the addressing of the banks.

This example demonstrates how the address calculation works:

$pfn = 011111111111101101110001110111101111111110000000$

$msk = 000000000000000000000000000001101100111000000$

When applying a bitwise XOR to pfn & msk, the result corresponds to the address bit determined by msk:

$$\begin{array}{ll} & 011111111111101101110001110111101111111110000000 \\ \& & 000000000000000000000000000000011011001110000000 \\ \hline = & 000000000000000000000000000000010011001110000000 \end{array}$$

$XOR(000000000000000000000000000000010011001110000000)$

$= 1 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 1$

$= 0$

The address function can be reverse-engineered by guessing masks and checking if they are plausible using multiple criteria:

- The mask should have "1" as result for one half of the groups and "0" for the other half of them.
- The mask should have the same result for all addresses in the same group (either all "1" or all "0").
- Every bit specified in the mask should have an effect to the result. If there would be a bit set in the mask which is either "1" or "0" for all HPAs that are checked, there is a mask with fewer bits which is equivalent or anti-equivalent because the not significant bit can be removed.

If these criteria match, the mask is added as a candidate. After a big amount of possible masks was checked, the list of found candidates is simplified by removing equivalent or anti-equivalent masks.

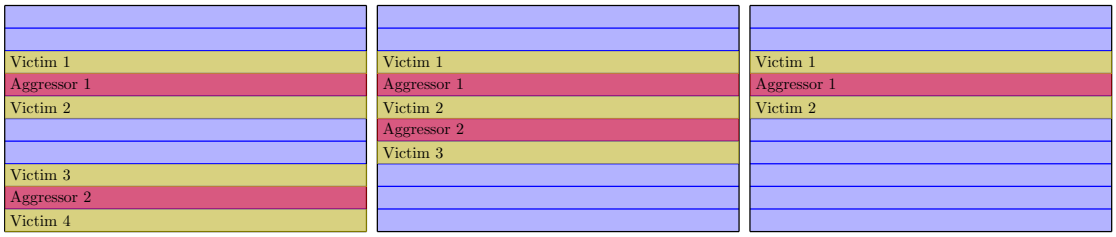
Now, $log_2(n)$ masks should be left (n is the number of banks).

When the function for addressing the banks is known, a bigger chunk of memory can be allocated. After this, the row and bank for each allocated address can be calculated using the function found before.

The next step is to find candidates that can be hammered. A candidate consists of a list of aggressor rows and a list of victim rows. The calculation of these candidates is based on rules defined by the parameters submitted to the function.

With this approach it is possible to define different patterns for hammering. It is more flexible than implementing only some well-known hammer patterns as single-sided (two aggressor rows with some random space between them, see Figure 12a), double-sided (two aggressor rows with exactly one row between them, see Figure 12b) or one-location (one aggressor row, see Figure 12c). There is no general rule which of those access patterns works best on a system. For example, one-location can induce bit flips in systems with mitigations based on memory access patterns [29].

Now, the value `0x00` is written to all victim addresses. Then, the aggressor rows are accessed in



(a) Single Side (b) Double Side (c) One Location

Figure 12: Typical row access patterns used in rowhammer attacks (aggressors are read multiple times, victims are likely to have bit flips after that)

a round-robin fashion using `flush + reload` [12] to access the DIMM every time rather than loading the data from the cache. When a defined amount of accesses is done, the content of the victim rows is checked. If there is any value which is not `0x00`, a bit flip was found. This procedure is repeated with the value `0xFF` written to all victim addresses. Thereby, flips from "1" to "0" as well as flips from "0" to "1" will be detected.

3.4.2 Usage

An extended version with additional comments of the listings in this section can be found in the appendix.

Hammerlib can be used to reverse-engineer the address function as shown in Listing 24.

```
1  #include<stdio.h>
2
3  #include"hammerlib/afunc.h"
4  #include"hammerlib/hammer.h"
5
6  int main(int argc, const char **argv) {
7      int nTries = 1;
8      int nChecks = 1;
9      int errtres = 50;
10     int iter = 10;
11     int accessTimeCnt = 0;
12     int verbose = 1;
13     int getTime = 0;
14     int vMode = 1;
15     int scale = -1;
16     int blockSize = (1<<7);
17     int nInitSets = 1;
18     int measureRowSize = 0;
19     int maxMaskBits = 7;
20     char *exportFilename = "export.cnf";
21
22     int banks = getBankNumber(nTries, nChecks, errtres, iter, &
           accessTimeCnt, verbose, getTime, vMode, scale);
23     addressGroups *aGroups = constructAddressGroups(banks * 2, blockSize);
24
```

```
25      int totalErrors = 0;
26
27      for(int i = 0; i < nInitSets; i++) {
28          printf("\rStarting set %3d of %3d", i + 1, nInitSets);
29          if(verbose >= 2) {
30              printf("\n");
31          }
32          fflush(stdout);
33          totalErrors += scanHugepages(1, aGroups, verbose, banks, nChecks,
                iter, &accessTimeCnt, getTime, vMode, i, scale, blockSize,
                measureRowSize);
34      }
35
36      aGroups->nItems = banks;
37
38      maskItems *mItems = constructMaskItems();
39      findMasks(aGroups, maxMaskBits, mItems, verbose);
40
41      if(1L<<mItems->nItems != banks) {
42          printf("Number of masks does not match number of banks.\n");
43          return EXIT_FAILURE;
44      }
45
46      printf("Found addressing bitmasks:\n");
47      for(int i = 0; i < mItems->nItems; i++) {
48          printf("\t");
49          printBinaryPfn(mItems->masks[i], mItems->nItems);
50      }
51
52      exportConfig(exportFilename, banks, mItems);
53
54      return EXIT_SUCCESS;
55  }
```

Listing 9: Example of reverse-engineering the address function with `hammerlib`

This code will try to measure the amount of banks and reverse-engineer the address function. If it was successful, the found amount of banks and the address function is exported to a file.

That file can be imported as shown in Listing 25 to execute a rowhammer attack based on the found information. It is also possible to reverse-engineer the address function and continue with the hammering directly without exporting and importing the configuration. The advantage of storing the configuration is the increased stability which means the configuration is the same each time so the results will be consistent because there can not be any differences due to errors in the reverse-engineered function.

```
1   #include<stdio.h>
2
3   #include"hammerlib/afunc.h"
4   #include"hammerlib/hammer.h"
```

```
5
6  int main(int argc, const char **argv) {
7      int nSets = 1;
8      int nHammerOperations = 1000000;
9      int omitRows = 0;
10     int multiprocessing = 0;
11     char *importFilename = "export.cnf";
12     int vMode = 1;
13     int verbose = 1;
14     int banks = 0;
15     maskItems *mItems = constructMaskItems();
16
17     importConfig(importFilename, &banks, &mItems);
18     long unifiedMask = 0;
19     for(int i = 0; i < mItems->nItems; i++) {
20         unifiedMask |= mItems->masks[i];
21     }
22
23     int blockSize = 0;
24     for(int i = 0; i < sizeof(long) * 8; i++) {
25         if((unifiedMask >> i) % 2 == 1) {
26             blockSize = 1 << i;
27             break;
28         }
29     }
30
31     if(blockSize > 4096) {
32         blockSize = 4096;
33     }
34
35     addressGroups *aGroups = constructAddressGroups(banks, blockSize);
36     for(int i = 0; i < nSets; i++) {
37         printf("\rStarting set %3d of %3d", i + 1, nSets);
38         fflush(stdout);
39         addHugepagesToGroups(1, aGroups, mItems, vMode, blockSize);
40     }
41     printf("\n");
42
43     long foundBitflips = 0;
44     foundBitflips += hammer(aGroups, nHammerOperations, verbose,
           HMASK_DOUBLESIDE, ROWNUM_DOUBLESIDE, omitRows, multiprocessing);
45     printf("Found %d hammers.\n", foundBitflips);
46
47     return EXIT_SUCCESS;
48 }
```

Listing 10: Example of executing a rowhammer attack with `hammerlib`

3.5 Command-line Tool Hammertool

`Hammertool` is a command-line tool that enables users to use the functionality of `hammerlib` and change its internal configuration parameters without the requirement to write separate tools.

