WE DON'T NEED NO EXPLOITATION

Systematic Investigation of Process-based Rowhammer Exploitation

martin heckel

november 22, 2022

Martin Heckel: *We don't need no Exploitation*, Systematic Investigation of Process-based Rowhammer Exploitation

Academic supervisor: Prof. Dr. Florian Adamsky

Abstract

Rowhammer is a hardware vulnerability that enables an attacker to flip bits in memory without accessing them [1], resulting in the possibility of writing to memory locations that are not under the attacker's control. Rowhammer is based on spatial adjacency of the physical locations where the data are stored in the Dynamic Random-Access Memory (DRAM) Integrated Circuits (ICs) on the Dual In-line Memory Modules (DIMMs). Hence, exploitation is tricky and depends on multiple preconditions. Furthermore, there are mitigations like Target Row Refresh (TRR) implemented in current systems, so it is required to bypass these mitigations.

In the beginning, Rowhammer was known as a hardware vulnerability that could not be exploited reliably and was therefore seen as a rather academic problem. However, when Seaborn and Dullien [2] released an exploit that uses Rowhammer to flip bits in Page Table Entries (PTEs) to gain kernel-level privileges in memory, vendors started to release Basic Input/Output System (BIOS) updates to mitigate Rowhammer. Later, there were mitigations implemented directly in the DRAM hardware. Due to those mitigations, the initial Rowhammer exploits did not work anymore.

In the subsequent years, many publications related to Rowhammer mitigation approaches [3]–[5] and how to bypass these mitigations [6]–[20] were released. However, in many cases, the actual exploitability was not analyzed anymore. Often, only the number of bit flips or their offsets were analyzed: If a bit flips at offset n within a page, it is considered to be exploitable because it could be used for e.g. PTE exploitation if that page would store PTEs.

Current exploits rely on the page cache [11], PTEs [2], or on special memory mechanisms like Kernel Samepage Merging (KSM) [8] to escalate a user's privileges.

In this thesis, another approach based on the exploitation of memory mapped by processes (e.g. on the *stack* or *heap*) is introduced: If it is possible to use the memory mapping mechanisms of Linux in a way that a vulnerable page is mapped at a specified offset, it should be possible to execute a binary and afterwards flip a bit on that page which results, for example, in the inversion of a variable used in a condition. For example, that could be used to modify conditions on a binary with the Set User IDentification (SUID) bit set, e.g. by modifying *sudo* to permit access only when the password is incorrect.

Additionally, the automated testing system developed during my practical [21] and bachelor's [22] theses was improved. It was used on a new test setup to reproduce the results shown by Jattke, van der Veen, Frigo, *et al.* [17] on 60 DIMMs and to find DIMMs vulnerable to Rowhammer for further experiments. The number of bit flips identified in the experiments was much lower than described by the authors of the publication. Their results could not be reproduced.

With the experiments performed in this thesis, it is impossible to *remap* vulnerable pages at specified offsets reliably. For that reason, the approaches that were tested and tools that were

developed are presented. To the best of my knowledge, the approaches described in this thesis are not published yet if not noted otherwise. In detail, the following approaches, tools and results are introduced:

- The HybridGrouping approach enables faster grouping of addresses by DRAM bank without the requirement to access Page Frame Numbers (PFNs) or use Transparent HugePages (THPs).
- It is shown that the vast majority (more than 97 %) of the pages freed by a parent process are allocated by a child process spawned directly afterwards when Central Processing Unit (CPU) pinning is enabled.
- The offsets of the remappings between the parent and child process are analyzed.
- AllocTrace, a kernel module to dump the PFNs currently in the buddy allocator's orders and per-CPU lists of the different logical CPU cores, is introduced.
- MemCP, a multi-stage copying approach to reduce memory noise while creating a snapshot of the files provided by AllocTrace, is presented.
- Using AllocTrace and MemCP, the correlation between PFNs within the buddy allocator at different times and within the parent and child process is analyzed.

Zusammenfassung

Rowhammer ist eine Hardware Schwachstelle, die es einem Angreifer ermöglicht, Bits im Dynamic Random-Access Memory (DRAM) zu flippen, ohne direkt darauf zuzugreifen [1]. Dadurch bekommt der Angreifer die Möglichkeit, effektiv in Speicherbereiche zu schreiben, auf die er keinen Zugriff hat. Rowhammer setzt räumliche Nähe von den physischen Speicherstellen, an denen die Daten auf den DRAM Integrated Circuit (IC) gespeichert werden, voraus. Aus diesem Grund ist das Ausnutzen von Rowhammer nicht trivial und hängt von mehreren Vorbedingungen ab. Zusätzlich gibt es auf aktuellen Systemen Mitigationen wie Target Row Refresh (TRR), die zuerst umgangen werden müssen.

Ursprünglich wurde Rowhammer als Hardware Schwachstelle, die nicht zuverlässig ausgenutzt werden kann, angesehen und aus diesem Grund als akademisches Problem behandelt. 2015 veröffentlichten Seaborn und Dullien [2] ein Exploit, in dem Rowhammer verwendet wird, um Bits in Page Table Entries (PTEs) zu flippen. Dadurch ist es einem Angreifer möglich, Speicherzugriff auf Kernel Ebene zu erlangen. Nach der Veröffentlichung dieses Exploits begannen Hersteller, Basic Input/Output System (BIOS) Updates bereitzustellen, die Rowhammer mitigieren. Später wurden entsprechende Mitigationen direkt in der DRAM Hardware implementiert. Durch diese Mitigationen funktionierten vorher veröffentlichte Exploits nicht mehr.

In den folgenden Jahren gab es viele Veröffentlichungen zu Rowhammer Mitigationen [3]–[5] und Möglichkeiten, diese zu umgehen [6]–[20]. In vielen Fällen wurde allerdings die tatsächliche Möglichkeit zum Exploiten nicht untersucht. Häufig wurde nur die Anzahl der Bit Flips sowie deren Offset analysiert: Wenn ein Bit an Offset n innerhalb einer Page flippt, wird es als exploitbar betrachtet, wenn es für z. B. PTE Exploitation verwendet werden könnte, sofern PTEs in der Page gespeichert wären.

Aktuelle Exploits nutzen den Page Cache [11], PTEs [2], oder besondere Mechanismen zur Speicherverwaltung wie Kernel Samepage Merging (KSM) [8] aus, um die Berechtigung eines Benutzers auszuweiten.

Im Rahmen dieser Masterarbeit wird ein anderer Ansatz zum Exploiten des Speichers, der von normalen Prozessen alloziert wird (z. B. auf dem *Stack* oder *Heap*), vorgestellt. Sofern es möglich ist, die Mechanismen zur Speicherverwaltung in Linux so zu nutzen, dass eine anfällige *Page* innerhalb des Kindprozesses an einem bestimmten Offset gemappt wird, sollte es möglich sein, den Bit Flip anschließend erneut zu provozieren. Somit tritt der Speicherfehler in der Page des Kindprozesses auf. Dadurch kann der Programmablauf des Kindprozesses manipuliert werden, indem eine in einer Bedingung verwendete Variable modifiziert wird. Dieser Ansatz könnte auf ausführbare Dateien, bei denen das Set User IDentification (SUID) Bit gesetzt ist, angewandt werden, z. B. um *sudo* so zu manipulieren, dass nur Zugriff gewährt wird, wenn das eingegebene Passwort nicht korrekt ist.

Zusätzlich werden einige Verbesserungen an dem automatisierten Testsystem, das im Rahmen meiner Praxisarbeit [21] und Bachelorarbeit [22] vorgestellt wurde, beschrieben. Das Testsystem wurde anschließend benutzt, um die Ergebnisse von Jattke, van der Veen, Frigo u. a. [17] auf 60 Dual In-line Memory Modules (DIMMs) zu evaluieren. Dadurch sollten außerdem DIMMs gefunden werden, die sehr anfällig für Rowhammer sind und in weiteren Experimenten verwendet werden können. Die Anzahl der im Rahmen dieser Experimente identifizierten Bit Flips war deutlich niedriger als von den Autoren beschrieben. Aus diesem Grund konnten die veröffentlichten Ergebnisse nicht reproduziert werden.

Mit den Experimenten, welche in dieser Arbeit vorgestellt werden, ist es nicht möglich, anfällige Pages stabil zu *remappen*, sodass diese immer an einem spezifizierten Offset innerhalb des Kindprozesses gemappt werden. Stattdessen werden die Ansätze und Programme, die im Rahmen dieser Masterarbeit erstellt und evaluiert wurden, vorgestellt. Sofern nicht anders angegeben ist mir zum jetzigen Zeitpunkt nicht bekannt, dass die in dieser Masterarbeit vorgestellten Ansätze bereits veröffentlicht wurden. Im Detail werden die folgenden Ansätze, Tools und Ergebnisse vorgestellt:

- Der HybridGrouping Ansatz ermöglicht es, virtuelle Adressen schnell auf Basis der *Bank* zu gruppieren, auf der diese im physisch im DRAM liegen. Dieser Ansatz funktioniert ohne Zugriff auf Page Frame Numbers (PFNs) oder die Verwendung von Transparent HugePages (THPs).
- Es wird gezeigt, dass der Großteil (mehr als 97 %) der Pages, die von einem Elternprozess freigegeben werden, von einem direkt danach gestarteten Kindprozess erneut alloziert werden, sofern Central Processing Unit (CPU) Pinning aktiviert ist.
- Die Offsets der *Remappings* zwischen dem Eltern- und Kindprozess werden analysiert.
- AllocTrace, ein Kernelmodul zum Ausgeben der PFNs, die sich aktuell auf den verschiedenen Ordnungen des Buddy Allocators oder in den per-CPU Listen der verschiedenen logischen CPU Kerne befinden, wird vorgestellt.
- MemCP, ein Ansatz zum mehrstufigen Kopieren, wird vorgestellt. Durch diesen Ansatz ist es möglich, die Störungen durch Speicherzugriffe zu reduzieren, während eine Kopie der von AllocTrace bereitgestellten Dateien durchgeführt wird.
- Unter Verwendung von AllocTrace und MemCP wird die Korrelation zwischen den PFNs, die vom Eltern- oder Kindprozess alloziert wurden, und den PFNs, die sich zu verschiedenen Zeitpunkten in den Listen des Buddy Allocators befinden, untersucht.

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 [23]

Acknowledgements

While performing the experiments and writing this master's thesis, I received much support and assistance.

While the experiments were performed, there were many situations where the results looked not as expected, and I could not explain them. Especially but not only in these situations, I had the possibility to discuss the experiments, results, possible explanations, etc., with several people. At this point, I want to thank Prof. Dr. Florian Adamsky, Adrian Märtins, and Dominik Thalhammer for the many discussions we had in the last months. Lots of the approaches presented in this thesis emerged during these discussions.

In order to increase the quality of the results, there were some experiments for which measurements on additional systems were useful to evaluate the hypotheses more completely. I want to thank Prof. Dr. Florian Adamsky, Sebastian Pahl, and Dominik Thalhammer for performing some of the experiments on their systems and, thereby, providing more data to evaluate the hypotheses.

In this thesis, there is an experiment where 60 Double Data Rate 4 (DDR4) Dual In-line Memory Modules (DIMMs) were tested for Rowhammer susceptibility. I want to thank the research group System and Network Security (SNS) and the Institute for Information Systems (iisys) at Hof University for providing the hardware[1] that made this experiment possible.

Finally, I want to thank Nico Bretschneider, Johanna Heckel, Michelle Madeline Krebs, Adrian Märtins, Angelina Scheler and Peter Siedentopf for proofreading this thesis.

[1] Two computers and 60 DDR4 DIMMs

Contents

Contents

List of Figures

List of Tables

Listings

Abbreviations

BIOS	Basic Input/Output System
CPU	Central Processing Unit
DDR3	Double Data Rate 3
DDR4	Double Data Rate 4
DIMM	Dual In-line Memory Module
DoS	Denial of Service
DRAM	Dynamic Random-Access Memory
ECC	Error Correction Code
EEPROM	Electrically Erasable Programmable Read-Only Memory
IC	Integrated Circuit
iisys	Institute for Information Systems
IPC	Inter-Process Communication
KSM	Kernel Samepage Merging
KVM	Kernel-based Virtual Machine
L1	Level 1
L2	Level 2
L3	Level 3
LLC	Last Level Cache
MAC	Media Access Control
MC	Memory Controller
MMU	Memory Management Unit
NIC	Network Interface Controller
OOM	Out Of Memory
OS	Operating System
PFN	Page Frame Number
PGD	Page Global Directory
PUD	Page Upper Directory
PMD	Page Middle Directory
PTE	Page Table Entry
PoC	Proof of Concept
RDMA	Remote Direct Memory Access
SGX	Software Guard Extensions
SMASH	Synchronized MAny-Sided Hammering
SNS	System and Network Security
SPD	Serial Presence Detect
SUID	Set User IDentification
THP	Transparent HugePage
TLB	Translation Lookaside Buffer

TRR	Target Row Refresh
TSC	TimeStamp Counter
pTRR	pseudo Target Row Refresh
VM	Virtual Machine

1 Introduction

In 2014, Kim, Daly, Kim, *et al.* [1] demonstrated that many Double Data Rate 3 (DDR3) Dual In-line Memory Modules (DIMMs) are susceptible to memory errors. 110 out of the 129 DDR3 DIMMs tested were prone to errors induced by specific access patterns. It was possible to flip bits in memory without accessing them by reading other data from memory. They mentioned that such disturbance errors can be exploited but did not provide a Proof of Concept (PoC). Therefore, their results were not relevant for real-world use cases and were seen as an academic problem then. Later, the attack they introduced got known as Rowhammer.

One year later, Seaborn and Dullien [2] provided a PoC that exploits Rowhammer to flip bits in Page Table Entries (PTEs), thereby gaining kernel-level privileges in memory. It was shown that Rowhammer was not a purely academic topic anymore but had an impact on the everyday usage of Dynamic Random-Access Memory (DRAM), which is an integral part of many computer systems.

In the following years, the topic of Rowhammer gained more attention, and many publications presented new offensive [6]–[20] and defensive [3]–[5] approaches. It was shown that Rowhammer could be exploited from the JavaScript code running in a browser [6], [19], on mobile devices [9], across different Virtual Machines (VMs) [8], and over the network [12], [13]. It was also shown that Rowhammer could be used to read memory without accessing it [16].

1.1 Objective of this Thesis

In the scope of my practical [21] and bachelor's [22] theses, two novel attacks that increase the amount of bit flips found in a given time by a hundredfold were introduced. Due to a lack of Double Data Rate 4 (DDR4) Hardware susceptible to Rowhammer at the time of writing, these attacks were evaluated on DDR3 only. The market share of DDR3 DRAM has decreased strongly in the last few years. Therefore the relevance of the evaluation described in these theses has also decreased. For that reason, one objective of this thesis is to use a system with DDR4 DRAM and repeat the evaluation on that system. In the first step, the results presented in [17] should be reproduced in order to find DIMMs that are susceptible to Rowhammer.

Some approaches exploit the code pages stored in the page cache [11], use PTEs [2], or specific memory mechanisms like Kernel Samepage Merging (KSM) [8] to escalate a user's privileges using Rowhammer. However, to the best of my knowledge, there is no exploitation technique that flips bits in the memory allocated by normal processes to achieve a local privilege escalation.

The following research questions are addressed in this thesis:

RQ 1: Can the results presented in [17] be reproduced on 60 DDR4 DIMMs?

RQ 2: Is it possible to exploit Rowhammer to modify the memory mapped by processes by using the memory mapping mechanisms of the Linux kernel?

Because none of the tested systems with DDR4 DRAM is susceptible to Rowhammer (there are a couple of bit flips on some systems, but not enough to perform a severe evaluation), the evaluation of the approaches introduced in [21], [22] is not performed. The partly automated experimental setup and the results of the search for affected modules are presented in this thesis.

Due to time limitations, it was not possible to get the complete process-based exploitation approach running in the scope of this thesis. Therefore, the current state is described instead. A new approach of mapping virtual addresses to physical *banks* in memory is introduced. Additionally, some insights into the memory allocation done when a process starts are shown.

1.2 Structure of this Thesis

At the beginning of this thesis, some essential background topics are explained in Section 2.

Afterwards, the experimental setup and the results of the measurements to find vulnerable DDR4 modules are described in Section 3. Additionally, some experiments on the granularity of the TimeStamp Counters (TSCs) of the CPUs are described.

Section 4 describes the approach of exploiting Rowhammer using the memory allocation of normal processes in detail. As a part of that, a new approach of mapping virtual addresses to physical bank numbers is introduced. Next, the internal mechanisms of page allocation are analyzed in detail.

A short overview of related publications is shown in Section 5. Section 6 contains some ideas that could be researched in the future. Finally, in Section 7, this thesis is concluded.

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

2 Background

This section contains background information required to understand the rest of this thesis. The information within this section refer to the amd64 architecture.

2.1 Memory Allocation in Linux

In most modern Operating Systems (OSs), memory is virtualized. Therefore, the addresses used by applications or the kernel are not the same addresses where the data is stored [24].

With virtualized addresses, it is not required to split the physical address space, shared among all processes and the kernel, so that spatial requirements are met. When an array (or a string) is allocated in a low-level programming language like *C*, the addresses have to be in a continuous block to address the array's single elements by using the start address and the offset. Even though high-level programming languages abstract that fact from the programmer, they rely on the same principles as C, so they also require continuous blocks of virtual memory.

Another advantage of virtualized addresses is that they can be used for *paging* [25], so it is possible to move some blocks of memory to the hard drive when there is not much free physical memory. Then, if the block is accessed the next time, it can be loaded from the hard drive and restored to physical memory before access. That procedure is implemented in the kernel and transparent to the process that accesses the memory.

Virtual memory requires some kind of translation between virtual and physical addresses. Because these translations have to be calculated somehow, memory addresses are not handled one by one where each byte is mapped individually but in blocks, called *pages*, to reduce the required number of mapping resolution operations. There is a tradeoff between the required number of calculations for address resolution and the granularity in which memory can be mapped. The *page size* defines the size of blocks managed by one mapping and specifies the granularity vs expense tradeoff. On x86 Linux, it is defined to be 4 KiB.

In order to map virtual addresses to physical addresses, *page tables* consisting of *PTEs* are used. One PTE stores the mapping between the virtual address and the physical address of a page. In order to use addresses within a page, the page mapping is resolved using PTEs first. Afterwards, the offset of the address within the page is taken from the virtual address. Therefore, the offset for the virtual and the physical address is always the same. The base address of the physical address is called Page Frame Number (PFN), as shown in Figure 1.