3.5.1 Concepts

`Hammertool` makes use of `getopt` and `getopt_long` [30] to parse command-line parameters submitted by the user. Most parameters use default values so it is not required to specify them when this default value works for the setup.

After parsing the command-line parameters, the amount of banks and the address function are reverse-engineered when they are not imported from a file. Then, the found function is exported if specified. After this, a rowhammer attack is executed based on a submitted pattern.

The pattern can be specified on the command-line with two parameters: `--hammer-many-sides ⟨NUM⟩` and `--hammer-mask ⟨BIN⟩`. The parameter to specify the hammer mask is a binary string of "1" and "0". Thereby, a "1" stands for an attacker row (which is accessed multiple times), a "0" stands for a non-attacker. Based on the aggressor rows it is calculated which of the rows specified with "0" are victim rows. This is done in a way that any row adjacent to an attacker and not an attacker itself is handled as victim row. The parameter of `--hammer-many-sides` specifies a number of attacker rows that are chosen additionally to the rows specified in the hammer mask randomly.

`Hammertool` prints a basic timing histogram to `stdout` when started with at least verbosity level 2 (command-line flag `-vv`). The access time data shown in that histogram can be exported using the `--export-stats` command-line flag. The data is written to CSV-files in the `data` folder which is removed recursively and created when `hammertool` is started and timing statistics should be exported. The CSV files can be plotted using the script `plot.py` in the same folder as `hammertool`.

The tool provides two ways to measure access times: the `RDTSCP` instruction and the `clock_gettime` function. The command-line flag `--gettime` can be used to measure the time with `clock_gettime` (time in ns) rather than with `RDTSCP` (time in CPU ticks).

`Hammertool` has the command-line option `--print-timing ⟨NUM⟩` to access one memory location for `NUM` times. When a refresh or another memory access to the same bank is in progress, the access takes more time. Then, a diagram of access times is printed to `stdout`. This can be used to check if there are accesses on a regular basis every $\approx 7.8\,\mu s$ or every $\approx 3.9\,\mu s$. Thereby, the refresh interval can be measured.

There is a mode that detects chunks of addresses which have slow access times when accessed alternating with the first address of a THP. The addresses detected are at the same bank and at different rows. The greatest common divisor of the size of all detected chunks which contains at least 8 addresses is the size of data (in bytes) that is always written continuously on the system.

The level of at least 8 addresses was chosen because there are many very small chunks probably due to measuring errors. This can be used to deduce the effective row size. To print it, the command-line flag `--measureRowSize` can be used.

If the measured amount of data continuously written reaches at least the size of one memory page, the tool can operate on entire memory pages rather than addresses. This reduces the memory footprint and improves the runtime. To use this mode, the command-line flag `--searchPagesAddressesOnly` can be used. This flag is only relevant for the calculation of the address function. After this function is known, the size of continuously written data is calculated from the function and set accordingly.

It is possible to hammer multiple rows at the same time when they are at different banks. To increase the speed of the attack, `hammertool` implements a multithreading mode for hammering which can be enabled with the `--multithreading ⟨NUM⟩` command-line parameter. If `NUM` is submitted, there will be `NUM` threads that hammer in parallel. If it is omitted, as many threads as logical CPU cores in the system will be used.

Because it is not possible to resolve the mapping between virtual and physical addresses without root privileges, `hammertool` uses a virtual mode that operates only on virtual addresses by default. It seems that virtual addresses are at the correct offsets when THPs are used. So, the mask and bank calculations can be done using the virtual addresses directly. If one want to use physical addresses instead, there is the command-line flag `--physical`.

3.5.2 Usage

To scan for the address function of the banks on a system without root privileges or a VM, the tool can be run as shown in Listing 11.

```
./hammertool
```

Listing 11: Command to reverse-engineer address function in virtual mode with `hammertool`

In case of failed calculation the tool writes “No masks found” to `stdout` and the return code is smaller than 0. Due to this behaviour, a script could retry to calculate the address function by executing `hammertool` again.

When the number of banks at the system is known, it can be specified with the command-line option `--banks ⟨NUM⟩` (Listing 12). When this is done, `hammertool` does not need to scan for the number of banks. This reduces the runtime and prevents possible errors in the calculation of the number of banks.

```
./hammertool --banks 16
```

Listing 12: Command to specify the number of memory banks in `hammertool`

There are different modes for verbosity. To enable verbose mode, add the `-v` command-line flag. If the output should be more verbose, multiple `-v` can be added (e.g. `-vvv`). Listing 13 shows an

example of running `hammertool` in verbosity level two.

```
./hammertool -vv
```

Listing 13: Command to set the verbosity level to 2 in `hammertool`

There are several command-line options to change the amount of measurements of different parameters. Currently, the following options are supported:

- `--maxMaskBits ⟨NUM⟩`: `Hammertool` scans for masks by applying them and checking the result. To limit the runtime, the maximal number of mask bits is defined to be seven. Maybe, longer masks could be required. To be able to scan in such cases as well, the amount of maximum set bits in the mask can be changed.
- `--iterations ⟨NUM⟩`: One measurement of timing can be inaccurate (e.g. because a refresh is done at the same time slowing down the measured access). To solve this problem, `hammertool` measures access times multiple times and uses the integer median of the measurements as measured value. By default, each access is measured ten times, this can be changed with this option.
- `--check-bank-addresses ⟨NUM⟩`: When two addresses are at the same bank and in the same row, the access is fast and the addresses are not added to the same group. This problem can be solved by measuring timing against multiple addresses of the group. By default, an address is compared to eight addresses of a group. When the mean access time is greater than the threshold, the address is added to this group. By increasing the value of this parameter, more addresses are checked before an address is added or not added to a group.
- `--n-get-banks ⟨NUM⟩`: When no number of banks is specified, `hammertool` tries to measure the number of banks. This is done one time by default. On some systems, the measurement of the number of banks can be unstable. In this case, the amount of checks can be increased. After measuring for the specified times, the integer median is taken as number of banks.

The rowhammer attack can be done in different patterns. Basically, a pattern consists of aggressor and victim rows. The aggressor rows were accessed many times and the victim rows are likely to contain bit flips after the aggressor rows are accessed. There are two parameters to specify the pattern that should be used: `--hammer-many-sides ⟨NUM⟩` and `--hammer-mask ⟨BIN⟩`.

For simplicity, there are some shortcuts for typical hammer patterns:

- `--hammer-one-location` is equivalent to the options in Listing 14.

```
./hammertool --hammer-many-sides 0 --hammer-mask 1
```

Listing 14: Command to use `hammertool` for one-location rowhammer

- `--hammer-single-side` is equivalent to the options in Listing 15.

```
./hammertool --hammer-many-sides 1 --hammer-mask 1
```

Listing 15: Command to use `hammertool` for single sided rowhammer

- `--hammer-double-side` is equivalent to the options in Listing 16.

```
./hammertool --hammer-many-sides 0 --hammer-mask 101
```

Listing 16: Command to use `hammertool` for double-sided rowhammer

To reverse-engineer the address function in virtual mode, store the found configuration in the file `rowhammer.cnf` and execute a double-sided rowhammer attack on 256 MiB of memory, the command shown in Listing 17 can be used.

```
./hammertool --exportConfig rowhammer.cnf --hammer-double-side --sets 128
```

Listing 17: Command to execute a double-sided rowhammer in virtual mode with `hammertool`

3.6 Toolset Hammertest

There are several PoCs that can be used to check if a system can be affected by rowhammer. I chose three of them: `Rowhammer-Test` [19], `RowhammerJS` [20] from the RowhammerJS paper [31] and `TRRespass` [21] from the TRRespass paper [32].

The `Rowhammer-Test` PoC uses a probabilistic approach to select the virtual addresses. This means the PoC does not require the address function to calculate the addresses that are accessed. The other two PoCs don't use this approach and require the address function, so it is required to find the address function manually and compile the PoCs afterwards.

Then, the PoCs can be started with the corresponding parameters. The results are printed to `stdout` or `stderr` and can be analyzed.

To automate this test, I have written `hammertest`, a script that does the required steps automatically. Additionally, some information of the system that is running `hammertest` are collected. All results are sent to a server, so the user can simply start the script and wait for it to finish.

3.6.1 Concepts

`Hammertest` implements the following workflow:

- Download required data (source code of the PoCs and `hammertool` binary)
- Collect information about the system:
  - SPD information [14], [15] of the DIMMs
  - CPU information (content of `/proc/cpuinfo`)
  - Random-Access Memory (RAM) information (output of `decode-dimms` [33])

  - Hostname of the system
  - General system information (output of `dmidecode` [34])
- Search for address function
- Update the source code of the `RowhammerJS` and `TRRespass` PoCs and recompile both
- Execute the PoCs and `hammertool` in rowhammer mode
- Send the collected data to the server

The submitted reports are collected and ready to be analyzed at the server.

3.6.2 Usage

In order to run `hammertest`, there are some requirements the Linux system has to fulfill:

- For the `TRRespass` PoC, at least one 1 GiB hugepages has to be enabled by adding the following parameters to the kernel command line: `hugepagesz=1G hugepages=1`.
- The following software has to be installed: `zip`, `unzip`, `gnu-netcat`, `dmidecode`, `i2c-tools`, `make`, `gcc`.

When these requirements are met, the system needs to be connected to the internet. It has to be able to connect to a web server on port 443 to download the necessary files and to connect to a system on Port 4242 to send the reports.

After this, the script can be executed with the command shown in Listing 18 (root privileges are required).

```
sh run.sh
```

Listing 18: Command to start `hammertest`

If the test should run locally only (results should not be sent to the server) the parameter `noauto` can be used (Listing 19).

```
sh run.sh noauto
```

Listing 19: Command to start `hammertest` without sending results to the server

3.7 Toolset Hammeriso

`Hammeriso` is an ISO image based on Arch Linux which can be used to automatically execute `hammertest`. This brings several benefits.

Because it is not required to install the system, a properly configured system for testing can be started very quickly on a computer without the necessity to install it before. This saves time and does not require any hard drives to install the system to.

Additionally, all systems used for testing are equal in terms of software revisions, so there should not be any errors in measurements due to different software versions on the systems.

Because I have not found any systems vulnerable to rowhammer at first, I wanted other people to test their systems for me as well. With the ISO, everything these people had to do was downloading and burning the file to a DVD or flashing it to a USB drive. After this, they were able to boot their systems with the medium created before and connect them to the network with a curses-based tool which is started after the system has booted. Then, all tests are done automatically and the results are sent to the server.

That is the reason why it is simpler to repeat the test (e.g. with other versions of the tools) because all it takes is to boot the system with the ISO image again. It is not required to download a new image because the test files (entire content of `hammertest`) are downloaded automatically when the ISO is booted.

3.7.1 Concepts

I used Archiso to build the image. The components of the Archiso setup can be found in the appendix of this thesis.

After adding the required entries to the package list and creating the required files in the root file system, I modified the bootloader configurations to enable one 1 GiB hugepage. Then, I created the ISO image.

3.7.2 Usage

`Hammeriso` can be used by burning the ISO image to a DVD or flashing it to a USB thumb drive. Then, the system that should be tested can be booted with the DVD or thumb drive. If a manual network configuration is required, a dialog is shown. Afterwards, the tests run automatically and the results are sent to the server.

4 Practical Rowhammer Exploitation

After the implementation was done, I started to test the vulnerability of several systems to rowhammer. See Table 1 for a list of tested systems and configurations.

CPU	Vendor	Product	DIMM configuration	BIOS revision
i5-3320M	LENOVO	ThinkPad X230 Tablet	1x Samsung M471B1G73BH0-YK0 (8 GiB)	2.61
i5-3320M	LENOVO	ThinkPad X230 Tablet	1x Samsung M471B1G73BH0-YK0 (8 GiB) 1x Micron 16KTF51264HZ-1G6M1 (4 GiB)	2.61
i7-4800MQ	LENOVO	ThinkPad T540p	1x Samsung M471B1G73BH0-YK0 (8 GiB)	2.17
i7-4800MQ	LENOVO	ThinkPad T540p	2x Samsung M471B1G73DM0-YK0 (8 GiB)	2.17
i5-8250U	LENOVO	ThinkPad T580	2x Samsung M471A1K43CB1-CRC (2 GiB)	1.10

Table 1: Systems which were tested and not vulnerable to rowhammer

Because none of these systems seemed to be vulnerable to rowhammer (neither `hammertool` nor any of the other PoCs), I contacted the research groups that published the other PoCs (`IAIK` [5], `VUSec` [2]) and asked them which systems they have used. They gave me the hint that there may be a mitigation in Basic Input/Output System (BIOS) that doubles the refresh frequency.

4.1 Rowhammer Mitigations

To test if the notebooks used a double refresh rate, I measured the access time on the `ThinkPad X230 Tablet` and the `ThinkPad T540p`. It was 3.9 µs for both devices resulting in a refresh rate of 32 ms (see Section 2.4). Because of this, bit flips were very unlikely.

There are versions with the note "Mitigate risk of security vulnerability related to DRAM Row Hammering" [35], [36] in the BIOS changelogs of the notebooks I used (`T540p`, `X230T`).