In contrast to the simplified version of the mapping of pages shown in Figure 1, the process is more complex, and there are multiple layers of tables [24]: *Page Global Directory (PGD)*, *Page Upper Directory (PUD)*, *Page Middle Directory (PMD)*, and *PTE*. A virtual address is split into

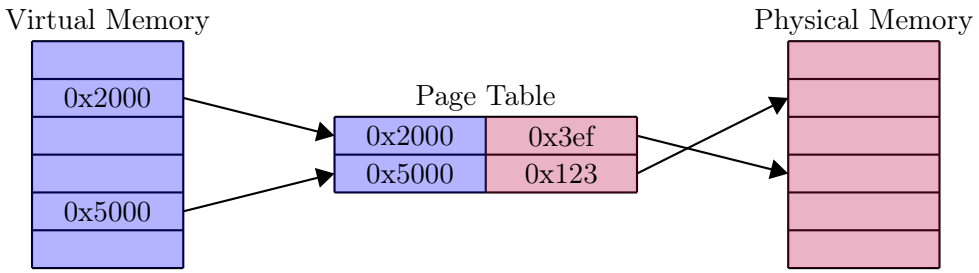


Figure 1: Simplified presentation of the mapping between virtual and physical addresses using page tables. Note that the virtual address's last 12 bit (three hexadecimal digits) are always zero. That is the case because the page size is 4 KiB which requires 12 bit for addressing. The address depicted as the physical address is the PFN (it does not contain that last 12 bit). The physical address is calculated using the PFN and appending the last 12 bit from the virtual address (`0x2042` would be mapped to `0x3ef042` in the depicted example). The image is based on [26].

several parts that correspond to the offset within the corresponding layer of the page tables, as shown in Figure 2.

The calculation of the physical address mapped to a virtual address is done by the Memory Management Unit (MMU), a part of the Central Processing Unit (CPU) in many modern systems. On the other hand, the translation between physical addresses and spatial locations, which are used to address data in DRAM (called *spatial address* in this thesis), is done by the Memory Controller (MC).

Because calculating a physical address from a virtual address is rather complex, the Translation Lookaside Buffer (TLB) is used to cache the last resolved mappings. Otherwise, it would be required to perform a complete lookup, as shown in Figure 2, for each memory access. According to [24], processes tend to have a locality of references, e.g. many memory accesses are performed on a small number of pages. Therefore, using a TLB can decrease the number of mapping resolutions. The mapping for each accessed page is loaded from the TLB. Only when there is a TLB miss, so the mapping for the page is not cached, it is resolved using the procedure depicted in Figure 2.

There are some applications where a considerable amount of (optionally continuous) virtual memory is required, e.g. for loading the content of a file to a buffer. In order to reduce the number of pages that have to be mapped (and resolved), the last 9 bit of the mapping can be added to the offset part, e.g. there is only one mapping for $2^9 = 512$ pages. The offset within that page is not 12 bit but 21 bit, so the page has a size of $2^{21} = 2097152$ B, equivalent to 2 MiB. These special pages are called Transparent HugePage (THP). If not specified otherwise, they are allocated automatically by the kernel if a memory mapping is big enough that the usage of a THP is considered beneficial. It should be noted that the THP, due to an offset of 21 bit, has to be aligned properly in virtual and physical address space (e.g. the last 21 bit of the first address of the THP have to be 0, the THP has to be allocated on a 2 MiB border).

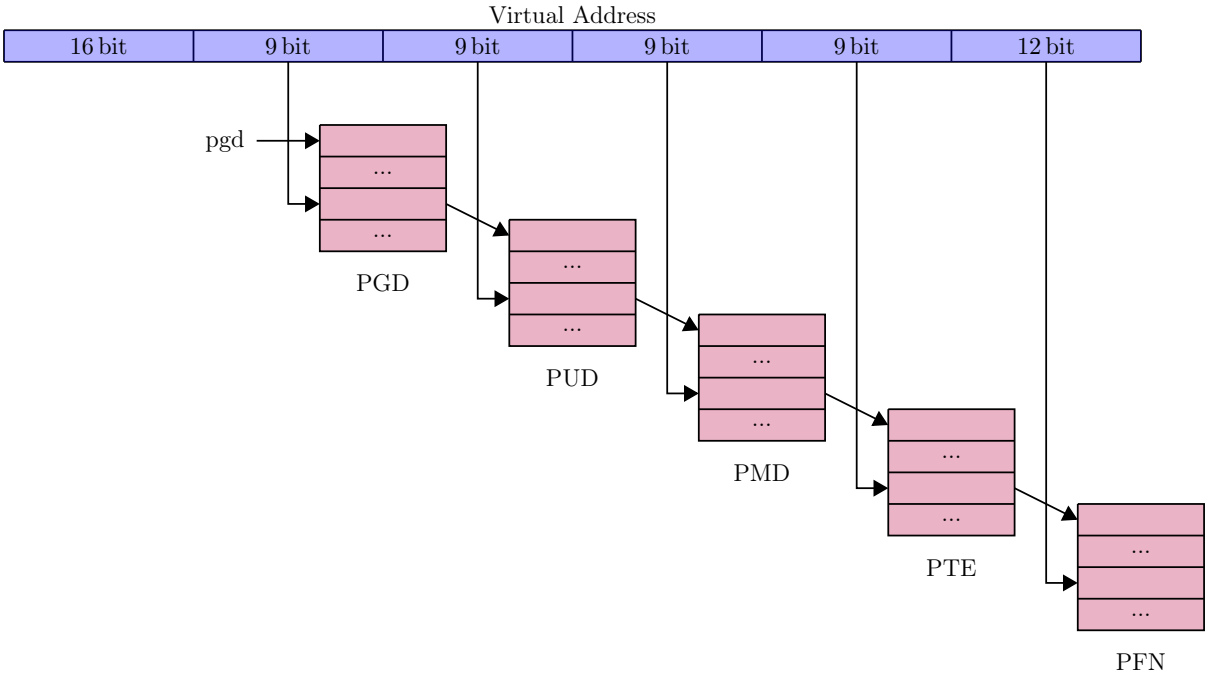


Figure 2: Detailed presentation of address resolution using page tables. Each process has a list of PGDs stored in *mm_struct->pgd* and depicted as *pgd* in the figure. The first 16 bit of the virtual address are the sign extension and currently not used for address mapping [27]. The next 9 bit specify an entry in the list of PGDs. That entry references another list of PUDs. The following 9 bit of the virtual address specify a PUD which references a list of PMDs. The next 9 bit of the virtual address specify the entry in the list of PMDs, which references to a list of PTEs The next 9 bit specify an entry in the list of PTEs that references to a PFN. The last 12 bit of the virtual address specify the offset within the page specified by the PFN. The sizes of the separate parts were measured using the approach described in Section A.1.

Besides the implicit and automatic allocation of THP, it is possible to use *madvise()* [28] with the advice value *MADV_HUGEPAGE* or *MADV_NOHUGEPAGE* to explicitly use or not use THP for the specified address range. Because pages are not allocated at the time of calling *mmap* [29] but at the time of the first write access, they are directly allocated as THP if *madvise()* is called before first writing to them.

When a page is allocated, the kernel adds entries to the according structures within the page tables to make the mapping between a virtual and a physical address accessible. However, the physical page that should be mapped has to be selected before that is possible. Therefore, the kernel must keep track of which pages are currently free to allocate them.

In Linux, that is not implemented with a linear list of free pages but with the *buddy allocator* [24]. That buddy allocator handles lists of blocks of 2^n pages, where n is called the *order* of the block. The buddy allocator on this system[2] uses orders 0 (blocks of 1 page) up to 10 (blocks of 1024 pages).

When all pages within an order are allocated (there are no free pages left), pages of a higher order can be split (e.g. one page of order 1 can be split into two pages of order 0). Both split blocks are marked as *buddies*. When both buddies are free at the same time, they will be merged again to a higher-order page. That principle can be chained over multiple orders (e.g. one page of order 8 is split into two pages of order 7, these two pages are split into 4 pages of order 6, two of those pages are split into four pages of order 5, etc.)

That approach brings the benefit of reduced fragmentation because the buddies of small allocations are merged again, resulting in bigger blocks of physically continuous pages available.

Because the buddy allocator is shared for the whole system (e.g. all logical CPUs access the same lists of free pages), the buddy allocator requires locking mechanisms to avoid multiple CPUs modifying the data structures simultaneously. In order to reduce that bottleneck, additional *per-CPU lists* of free pages are used as a cache between the page allocation mechanism and the buddy allocator.

When a page is freed, it is added to the per-CPU list accordingly, so it is not required to lock the buddy allocator. For page allocation, the same principle applies: The page is taken from the per-CPU list, so the buddy allocator does not have to be locked in that case either.

However, there are cases where the per-CPU list is insufficient: When a page should be allocated, but the list is empty, the buddy allocator is used to get a batch of pages, which requires locking the allocator for that allocation. The pages not required for the requested allocation are added to the per-CPU list so subsequent allocations can be handled with the pages in the list again. A similar principle applies when too many pages are in the list: Again, the buddy allocator is locked, and the pages from the list are returned to the buddy allocator.

In addition to the buddy allocator, the Slab allocator [24] is implemented on top of the buddy allocator. The Slab allocator allocates pages using the buddy allocator and provides them for allocating kernel objects. Since allocation and initialization of kernel objects would require much time when done directly via the buddy allocator, the Slab allocator keeps the allocated memory of the objects in an initialized state after they were freed. Thereby, the still allocated and pre-initialized memory can be used again when the next object of the same type is requested, which reduces the access time significantly. See Figure 3 for an overview of the memory allocation process.

[2] amd64 architecture with 32 GiB of DRAM

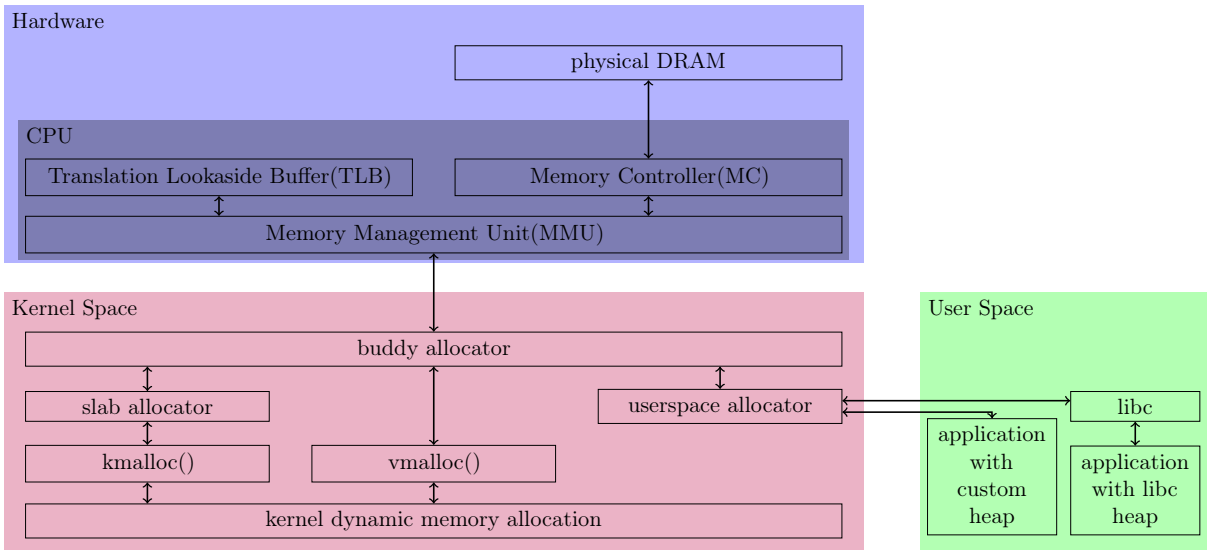


Figure 3: Overview of the memory management of Linux. All allocation mechanisms are based on the buddy allocator. Different higher-level mechanisms, e.g. the Slab allocator or *vmalloc()*, abstract the usage of the buddy allocator. The MMU does the mapping between virtual and physical addresses. The mapping between physical and spatial addresses is done by the MC. The image was initially created by Prof. Dr. Florian Adamsky at HAW Hof.

2.2 DRAM

A DRAM *cell* consists of a capacitor and a transistor, as shown in Figure 4.

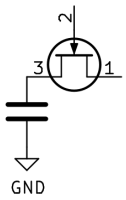


Figure 4: Single DRAM cell. One cell consists of a capacitor and a transistor. The image was created by myself as part of the practical thesis [21].

The capacitor stores the logical state in the form of an electrical charge. Depending on the electrical design of the cell, a discharged capacitor can be interpreted as logical 1 or logical 0. The cell is activated by applying a current to the pin depicted as 2 on the transistor. In that case, the transistor conducts and logically connects the pins depicted as 1 and 3.

In order to read the value that is currently stored in the cell, the voltage at the pin depicted as 1 is measured when the cell is activated. Writing to the cell works similarly: A voltage is applied at pin 1 while the cell is activated. If the applied voltage is V_{cc}, the capacitor is charged. If it is GND, the capacitor is discharged.

Reading the value of a DRAM cell discharges the capacitor. Therefore, it is required to restore the previous value after reading it (e.g. recharge the capacitor when it was charged before). Because the capacitor leaks charge over time, it is required to periodically recharge the cell, even if it was not accessed. On typical DDR3 and DDR4 systems, the DRAM cells are refreshed every 64 ms [30].

These DRAM cells are organized in an array of *rows* and *columns*, as shown in Figure 5. The access is always performed per row, meaning a row can only be activated completely. A row is activated by applying a voltage on the corresponding pin at the header labelled *Row select* in Figure 5. Depending on the access type (read or write), the charge of the capacitors can be measured or set at the pins labelled *Data*.

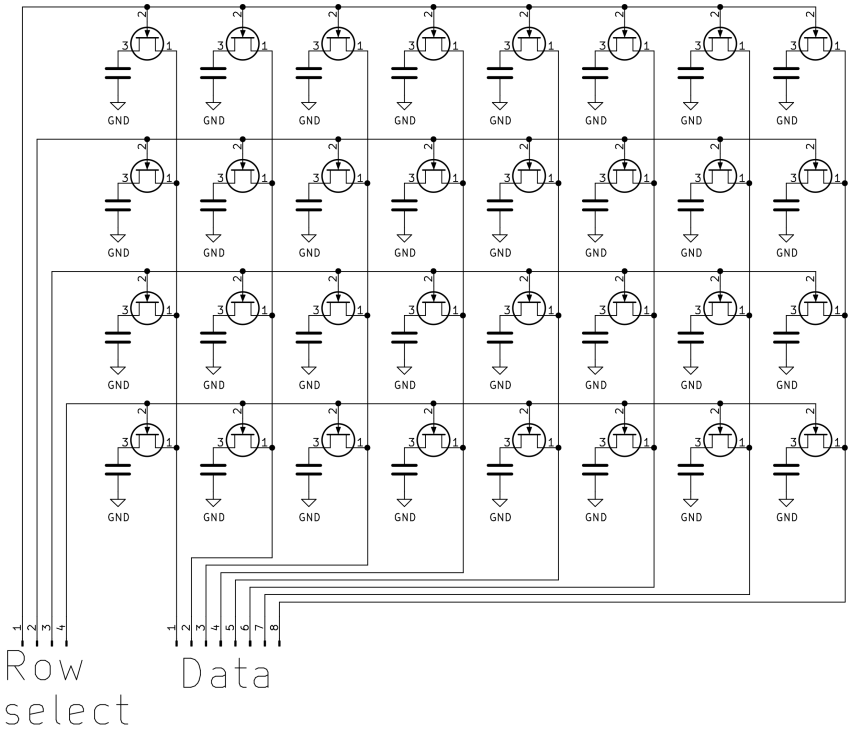


Figure 5: Array of DRAM cells. It is only possible to activate entire rows on the pins labelled *Row select*. When a row is activated, each cell within that row is electrically connected to one of the pins labelled *Data*. At these pins, the cells can be accessed as described for single cells, e.g. the charge can be measured to read the cells, or a charge can be applied to write to the cells. The image was created by myself as part of the practical thesis [21].

Because reading a row from the DRAM array destroys the data within the array, an additional buffer is required to cache the data until it is written back into the array. Since that buffer caches an entire row, it is called *row buffer*. There are amplifiers between the row buffer and the DRAM array. These components (DRAM array, amplifiers, and row buffer) are called a bank, as shown in Figure 6. Typically, one row within a bank has a size of 8 KiB so that it can fit two pages with a size of 4 KiB each. However, Pessl, Gruss, Maurice, *et al.* [7] showed that pages are not always

located in one bank but can be distributed over multiple banks.

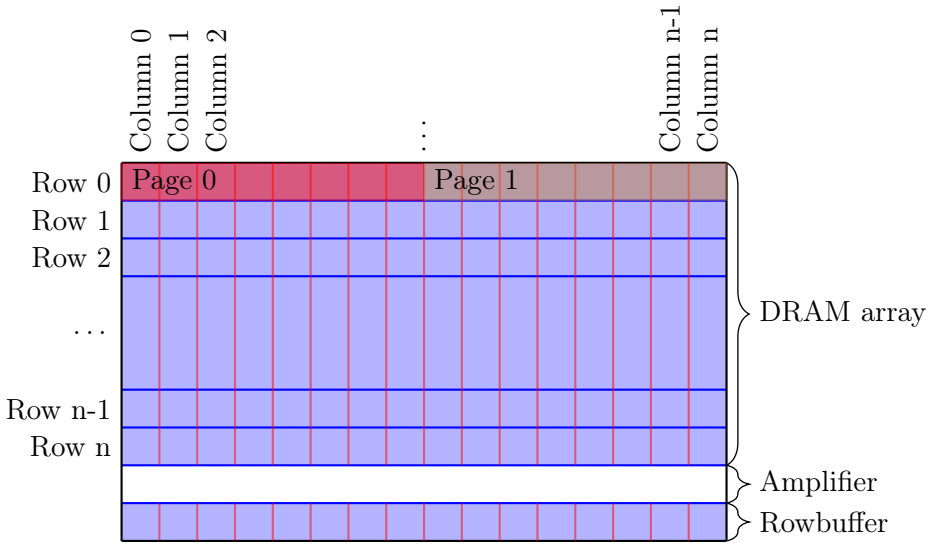


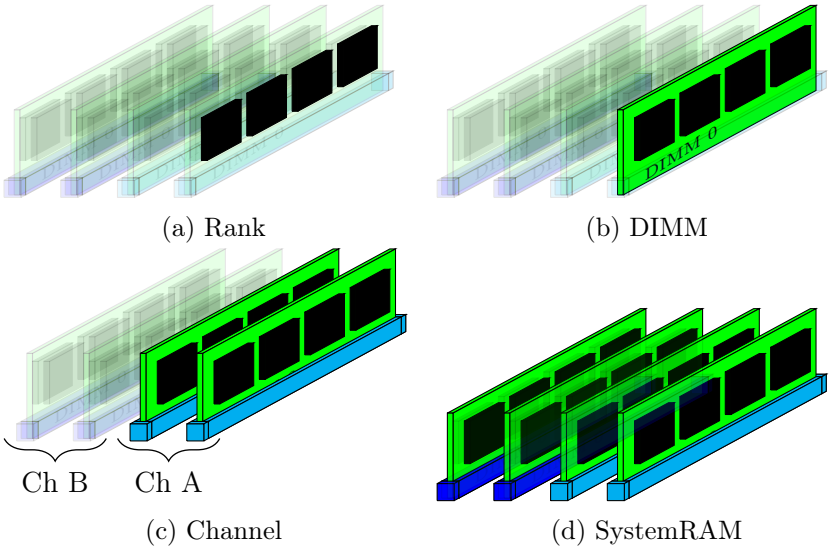
Figure 6: DRAM Architecture: Bank. One DRAM bank consists of an array of DRAM cells, amplifiers, and a row buffer. Typically, one row has a size of 8 KiB so that it can fit two pages of 4 KiB. Depending on the addressing, two following pages can be in a row (as shown in the image). It is also possible that one page spans across multiple banks, which is often the case in multi-channel setups. As in the DRAM array depicted in Figure 5, it is only possible to activate entire rows. When a row should be read, it is activated, and its content is written into the row buffer. If a write operation is performed, the content of the row buffer is written into the row. The image was created by myself as part of the practical thesis [21].