I tried to downgrade the BIOS to a version below but it was not possible due to the downgrade locks Lenovo uses at some BIOS versions.

To get the old BIOS versions to the Notebooks, I disassembled them and flashed the BIOS Electrically Erasable Programmable Read-Only Memories (EEPROMs) manually.

4.2 Manual BIOS Downgrade

I have done the steps described below on a `ThinkPad X230T` and a `ThinkPad T540p`. Both of these notebooks have one EEPROM with a size of 8 MiB and one EEPROM with 4 MiB. The EEPROMs are handled as continuous space with the 8 MiB EEPROM first, so there are 12 MiB of storage in total [37].

At the beginning, I needed some hardware to dump the current content of the EEPROMs. Because I did not have a programmer at this time, I used an Arduino nano with the frser-duino firmware [38].

In order to connect the Arduino with the PC, I have made use of a FT232R-based board [39] and set the voltage jumper to 3.3 V. I connected the Vcc pin of the board with the Arduino.

Typically, the Arduino operates at a voltage of 5 V. Because the EEPROMs are working at a voltage of 3.3 V, it is required to either run the Arduino on 5 V and shift the output levels from 5 V to 3.3 V or to operate the Arduino on 3.3 V.

I used Flashrom [40] to access the EEPROMs. Basically, the entire setup consisted of a computer with Flashrom installed. This tool communicates with the FT232R-based board via USB. The board translates the USB commands from Flashrom to serial commands. These are sent to the Arduino which translates them to Serial Peripheral Interface (SPI) commands. The output of the Arduino is linked to the EEPROMs.

There are programming clamps that can be connected to ICs in the Small Outline Integrated Circuit (SOIC8) package (Figure 13). With such clamps it is possible to interact with the EEPROMs without soldering them to a programming pad. So, the old BIOS can be read and the new one flashed without soldering.

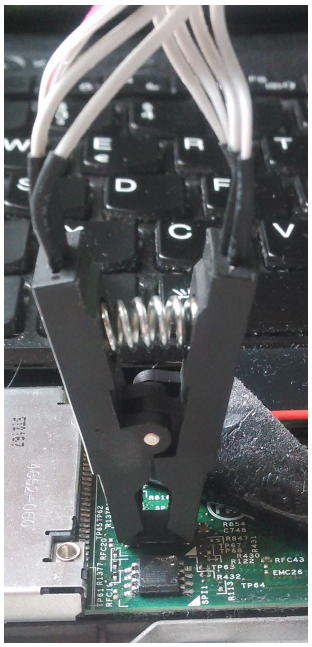


Figure 13: Clamp to program EEPROMs without the necessity to solder them to a programming pad first

Because the EEPROMs are connected to other components when still soldered on the mainboard, the power of the FT232R board may not be sufficient for all connected components. If that is the case (it was on the `X230T`), an external power supply is required.

Figure 14 gives an overview of the setup.

On the `X230T`, a `MX25L3273E` [42] and a `MX25L6473E` [43] are used and on the `T540p`, a `MX25L3206E` [44] and a `MX25L6406E` [45].

After I connected the FT232R board, the Arduino and the 8 MiB EEPROM as described above, I have done the steps described below with the `X230T`.

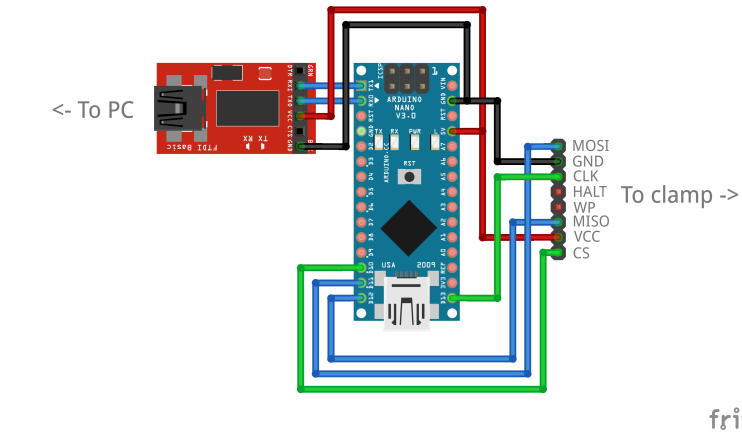


Figure 14: Connection diagram of the Arduino-based programmer setup drawn with Fritzing [41]

Because dumping and flashing with the Arduino is pretty slow (at a baud rate of 115 200, an entire dump of the 4 MiB EEPROM takes approx. 5 min), I have ordered a CH341-based [46] board for flashing after the first flash on the `X230T` succeeded with the Arduino setup. At a baud rate of 2 000 000, an entire dump of the 4 MiB EEPROM takes approx. 20 s. According to the frser-duino [38] project, the serial speed of the Arduino setup can be increased as well, but I have not tested it.

I was not able to get a valid dump of the `T540p` with the Arduino setup. Because of that I used a CH341-based board which works more stable than the Arduino setup. In order to use the CH341 instead of the arduino setup, the programmer specified at the flashrom command has to be changed from `serprog:dev=/dev/ttyUSB0:115200` to `ch341a_spi`.

To dump the content of the `MX25L6473E` two times the commands shown in Listing 20 can be used.

```
flashrom -r mx25l6473e_01.bin -p ch341a_spi -c MX25L6473E
flashrom -r mx25l6473e_02.bin -p ch341a_spi -c MX25L6473E
```

Listing 20: Command to dump a `MX25L6473E` EEPROM

Then, I verified the checksums with the commands from Listing 21. If they are not equal, the files are not identical and something might have gone wrong wile reading.

```
sha512sum mx25l6473e_01.bin
sha512sum mx25l6473e_02.bin
```

Listing 21: Command to verify the checksums of the two dumps for `MX25L6473E`

If both checksums are equal, the files are identically and the dump did work correctly. Now, this has to be repeated for the 4 MiB EEPROM. After that is done, there should be four files: mx25l6473e_01.bin, mx25l6473e_02.bin, mx25l3273e_01.bin and mx25l3273e_02.bin.

Because the BIOS is continuous over both EEPROMs, the files can be concatenated like shown in Listing 22.

```
cat mx25l6473e_01.bin mx25l3273e_01.bin > bios_dump.bin
```

Listing 22: Command to concatenate the two dumps of `MX25L6473E` and `MX25L3273E`

Now, the entire BIOS dump can be analyzed. To do this, I used UEFITool [47]. When loading the file, there is an Intel image with the size of 12 MiB. Besides other parts, the image contains a part named "BIOS region" with a size of 7 MiB. The first part of that is a Padding of 3 MiB. So, the last 7 MiB − 3 MiB = 4 MiB contain the BIOS. Because of this, it is only required to flash the 4 MiB EEPROM.

When ignoring the first Padding, the "BIOS region" consists of the following parts:

- 7A9354D9-0468-444A-81CE-0BF617D890DF (size: `0x2B0000`)
- Padding (size: `0x1000`)
- FFF12B8D-7696-4C8B-A985-2747075B4F50 (size: `0x66000`)
- 00504624-8A59-4EEB-BD0F-6B36E96128E0 (size: `0x29000`)
- 7A9354D9-0468-444A-81CE-0BF617D890DF (size: `0xC0000`)

As expected, the total size is `0x2B0000` + `0x1000` + `0x66000` + `0x29000` + `0xC0000` = `0x400000`.