DRAM banks are distributed over the Integrated Circuits (ICs) that form a *rank*. There are layouts where one IC contains one bank, but there are also cases where one IC contains multiple banks or one bank is split across multiple ICs. Figure 7a shows one DRAM rank.

One or more ranks are part of a *DIMM* which is shown in Figure 7b. One or multiple DIMMs are at a bus connected to the CPU called *channel* shown in Figure 7c. There are systems with one or multiple channels. Everything together is the physical DRAM of the system shown in Figure 7d. Figure 7 depicts an overview of the different levels of the DRAM architecture described before.

As shown above, there is a row buffer with the size of a single row in each bank. When a row is read, its content is loaded into the row buffer and, at the same time, destroyed in the DRAM array. Afterwards, the data is not restored immediately but stays in the row buffer until another row is accessed. When that happens, the row buffer would be overwritten with the other row's content, resulting in a loss of the data currently stored in the row buffer (which was destroyed in the DRAM array when loaded into the row buffer). Therefore, the row buffer's content must be written back to the DRAM array before another row can be loaded into the row buffer.

There are two cases when data is read from DRAM: The requested row can already be in the row buffer, which is called *row hit*, or it can not be in the row buffer, which is called *row conflict*.



(a) Rank (b) DIMM

(c) Channel (d) SystemRAM

Figure 7: DRAM Architecture overview. The images were created by myself as part of the practical thesis [21].

When there is a row hit, the data can be directly returned without accessing the DRAM array. In the case of a row conflict, the current data in the row buffer has to be written first into the DRAM array. After that, the requested data is loaded into the row buffer. Then, the data can be returned from the row buffer.

2.3 CPU Cache

In order to increase the speed of accessing data, it is not always required to request it from DRAM, but it can be fetched from the CPU cache if stored in the cache. When read access to an address occurs, it is first checked if the data is already in the CPU cache (*cache hit*). If that is the case, the data is directly returned from the cache, and there is no access to the system DRAM. On the other hand, if the data is not in the cache (*cache miss*), the data is requested from DRAM. As described before, there can be a row hit or a row conflict in DRAM. In the case of a row hit, the data is returned faster than in the case of a row conflict.

Due to the differences in timing, it is possible to measure the access time and derive whether it was a cache hit, row hit, or row conflict based on the measured time.

In current CPUs, there are multiple levels of cache: Typically, the Level 1 (L1) and Level 2 (L2) caches belong to a CPU core and store only data accessed by that core. The Level 3 (L3) cache, also known as the Last Level Cache (LLC), is shared between all CPU cores [31]. There are different cache policies that specify whether data in one cache level must or must not be in other cache levels. Often, there is the *inclusive policy*, which means that data stored in the lower level

caches has to be stored in the higher level caches as well (e.g. data that is in the L1 cache has to be in the L2 cache and LLC as well).

2.4 DRAM Address Functions

As described in Section 2.1, virtual addresses used by processes or the kernel are translated to physical addresses by the MMU. However, DRAM does not use physical addresses directly but is addressed spatially, e.g. by specifying the channel, DIMM, rank, bank, row, and column of the data. That translation between physical and spatial address is done by the MC that is integrated into the CPU in many modern systems.

The mapping between virtual and physical addresses can be obtained using the file */proc/self/pagemap* [32] provided by the Linux kernel. However, to get the mapping from the physical address to a spatial one, it is required to know the mapping mechanisms implemented in the MC, which are not publicly available in many cases.

Often, it is not essential to get all spatial information, but some information can be abstracted. For example, on a system with 16 banks in total, it might be sufficient to know the number of the bank. It would not be required to have information about the rank in which the bank is organized, the DIMM the rank is on, and the channel the DIMM is connected to, as long as all banks have a continuous number (e.g. bank 0 to bank 15).

As described in Section 2.3, it is possible to use the access timing information to derive whether an access was a cache hit, a row hit, or a row conflict. When two addresses are accessed alternately and removed from the CPU cache before each access, there are only two possibilities: The access can be fast (which would indicate a row hit) or slow (which would indicate a row conflict). Since DRAM needs to be refreshed and other processes access the DRAM simultaneously, there is some noise in the measurement. However, when that measurement is repeated several times, there are clear timing differences between two addresses accessed with row hits or row conflicts.

In a spatial view, a row hit means that the addresses are either in different banks (because each bank has its own row buffer that stores the row that was last accessed) or at the same bank and in the same row (because the row buffer of that bank stores always the same row). When there is a row conflict, the addresses have to be on the same bank and in different rows because access to another row requires the current content of the row buffer to be written back to the DRAM array before the next row can be loaded into the row buffer.

When there are enough physical addresses grouped by spatial banks, it is possible to calculate the address function for the banks: The number of banks is a power of 2, so there are $log_2(n)$ bits required to address n banks. Each bit of the bank number is calculated by applying *XOR* to the physical address and the *address mask*. Afterwards, *XOR* is applied to that value: when there is an even number of ones, the result is 0. If the number of ones is odd, the result is 1 [7], for example:

Physical Address (binary)		
00011111001110010101000000000000		
Mask (binary)	**Result (Physical Address XOR Mask)**	**Result Bit**
00000000000000000010000000000000	00000000000000000000000000000000	0
00000000000001001000000000000000	00000000000000000000000000000000	0
00000000000001001000000000000000	00000000000000000100000000000000	1
00000000000010010000000000000000	00000000000010010000000000000000	0

In that example, the binary result of the number of the bank would be `0010`, which is equal to 2. However, it should be noted that there is no additional information about the order of the masks, so depending on that arbitrarily chosen order, the bank number might be any combination of the result bits. However, it is possible to calculate if two physical addresses are within the same bank when an arbitrarily chosen but fixed order is used.

It should be noted that, as stated before, pages are not always stored in one row but can span across multiple rows [7]. Therefore, a sufficient resolution of the addresses used in the first stage of grouping them by the bank they belong to is required. Depending on the system, it might not be sufficient to use only the base address of each page (e.g. setting the last 12 bit of the physical address to 0).

Based on the address information about the banks, row adjacency can be guessed: When a sequence of physical addresses is grouped by banks, they are in the same row within that bank as long as the bank stays the same (e.g. bank 2). When the bank changes (e.g. bank 4), the same principle also applies to that bank. The data is in the next row when the bank is the same one as before again (e.g. bank 2). As shown by Cojocar, Kim, Patel, *et al.* [33], the mapping of the physical address to the spatial rows is not always linear, so the approach of guessing the row information is not always correct. Because there is no good way to measure row adjacency in software, the authors built a hardware fault injector to get information about row adjacency.

To conclude: It is possible to reverse engineer the used address functions to calculate the bank number from a physical address in software without requiring additional hardware. That approach works as well when a physical page spans multiple banks. However, it does not provide semantic information about the calculated bits (e.g. which bit specifies the rank, which specifies the DIMM, etc.) In contrast, there is no good way to reverse engineer spatial information about the rows in software: Additional hardware is required. It is still possible to guess spatial row information, which might not be correct.

2.5 Rowhammer

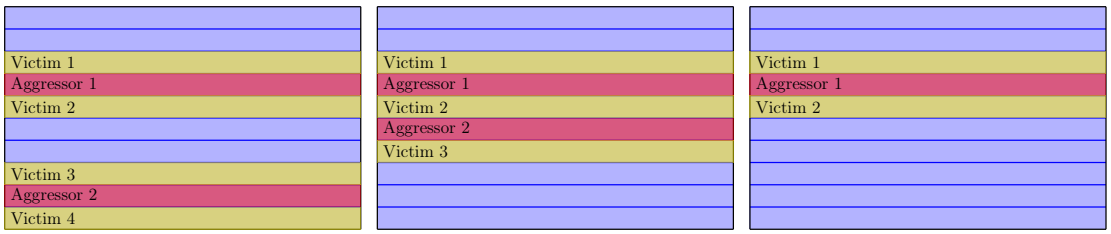
As stated in Section 2.2, a bank in DRAM consists of an array of DRAM cells, amplifiers, and row buffer. Depending on the current content in the row buffer, there can be a row hit (requested data is already in the row buffer) or a row conflict (requested data is not in the row buffer). In

the case of a row conflict, the row buffer data must be written to the DRAM array to avoid data loss.

Reading data from or writing data to the DRAM array is based on electrical currents flowing from or into the DRAM cells. Those currents can lead to disturbance errors in nearby cells by leaking charge from or into the capacitors of these cells. If that leaked charge is strong enough, so the capacitor charge level is interpreted as *full* when it was *empty* before or the other way around at the next refresh of the cell, the bit stored in the cell *flipped*: A logical 1 became a logical 0, or a logical 0 became a logical 1.

Within normal operation, that case should not happen because that would result in memory errors everywhere. Therefore, the cells are designed so that typical operations do not (or not often) lead to the effect described above. However, the physical problem does still exist.

Rowhammer is the approach to exploit these effects actively by using specific *access patterns* that maximize the number of such *bit flips*. An access pattern consists of a list of *aggressor rows* that are read and a list of *victim rows* that might contain bit flips after the aggressor rows were read several times. Reading the aggressor rows sequentially is called *hammering*. Because the charge of the capacitors is restored at each refresh, the aggressors should be hammered for two refresh cycles so that there is at least one complete cycle. There are statically defined patterns, such as those shown in Figure 8.



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

Figure 8: Some typical patterns for Rowhammer exploitation. The *Aggressors* are the rows accessed alternatingly and, thereby, loaded into and restored from the row buffer. The *Victims* are the rows that will likely show flipped bits after a sufficiently high amount of accesses to the *Aggressors*. The images were created by myself as part of the practical thesis [21].

In 2015, Seaborn and Dullien [2] published an exploit that uses Rowhammer to flip bits in PTEs and, thereby, get a PTE to point to the physical address of another PTE. It is possible to use the virtual address mapped by the manipulated PTE to write a PTE currently mapped into the process's address space. Due to that, the attacker can map arbitrary physical addresses into the virtual address space of its own process, which means that he can arbitrarily read and write the system's entire memory.

Their exploit demonstrated that Rowhammer is not a purely academic problem but can also be used for the exploitation of productive systems. At that time, vendors started to develop and roll

out mitigations against Rowhammer. One of the first mitigations was to double the refresh rate since the accesses have to be done between two refreshes. With a double refresh rate, this results in half the number of possible accesses between two refreshes.

Another mitigation approach is to detect if a Rowhammer attack is running, find the victim rows and refresh them. Initially, that was done by the MC: Each DDR3 DIMM has an entry in its Serial Presence Detect (SPD)[3] how many activations require a refresh. That approach is called pseudo Target Row Refresh (pTRR). In DDR4, a similar approach can be directly implemented at the DIMM. In that case, the refresh is done by the DIMM and not by the MC.

It was assumed that Error Correction Code (ECC) DRAM is not affected by Rowhammer because single-bit flips should be automatically detected and corrected. However, as shown by Cojocar, Razavi, Giuffrida, *et al.* [14], that assumption was wrong, and it is possible to exploit Rowhammer on ECC DRAM.

Kim, Daly, Kim, *et al.* [1] mentioned that a double refresh rate does not fully mitigate Rowhammer. As shown in my practical [21] and bachelor's [22] theses, other mechanisms, in detail parallel hammering and execution of an additional sequence of *x86* instructions (together called *flipper*), can be used to increase the number of bit flips found in a given time significantly. Depending on the DIMM, such mechanisms can render the mitigation of a double refresh rate useless (the number of bit flips in a given time on a system without mitigation is equal to the number of bit flips on the same system with the mitigation and amplification).

Because pTRR and Target Row Refresh (TRR) rely on detecting Rowhammer patterns, it would be required to find a pattern that induces bit flips without being detected by the implementation of pTRR or TRR. However, new approaches to bypass these mitigations [15], [17], [19], [20] have been published in recent years.

[3] Electrically Erasable Programmable Read-Only Memory (EEPROM) on the DIMM that stores information about the DIMM which is read by the Basic Input/Output System (BIOS) at initialization time. SPD [34], [35] can be accessed from the OS as well.

3 Flipper on DDR4

As mentioned before, *flipper* [21], [22], the approach that increases the number of bit flips found in a given time by a hundredfold [22], was only evaluated on DDR3 DRAM due to hardware limitations[4] when the experiments were performed.

Within this thesis, there is a test setup with two systems and 60 DDR4 DIMMs with 8 GiB each. Therefore, the experiments done before can be repeated on those systems to verify if flipper increases the number of bit flips found in a given time on DDR4 the same way it does on DDR3.

If no bit flips are found, it is not certain if the PoC does not work or if the system is not affected by Rowhammer. For that reason, the PoC published by Jattke, van der Veen, Frigo, *et al.* [17] was used to scan all 60 DIMMs first. That PoC was chosen because the authors claim to have found bit flips on 40 out of 40 tested DDR4 DIMMs.

In the following sections, the automated test setup used for those experiments is described in detail, and the results of the experiments are shown.

3.1 Automated Test Setup

I have developed an ISO image based on Archiso [36] to perform automated Rowhammer experiments in the scope of the practical [21] and bachelor's [22] theses. The usage of such a setup brings multiple benefits:

- It is not required to install a test system (e.g. install the required software, adjust the configuration, etc.).
- The software versions on all systems are the same over the entire experiment, even if the systems are used for other tasks and, for that reason, updated regularly.
- Most parts of the experiments can be automated to minimize the effort required to perform them.
- Due to the reduced manual interaction, the experiments are less prone to errors made by humans and are performed precisely the same way for each iteration.

Initially, the setup was meant to perform the experiments on any system that boots the ISO image. After the experiments are done, they are uploaded to a server to analyze the results. That mode of operation is called *remote mode* in the current version. In addition, a *local mode* connects to a shared storage (in the current version, *sshfs* [37] is used for that). Thereby it is possible to have a shared state over multiple systems and reboots. The Media Access Control (MAC)

[4] There were two systems with DDR4 DRAM. One had an AMD CPU, so the implemented approach of reverse engineering the address functions did not work [22]. The other one did not seem to be affected by Rowhammer.

address of the primary Network Interface Controller (NIC) is used to distinguish the systems on which the experiments are performed.

For the experiments, 60 DIMMs were tested on two systems. First, it is required to map the physical DIMM to the result set which was uploaded. That is done by physically labelling the DIMMs (D4M01 — D4M60). Afterwards, a state file in the shared storage is created for each system. The file is named like the MAC address of the system in the shared storage. Each file contains the number of the DIMM that will be tested next and the step width, which was two for the experiment.

During execution, the number of the next DIMM is written to the file after the data is successfully uploaded. Due to that, it is only required to plug in the next DIMM and start the system. Afterwards, the experiments are executed automatically. Finally, the system was powered off if everything was successful, and the DIMM could be replaced.

The initially generated ISO image was specially adjusted to match the requirements of the experiments at that time. However, due to that, the architecture made it hard to extend the experiments. For that reason, the architecture was adjusted in order to make it easier to add new experiments.

In the current architecture, there are four stages:

- **Collect Information**: General system-wide information about the CPU, DIMMs, SPD, etc., is collected.
- **Reverse Engineering of the Address Functions**: The address functions are reverse-engineered, as described in Section 2.4.
- **Execution of Experiments**: Depending on the PoC, it might be required to adjust the source code and recompile it if the address functions are hard coded. It might also be required to modify a configuration file or adjust command-line parameters. After that, the PoCs are executed.
- **Evaluation of Results**: The results (e.g. the number of bit flips) are evaluated and, in the end, printed in a brief overview.

Based on that architecture and the features of Bash [38], adding a new PoC requires the creation of a new file, *myExperiment.sh*, in the *Scripts/PoCModules* directory. An example file is shown in Listing 1

```
1 #!/bin/sh
2
3 setvar_MyExperiment() {
4   # Define the name of the PoC
5   POCNAME="New PoC"
6
7   # Define the number of iterations (e.g. how often the experiment is executed)
```

```
  # If that number is greater than 1, there should be a loop in the
  # measure_MyExperiment function that calls Set_Step before each execution.
  # This is used to update the percentage display after each execution and get
  # a more precise forecast about the time required until finished.
  POCITERATIONS=1
}

init_MyExperiment() {
  # Build the PoC only when it was not already built (depending on the
  # experiment configuration multiple experiments can share the same PoC and run
  # it with different parameters).
  if [ "$BUILT_MyPoC" != "Y" ]; then
    cd PoC/mypoc && make; cd ../..
    export $BUILT_MyPoC="Y"
  fi

  # If Hugepage support is required, it should be enabled if not already enabled
  # by another experiment file.
  if [ "$BUILT_HugePage" != "Y" ]; then
    sh ./PoC/trrespass/hugepage.sh
    export BUILT_HugePage="Y"
  fi
}

measure_MyExperiment() {
  # Perform the actual measurement by executing the PoC. Use the TIMEOUT
  # variable in a way that the PoC terminates after that time. This can be done
  # with timeout as shown below, or with something else. It is only important
  # that the PoC terminates after that time.
  # Additional, there is the LOGDIR variable that specifies the directory where
  # all experiment logs are stored. Make sure that all files the PoC creates are
  # stored in that directory.
  (time timeout -s SIGINT $TIMEOUT ./PoC/mypoc/mypoc) 2>&1 | \
    tee $LOGDIR/MyExperiment.log
}

eval_MyExperiment() {
  # Perform the evaluation of the experiment by counting the number of bit flips
  # that occurred from the log file created before. Write the result to a file
  # in EVALDIR.
  cat $LOGDIR/MyExperiment.log | grep flip | wc -l > $EVALDIR/MyExperiment.txt
}

print_MyExperiment() {
  # Print the number of bit flips found that was calculated in the eval
  # function. Normally, it should be sufficient to just print the content of the
  # file in the EVALDIR directory.
  echo "MyExperiment: $(cat $EVALDIR/MyExperiment.txt)"
}
```

Listing 1: Example of a file describing an experiment for the test setup.

After the experiment definition file is created, it is required to enable the experiment. This can be done by editing the files *Scripts/localConfig.sh* and *Scripts/remoteConfig.sh*. Add lines with the content shown in Listing 2 to enable the experiment. The entry in the configuration must have the same name as the file created before because the experiment is matched by the file's name.

```
1 BASELINE_MyExperiment="Y"
2 FLIPPER_MyExperiment="Y"
```

Listing 2: Lines to enable the experiment created before

More information about the automated test setup can be found in Section A.2.

3.2 BlackSmith Experiment

In order to find the DIMMs most susceptible to Rowhammer, the BlackSmith PoC [17] introduced by Jattke, van der Veen, Frigo, *et al.* is executed on 60 DIMMs using the ISO image described in Section 3.1. Two identical test systems with an *Intel(R) Core(TM) i9-10900K* CPU are used for the experiment. For reference, the systems are named *hammertest01* and *hammertest02*. The DIMMs are labelled for reference (*D4M01* — *D4M60*). The ones with odd numbers are tested on *hammertest01*, and those with even numbers on *hammertest02*. Each DIMM is tested for a total time of 6 h.

The publication [17] states that 40 out of 40 DDR4 DIMMs were susceptible to Rowhammer. The authors tested 256 MiB of memory for each DIMM and measured an average amount of 720.45 bit flips. For that reason, it was decided to use that *PoC* because the results from the paper looked promising. Therefore, it was assumed that the PoC should find bit flips on a DIMM when it is susceptible to Rowhammer.

In addition to the number of bit flips, the authors measured the time until the first exploitable bit flip occurred. As stated before, a bit flip is considered to be exploitable when it occurred at a specific offset within a page that could be exploited if the page was used accordingly. In the case of the publication, the exploitation of PTEs, RSA-2048 and sudo was analyzed. The longest time measured was 4 h 2 min for an offset that could be exploited with RSA-2048. Since 34 of 40 DIMMs had exploitable bit flips (which are a subset of the bit flips found in total) within that time it was assumed that 6 h should be sufficient to find bit flips on most DIMMs.

However, bit flips are only found on 4 out of 60 DIMMs during the evaluation. So, the results from the paper can not be reproduced with the test setup used for that experiment. Table 1 lists the details of the DIMMs that are affected. On all other DIMMs, no bit flips were found within the tested time frame of 6 h.

As shown in Table 1, only DIMMs with odd numbers are affected by Rowhammer. Due to the test setup, all those DIMMs were tested with *hammertest01*. Therefore, it is evaluated if these results can be reproduced on *hammertest02* to exclude a problem with the test setup.

Module ID	Number of Bit flips
D4M001	9
D4M021	6
D4M037	4
D4M043	2

Table 1: Number of Bit Flips found within 6 hours by the BlackSmith PoC without flipper

It was not possible to reproduce the results measured on *hammertest01* with the same DIMMs on *hammertest02*, so bit flips did only occur when a DIMM was tested on *hammertest01*. Since *hammertest01* and *hammertest02* have identical hardware and the same BIOS version, that is a strange effect. Because there was not enough time to analyze this further in the scope of this master thesis, it is suggested to inspect that behaviour in the future. See Section 6 for a description of what should be analyzed concerning that effect.

The DIMMs with bit flip were tested with the current version of the PoC hammertest [21]. However, no bit flips were found during those tests. That was expected since hammertest implements classical Rowhammer patterns that are not TRR aware.

Because the experiment results show that none of the tested DIMMs is strongly susceptible to Rowhammer, it is decided to skip further experiments. Instead, it is focused on the process-based Rowhammer exploitation described in Section 4.

4 Exploitation Approach

As shown by Gruss, Lipp, Schwarz, *et al.* [11], it is possible to use Rowhammer to exploit the *page cache*, a caching mechanism that stores the content of files in memory to increase access speed. In detail, an executable binary, library, etc. is loaded into the page cache before execution. If the files on disc were not modified since the last execution and the data is already in the page cache, furthermore executions access the binary directly via the page cache.

The authors target the *opcodes* of userspace binaries stored in the page cache by flipping bits in the instructions. Thereby, the targeted instructions are changed and the binary executes the modified instructions resulting in a change of executed instructions. If that attack is performed on a binary which is executed by the root user since the permissions are set accordingly, it is possible to escalate privileges by executing modified instructions as root.

In addition to exploiting a userspace binary by flipping bits in the opcodes stored in the page cache, it should be possible to flip bits in the memory used at runtime. In the *templating phase*, the attacker's process allocates memory and scans it for bit flips by executing Rowhammer. If a vulnerable page is found, that page is freed so that it is allocated by another process spawned by the attacker at a specific offset in the *reallocation phase*. After that process has started, the bit flip can be triggered again in the *attacking phase* by accessing the aggressor rows that induced the bit flip during templating. The child process should have some wait condition (e.g. waiting for user input) for that approach to work reliably. If the bit flip was induced again, it occurred in the memory used by the child process. A graphical representation of that exploitation approach is shown in Figure 9.

Suppose it is possible to get the vulnerable page remapped in the spawned process at a location chosen by the attacker. In that case, it should be possible to flip bits in arbitrary pages for arbitrary binaries. However, if there is no wait condition within the binary, there is the additional challenge of performing the Rowhammer attack at the correct time. When that happens with a binary where the Set User IDentification (SUID) bit is set, an attacker can modify variables in the binary's memory. If the correct variables are modified, it should be possible to change the control flow of the process and escalate the attacker's privileges.

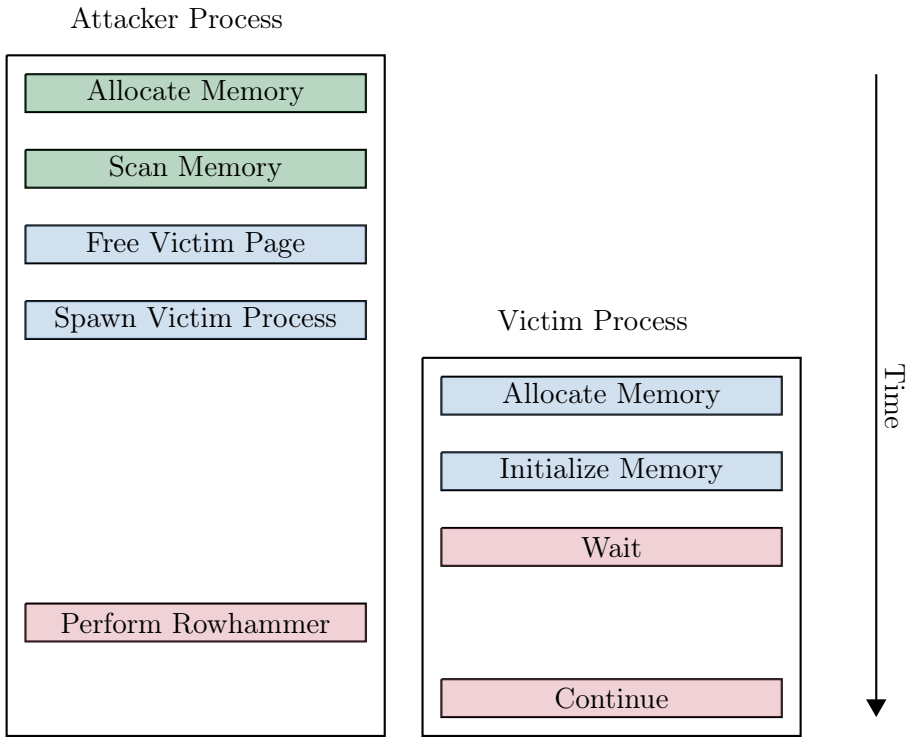


Figure 9: Exploitation approach for attacking process memory. The steps of the *templating phase* are shown in green, the steps of the *reallocation phase* in blue, and the steps of the *attacking phase* in red.

4.1 Hybrid Grouping

As shown in Section 2.1, virtual addresses are mapped to physical addresses using page tables. These mappings can be accessed in Linux by reading the file */proc/PID/pagemap*, where *PID* is the ID of the process whose mappings should be inspected. However, access to the mappings is restricted to users with the *CAP_SYS_ADMIN* privilege to avoid Rowhammer exploitation [39]. This problem can be partly solved by using THP since the last 9 bit of the PFN are directly mapped, e.g. the same for the virtual and the physical address. Thereby, the last 9 bit of the PFN are known.

However, it is impossible to free single pages from a THP, so they are allocated again by another process. A short test implementation[5] shows that single pages can be freed using, for example, madvise [28] with the *MADV_DONTNEED* flag or *munmap* [29]. However, they are not allocated to another process until there is an Out Of Memory (OOM) condition and, thereby, the parent

[5]A THP is allocated using *mmap()* and *madvise()*, afterwards a single page of that THP is freed using either *munmap()* or *madvise()*. Then, a write access is performed to the virtual address. During that, the PFN is retrieved from the *pagemap* file in procfs. After the allocation, there is a PFN at the virtual address. After freeing the page, the PFN is *0x00*. When the write access took place, there is another PFN that is different than the one before.

process is killed. That behaviour was measured with another short experiment[6]. Afterwards, the experiment was slightly modified by killing the parent process manually which led to the reallocation of some freed pages as well.

Because the exploitation approach described before needs both: Correct groups of addresses for the Rowhammer exploitation and the possibility to free single pages afterwards, there was the requirement to find a new approach for grouping the virtual addresses.

The trivial approach would be to group all addresses manually, e.g. create a list of addresses per bank and measure the access times of the addresses against n addresses of each group. Since there should be row conflicts when the addresses are in the same bank, the address is added to the group with the slowest access time afterwards. However, many measurements are required for this approach which, therefore, is pretty slow.

Since the address functions are still used by the MC but can not be applied directly to virtual addresses, another approach is to mix both mechanisms: Perform time-based measurements where necessary and use address function related effects where possible.

As stated in Section 2.1, the buddy allocator handles blocks of 2^n physically continuous pages where n is called the *order* of that block. The numbers of the banks in which these addresses are have to follow the address functions implemented in the MC. If it is known which order the current block from the buddy allocator has and how the addresses are grouped based on the address functions, most banks could be mapped using the address functions. The number of required measurements is reduced significantly, leading to faster results.

HybridGrouping is an approach that combines manual measurements with knowledge about the addressing functions. Figure 10 shows an overview of that approach. In the following, the different steps are explained in detail.

[6] One parent process allocated multiple THPs and freed single pages out of it using *munmap* [29]. Afterwards, it spawned child processes that allocate pages, write to them, and verify if the allocated pages have one of the freed PFNs. It showed that that was just the case after an OOM condition occurred and the parent process was killed.

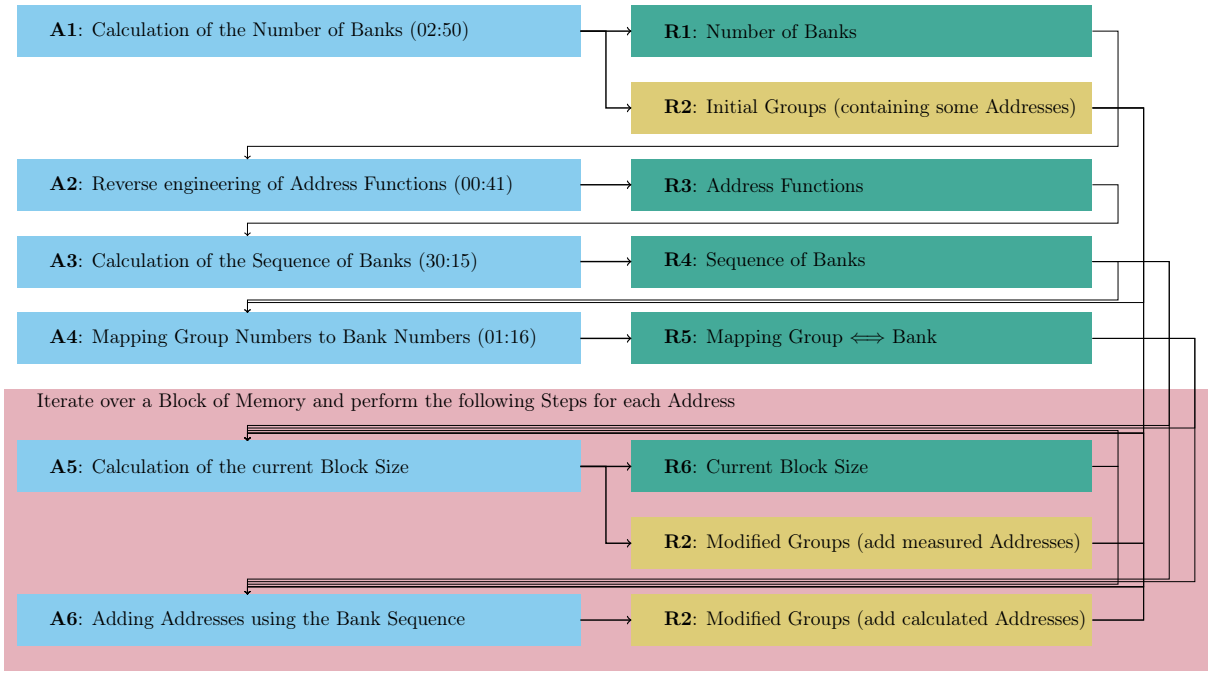


Figure 10: Overview of the HybridGrouping approach. On the left side, the different **A**ctions performed by the PoC are depicted in blue. The right side shows the **R**esults of those steps. The groups that contain virtual addresses by bank and, therefore, are the final result are depicted in yellow. Intermediate results are depicted in green. The depicted times the different actions require are rough estimations measured on one system (average over 5 measurements). It should be noted that the required time depends strongly on the parameters of the PoC and the system it is executed on.

A1: Calculation of the Number of Banks

A threshold time between row hit and row conflict is calculated in the beginning. This is done by allocating a block of memory and comparing the first address in the block with every other address. Two addresses are compared by reading them multiple times (e.g. 1000 times[7] for the HybridGrouping approach) alternatingly and flushing the data from the cache afterwards. Thereby, each read results in access to the DRAM. During the read sequence, the time is measured. There should be many address pairs with a *fast* access time and some address pairs with a *slow* access time. Based on that, the threshold is calculated as the average of the median *fast* and the median *slow* access time.

The number of banks is calculated by creating an initially empty list of address groups. After that, a block of memory[8] is allocated, and each address of that block is compared to n addresses randomly chosen from each existing address group. Multiple addresses are compared one after

[7] A higher amount of measurements increases the accuracy and the time that is required to measure. 1000 is a good compromise on the systems the experiment was performed on.

[8] In the current implementation, 512 pages (2 MiB) are used. Since the system the tests were performed on has 16 banks, there is an average 32 addresses in each group, which yielded good results during experimentation.

another. In the end, the median access time is taken as a result. Because the pages do not span multiple banks on the test system, it is sufficient to use only the virtual addresses at a page border (e.g. where the last 12 bit are zero).

When there are address groups with a median access time bigger than the threshold calculated, the address is added to the group with the biggest access time. Due to noise in the system, multiple groups with an access time bigger than the threshold might exist. However, the access time to the correct group should be the biggest). If no group has an access time bigger than the threshold, a new group is created, and the address is added to that group.

Because a row within a bank has a size of 8 KiB, it can store 2 pages with a size of 4 KiB each. For that reason, two pages can be in the same bank and the same row if pages do not span multiple banks, which is the case on the test system. Often, this happens to pages physically following each other. Because both pages are in the same bank and row, there is a row hit in that case, so the second page is added to a newly created group. That problem can be solved by adding a *regrouping* step. First, each group is removed (one after another) from the list of groups. Afterwards, the addresses of the removed group are added one after another, as described before. Finally, insignificantly small groups are removed (e.g. groups that contain fewer addresses than $\frac{1}{n}$ of the median of all groups, in the current version, less than $\frac{1}{4}$).

Afterwards, the number of groups should equal the number of banks in the system (which should be a power of 2). Additionally, the groups contain some addresses. Therefore the effect of wrong grouping because two addresses are in the same bank and the same row (as described above) should not happen anymore.

A2: Reverse engineering of Address Functions

The reverse engineering of the address functions is done using HammerLib, a Rowhammer library written as a part of my practical thesis [21]. The basic approach of reverse engineering DRAM address functions is described in Section 2.4.

A3: Calculation of the Sequence of Banks

Because the PFNs of the addresses can not be resolved without root privileges, it is not possible to apply the address functions directly to the PFNs. However, applying the address functions to addresses that follow each other results in a *sequence of banks* that repeats itself. That sequence is used for the calculating of the current block size as described at A5: Calculation of the current Block Size and for adding the addresses to the correct groups as described at A6: Adding Addresses using Bank Sequence.

The sequence contains the bank numbers for a *hypothetical PFN* ($0 \leq \text{PFN} < x$), where x is the unique length of the sequence before it repeats itself. It is generated by assuming the hypothetical

PFN to be 0 in the beginning and calculating the number of the bank using the address functions as described in Section 2.4. Afterwards, the hypothetical PFN is increased to be 1, etc., until it is equal to x. Figure 11 depicts such a bank sequence with a length of 512.