Now, a 4 MiB BIOS image that can be flashed to the EEPROM is required. Lenovo does not supply a corresponding file in the `driver and downloads` section of the support pages. However, there is an ISO image and a Windows tool that can be used to upgrade the BIOS.

When the windows tool is executed, it extracts some files to `C:\DRIVERS\FLASH\BIOS-CODENAME` which contains a sub folder named like the ID of the BIOS version. In this folder, there is one `FL1` file with the size of 4.7 MiB (8.1 MiB for the `X230T`).

When opening this file in UEFITool, it contains the parts described above. Now, the parts can be extracted in a new image file which should have a size of 4 MiB. This file can be written to the 4 MiB EEPROM using the command shown in Listing 23.

```
flashrom -w bios.bin -p ch341a_spi -c MX25L3273E
```

Listing 23: Command to write data to a `MX25L3273E`

Afterwards, the new BIOS is flashed and ready to use.

4.3 Results

With the BIOS downgraded to the version before the patched one, I measured a refresh every ≈ 7.8 µs which equals a refresh period of 64 ms. So, the downgrade reduced the refresh period to the original value.

Afterwards, I tested the `RowhammerJS` PoC again and found some bit flips on both machines (I used the same DIMM (`Samsung M471B1G73BH0-YK0`) on both, so bit flips should occur either at both or at none of them).

5 Related Work

In the paper with the title "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors" [3], Kim et al. described the technical problems behind the rowhammer bug.

One year later, Seaborn and Dullien published an exploit of the rowhammer bug that can be used for local privilege escalation [48]. Seaborn developed a tool for automated rowhammer testing as well [19]. This tool uses a probablistic approach which means that it is not required to know the mapping between addresses and memory locations. It is part of the `hammertest` toolset.

In the same year, Gruss et al. published "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript" [31]. This paper describes the approach of executing a rowhammer attack without the usage of the `CLFLUSH` instruction by evicting the lines from the CPU cache. The PoC published in the scope of that paper included a native component with `CLFLUSH` as well. That tool makes it possible to find bit flips when the address function of the MMU is known. This PoC is part of the `hammertest` toolset.

In 2016, Pessl et al. described an approach of time-based reverse-engineering of the address function in their paper "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks" [12]. For this thesis, I used the time-based approach described in that paper for the reverse-engineering of the address function.

In the same year, Razavi et al. published the paper "Flip Feng Shui: Hammering a Needle in the Software Stack" [1] which describes an approach of changing one arbitrary bit in a chosen page in memory in a virtualised environment. In order to get the correct address mappings, the authors used THPs. `Hammerlib` uses THPs as well. This approach is used to get continuous physical memory and thereby not needing the exact physical addresses.

In 2020, Frigo et al. published "TRRespass: Exploiting the Many Sides of Target Row Refresh" [32]. This paper describes an approach that exploits rowhammer on DDR4 memory with implemented mitigations. I used the PoC published in the scope of the paper as part of `hammertest`.

In difference to all of these PoCs, `hammertool` implements a comfortable all-in-one solution for rowhammer testing. It is not longer required to reverse-engineer the addresses, edit the source code of a PoC, compile and run it afterwards.

The kernel module makes it possible to access and inspect the entire address mapping of a virtualised system. Of course this can not be used for a practical exploit because it requires root privileges on both the host and the VM but it can help to understand the behaviour of different mapping mechanisms, e.g. KSM.

`Hammertest` and `hammeriso` provide a more automated way of testing systems. With these tools, many machines can be tested automatically.

6 Conclusion

At first, I developed some tools to be able to analyze the address mappings. Because it was not possible to access the host addresses that are mapped to a VM, I have written a kernel module that runs on the host and makes it possible to access these mappings from the user space.

Next, I wrote a tool that can be used to test if a system is vulnerable to rowhammer. Because the first systems tested were not vulnerable, I have built a tool set for automated testing, so it was not necessary to do the required steps on every system I tested. Then, I created an ISO image that contains all required tools and settings so the test can be started by just booting it on a computer that should be tested.

I tested several computers with different memory configurations with the tools I have developed. However, none of them was vulnerable to rowhammer. I added some measurement functions to the tools I wrote before and found the reason why they did not discover any bit flips: There was a mitigation in the BIOS. Because it was not possible to downgrade the BIOS with the utilities from the vendor, I have done a manual BIOS downgrade. Afterwards, some bit flips occurred on the system.

With the tools described in this practical thesis it is possible to analyze the mappings between virtual and physical addresses as well as the mapping between addresses in a VM and on the host. `Memtool` makes this functionality accessible as command line tool. With `hammertool`, it is possible to reverse-engineer the functions used to calculate the memory banks data is stored on in physical memory.

The manual BIOS downgrade on the `X230T` and the `T540p` removed the mitigations and resulted in both systems being vulnerable to rowhammer attacks.

References

[1] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack", in USENIX Security, Jun. 2016. [Online]. Available: https://download.vusec.net/papers/flip-feng-shui_sec16.pdf.

[2] (2020). "VUSec", [Online]. Available: https://www.vusec.net/ (visited on 11/16/2020).

[3] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors", SIGARCH Comput. Archit. News, vol. 42, no. 3, pp. 361–372, Jun. 2014, ISSN: 0163-5964. DOI: 10.1145/2678373.2665726. [Online]. Available: https://doi.org/10.1145/2678373.2665726.

[4] D. Adams, The Hitchhiker's Guide to the Galaxy.

[5] (2020). "IAIK", [Online]. Available: https://www.iaik.tugraz.at/ (visited on 11/16/2020).

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space", in 27th USENIX Security Symposium (USENIX Security 18), 2018.

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution", in 40th IEEE Symposium on Security and Privacy (S&P 19), 2019.

[8] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX", in 41st IEEE Symposium on Security and Privacy (S&P'20), 2020.

[9] M. Heckel, "OpenFFS", yet unpublished bachelor thesis, title may be changed before it will be published, 2021.

[10] N. Honarmand. (2015). "Main Memory and DRAM", [Online]. Available: https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp16/cse502/slides/05-main_mem.pdf (visited on 11/16/2020).

[11] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob, "DRAM Refresh Mechanisms,Penalties, and Trade-Offs", IEEE TRANSACTIONS ON COMPUTERS, vol. 64, 2015.

[12] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks", in USENIX Security Symposium, USENIX Association, 2016, pp. 565–581.

[13] M. Seaborn. (2015). "Measuring the DRAM refresh rate by timing memory accesses". commit 2013bb50a3db541d4c969ca87471e16e866415bd, [Online]. Available: https://github.com/google/rowhammer-test/blob/master/refresh_timing/README.md (visited on 11/16/2020).

[14] (2014). "Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules", JEDEC Solid State Technology Association, [Online]. Available: https://www.jedec.org/standards-documents/docs/spd-4010211 (visited on 12/04/2020).

[15] (2015). "Annex L: Serial Presence Detect (SPD) for DDR4 SDRAM Modules", JEDEC Solid State Technology Association, [Online]. Available: https://www.jedec.org/standards-documents/docs/spd412l-3 (visited on 12/04/2020).