0	0	8	8	2	2	10	10	4	4	12	12	6	6	14	14	1	1	9	9	3	3	11	11	5	5	13	13	7	7	15	15
2	2	10	10	0	0	8	8	6	6	14	14	4	4	12	12	3	3	11	11	1	1	9	9	7	7	15	15	5	5	13	13
4	4	12	12	6	6	14	14	0	0	8	8	2	2	10	10	5	5	13	13	7	7	15	15	1	1	9	9	3	3	11	11
6	6	14	14	4	4	12	12	2	2	10	10	0	0	8	8	7	7	15	15	5	5	13	13	3	3	11	11	1	1	9	9
1	1	9	9	3	3	11	11	5	5	13	13	7	7	15	15	0	0	8	8	2	2	10	10	4	4	12	12	6	6	14	14
3	3	11	11	1	1	9	9	7	7	15	15	5	5	13	13	2	2	10	10	0	0	8	8	6	6	14	14	4	4	12	12
5	5	13	13	7	7	15	15	1	1	9	9	3	3	11	11	4	4	12	12	6	6	14	14	0	0	8	8	2	2	10	10
7	7	15	15	5	5	13	13	3	3	11	11	1	1	9	9	6	6	14	14	4	4	12	12	2	2	10	10	0	0	8	8
0	0	8	8	2	2	10	10	4	4	12	12	6	6	14	14	1	1	9	9	3	3	11	11	5	5	13	13	7	7	15	15
2	2	10	10	0	0	8	8	6	6	14	14	4	4	12	12	3	3	11	11	1	1	9	9	7	7	15	15	5	5	13	13
4	4	12	12	6	6	14	14	0	0	8	8	2	2	10	10	5	5	13	13	7	7	15	15	1	1	9	9	3	3	11	11
6	6	14	14	4	4	12	12	2	2	10	10	0	0	8	8	7	7	15	15	5	5	13	13	3	3	11	11	1	1	9	9
1	1	9	9	3	3	11	11	5	5	13	13	7	7	15	15	0	0	8	8	2	2	10	10	4	4	12	12	6	6	14	14
3	3	11	11	1	1	9	9	7	7	15	15	5	5	13	13	2	2	10	10	0	0	8	8	6	6	14	14	4	4	12	12
5	5	13	13	7	7	15	15	1	1	9	9	3	3	11	11	4	4	12	12	6	6	14	14	0	0	8	8	2	2	10	10
7	7	15	15	5	5	13	13	3	3	11	11	1	1	9	9	6	6	14	14	4	4	12	12	2	2	10	10	0	0	8	8

Figure 11: Example of the usage of bank sequences. The current block size is 8, and the offset is 0. The fields in red are the measured offsets, the blue one is the verification offset, and the green ones are additional fields in the block that have not to be measured. The block itself can be seen as a sliding window that is shifted over the bank sequence until the measured offsets (in red) and verification offset (in blue) match the measurement. If no valid sub-sequence is found within the sequence, the block size is reduced, and the procedure is repeated (with a block size of 4). At the same time, the procedure is performed for double the current block size (in this example 16). If the requirements described above are met, the block size is doubled.

As stated before, the pages do not span multiple banks on the test systems, so using the base addresses for each hypothetical PFN (e.g. append 12 bit with a value of zero) works. However, on other systems where the pages do span multiple banks, it is required to add multiple offset addresses depending on the exact span of the pages over the banks (e.g. when a page spans over 8 banks, 8 addresses have to be calculated for each hypothetical PFN with the corresponding offsets).

The result of this calculation is a sequence of bank numbers that matches for addresses following each other. When n PFNs are physically continuous, there has to be at least one *sub-sequence* of length n with matching bank numbers. In Figure 11, the sub-sequence is depicted as the colored part (e.g. the first eight items in the sequence).

If the PFNs are not continuous, there should be no matching sub-sequence. Depending on the address functions there is a probability that there is a matching sub-sequence even though the PFNs are not continuous. However, that is not a problem since the address functions do still apply when that case occurs (e.g. the PFNs are not continuous but follow the address functions and, for that reason, follow the sequence).

Therefore, a sequence of n virtual addresses is mapped to PFNs that are either continuous or aligned in a way that they follow the address functions if and only if the measured bank sequences of those addresses matches a sub-sequence of length n of the calculated sequence.

Since the search for a sub-sequence of length n would require n measurements, there would be no advantage compared to the trivial approach of measuring the bank of each address. Therefore, the next step is to minimize the number of required measurements in a way that they specify a sub-sequence uniquely. Those measurements are specified by *offsets* within the length (n) of the sub-sequence that is searched. Each length requires different offsets depending on the address functions and the total length of the bank sequence. Due to the operation of the buddy allocator, n is always a power of 2.

For example, a sub-sequence of length 16 could be used to group a list of 16 virtual addresses to banks. If that would require 16 measurements, it would be equivalent to the trivial approach of measuring the bank of each address. However, when it is possible to uniquely identify a sub-sequence of length 16 with only 4 measurements, that would be faster since fewer measurements are required to group the same number of addresses. In Figure 11, the offsets are depicted in red within the sub-sequence.

In addition to the offsets that are required to uniquely identify a sub-sequence an additional *verification offset* is used to avoid the case that the blocks do not follow the sub-sequence but the offsets match anyway. Therefore, the verification offset is calculated in addition to the offsets. The verification offset is depicted in blue in Figure 11.

Since the calculation of the offsets is slow (action A3 took approximately 30:15), the results are stored in a cache file and loaded on subsequent runs. The results of this step depend only on the address functions. Therefore, the address functions are stored in the cache file as well and a restore is only performed when the address functions measured in A2: Reverse engineering of Address Functions are equal to the ones stored in the cache file.

A4: Mapping Group Numbers to Bank Numbers

The bank sequence calculated in A3: Calculation of the Sequence of Banks uses *real bank numbers*. However, the addresses are only grouped by banks, so it is not ensured that the index of a group is equal to its bank number according to the address functions (e.g. there is a permutation between bank number and group index).

In this step, a THP is allocated, and the single addresses within the THP are grouped as described before. Because the last 9 bit of the page address within a THP are equal to the last 9 bit of the corresponding PFN, the address functions can be applied to the virtual address within the THP to calculate the correct bank number if there are not more than the last 9 bit of the PFN used for bank number calculation, which is the case on the test systems.

At this point, the number of the group to which a virtual address from the THP was added and the number of the bank that was calculated using the address functions are known. With this information, a mapping between the group index and bank number can be created to calculate the *real bank number* from a group index afterwards.

A5: Calculation of the current Block Size

As stated before, continuous PFNs must follow the bank sequence calculated. Due to the buddy allocator described in Section 2.1, PFNs are allocated in blocks of 2^n pages, where n is called the *order* of that block. For that reason, the initial *block size* is assumed to be 1, e.g. pages allocated from order 0. This is done because that criterion (block of 1 page) is met by all allocations which are performed for whole pages only.

For the calculation of the block size, it is required to identify sub-sequences with a minimum amount of known banks uniquely (e.g. measure the group index at the specified offsets and map it to the bank number afterwards). The smaller the fraction of measured addresses is, the faster is the entire grouping because the measurements are the slow part. Therefore, the offsets and verification offset calculated in step A3: Calculation of the Sequence of Banks are used.

Based on the offsets and the verification offset, it is possible to search sub-sequences of the bank sequence that match the current measurements. When the current block size is n, it is checked if there is a valid sub-sequence for a block size of n and a block size of $2n$. If there is no valid sub-sequence for a block size of n, the block size has to be reduced. On the other hand, if there is a valid sub-sequence for a block size of $2n$, the block size can be increased. In order to improve the stability, the block size is not increased at the first sub-sequence match for $2n$, but after x following sub-sequence matches for $2n$. However, since the block size would be $2n$ in the successful case, only every second comparison has to match (the other comparisons have an offset of a half block, so they do not match).

A6: Adding Addresses using Bank Sequence

When a valid sub-sequence is found, all addresses in the block of the current block size can be added to the groups based on the previously identified sub-sequence. However, because there is a mapping between the group index and the bank number, it is required to calculate the group index. That can be done by using the sub-sequences bank number. Afterwards, the addresses can be added to the corresponding groups.

Evaluation

In the current version of the PoC, the PFNs are also accessed. Accessing the PFNs requires root privileges and has to be disabled for real use cases because the attacker does not have root

privileges in the typical attack scenario. However, it can be used to verify how accurately the approach works.

When the PoC is executed, it dumps the offsets where a specific block size is reached the first time. If the block size is decreased again, that is shown as well. These information are calculated from the PFNs, so they are accurate. After an initialization phase (see Figure 10), the addresses are grouped using the HybridGrouping approach. Again, the offsets are shown when the block size changes. In order to increase the speed, the first x percent of the pages are skipped (currently, the first 20 %) because the smaller blocks are allocated first. Afterwards, the offsets where the block size changes should be similar to those shown before. Due to the approach, it is impossible to detect bigger block sizes immediately, so there can be a slight lack. An execution on the test system showed that this was the case. See Section A.3 for more information about the results and detailed reproducing instructions.

RDTSCP Resolution

The implementation of the HybridGrouping approach described above uses the *rdtscp* instruction to measure the time. Therefore, the current counter value is stored in a variable. Afterwards, the addresses are accessed, and the final counter value is stored in another variable. The tests showed that the counter does not have a resolution of 1 but a bigger value (e.g. it was only possible to measure counter differences in discrete steps). Table 2 provides an overview of the measured resolutions on different systems. See Section A.4 for a detailed description of how the results can be reproduced.

CPU	rdtscp resolution	gettime resolution
Intel(R) Core(TM) i5-3320M	4	1
Intel(R) Core(TM) i5-4210M	1	1
Intel(R) Core(TM) i7-4800MQ	1	1
Intel(R) Core(TM) i7-8550U	1	1
Intel(R) Core(TM) i9-10900K	2	1
Intel(R) Core(TM) i7-12700H	2	1
AMD A4-5000	1	1
AMD Phenom(tm) II X6 1055T	1	1
AMD Ryzen 7 3700X	36	10
AMD Ryzen 7 3800X	39	10
AMD Ryzen 9 3900X	38	10

Table 2: Resolution of the rdtscp counter on different systems

4.2 Page Reallocation

As stated before, the exploitation approach relies on the possibility of getting single vulnerable pages mapped in the virtual address space of other (e.g. newly spawned) processes at an offset specified by the attacker. Therefore it is analyzed how the memory allocation system of Linux works in detail. The general procedure of page allocation is described in Section 2.1.

In the rest of this section, the general concept of page reallocation is explained in more detail and evaluated. Afterwards, the offsets of the remapped ranges are analyzed to find stable procedures for freeing pages to get them allocated at a specified offset in the child process. Finally, the pages are tracked so that it is possible to determine what the pages were used for before or after the allocation in the PoC.

It should be noted that the experiments described in this section were performed to get a better understanding of the memory mapping mechanisms. Since many of them require access to PFNs to identify pages uniquely, it is required to execute the PoCs with root privileges. Therefore, the experiments and results should be seen as a way to better understand the memory mapping mechanisms and their behaviour and not as an exploit for privilege escalation.

4.2.1 Reallocation Percentage of freed Pages

When a process allocates many pages and frees them just before calling *fork* and *exec*, some of the pages freed before should be allocated by the child process. That is the case because the freed pages are returned to the per-CPU list of the logical CPU core the parent process is executed on when freeing the pages. If too many pages are there in that per-CPU list, they are returned to the free lists of the buddy allocator in the corresponding order. When the pages were allocated one by one, there were only requests for single pages. Therefore, they were served from order 0. So the pages will be in order 0 of the buddy allocator if their buddies are not freed. If both buddies were freed, they would be merged into a higher-order block.

Depending on the PFNs of the freed pages, it is possible to enforce most freed pages to be added into order 0 of the buddy allocator. When the pages are allocated and freed one by one, but the freeing is performed in a random order, most of the pages of a formerly split block from the buddy allocator stay allocated. That is the case because the buddies of the freed pages will probably still be allocated, so the allocator cannot merge them to a block bigger than order 0 (e.g. single pages).

The child process will allocate the pages from the per-CPU list of the logical CPU core it is executed on. When the list is drained, more pages are allocated from the free lists of the buddy allocator, beginning (for single page requests) at order 0, where the pages freed by the parent process are stored. It is possible to use CPU pinning to force the child process to run on the same logical CPU core as the parent process so they share the same per-CPU list.

For that reason, it is assumed that, depending on the noise on the system (e.g. memory allocation and free requests from other processes), a high percentage of the pages freed by the parent process should be allocated by the child.

A PoC was written to verify that assumption: First, it allocates a specified amount of pages one by one (e.g. calling *mmap* to map one page, writing to that page to actually allocate it, calling *mmap* again for the next page, writing to it, etc.). Since the pages are mapped one by one, there is a high amount of mappings. The maximum number of mappings is limited for a process (see */proc/sys/vm/max_map_count* [40]), so the number of pages that can be allocated with that approach is limited[9]. It should be noted that the process also has implicit mappings (e.g. required for shared libraries, general allocations using *malloc*, etc.), so the number of pages that can be mapped with *mmap* is lower than the mapping limit depending on the implementation.

Next, the PFNs of the process are dumped. Therefore, an export file is created for all mappings grouped by the mapping name retrieved from */proc/self/maps*. Another export file is created for the PFNs that are part of the pages explicitly allocated before. The pages of the explicitly allocated area are also part of the general mapping.

Afterwards, a specified amount of pages is freed, the process is forked, and the binary is executed again. Like before, it allocates a specified amount of pages and dumps the PFNs of the mappings.

Finally, the PFNs of the explicitly mapped pages of the parent can be compared with all PFNs mapped in the child. If a PFN is part of both, it is counted. For the evaluation, the fraction of duplicated PFNs of the number of freed PFNs from the parents mapping can be calculated. See Table 3 for the results of the measurements.

For all systems depicted in Table 3, the reallocation percentage is higher with CPU pinning enabled than without. This is the case because CPU pinning forces the parent and the child process to run on the same logical CPU core, so they share the same per-CPU list (e.g. the child can reallocate the pages that the parent freed and that are in the per-CPU list).

The remapping percentage without CPU pinning differs strongly between the different systems. It seems to depend on the generation of the CPU in the case of Intel[10], but there are not enough systems with AMD CPUs to assume anything about AMD in that respect. However, it is not possible to verify that assumption within this thesis since there are not enough systems on which the experiment can be performed.

See Section A.5 for a detailed description of how the results described above can be reproduced.

[9] On the systems that were used for testing, it was set to 65 530 at the time of writing this thesis.

[10] A first assumption was that it is related to the number of logical CPU cores, but, among others, the *i5-3320M* and the *i5-4210M* have four logical cores each, and the *i7-4800MQ* and the *i7-8550U* have eight logical cores each, so it seems not to be directly related to the number of cores.

CPU	Kernel	CPU Pinning	Remapped	Remapping Percentage
Intel(R) Core(TM) i5-3320M	5.19.13-arch1-1	✗	140.39	3.42 %
		✓	3983.67	97.25 %
Intel(R) Core(TM) i5-4210M	5.19.13-arch1-1	✗	2751.90	67.18 %
		✓	4067.63	99.30 %
Intel(R) Core(TM) i7-4800MQ	5.19.13-arch1-1	✗	2630.57	64.22 %
		✓	4083.08	99.68 %
Intel(R) Core(TM) i7-5600U	5.19.13-arch1-1	✗	431.18	10.52 %
		✓	4051.97	98.92 %
Intel(R) Core(TM) i7-5600U	5.19.13-arch1-1	✗	390.83	9.54 %
		✓	4086.26	99.76 %
Intel(R) Xeon(R) CPU E5-2680 v4	5.19.13-arch1-1	✗	3763.50	91.88 %
		✓	4086.99	99.78 %
Intel(R) Core(TM) i7-8550U	5.19.13-arch1-1	✗	3097.95	75.63 %
		✓	4059.78	99.11 %
Intel(R) Core(TM) i7-8665U	5.19.13-arch1-1	✗	3063.22	74.78 %
		✓	4081.99	99.65 %
Intel(R) Core(TM) i9-10900K	5.19.13-arch1-1	✗	3937.94	96.14 %
		✓	4086.73	99.77 %
Intel(R) Core(TM) i7-12700H	5.19.13-arch1-1	✗	3684.10	89.94 %
		✓	4085.63	99.74 %
AMD A4-5000	5.10.0-19-amd64	✗	3702.77	90.39 %
		✓	4042.97	98.70 %
AMD Ryzen 9 3900X	5.19.13-arch1-1	✗	3948.78	96.40 %
		✓	4086.81	99.77 %

Table 3: Remapping percentage of freed pages. The parent process allocates 32 728 pages and frees 4096 randomly selected pages afterwards. Next, the child process is started and allocates 32 728 pages. Next, the number of pages that were in the mapped pages of the parent (e.g. the 32 728 pages) and are mapped again somewhere in the child (e.g. any mapping, not limited to the 32 728 pages) is counted. Afterwards, the percentage of them (from the 4096 freed pages) is calculated. The experiment was performed 100 times, and the average number of remapped pages and the according percentage are depicted in this table.

4.2.2 Offsets of reallocated Pages

With the approach introduced in Section 4.1, it is possible to group virtual addresses by the spatial bank number the PFNs mapped to those addresses are stored on faster than measuring each address.

The experiment shown in Section 4.2.1 confirmed the hypothesis that most pages freed by a parent process are allocated by a child process that is created afterwards. Even though that behaviour could not be verified on some systems without CPU pinning, it was confirmed on all systems when CPU pinning was enabled.

The HybridGrouping approach introduced in Section 4.1 requires mappings of higher orders from

the buddy allocator to be effective (the higher the order of the block is, the fewer measurements have to be performed to group the same amount of pages). So, it works efficiently when many pages are allocated, and the first part of the allocation is skipped. In contrast, the approach used for page reallocation introduced in Section 4.2.1 requires single mappings for each page to free the pages afterwards. That approach limits the number of pages that can be mapped since there is a limit for mappings for each process [40].

Since the number of mappings is limited per process, the HybridGrouping approach could be modified to a multi-process architecture. However, that is not done in the scope of this thesis. See Section 6 for a detailed description of the requirements and the discussion of a possible architecture.

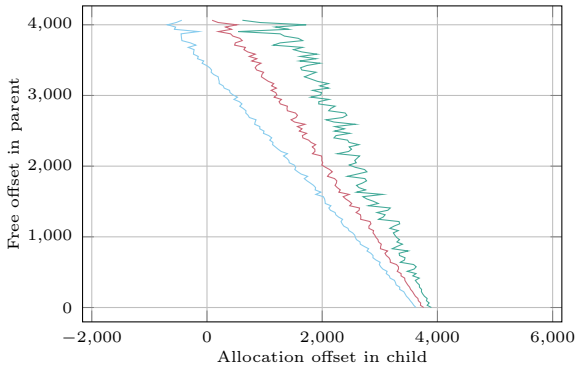
Next, it is required to find a deterministic approach to get the freed pages mapped at a specific offset within the child process. The last experiment confirmed that there is a high remapping percentage, but the exploitation approach requires a vulnerable page to be mapped at a specific offset within the child process. In this section, the offsets of the freed and reallocated pages are analyzed in more detail to find such an approach.

The PoC described in Section 4.2.1 was used again for this experiment. The procedure is the same as described before: The parent process is started, allocates pages, shuffles their order, and frees some of them (which are randomly selected due to the shuffle done before). Afterwards, the child process is spawned. In contrast to the last experiment, the offsets of the remappings are analyzed. Since the majority of pages were reallocated in the area that is manually allocated by the child process (exact percentages are shown on the individual results), the offsets were only analyzed for that case (e.g. the pages explicitly freed in the parent process and reallocated in the *mmap*ed area of the child). See Figure 12 for the results of the experiment. See Section A.6 for a detailed description of the experiment and instructions to reproduce the results.

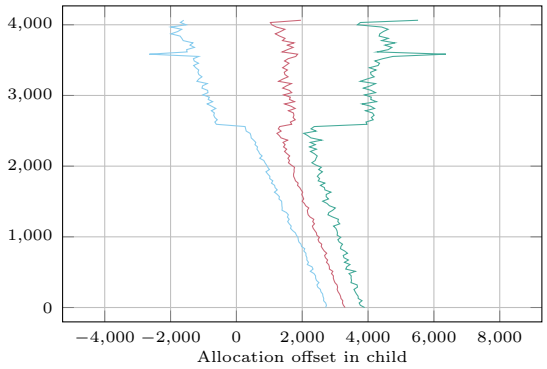
The experiment was performed on two systems with an *Intel(R) Core(TM) i7-12700H* and an *AMD Ryzen 9 3900X*. The remapping offsets were measured with and without CPU pinning enabled.

On both systems, the average graph is nearly a linear function (with some noise) when CPU pinning is enabled. However, the standard deviation increases with the offset of the page freed by the parent process, e.g. the later that page was freed, the bigger the standard deviation (and thereby, the difference between the single measurements) is. The noise in the average graph seems to be related to the noise in the standard deviation since the graph showing $\mu - \sigma$ (depicted in blue) is nearly linear, e.g. has less noise than the graph showing μ. On the other hand, the graph showing $\mu + \sigma$ (depicted in green) shows more noise than the average graph.

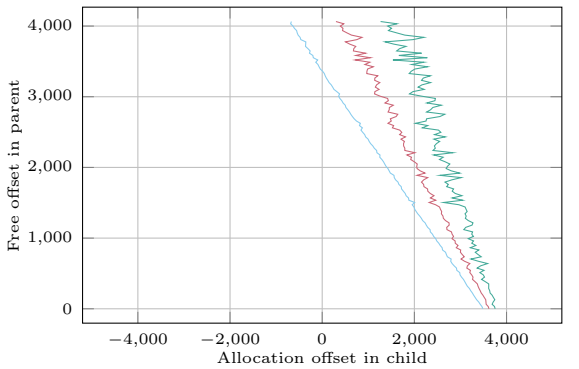
When CPU pinning is not enabled, the results look similar for a parent offset up to approximately 3000 on the *AMD Ryzen 9 3900X* and up to 2500 on the *Intel(R) Core(TM) i7-12700H*. After that parent offset, the allocation offset in the child shifts to the right, and the standard deviation increases significantly.



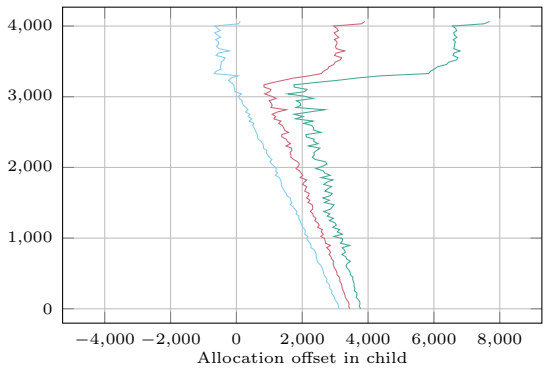
(a) i7-12700H with CPU pinning. 100 % of the reallocated pages are in section *anonymous* and, therefore, presented in this plot.



(b) i7-12700H without CPU pinning. 89.28 % of the reallocated pages are in section *anonymous* and, therefore, presented in this plot.



(c) Ryzen 9 3900X with CPU pinning. 99.46 % of the reallocated pages are in section *anonymous* and, therefore, presented in this plot.



(d) Ryzen 9 3900X without CPU pinning. 97.18 % of the reallocated pages are in section *anonymous* and, therefore, presented in this plot.

Figure 12: Remapping offsets between parent and child process. Since the offsets are only considered for the *mmap*ed part of the pages of the child process (in contrast to the experiment before where the entire memory was compared), the percentages of the remappings to the analyzed memory are shown in the sub-figures (e.g. how many percent of the pages freed by the parent process were remapped in the explicitly *mmap*ed part of the child processes memory). This results are unlike the ones from the experiment described in Section 4.2.1, where the remapping percentage within all memory mappings was measured. The depicted data are the result of 100 measurements. In order to simplify the graphs, 32 parent offsets (shown on the y-axis) are concluded to 1 point in the graph by calculating the average x and y values of the measured values. The red graph shows the average value over the measurements (e.g. average allocation offset in the child process). The other graphs depict the standard deviation (the blue graph is $\mu - \sigma$ and the green one is $\mu + \sigma$).

Because the parent process starts freeing at offset 0 up to 4096, the pages with a higher offset in the parent process were freed later than the ones with a lower offset. Therefore, the graphs without CPU pinning differ significantly from the ones with CPU pinning for the pages the parent process freed. Since there are per-CPU lists that effectively buffer free pages for each CPU core, the pages that the parent last frees are probably in the per-CPU list of the according core. When the maximum number of pages in the per-CPU list is reached, the pages are freed and added to the list of the different orders of the buddy allocator.

With CPU pinning, the parent and child run on the same CPU core, so they share the same per-CPU list, and the child reallocates the pages that were added to the per-CPU list by the parent before. Without CPU pinning, it is unlikely that parent and child will run on the same CPU core (depending on the scheduler, that may be the case sometimes). So, they often do not share the same per-CPU list. When more pages are freed by processes running on the same CPU core than the parent, the upper limit is reached at some point, and the pages from the list are added to the buddy allocator, so they are allocated by the child again. Since that does not happen at a specified time and depends on other processes running on the same CPU core, there are significant differences between the single measurements and, therefore, a much bigger standard deviation when CPU pinning is not enabled.

4.2.3 Tracing of Pages

Since the reallocation offsets of freed parent pages differ strongly between multiple measurements, as described in Section 4.2.2, it would be helpful to gain more insights into the inner working of the memory allocation mechanism. Because some pages are not reallocated (even if it is a small fraction of the pages that were freed), the question: *What happens to the pages that are not reallocated?* should be answered as well.

In order to get a better understanding of the usage of pages, *ftrace* [41], the function tracer of the Linux kernel, can be used. The command *trace-cmd* [42] provides the possibility to access these tracing functionalities from user space. It is possible to trace called functions and their parameters, e.g. *mm_page_free* or *mm_page_alloc*.

A short evaluation showed that other processes used the pages that were not reallocated, e.g. there were calls to *mm_page_alloc* and *mm_page_free* from other processes as well for the same PFNs within the measured time frame[11].

Next, the problem with the allocation offsets is analyzed in more detail. Since it is not possible to directly access the PFNs of pages that are in the lists of the buddy allocator, the kernel module AllocTrace was written to provide access to these information. When the module is loaded, it creates a new directory */proc/allocTrace*. Within this directory, directories are created for each zone, containing directories for each migrate type. In those two directories, *buddy_pages* and

[11] For this experiment, the functions were traced for 10 s before, during, and 10 s after the execution of the process

per_cpu_pages are created. In the *buddy_pages* directory, there are files for each buddy order named like the order (*00 — 10*). In the *per_cpu_pages* directory, a file is created for each logical CPU core following the naming scheme of the buddy files.

Each file supports the operations *read*, *lseek*, *open* and *release*. When a file is read, the PFNs related to the path (e.g. in the correct zone, with the correct migration type, and in the case of the buddy system, the correct order, or for per-CPU lists, the correct logical CPU) are returned in a readable form (e.g. *0x0af842*).

Due to that, it is possible to access the PFNs in the free lists of the different orders of the buddy allocator or at the per-CPU lists of the different logical CPU cores. See Section A.7 for detailed instructions on how AllocTrace can be used.

It is possible to create a snapshot of the free pages within the buddy allocator system by copying the directory created by AllocTrace recursively. However, since the kernel module does not store any data but accesses and returns live data, modifications might occur while the snapshot is done. That is the case when a process allocates or frees pages while the directory structure is copied.

Since copying files requires memory, e.g. allocates and frees pages while running, MemCP, a tool that splits the copy procedure into multiple stages triggered after each other, was implemented.

In the first stage, a list of files is created by performing a directory walk at the specified base directory. During that, the size of the files is measured as well. If that is possible by using *stat()* [43], this is done. However, for some files, *stat()* returns a size of 0, even if their size is not. This is, for example, the case for files in the *procfs*. Therefore, a manual measurement mechanism is implemented: When the size returned by *stat()* is 0, the file is opened, and its content is read. During that, the size of the file is measured by counting the number of bytes read. Then, the required size is calculated by multiplying the sum of the number of bytes required for the files and the number of bytes required for the file path strings by 2. A buffer of at least the required size is allocated using THPs. If the size is not on a 2 MiB border, it is rounded up.

After that, MemCP waits for the signal *SIGUSR1*. If it receives that signal, it opens one file after another and writes their paths and content to the buffer that was allocated before. Thereby, the path of the file, followed by its content, is written. In order to distinguish the path from the content, a carriage return character (`\r`) is added in front of the path. For that reason, there would be a problem when files contain that character which is not the case for files created by AllocTrace.

Then, MemCP waits for the signal *SIGUSR1* again. When the signal is received, the buffer's content is restored to the target directory by creating the directory structure (based on the file paths stored in the buffer) and writing the contents to the files.

Thereby, the operations performed when the snapshot is created are limited to the bare minimum. It is still required to read the file and write to the buffer, but there are no additional steps of

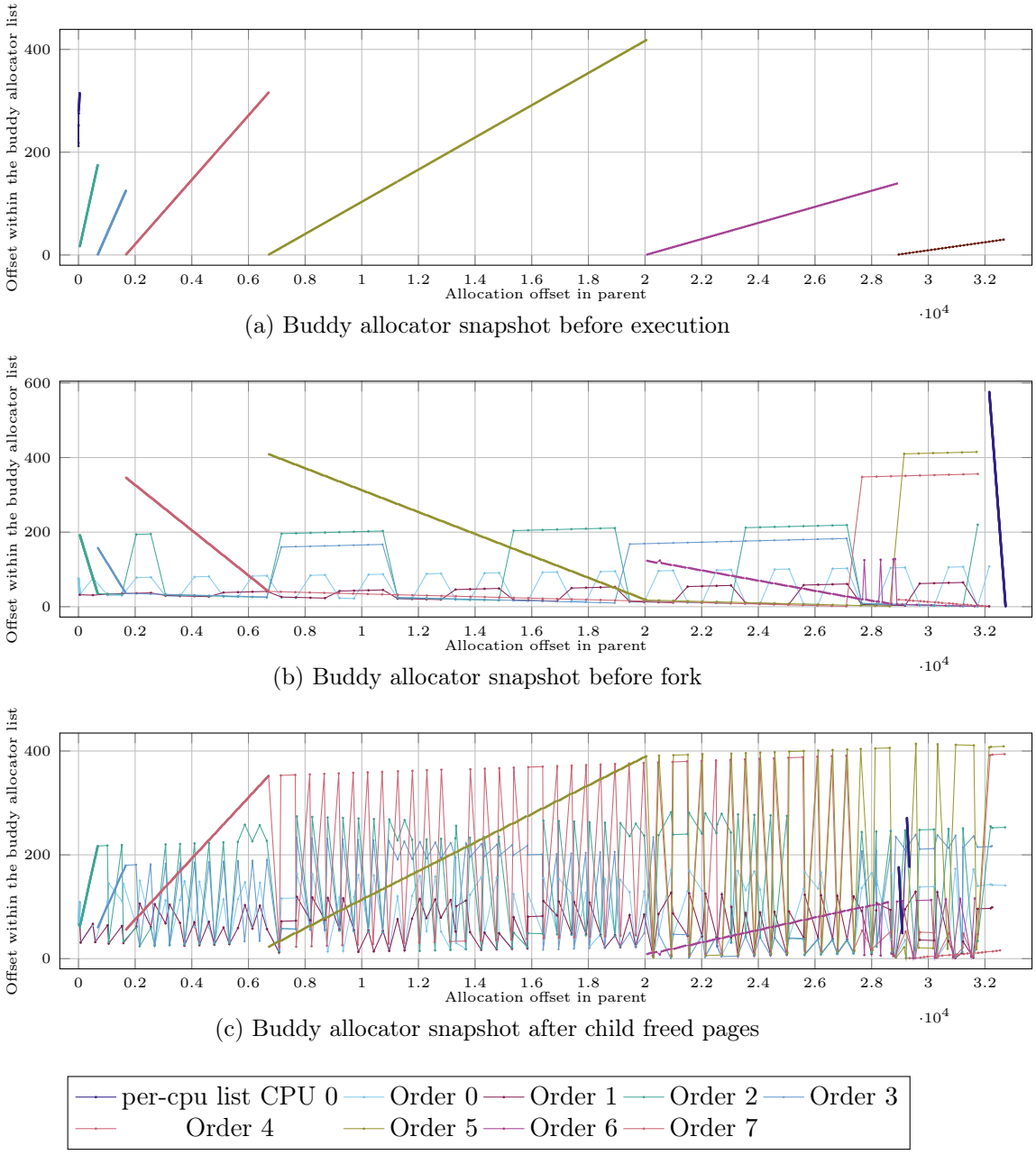
dynamic memory allocation. Also, the creation of output files and freeing of the allocated buffers is delayed to not interfere during the snapshot. Even though that approach is not perfect, it reduces the memory interactions that are done during a snapshot is created. A detailed description of how MemCP can be compiled and used is provided in Section A.8.

With MemCP, three snapshots of the buddy allocator lists are created during the experiment: The first one before the parent process is started, the second one before the parent process forks, and the third one after the pages the child process allocated before being freed. As already used for the experiments described in the last sections, the PoC introduced in Section 4.2.1 was used. Since the interaction with the buddy allocator is inspected, the shuffle of the addresses was disabled (the graphs will show the offsets within the allocated area and within the allocator's lists, so the shuffle would scramble the graph). Additionally, the sending of the signal to the MemCP process is implemented in the parent and child. Both processes free all pages manually mapped before, not only the first 4096 in the parent.

After the execution, the three dumps of the pages within the buddy allocator can be evaluated using the PFNs dumped by the parent and child process. Next, three graphs showing the correlation between the offset within the mapping in the process and the offset within the lists of the buddy allocator are generated for each process (one for each dump of the allocator). Figure 13 depicts the results for the parent process, and Figure 14 depicts the results for the child process. The measurement was performed on the system with the *Ryzen 9 3900X*.

The parent allocates and frees the pages between the initial snapshot and the snapshot before the fork, and the child allocates and frees the pages between the fork snapshot and the final one. Therefore these measurements are similar: In both cases (Figure 13a and 14b), the allocation starts with the pages from the per-CPU list of the corresponding logical CPU core (in this case, CPU 0). Afterwards, the pages are allocated from the increasing orders of the buddy allocator. While there are pages from all orders in Figure 14b, the allocation in Figure 13a starts at order 2, which is the case because the lower orders were empty when the process started.

Since the different lists of the buddy allocator are implemented as linked lists, the allocation is performed in increasing order, e.g. the first page in the list is allocated first. For that reason, the single lines for the different orders are ascending. Also, the different sizes of the blocks can be seen in the slope of the different lists depicted in both figures: It is divided by two every time the order number increases (but is the same for order 0 and the per-CPU list since they both use a block size of 1 page).

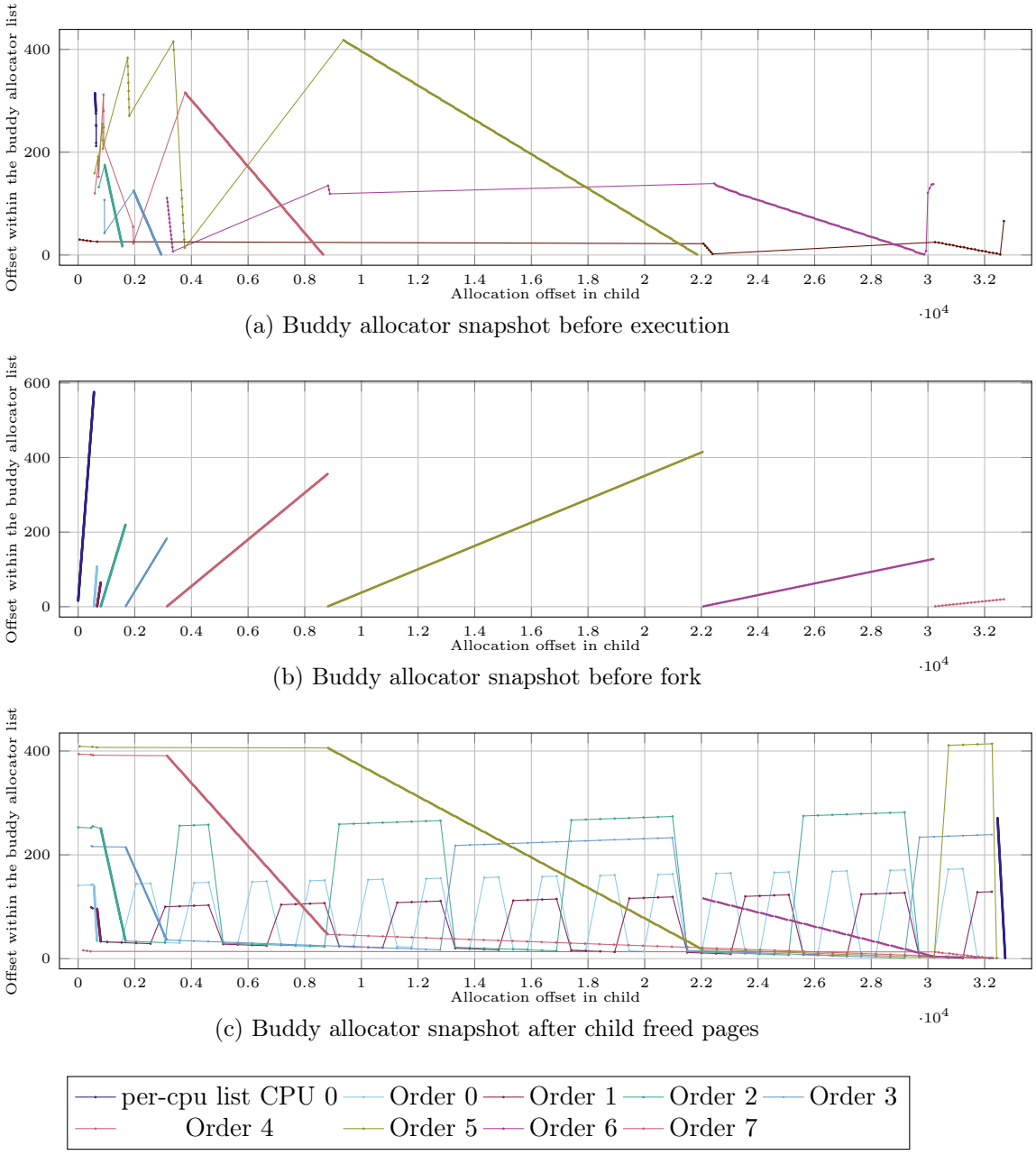


(a) Buddy allocator snapshot before execution

(b) Buddy allocator snapshot before fork

(c) Buddy allocator snapshot after child freed pages

Figure 13: Offsets within the buddy allocator (parent process). First, a snapshot of the buddy allocator named *before* was created. Next, the parent process was started and allocated 32 728 pages (the PFNs of those pages were dumped). Then, the parent freed all pages, and a second snapshot of the buddy allocator named *between* was created. Afterwards, the parent forked and executed a child process (basically the same binary as the parent with modified command line parameters), which allocated 32 728 pages as well, dumped their PFNs and freed them afterwards. Finally, a third snapshot called *after* was created. The graphs show the offset of PFNs that were allocated in the parent process and their offset in the buddy allocator at the time the snapshot was created.