[16] M. Gorman, "An Investigation into the Theoretical Foundations and Implementation of the Linux Virtual Memory Manager", M.S. thesis, University of Limerick, Apr. 2003, pp. 1–236. [Online]. Available: https://www.kernel.org/doc/gorman/pdf/thesis.pdf (visited on 12/04/2020).

[17] (2020). "AMD Developer Guides, Manuals and ISA Documents", [Online]. Available: https://developer.amd.com/resources/developer-guides-manuals/ (visited on 12/22/2020).

[18] (2017). "Linux 2.6.32", [Online]. Available: https://kernelnewbies.org/Linux_2_6_32#Kernel_Samepage_Merging_.28memory_deduplication.29 (visited on 11/16/2020).

[19] (2015). "Program for testing for the DRAM "rowhammer" problem". commit c1d2bd9f629281402c10bb10e52bc1f1faf59cc4, Google, [Online]. Available: https://github.com/google/rowhammer-test (visited on 12/04/2020).

[20] (2017). "Program for testing for the DRAM 'rowhammer' problem using eviction". commit 62c9f3199d0a7ff5f7e06722fbcc1e186cf37591, IAIK, [Online]. Available: https://github.com/IAIK/rowhammerjs (visited on 12/04/2020).

[21] (2020). "TRRespass". commit 9d72093fb026fd736582719e5d61a2a1b7bf9cd9, VUSec, [Online]. Available: https://github.com/vusec/trrespass (visited on 12/04/2020).

[22] (Aug. 2020). "PROC(5) Linux Programmer's Manual", [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 12/04/2020).

[23] (2013). "LSEEK(3P) POSIX Programmer's Manual", [Online]. Available: https://man7.org/linux/man-pages/man3/lseek.3p.html (visited on 12/04/2020).

[24] (Feb. 2018). "READ(2) Linux Programmer's Manual", [Online]. Available: https://man7.org/linux/man-pages/man2/read.2.html (visited on 12/04/2020).

[25] P. J. Salzman, M. Burianm, and O. Pomerantz. (2007). "The Linux Kernel Module Programming Guide", [Online]. Available: https://tldp.org/LDP/lkmpg/2.6/html/x121.html (visited on 11/16/2020).

[26] (2020). "Adding a New System Call", [Online]. Available: https://www.kernel.org/doc/html/v4.12/process/adding-syscalls.html (visited on 12/04/2020).

[27] J. Corbet. (2009). "DebugFS", [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/debugfs.html (visited on 12/04/2020).

[28] (Apr. 2008). "SSHFS(1) User Commands", [Online]. Available: https://man7.org/linux/man-pages/man1/SSHFS.1.html (visited on 12/04/2020).

[29] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses", CoRR, vol. abs/1710.00551, 2017. arXiv: 1710.00551. [Online]. Available: http://arxiv.org/abs/1710.00551.

References

[30] (Jun. 2020). "GETOPT(3) Linux Programmer's Manual", [Online]. Available: https://man7.org/linux/man-pages/man3/getopt.3.html (visited on 01/19/2021).

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript", CoRR, vol. abs/1507.06955, 2015. arXiv: 1507.06955. [Online]. Available: http://arxiv.org/abs/1507.06955.

[32] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh", in S&P, Best Paper Award, May 2020. [Online]. Available: https://download.vusec.net/papers/trrespass_sp20.pdf.

[33] (Nov. 2017). "decode-dimms(1) User Commands", [Online]. Available: https://manpages.debian.org/testing/i2c-tools/decode-dimms.1.en.html (visited on 12/04/2020).

[34] (Mar. 2012). "DMIDECODE(8) System Manager's Manual", [Online]. Available: https://manpages.debian.org/stretch/dmidecode/dmidecode.8.en.html (visited on 12/04/2020).

[35] (2015). "BIOS Update Utility README (T540p)", [Online]. Available: https://download.lenovo.com/pccbbs/mobiles/gmuj16us.txt (visited on 11/16/2020).

[36] (2015). "BIOS Update Utility README (X230T)", [Online]. Available: https://download.lenovo.com/pccbbs/mobiles/gcuj22us.txt (visited on 11/16/2020).

[37] (2020). "Skulls - Thinkpad X230T". commit 664b1cf2162a5b1b648f47b1efa3e9378988ad4d, [Online]. Available: https://github.com/merge/skulls/blob/master/x230t/README.md (visited on 11/16/2020).

[38] (2018). "frser-duino". commit 6a8a98eab65a7ab2d79450cbb6df136da042a3a0, [Online]. Available: https://github.com/tomvanveen/frser-duino (visited on 12/04/2020).

[39] (2020). "FT232R USB UART IC Datasheet", Future Technology Devices International Ltd., [Online]. Available: https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf.

[40] (2020). "Flashrom". commit 74fd0300b8d4a58d3838963f60c9f16451d1db78, [Online]. Available: https://github.com/flashrom/flashrom (visited on 12/04/2020).

[41] (2020). "Fritzing", [Online]. Available: https://fritzing.org/ (visited on 01/19/2021).

[42] (2013). "MX25L3273E HIGH PERFORMANCE SERIAL FLASH SPECIFICATION", Macronix International Co., Ltd., [Online]. Available: https://www.mxic.com.tw/Lists/Datasheet/Attachments/7351/MX25L3273E,%203V,%2032Mb,%20v1.1.pdf (visited on 12/04/2020).

[43] (2015). "MX25L6473E", Macronix International Co., Ltd., [Online]. Available: https://www.mxic.com.tw/Lists/Datasheet/Attachments/7380/MX25L6473E,%203V,%2064Mb,%20v1.4.pdf (visited on 12/04/2020).

[44] (2013). "MX25L3206E DATASHEET", Macronix International Co., Ltd., [Online]. Available: https://www.mxic.com.tw/Lists/Datasheet/Attachments/7355/MX25L3206E,%203V,%2032Mb,%20v1.5.pdf (visited on 12/04/2020).

[45] (2014). "MX25L6406E DATASHEET", Macronix International Co., Ltd., [Online]. Available: https://www.mxic.com.tw/Lists/Datasheet/Attachments/7370/MX25L6406E,%203V,%2064Mb,%20v1.9.pdf (visited on 12/04/2020).

[46] (2020). "USB bus convert chip CH341", [Online]. Available: http://anok.ceti.pl/download/ch341ds1.pdf (visited on 12/04/2020).

[47] (2020). "UEFITool". commit d9642c53e7b873fcce858ec2bea577f099b966d5, LongSoft, [Online]. Available: https://github.com/LongSoft/UEFITool (visited on 12/04/2020).

[48] M. Seaborn and T. Dullien. (2015). "Exploiting the DRAM rowhammer bug to gain kernel privileges", [Online]. Available: https://www.cs.umd.edu/class/fall2019/cmsc818O/papers/rowhammer-kernel.pdf (visited on 11/16/2020).

Appendix

Contents

A File structures

A.1 0x01_kvmModule

- **kvmModule.c**: Implementation of the kernel module
- **kvmModule.h**: Headers and type definitions of the kernel module
- **Makefile**: File with instructions to automatically compile the module. Use the command ”make“ in the directory to do this

A.2 0x02_memlib

- **memoryInspect.c**: Implementation of the memory inspection functionality. This means in detail:
  - Create and destroy addrInfo data structures (used to handle information in the library)
  - Create addrInfo data structures from a maps file in procfs or a submitted array of virtual addresses
  - Add physical to virtual addresses from a pagemap file in procfs
  - Add host addressing information (when running in a VM) based on the procfs files created by kvmModule
  - Group addrInfo data structures by PFN (for inspection of continuous PFNs)
  - Print information about addrInfo data structures and KSM state of pages
- **memoryInspect.h**: Headers and type definitions of the memory inspection functionality (see above for details)
- **pfnInspect.c**: Implementation of the pfn inspection functionality. This means in detail:
  - Get the number how often a pfn is mapped from /proc/kpagecount
  - Get the flags of a pfn from /proc/kpageflags
- **pfnInspect.h**: Headers and type definitions of the pfn inspection functionality (see above for details)
- **util.c**: Several utility functions used by the other files, in detail:
  - Read an int64 from a file at a specified offset (for files like pagemap in procfs)
  - Convert a hex string to an integer
  - Read the content of a file to a dynamically allocated buffer
  - Comparator functions (required for qsort)

  - Get the PID of a VM process on the host from the guest (libvirt path from the host has to be mounted to the guest for this)

- **util.h**: Headers and type definitions of the utility functionality (see above for details)

A.3 0x03_memtool

- **memtool.c**: Implementation of the user tool based on memlib. In detail, the following features are supported:
  - Scan all memory mappings of a process with a submitted PID or the own process (the tool inspects its own memory mappings)
  - Add more information to the found pfns (based on the flags from /proc/kpageflags)
  - Get a PFN for a submitted GFN
- **Makefile**: File with instructions to automatically compile the tool. Use the command "make" in the directory to do this

A.4 0x04_hammerlib

- **afunc.c**: Implementation of functionality used to reverse engineer the address function and group addresses. In detail, this means:
  - Add a page to an address group based on measured access times
  - Add a page to an address group based on known address functions
  - Allocate a THP and populate a corresponding aInfo data structure from memlib
  - Automatically determine the threshold value between "fast" and "slow" memory accesses
  - Regroup an address group so addresses grouped wrong before are grouped correctly (grouping can be wrong when the groups have very few elements in the beginning)
  - Calculated address function mask based on address groups
  - Measure the number of memory banks in the system
  - Measure the effective size of physical continuous memory (how many bytes are written before the bank or row changed)
- **afunc.h**: Headers and type definitions of the reverse engineering functionality (see above for details)
- **asm.h**: Wrapper functions of assembler instructions
- **hammer.c**: Implementation of the rowhammer execution part, in detail:

  - Calculation of rowhammer candidates (sets of aggressor and victim rows) based on a submitted hammer pattern
  - Execution of a rowhammer attack on found candidates and check if there are bit flips in the victim rows

- **hammer.h**: Headers and type definitions of the rowhammer functionality (see above for details)
- **util.c**: Some utility functionality, in detail:
  - Construct and destruct items of several data structures used in hammertool
  - Measure the access time using rdtscp instruction or clock_gettime() function
  - Comparator functions (required for qsort)
  - Print and export data
  - Generate bit mask candidates
  - Calculate physical addressing components (row, bank, column) from a pfn
  - Print timing for continuously accessing one address (can be used to deduce the refresh period)
- **util.h**: Headers and type definitions of the utility functionality (see above for details)

A.5 0x05_hammertool

- **main.c**: Implementation of the main program which parses the command line options and calls other functions
- **Makefile**: File with instructions to automatically compile the tool. Use the command "make" in the directory to do this
- **plot.py**: Script to create plots based on the access time data exported by hammertool

A.6 0x06_hammertest

- **run.sh**: Script that can be executed on a system which should be tested. The following functionality is implemented:
  - Download the current data (hammertest.zip) and extract it
  - Collect several information about the system
  - Find the address functions and re-compile the tools if this is required
  - Execute the PoCs (rowhammer-test, RowhammerJS, TRRespass, hammertool)
  - Send the results to the specified server

- **hammertest**: Contains the source code and/or binaries of the tools that should be executed
- **server.sh**: Script that can be executed on the server to automatically store reports submitted by run.sh
- **Makefile**: File with instructions to automatically rebuild the zip file with the source code and executables. Use the command ”make“ in the directory to do this

A.7 0x07_hammeriso

- **airootfs**: Content of the root file system of the ISO image, this will be copied to the ISO root file system before any packages are installed
- **efiboot**: Boot loader for Unified Extensible Firmware Interface (UEFI) boot mode
- **syslinux**: Boot loader for BIOS (Legacy) boot mode
- **isolinux**: Boot configuration for syslinux when the system is booted from an optical disc
- **Makefile**: File with instructions to automatically build the iso file. Use the command “make” in the directory to do this
- **packages.x86_64**: List of packages that should be installed in the ISO image
- **pacman.conf**: Configuration for the package manager pacman which is used instead of the system-wide configuration when the packages are installed into the root file system of the ISO image

B Source code

The source code of the tools can be accessed at `http://rowhammer.xitokero.de/code_submit.zip`. In order to ensure the source code is the same as when this thesis was submitted, I added the `SHA256` hash of the zip file: `d0dea04c17c6afa28765ace410e8a4dc8982e56839e0e05e98475bb9b2f9bc1d` .