(a) Buddy allocator snapshot before execution

(b) Buddy allocator snapshot before fork

(c) Buddy allocator snapshot after child freed pages

Figure 14: Offsets within the buddy allocator (child process). First, a snapshot of the buddy allocator named *before* was created. Next, the parent process was started and allocated 32 728 pages (the PFNs of those pages were dumped). Then, the parent freed all pages, and a second snapshot of the buddy allocator named *between* was created. Afterwards, the parent forked and executed a child process (basically the same binary as the parent with modified command line parameters), which allocated 32 728 pages as well, dumped their PFNs and freed them afterwards. Finally, a third snapshot called *after* was created. The graphs show the offset of PFNs that were allocated in the child process and their offset in the buddy allocator at the time the snapshot was created.

After the pages were freed, as shown in Figures 13b and 14c, the offsets in the lists are decreasing, e.g. the page that was freed first has the highest index, etc. This is the case since an item appended to a linked list moves all items by one and gets the new first item. The slope of the graphs is similar to the increasing ones described before: it divides by two every time the order of the buddy allocator list increases. In contrast to the allocation, the pages in the per-CPU list are the ones that were freed last. This is the case since the per-CPU lists perform as a cache between the buddy allocator shared between all logical CPU cores and the processes assigned to one core.

The per-CPU lists have two relevant parameters in that aspect: *High* and *Batch*. *High* specifies the maximum number of pages in the list before it is flushed, e.g. the pages in the list are returned to the buddy allocator. *Batch* specifies the number of pages that should be allocated when the list is empty. Since allocating one page would require locking the buddy allocator every time a page is requested, and no pages were added to the list, multiple pages are added to the list at once. On the system the test was performed on[12], the *High* value for the *Normal* zone was 932, and the *Batch* value was 63. These values were measured as described in Section A.7. See Section A.9 for a detailed description of how the results can be reproduced.

The graphs depicting the process of freeing pages (depicted in Figure 13b and Figure 14c) contain some *noise* in addition to the linear graphs described above. It is assumed that that is the case because other processes interact with the buddy allocator and per-CPU lists at the same time the experiments are performed.

When another process allocates a page that was just freed as one of the pages of a block of a higher order (e.g. order 6), the buddy of the page is not available anymore, so there would be one block in order 0 whose buddy is not free because it was allocated by the other process. Due to that, the buddies can not be merged to a block of order 1, and another block in order 1 does not have a buddy and can not be merged to a block of order 2, etc.

The graphs outside the execution scope shown in Figure 13c and Figure 14a have a considerable time lag between the snapshot was taken and the process interacted with the memory. Additionally, in both cases, there were other processes (for the child, the parent process and for the parent, the child process) that performed heavy memory interactions. For that reason, they provide a low informative value and are only depicted for completeness.

The allocation of memory performed by processes uses the lists of the buddy allocator as well as the per-CPU list of the core the process is executed on. Therefore, this section verified that the buddy allocator on the test system works as described in Section 2.1. So, the differences in the remapping offsets described in Section 4.2.2 can not be explained by a different behaviour of the buddy allocator.

[12]The system with the *AMD Ryzen 9 3900X*

5 Related Work

In 2014, Kim, Daly, Kim, *et al.* [1] showed that modern DRAM (when writing DDR3) is affected by bit flips induced by an attacker reading memory locations that are spatial near the ones the bit flip should occur afterwards. In their experiment, 110 out of 129 DDR3 DIMMs were affected by bit flips. They also stated that the industry was aware of the problem since at least 2012 because then, Intel wrote patent applications regarding „the problem of row hammer“.

Seaborn and Dullien [2] published a PoC that exploits Rowhammer to gain kernel privileges by flipping bits in page tables one year later. The general approach is to fill the memory with PTEs by using *mmap* to map a file multiple times. Afterwards, Rowhammer is applied to flip a bit in memory. Suppose the bit flip occurred in a PTE in a way that it points to the PFN of another PTE. In that case, it is possible to modify that PTE which allows an attacker to map arbitrary PFNs in its virtual address space and gain kernel-level memory access. After that PoC was released, vendors started to roll out mitigations against Rowhammer.

Gruss, Maurice, and Mangard [6] demonstrated that it is possible to perform Rowhammer without the requirement of the *CLFLUSH* instruction used in previous PoCs in the same year. Their approach is to find patterns that can be used to perform *cache eviction*, e.g. getting data flushed from the cache by requesting other data instead of explicitly flushing it. They implemented a PoC that exploits Rowhammer from the JavaScript code within a browser.

In 2016, Pessl, Gruss, Maurice, *et al.* [7] introduced a software-based approach for reverse engineering the DRAM address functions, so it is no longer required to measure the address functions physically. Additionally, they described a new covert channel that exploits DRAM addressing.

Razavi, Gras, Bosman, *et al.* [8] demonstrated in the same year that it is possible to exploit Rowhammer in combination with KSM to flip bits on pages in another Kernel-based Virtual Machine (KVM) based VM on the same host.

In 2017, Jang, Lee, Lee, *et al.* [44] introduced an exploit based on Rowhammer that flips bits inside Software Guard Extensions (SGX) enclaves. Since the integrity of data within these enclaves is ensured by an integrity check, a bit flip within the enclave (even though the data is encrypted) results in the check failing. When an integrity check fails, the CPU locks itself, so a system reboot is required. According to the authors, this Denial of Service (DoS) is a severe threat, especially for public cloud providers who execute unknown and untrusted enclave programs for clients, which can impact the availability of the entire system.

Gruss, Lipp, Schwarz, *et al.* [11] introduced a new pattern for Rowhammer in the same year: *one-location*. Additionally, they demonstrate that it is possible to induce bit flips in the page cache where binaries are cached across multiple executions, which makes it possible to flip bits in a SUID binary to escalate privileges. By using an SGX enclave to execute the attack, it is

hidden from the OS. In contrast to their approach, the approach introduced in this thesis attacks the memory allocated by the userland processes and not the page cache.

Tatar, Konoth, Athanasopoulos, *et al.* [13] showed one year later that it is possible to induce bit flips not only by executing code on a local machine but via network from a remote machine. Their PoC demonstrated that for a fast, Remote Direct Memory Access (RDMA) enabled network, which is widely used in cloud environments and data centres, according to the authors.

In 2019, Cojocar, Razavi, Giuffrida, *et al.* [14] introduced an exploit that can be used to get bit flips using Rowhammer on systems where ECC is enabled, so the assumption that systems with ECC are not affected by Rowhammer was refuted.

Frigo, Vannacci, Hassan, *et al.* [15] introduced a new approach to bypass the TRR mitigation in 2020. Even though the DDR4 standard does not require it, TRR is implemented in many newer DDR4 DIMMs. The author's approach was to replace the well-known Rowhammer patterns with new, *many-sided* patterns that are generated automatically.

In 2021, Heckel [21], [22] introduced a novel approach that increases the number of bit flips found in a given time by a hundredfold. Some of the PoCs developed during that theses were included in the PoCs of this thesis.

Ridder, Frigo, Vannacci, *et al.* [19] introduced Synchronized MAny-Sided Hammering (SMASH) in the same year. SMASH is an approach to exploit bit flips on DIMMs that implement the TRR mitigation from JavaScript. Since the speed of a previous browser-based PoC [6] is insufficient for the many-sided patterns introduced in [15], SMASH provides a faster way to exploit Rowhammer from JavaScript.

In the same year, Jattke, van der Veen, Frigo, *et al.* [17] introduced a new, highly efficient approach that can be used to bypass the TRR mitigation. The authors stated that bit flips occurred on 40 out of 40 DDR4 DIMMs they tested. This thesis uses their PoC to find a system where many bit flips occur. However, in the experiments performed in this thesis, it was not possible to reproduce their results since bit flips were only found on 4 out of 60 DIMMs tested. Because the time for writing this thesis is limited, it was not possible to contact the authors and repeat the experiment. However, this will be done in the future.

6 Future Work

As described in Section 3.2, all DIMMs where bit flips were found were tested on *hammertest01*. A short experiment where the same DIMM was tested on *hammertest02* did not show any bit flips. In the future, it should be analyzed why bit flips occurred only on *hammertest01*, even though both systems have the same hardware.

An approach for further analysis would be to swap the hardware components one after another and verify if bit flips are found after swapping one component. If that is the case, the component responsible for the change should be inspected in more detail to find the reason for that behaviour.

It was not possible to confirm the results published by Jattke, van der Veen, Frigo, *et al.* [17] as described in Section 3.2. Since the code of the PoC is not well designed, it might be that some parameters have not been adjusted according to the author's intention during the automated performance of the experiments. In the future, the publication's authors should be contacted to find the reason for the reproduction problems and solve them. Afterwards, the experiments should be repeated to get better results and find DIMMs more susceptible to Rowhammer. In the end, those DIMMs can be used to evaluate if the approaches introduced in [21], [22] work on DDR4 as well.

The approach of HybridGrouping introduced in Section 4.1 is currently implemented as a single process, and, therefore, the total number of mappings is limited to 65 530. Since the experiments shown in Section 4.2 use one mapping per page, the number of pages would be limited to approximately 64 000, depending on the number of mappings done by the process additionally.

With a multi-process approach, the limit of mappings would count per process so that each process could have 65 530 mappings. So, it should be possible to get an arbitrary number of pages mapped using one mapping per page. However, a multi-process approach would require Inter-Process Communication (IPC) to synchronize the state between the different processes.

In the future, such an approach should be implemented to remove the limit on the number of pages that can be mapped.

In [21], the implementation of the reverse engineering approach introduced by Pessl, Gruss, Maurice, *et al.* [7] was shown. However, that implementation is limited to Intel x86 CPUs in the current version and does not work with AMD CPUs. With the insights about the timestamp counter resolution of AMD CPUs described in Section 4.1, it should be possible to modify the implementation to work on AMD CPUs by adjusting the calculation of the threshold and group timings with respect to the timer resolution. In the future, the implementation of the reverse engineering approach for address functions should be adjusted to work also on AMD CPUs.

The experiment results shown in Section 4.2.1 suggests that the reallocation percentage depends on the generation of the CPU. However, in this thesis, it was not possible to evaluate that assumption because there were not enough CPUs to perform the experiment on.

In the future, the experiment should be repeated on more CPUs of different generations to verify that assumption.

In Section 4.2.2, an experiment to measure the offset of reallocated pages was introduced. The measurements for that experiment were performed on systems with *normal* load, e.g. desktop environment, some services, etc.

In the future, the experiment should be repeated on a system with heavy allocation load: While the measurement is done, other processes should allocate memory. Depending on the allocation amount, the difference between the results could be analyzed to get more information about the correlation of memory allocation load and allocation offsets.

This thesis introduced a new approach for a privilege escalation exploit utilizing Rowhammer (see Section 4). However, it was not possible to build a working PoC since no stable way to modify the offset a freed page is allocated in a newly spawned process was found.

In the future, the tools introduced in this thesis should be used to continue the analysis of the remapping of pages between processes to find a stable way to get freed pages mapped at a specified offset in the child process. If such a way is found, a working PoC should be created to utilize that approach to exploit Rowhammer for privilege escalation.

7 Conclusion

In the scope of this thesis, an improved version of the test setup introduced in [21] was created. In addition to multiple architectural improvements, that new version supports a *local* mode which makes it possible to access shared storage (in the form of an *sshfs* mount) to store shared states between reboots and different systems. Thereby it is possible to perform experiments on many DIMMs where the only required interaction is to switch DIMMs after the system is powered off.

That automated test setup was used to evaluate the results presented by Jattke, van der Veen, Frigo, *et al.* [17]. During the experiment, their PoC was executed on 60 DIMMs. However, only 4 were susceptible to Rowhammer within the tested time of 6 h, so it was not possible to reproduce their result of 40 out of 40 DIMMs being affected by Rowhammer.

After that, a new approach for exploiting Rowhammer for privilege escalation using the memory allocated by userland processes was introduced: The parent process allocates memory and scans it for bit flips. If a vulnerable location is found, the *victim page* is freed, and a child process is created. It is required to get the freed victim page allocated at a specific offset within the child process. Afterwards, the bit flip can be triggered again, so the bit flips in the page that now belongs to the child process. Thereby it should be possible to modify the memory of the child process which leads, if an exploitable location is flipped, to a change in the child execution flow and finally to a privilege escalation when a *SUID* binary was executed.

To get the remapping working, it is required to free a page in a way that it gets allocated again by a child process spawned after the page was freed. This does not work when THPs are allocated since it is possible to free single pages within a THP, but they are not allocated again. For that reason, THPs can not be used. To increase the speed of the grouping of addresses by DRAM bank and use the known address functions (which can be reverse engineered before), a new approach named HybridGrouping is introduced.

The measurements for HybridGrouping showed that the timestamp counter on different CPUs has different resolutions. That was evaluated with multiple CPUs to understand the differences better. Even though that is not directly relevant to HybridGrouping, that insight can be used in the future to improve the reverse engineering tool introduced in [21] to work on AMD CPUs as well.

Since HybridGrouping can be used to group addresses by CPU core, the next step would be to analyze the reallocation percentage, e.g. how many of the pages freed by the parent process are allocated in the child process. That experiment was performed, and the results showed that if CPU pinning is enabled, the majority of pages (more than 97 %) freed by the parent are allocated by the child.

Next, the offset of the reallocated pages was analyzed. The results showed that there is a clear pattern of the offsets. However, the standard deviation is relatively high, so it is not possible to

determine a precise offset. So, the pages are remapped to the child process, but the offsets differ between multiple executions.

Since the offsets are unstable the behaviour has been inspected in more detail. This was achieved by implementing the kernel module AllocTrace, which provides an interface in procfs which can be used to access the PFNs currently in the buddy allocator's different orders or the different per-CPU lists.

In order to reduce memory noise while the snapshots of the files provided by AllocTrace are created, an in-memory copy tool MemCP was implemented. It splits the copy process into three stages: Initialization, snapshot creation, and snapshot storing. It is possible to perform the initialization step that allocates the memory required for the buffers before creating the snapshot. Then, the data is stored on disk, which frees the buffers after the snapshot is done.

Using AllocTrace and MemCP, three snapshots were created of the buddy allocator during the execution of the parent and child processes: The first one before the parent process is started, the second one after the parent freed the pages and before the child is started, and the third one after the child freed the pages.

Those snapshots were set in relation to the PFNs allocated by the parent and the child to better understand the buddy allocator including per-CPU lists.

References

[1] Y. Kim, R. Daly, J. Kim, *et al.*, „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 (visited on 11/22/2022).

[2] M. Seaborn and T. Dullien. „Exploiting the DRAM rowhammer bug to gain kernel privileges“. (2015), [Online]. Available: https://www.cs.umd.edu/class/fall2019/cmsc818O/papers/rowhammer-kernel.pdf (visited on 10/07/2022).

[3] R. K. Konoth, M. Oliverio, A. Tatar, *et al.*, „ZebRAM: Comprehensive and compatible software protection against rowhammer attacks“, in *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, p. 15. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/konoth (visited on 10/07/2022).

[4] Z. B. Aweke, S. F. Yitbarek, R. Qiao, *et al.*, „ANVIL: Software-based protection against next-generation rowhammer attacks“, in *Proceedings of the 21th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, Mar. 25, 2016, pp. 743–755. DOI: 10.1145/2872362.2872390.

[5] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, „CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory“, presented at the Proceedings of the 26th USENIX Security Symposium, 2017, pp. 117–130, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser (visited on 10/07/2022).

[6] 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 (visited on 11/22/2022).

[7] 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.

[8] 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 (visited on 11/22/2022).

[9] V. van der Veen, Y. Fratantonio, M. Lindorfer, *et al.*, „Drammer: Deterministic Rowhammer Attacks on Mobile Platforms“, in *CCS*, Pwnie Award for Best Privilege Escalation Bug, Android Security Reward, CSAW Best Paper Award, DCSR Paper Award, Oct. 2016. [Online]. Available: https://vvdveen.com/publications/drammer.pdf (visited on 11/22/2022).

[10] R. Qiao and M. Seaborn, „A New Approach for Rowhammer Attacks“, in *Proccedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 161–166. DOI: 10.1109/HST.2016.7495576.

[11] D. Gruss, M. Lipp, M. Schwarz, *et al.*, „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 (visited on 11/22/2022).