C Extended Listings

Some listings in this thesis are shorter versions without comments. This sections contains a more commented version of those listings.

```c
#include<stdio.h>

#include"hammerlib/afunc.h"
#include"hammerlib/hammer.h"

int main(int argc, const char **argv) {
    /*
     * Do one measurement of timing between two addresses before
     * deciding if the access is 'fast' or 'slow'
     */
    int nTries = 1;

    /*
     * Measure the amount of banks 1 time (If there are errors in the
     * measurement, this value can be increased. In that case, the
     * median will be taken as number of banks. If the median would
     * be two values, the smaller value is taken, not the mean value
     * of them)
     */
    int nChecks = 1;

    /*
     * Allow up to 50 addresses to be not added to any group to assume
     * the number of groups as correct.
     */
    int errtres = 50;

    /*
     * One measurement consists of 10 single measures. The median of
     * them is taken as real value. As stated above, no mean will be
     * calculated
     */
    int iter = 10;

    /*
     * Global counter for statistics export (it will be increased with
     * each exported access time statistic. This is used for the
     * filenames (aka accessTime-<num>.csv)
     */
    int accessTimeCnt = 0;

    /*
     * Set the verbose mode. The higher the more output will be there.
     */
    int verbose = 1;
```

```
46
47      /*
48       * Don't use clock_gettime() instead of the rdtscp instruction. If
49       * this is 1, clock_gettime() will be used instead.
50       */
51      int getTime = 0;
52
53      /*
54       * Run in virtual mode, don't translate virtual to physical
55       * address but do calculations on virtual addresses
56       */
57      int vMode = 1;
58
59      /*
60       * Do automatic scaling of the histogram plots and minimal values.
61       * If something in the detection of the treshold does not work,
62       * this can be used to change the minimum amount manually. The
63       * plots will match this value (one '#' will be equal to the
64       * submitted scale value)
65       */
66      int scale = -1;
67
68      /*
69       * There are always at least blockSize of continuous bytes in
70       * memory. On single channel setups, this is 8192, on dual
71       * channel it is 128. Because at this point it is not known
72       * if the system is single or dual channel, 128 should be used.
73       */
74      int blockSize = (1<<7);
75
76      /*
77       * Scan one transparent hugepage to find the addressing
78       * functions
79       */
80      int nInitSets = 1;
81
82      /*
83       * Don't measure the row size. If this is enabled, hammerlib
84       * tries to measure the actual number of continuous bits
85       * and use them. When running with at least verbosity
86       * level 1, there is some output of the found size.
87       */
88      int measureRowSize = 0;
89
90      /*
91       * Scan for masks with maximal 7 bits. The higher the number
92       * of maximal bits, the longer the scan will take. Based on
93       * the drama paper there should be a maximum of 7 bits
94       * required If no masks are found and the addresses seem
95       * to be grouped correctly, this value can be increased.
```

```
96          */
97          int maxMaskBits = 7;
98
99          /*
100          * Name of the file that should be used to export the found
101          * configuration
102          */
103         char *exportFilename = "export.cnf";
104
105         int banks = getBankNumber(nTries, nChecks, errtres, iter, &
                accessTimeCnt, verbose, getTime, vMode, scale);
106
107         /*
108          * Create a new group for the double amount of addresses
109          * (when there are some initial errors in calculation due
110          * to nearly empty groups, the addresses are added to
111          * higher groups. After some unifying steps, they are
112          * removed at a later point
113          */
114         addressGroups *aGroups = constructAddressGroups(banks * 2, blockSize);
115
116         int totalErrors = 0;
117
118         for(int i = 0; i < nInitSets; i++) {
119             printf("\rStarting set %3d of %3d", i + 1, nInitSets);
120             if(verbose >= 2) {
121                 printf("\n");
122             }
123             fflush(stdout);
124             totalErrors += scanHugepages(1, aGroups, verbose, banks, nChecks,
                    iter, &accessTimeCnt, getTime, vMode, i, scale, blockSize,
                    measureRowSize);
125         }
126
127         /*
128          * Truncate the address group to the number of banks. All
129          * unifying steps are done at this point and addresses that
130          * are grouped outside of the amount of banks should be
131          * removed
132          */
133         aGroups->nItems = banks;
134
135         maskItems *mItems = constructMaskItems();
136         findMasks(aGroups, maxMaskBits, mItems, verbose);
137
138         /*
139          * Check if the amount of found masks is equal to log_2(banks)
140          */
141         if(1L<<mItems->nItems != banks) {
142             printf("Number of masks does not match number of banks.\n");
```

```
143         return EXIT_FAILURE;
144     }
145
146     printf("Found addressing bitmasks:\n");
147     for(int i = 0; i < mItems->nItems; i++) {
148         printf("\t");
149         printBinaryPfn(mItems->masks[i], mItems->nItems);
150     }
151
152     exportConfig(exportFilename, banks, mItems);
153
154     return EXIT_SUCCESS;
155 }
```

Listing 24: Example of reverse-engineering the address functions with `hammerlib` (extended)

```
1  #include<stdio.h>
2
3  #include"hammerlib/afunc.h"
4  #include"hammerlib/hammer.h"
5
6  int main(int argc, const char **argv) {
7      /*
8       * Number of sets that should be used for the rowhammer attack.
9       * One set is equal to one THP (2MiB)
10      */
11     int nSets = 1;
12
13     /*
14      * Access the aggressors 1000000 times before checking the
15      * victim rows. In one refresh cycle of 64ms, ca. 300000
16      * accesses are possible which makes 600000 accesses in
17      * two refresh cycles (this is required to make sure that
18      * one complete refresh cycle is hammered at least). So,
19      * the value should be at least 600000. 600000 may be too
20      * less on some systems. Using 1000000 should be OK.
21      */
22     int nHammerOperations = 1000000;
23
24     /*
25      * Hammer every row that is possible in the specified
26      * number of sets. Don't skip. If this is set to a
27      * non-zero value, omitRows will be skipped so a bigger
28      * area of memory can be scanned in a shorter time.
29      * Mind that this may skip vulnerable rows as well.
30      */
31     int omitRows = 0;
32
33     /*
34      * Don't run the rowhammer test in multiprocessing mode
35      * (one subprocess per logical CPU core). Till now, I have
```

```
36          * not found any bit flips in multiprocessing mode but
37          * this could increase testing speed significantly. Test
38          * if you find bit flips in multiprocessing mode if you
39          * want and tell me if you found some :)
40          */
41         int multiprocessing = 0;
42
43         /*
44          * Name of the file that should be used to export the
45          * found configuration
46          */
47         char *importFilename = "export.cnf";
48
49         //Virtual mode
50         int vMode = 1;
51
52         //Verbose mode level 1
53         int verbose = 1;
54
55         int banks = 0;
56         maskItems *mItems = constructMaskItems();
57         importConfig(importFilename, &banks, &mItems);
58
59         /*
60          * Get the least significant bit set in the addressing masks
61          * and calculate the size of continuous blocks in bytes
62          */
63         long unifiedMask = 0;
64         for(int i = 0; i < mItems->nItems; i++) {
65             unifiedMask |= mItems->masks[i];
66         }
67
68         int blockSize = 0;
69         for(int i = 0; i < sizeof(long) * 8; i++) {
70             if((unifiedMask >> i) % 2 == 1) {
71                 blockSize = 1 << i;
72                 break;
73             }
74         }
75
76         if(blockSize > 4096) {
77             /*
78             * Allocation works only in entire pages, even if a row
79             * is 8192 (2 pages)
80             */
81             blockSize = 4096;
82         }
83
84         addressGroups *aGroups = constructAddressGroups(banks, blockSize);
85         for(int i = 0; i < nSets; i++) {
```

```
86          printf("\rStarting set %3d of %3d", i + 1, nSets);
87          fflush(stdout);
88          addHugepagesToGroups(1, aGroups, mItems, vMode, blockSize);
89      }
90      printf("\n");
91
92      long foundBitflips = 0;
93      foundBitflips += hammer(aGroups, nHammerOperations, verbose,
            HMASK_DOUBLESIDE, ROWNUM_DOUBLESIDE, omitRows, multiprocessing);
94      printf("Found %d hammers.\n", foundBitflips);
95
96      return EXIT_SUCCESS;
97  }
```

Listing 25: Example of executing a rowhammer attack with `hammerlib` (extended)

Eidesstattliche Erklärung

Ich versichere durch meine Unterschrift, dass ich diese Praxisarbeit mit dem Titel „Hammerkit–Toolset for Memory inspection and automatic Rowhammer detection" selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die ich wörtlich oder dem Sinn nach aus Veröffentlichungen entnommen habe, sind als solche kenntlich gemacht.

Diese Arbeit wurde bisher keiner anderen Prüfungsinstitution vorgelegt und auch noch nicht veröffentlicht.

Steinberg, 21. Januar 2021 ....

Ort, Datum Martin Heckel