[12] M. Lipp, M. Schwarz, L. Raab, *et al.*, „Nethammer: Inducing Rowhammer Faults through Network Requests“, in *IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, 2020. DOI: 10.1109/EuroSPW51379.2020.00102.

[13] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, „Throwhammer: Rowhammer Attacks over the Network and Defenses“, in *USENIX ATC*, Pwnie Award Nomination for Most Innovative Research, Jul. 2018. [Online]. Available: https://download.vusec.net/papers/throwhammer_atc18.pdf (visited on 11/22/2022).

[14] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, „Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks“, in *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, May 2019. [Online]. Available: https://download.vusec.net/papers/eccploit_sp19.pdf (visited on 10/07/2022).

[15] P. Frigo, E. Vannacci, H. Hassan, *et al.*, „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 (visited on 11/22/2022).

[16] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, „RAMBleed: Reading Bits in Memory Without Accessing Them“, in *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020, pp. 695–711. DOI: 10.1109/SP40000.2020.00020.

[17] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, „BLACKSMITH: Rowhammering in the Frequency Domain“, in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (IEEE S&P)*, Nov. 2021. [Online]. Available: https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf (visited on 10/07/2022).

[18] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, „Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications“, in *Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 1198–1213. DOI: 10.1145/3466752.3480110.

[19] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, „SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript“, in *Proceedings of the 30th USENIX Security Symposium*, Aug. 2021. [Online]. Available: https://download.vusec.net/papers/smash_sec21.pdf (visited on 10/07/2022).

References

[20] A. Kogler, J. Juffinger, S. Qazi, *et al.*, „Half-Double: Hammering from the next row over“, in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3807–3824, ISBN: 978-1-939133-31-1. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double` (visited on 11/22/2022).

[21] M. Heckel, „Hammerkit – Toolset for Memory inspection and automatic Rowhammer detection“, Hof University of Applied Sciences, 2021.

[22] M. Heckel, „RAEAX – Rowhammer Amplification by Execution of Additional X86 instructions“, Hof University of Applied Sciences, 2021.

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

[24] M. Gorman, „An Investigation into the Theoretical Foundations and Implementation of the Linux Virtual Memory Manager“, M.S. thesis, University of Limerick, Apr. 2003. [Online]. Available: `https://www.kernel.org/doc/gorman/pdf/thesis.pdf` (visited on 10/07/2022).

[25] „SWAPON (8) system administration“. (Apr. 2, 2021), [Online]. Available: `https://man7.org/linux/man-pages/man8/swapon.8.html` (visited on 09/06/2022).

[26] F. Adamsky, „Low-Level Fundamentals“, 2020, Slides for the lecture IT-Sicherheit.

[27] „AMD64 Architecture Programmer's Manual Volume 2: System Programming“, AMD. (Oct. 2022), [Online]. Available: `https://www.amd.com/system/files/TechDocs/24593.pdf` (visited on 11/09/2022).

[28] „MADVISE(2) linux programmer's manual“. (Mar. 22, 2021), [Online]. Available: `https://man7.org/linux/man-pages/man2/madvise.2.html` (visited on 09/06/2022).

[29] „MMAP(2) linux programmer's manual“. (Mar. 22, 2021), [Online]. Available: `https://man7.org/linux/man-pages/man2/mmap.2.html` (visited on 09/06/2022).

[30] M. Seaborn. „Measuring the DRAM refresh rate by timing memory accesses“. `commit 2013bb50a3db541d4c969ca87471e16e866415bd`. (2015), [Online]. Available: `https://github.com/google/rowhammer-test/blob/master/refresh_timing/README.md` (visited on 11/22/2022).

[31] „Improving Real-Time Performance by Utilizing Cache Allocation Technology“, Intel. (2015), [Online]. Available: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf` (visited on 09/07/2022).

[32] „Examining process page tables“. (2019), [Online]. Available: `https://www.kernel.org/doc/html/latest/admin-guide/mm/pagemap.html` (visited on 11/22/2022).

[33] L. Cojocar, J. Kim, M. Patel, *et al.*, „Are we susceptible to rowhammer? an end-to-end methodology for cloud providers“, in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 712–728. DOI: `10.1109/SP40000.2020.00085`.

[34] „Annex K: Serial presence detect (SPD) for DDR3 SDRAM modules“, JEDEC Solid State Technology Association. (2014), [Online]. Available: https://www.jedec.org/standards-documents/docs/spd-4010211 (visited on 10/07/2022).

[35] „Annex L: Serial presence detect (SPD) for DDR4 SDRAM modules“, JEDEC Solid State Technology Association. (2015), [Online]. Available: https://www.jedec.org/standards-documents/docs/spd412l-3 (visited on 10/07/2022).

[36] „Archiso“. (Aug. 25, 2022), [Online]. Available: https://wiki.archlinux.org/title/Archiso (visited on 09/09/2022).

[37] „SSHFS(1) user commands“. (Apr. 2008), [Online]. Available: https://man7.org/linux/man-pages/man1/SSHFS.1.html (visited on 10/07/2022).

[38] „Bash(1) — linux manual page“. (Oct. 29, 2020), [Online]. Available: https://man7.org/linux/man-pages/man1/bash.1.html (visited on 10/31/2022).

[39] K. A. Shutemov. „Pagemap: Do not leak physical addresses to non-privileged userspace“. (2015), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce (visited on 09/12/2022).

[40] „Documentation for /proc/sys/vm/“. commit dff033818a06e7d0bf79271e34bda11c2d9d98d0. (Jun. 28, 2022), [Online]. Available: https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/sysctl/vm.rst (visited on 09/19/2022).

[41] „Ftrace - function tracer“. (2017), [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (visited on 09/26/2022).

[42] „Trace-cmd(1) — linux manual page“. (May 27, 2020), [Online]. Available: https://man7.org/linux/man-pages/man1/trace-cmd.1.html (visited on 09/26/2022).

[43] „Stat(2) — linux manual page“. (Aug. 27, 2021), [Online]. Available: https://man7.org/linux/man-pages/man2/lstat.2.html (visited on 09/27/2022).

[44] Y. Jang, J. Lee, S. Lee, and T. Kim, „Sgx-bomb: Locking down the processor via rowhammer attack“, Oct. 2017, pp. 1–6. DOI: 10.1145/3152701.3152709.

[45] „Ncat(1) — linux manual page“. (Jun. 8, 2021), [Online]. Available: https://man7.org/linux/man-pages/man1/ncat.1.html (visited on 10/27/2022).

[46] „TASKSET(1) — linux manual page“. (Apr. 2, 2021), [Online]. Available: https://man7.org/linux/man-pages/man1/taskset.1.html (visited on 09/19/2022).

[47] „Gnuplot homepage“. (Oct. 2020), [Online]. Available: http://www.gnuplot.info/ (visited on 10/05/2022).

Appendix

Contents

A Reproduction Guide

The source code of the tools described in this section can be accessed at `https://rowhammer.xitokero.de/master_submit.zip`. In order to ensure that the source code is the same as it was at the submission of this thesis, the `SHA256` hash of the zip file shown in Listing 3 can be used.

```
b41d102a41e2565af408ff4e69c4162bb40d11ff77535f2e41e5d062bc4b70da
```

Listing 3: `SHA256` hash of the zip file

It should be noted that lines that should be typed into a terminal as normal user start with a "`>`" in the following sections. Lines that should be typed into a terminal as superuser start with "`#`". Lines not starting with one of these characters represent output.

A.1 Dumping Offsets of virtual Addresses

The sizes of the different parts of the virtual address depicted in Figure 2 were measured using the kernel module in the directory *MeasurePageSize* of the submission. Since compiling the kernel module requires the kernel headers, they have to be installed on the system if that is not already the case. For arch linux with the default kernel, the headers can be installed like this:

```
# pacman -S linux-headers
```

In order to get the information, the kernel module has to be built afterwards:

```
> cd MeasurePageSize
> make
```

Afterwards, the compiled module can be executed on the current system using the following command:

```
# insmod measurePageSize.ko
```

The output can be retrieved using *dmesg*:

```
# dmesg
```

A.2 ISO image for automated testing

The automated test setup can be modified as described in the *README.md* in the directory *TestSetup*. After the modifications (e.g. adding a PoC or changing the configuration) are done, the ISO for the *local* mode can be built with the following commands:

```
> cd TestSetup
> make local
```

It should be noted that the mount of the *sshfs* will fail in the submitted state since it has to be adjusted to the local setup. The following adjustments are required:

- Add the fingerprint of the *sshfs* server to the file *ISO/localroot/root/.ssh/known_hosts*.
- Adjust the username, IP, and path of the *sshfs* server to the file *ISO/localroot/root/autostart.sh* at line 8.
- Create an *SSH* key pair without using a passphrase and copy the *private key* (named *id_rsa* by default) to *ISO/localroot/root/.ssh/id_rsa*. Add the corresponding *public key* to the file */home/USER/.ssh/authorized_keys* on the *sshfs* server.
- Create a file named *<MAC address>_module.txt* for each system that should be used, e.g. *f4b5203a1337_module.txt* and add the following content to the files:

```
steps=1
dimm=1
```

  Thereby, *steps* is the amount the module should be increases after each run and *dimm* is the number of the next DIMM that should be tested.

Afterwards, the *sshfs* share should be automatically mounted and the experiments should be performed according to the configuration.

For the remote mode, the adjustments described above are not required. Instead, the *SERVER* and *PORT* variables in *run.sh* have to be adjusted to be able to connect to the server. In the current version, the server is a simple *netcat* [45] based script. It is required to adjust the *UPORT* variable in the *server.sh* file. Afterwards, *server.sh* can be executed on the system that should be the server.

Afterwards, the communication with the server can be enabled by removing the *noauto* command line parameter to *run.sh* in the file *ISO/remoteroot/root/autostart.sh* at line 4. If that is not done, the results are only shown at the local console of the test system and not sent to any server.

When these modifications are done, the *remote* ISO image can be created using the following commands:

```
> cd TestSetup
> make remote
```

A.3 Hybrid Grouping

The PoC for the HybridGrouping approach introduced in Section 4.1 can be found in the directory *HybridGrouping*. In the current version, it is implemented for systems where one page does not span multiple banks which was the case for all test systems (typically, this is the case for all systems where only one channel is used). In order to execute the PoC, it has to be built first:

```
> cd HybridGrouping
> make
```

Afterwards, it can be executed using the following command:

```
# ./bin/hybridGrouping
```

The execution requires root privileges to dump the real block sizes which is needed for evaluation.

The results of an execution on one of the test systems is shown in the following listings. In order to increase the readability, all outputs that are not related to the block size are omitted (lines starting with # are comments and not part of the output).

```
# Block sizes measured based on the PFN
[00967] Assuming a block size of 2
[05975] Assuming a block size of 4
[18495] Assuming a block size of 8
[45847] Assuming a block size of 16
# ...
# Block sizes found with hybrid grouping
[13123] Increasing block size to 2
[13179] Increasing block size to 4
[18503] Increasing block size to 8
[45895] Increasing block size to 16
# ...
# Result stats
Grouped 52948 addresses with 0 errors (  0.00%).
```

Since the first 20% of the addresses are skipped to improve speed (by skipping small blocks), the HybridGrouping approach started at offset 13 107, so the offsets for block sizes 2 and 4 could not be detected correctly because the correct offsets were skipped.

A.4 Timing resolution tool to measure RDTSCP accuracy

The tool used to measure the rdtscp resolution can be found in the directory *MeasureTiming*. It can be built using the following commands:

```
> cd MeasureTiming
> make
```

Afterwards, it can be executed with the following command:

```
> ./bin/measureTiming
```

Because the tool was initially intended to analyze the correlation between the order of memory blocks allocated by the buddy allocator and the time required to allocate the memory (it was assumed that the split of a high order block to smaller blocks should take more time than directly using a lower order block), the tool has the possibility to access the PFNs, which requires root privileges. If it is executed without root privileges and the usage of PFNs is enabled, the PFN is always zero, so the groups of continuous PFNs are not highlighted correctly. The following command enables PFN mode:

```
# ./bin/measureTiming -p -d
```

It showed during the experiments that there was no correlation between block size (e.g. buddy allocator order) and allocation time. However, it showed that the access times are discrete with values bigger than 1, e.g. the *rdtscp* counter has a resolution worse than 1.

The greatest common divisor of all access times is equal to the resolution of the counter, so the tool can be used to measure the resolution of the *rdtscp* counter as well. There is an additional flag to measure the access times for the *gettime* function:

```
> ./bin/measureTiming -g
```

For more features and a help page with more information about features and additional command line flags, the following command can be used:

```
> ./bin/measureTiming -h
```

See Table 2 for a detailed overview of timer resolutions on different systems.

A.5 Calculation of remapping percentage

The PoC to measure the remapping percentage can be found in the directory *MeasureRemap*. It can be built using the following commands:

```
> cd MeasureRemap
> make
```

Then, it can be executed using the following command:

```
> sh performExperiment.sh
```

Since access to physical addresses is required for page tracking (e.g. if the same PFN is allocated in the parent and the child, the PoC itself is executed with *sudo* within *performExperiment.sh*.

By default, there is no CPU pinning, so the child might run on another logical CPU core than the parent. It is possible to pin both processes to a specified logical CPU core by using the following command:

```
> taskset 0x1 sh performExperiment.sh
```

Thereby, *0x1* is the number of the logical CPU, e.g. CPU 0. See the man page of taskset [46] for more options.

See Section 4.2.1 for a detailed description of the implemented approach and the results of the performed measurements.

A.6 Calculation of remapping offsets

The PoC used to measure the remapping offsets is the same that was used to measure the remapping percentage described in Section A.5. However, the evaluation was adjusted to measure the offsets of the remapping (e.g. on which offset the n^{th} page freed in the parent process was mapped in the child process instead of measuring the percentage of overall remapping). See Section 4.2.2 for a more detailed description of the experiment.

The first step required to reproduce the results is to compile the PoC and evaluation binaries:

```
> cd MeasureOffset
> make
```

Afterwards, the experiment can be performed by executing the *performExperiment.sh* script:

```
> sh performExperiment.sh
```

The script will perform 100 measurements which are done by calling the script *measureOffset.sh* for each measurement. Afterwards, the overall evaluation script *evaluateAll.sh* is called to perform the evaluation of all experiments (e.g. calculate average values, standard deviations, percentage of remapped pages in the *anonymous* section, etc.).

Finally, the remapping percentage is shown on standard output and a file *results.dat* is generated which contains the results of the experiment in the following format:

```
<parent offset> <μ_child − σ_child> <μ_child> <μ_child + σ_child>
```

The Number of parent offsets that are concluded to one line in the result file is hard coded in the file *evaluateAll.c* and can be adjusted, if that is required. Currently, it is set to 32, so 32 parent offsets are concluded by calculating the average for those data sets. Afterwards, only one line is stored in the result file.

As before, the experiment can be performed with CPU pinning by using *taskset*:

```
> taskset 0x1 sh performExperiment.sh
```

A.7 Access PFNs within the buddy allocator

The source code of the kernel module AllocTrace described in Section 4.2.3 can be found in the directory *AllocTrace*. Since compiling the kernel module requires the kernel headers, they have to be installed on the system if that is not already the case. For arch linux with the default kernel, the headers can be installed like this:

```
# pacman -S linux-headers
```

Afterwards, the kernel module can be compiled using the following commands:

```
> cd AllocTrace
> make
```

It should be noted that the kernel version the module was compiled on has to be equal to the kernel version it is used on, e.g. it is required to recompile the module after a kernel update. This can be done with the following commands:

```
> make clean
> make
```

After the module is compiled, it can be loaded with the following command:

```
# insmod allocTrace.ko
```

Loading the module writes to the *dmesg* log which can be accessed using the following command:

```
# dmesg
```

If everything was successful, the following output should be in the log:

```
[<time>] [allocTrace]: Zone DMA High: 4 Batch: 1.
[<time>] [allocTrace]: Zone DMA32 High: 252 Batch: 63.
[<time>] [allocTrace]: Zone Normal High: 932 Batch: 63.
[<time>] [allocTrace]: Zone Movable High: 4 Batch: 1.
[<time>] [allocTrace]: Zone Device High: 4 Batch: 1.
[<time>] [allocTrace]: Loaded
```

Now, there should be a directory */proc/allocTrace* with the structure as described in Section 4.2.3.

The module can be unloaded by using the following command:

```
# rmmod allocTrace
```

A.8 Multistage copy

The approach of copying files using preallocated buffers and splitting the copy process to multiple stages as described in Section 4.2.3 is implemented in the PoC in the directory *MemCP*. It can be compiled with the following commands:

```
> cd MemCP
> make
```

Afterwards, it can be executed with the following command:

```
> ./bin/memcp [basePath] [targetPath]
```

When the initialization is done, the following output is written to stdout:

```
Initialization done. Send SIGUSR1 to my PID <PID> to create an in-memory
snapshot.
```

Then, the signal can be sent using the following command in order to proceed to the next stage:

```
> kill -SIGUSR1 <PID>
```

When the copy of the files is done, the following output is shown in stdout:

```
In-memory snapshot done. Send SIGUSR1 to my PID <PID> to copy the data to the
submitted target directory.
```

After that, the command to send the signal to the process can be repeated to proceed to the final stage:

```
> kill -SIGUSR1 <PID>
```

It should be noted that, since the PoC was implemented for the files in */proc/allocTrace*, it does only support files that do not contain the carriage return (`\r`) character in the current version since that is used as control character to mark the lines that contain paths.

A.9 Buddy Allocator Inspection

The offsets within the range of allocated pages and within the lists of the buddy allocator (as well as per-CPU lists) as described in Section 4.2.3 can be measured by using the files in the *MeasureAllocation* directory. First, the components have to be compiled which can be done using the following command:

```
> cd MeasureAllocation
> make
```

For the performance of the experiment it is required to load AllocTrace as described in Section A.7. Next, the script *measure.sh* can be used to perform the experiment. Since it uses *kill* to send the required signals to MemCP, it is required to run the script in the current console (e.g. not execute it with *sh*, but with *source*). For some reason the sending of the signal SIGUSR1 fails when the script is executed in a new shell. Type the following command to perform the experiment:

```
> source measure.sh
```

Afterwards, the evaluated results are stored in the *data_tmp* directory. Additionally, graphs are generated in the *plots* directory using gnuplot [47].

Eidesstattliche Erklärung

Ich versichere durch meine Unterschrift, dass ich diese Masterarbeit mit dem Titel „We don't need no Exploitation– Systematic Investigation of Process-based Rowhammer Exploitation" 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, 22. November 2022 ....

Ort, Datum Martin Heckel