On Feasibility and Performance of Rowhammmer Attack

Varnavas Papaioannou

A dissertation submitted in partial fulfillment

of the requirements for the degree of

MSc Information Security

of

University College London.

Supervisor: Dr. Nicolas Courtois

September 4, 2017

Abstract

The aim of this thesis is to study the Rowhammer attack and evaluate its feasibility on practical exploitation scenarios in Linux. Currently, all the implementations released, capable of performing the Rowhammer attack require elevated privileges, which is a very strong requirement and somehow puts the attack into the theoretical spectrum. The purpose of this report is to explore and implement different techniques that would allow the execution of the Rowhammer attack in userspace. More specifically, we provide three implementation, each of them having different strength of requirements but with one characteristic in common: the capability of executing the Rowhammer attack without elevated privileges. We then proceed and evaluate the performance of the new implementations with comparison to the performance of the already existing tools. At the end, we see that not only it was possible to reach similar levels of performance with the programs that required elevated privileges but in some cases even outperform them, in both native and virtual environments.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Nicolas Courtois for his invaluable support, guidance and useful suggestions in this MSc project as well as his provision of resources necessary for the completion of this thesis.

It shouldn't go without mentioning the importance of the role of the cryptanalysis course COMPGA18 in the whole project. Throughout that course we had the initial exposure on the subject where along with a team of other students experimented and explored the topic of the current thesis.

Contents

1 Introduction 11

2 Background on Computer Memory 12

2.1 DRAM 12

2.2 Virtual Memory 14

3 The Rowhammmer Side Channel 16

3.1 Attack Foundations 17

3.1.1 Requirement 1: Bypassing the CPU Cache 17

3.1.2 Requirement 2: Bypassing the Rowbuffer 19

3.2 Inducing the Rowhammer Bug 22

3.2.1 Single Sided Rowhammer 23

3.2.2 Row Targeted Rowhammer 23

3.3 Rowhammer Bit Flips Exploitation 24

3.3.1 Spraying the Memory 25

3.3.2 Targeted Approach 27

4 User-Space Rowhammer Testing Implementations 28

4.1 THP Based Implementation 28

4.1.1 hprh 28

4.2 SBDR Timing Channel Based Implementations 29

4.2.1 tcrh 29

4.2.2 thrh 31

4.3 Further Experimentation 31

5 Experiments 33
5.1 Implementation Details of Already Existing Tools 33
5.2 Measurements 36
5.3 Discussion 36

6 Conclusion and Future Work 41

Bibliography 41

List of Figures

2.1 Memory Cell Structure, Source: [1] (Modified) 12

2.2 DRAM Structure, Source: [2] (Modified) 13

2.3 Virtual Memory Mapping 14

2.4 Virtual Address to DRAM Dataflow 15

3.1 Example of the Multiple Levels of Memory Hierarchy 17

3.2 Illustration of Sandy Bridge DRAM Mapping Found in [3] 20

3.3 Illustration of Double Sided Rowhammer, Source: [2] (Modified) . . 24

3.4 Page Table Exploitation 26

4.1 The New DRAM Mapping Function 30

List of Tables

3.1 Code for Inducing the Rowhammer Vulnerability 22

3.2 Sandbox Evasion Example 25

5.1 Overview of Rowhammer Testing Tools 35

5.2 Performance Results (Total Number of Induced Bit Flips) 37

Chapter 1

Introduction

A key assumption in modern computer security is that a memory location may only be modified by directly addressing that specific location. However, as it was shown by Kim, Daly, Fallin et al. in [4] in 2014, this assumption is not true. This is the core principle of the Rowhammer attack, where a sophisticated adversary may be able to alter memory locations owned by different applications by performing specific memory access patterns on his own memory. As it was shown over the last years, this is an extremely dangerous vulnerability and it can efficiently exploited under different settings and may expose a computer system to attacks even in absence of any software bugs. The seriousness of this vulnerability was acknowledged by the Linux community and they took some mititgation measures to restrict the execution of the attack to privileged users. All the currently available tools capable of inducing the Rowhammer vulnerability are prevented to run in userspace by those mitigation measures and that's why our goal is to find new ways to circumvent this restriction and assess whether or not they are practical to use for exploitation.

Chapter 2

Background on Computer Memory

2.1 DRAM

Modern DRAM modules hierarchy is organized into multiple levels. At its bottom, there are the memory cells, which are the main components for storing information within DRAM modules . A memory cell, as shown in Figure 2.1, is composed by one transistor and one capacitor. The transistor is used to enable the potential charging of the capacitor, which at the end will hold the encoded one-bit information. Due to the nature of the capacitor, there is current leakage through the access transistor, so in order to retain its value within the predefined noise margins, a periodic refreshing within the expected lifetime of the information is necessary. During the refreshing time, the capacitor gets recharged to its original levels.

The memory cells are then grouped into a two-dimensional grid with rows and columns to form the next level into the hierarchy, the bank. A collection of banks is

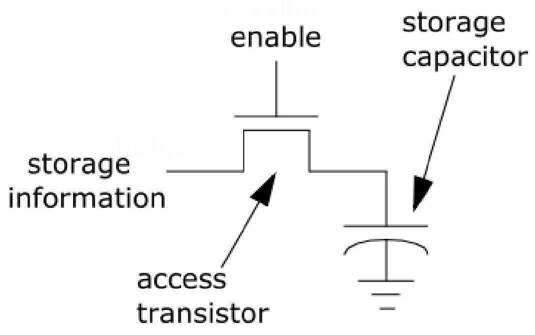

Figure 2.1: Memory Cell Structure, Source: [1] (Modified)

further combined to form a chip on the module. A chip has a specified data width (typically 4 or 8 bits) and so multiple chips are utilized in order to fill the bus width of the module (typically 64 bits or 72 with ECC) and form the top level of the hierarchy, the rank. Modern DRAM modules may have 1,2 or 4 ranks. Finally, since the total capacity of a DRAM module is limited, it is common for memory controllers to support access to multiple modules and even provide increased bandwidth by accessing them in parallel. In the latter case, the memory controller groups the DRAM modules in channels, where each channel can be accessed independently from the others. The operations on the DRAM module most of the times are directly controlled by the memory controller. When there is memory access request, the memory controller maps the corresponding physical address into the DRAM hierarchy using a bijective function. What that means, is that for each physical address, there is a corresponding channel, DRAM module, rank, bank, row and column. After mapping the address, the memory controller activates the row where the data resides in which causes the data to be sensed by the sense amplifiers and stored

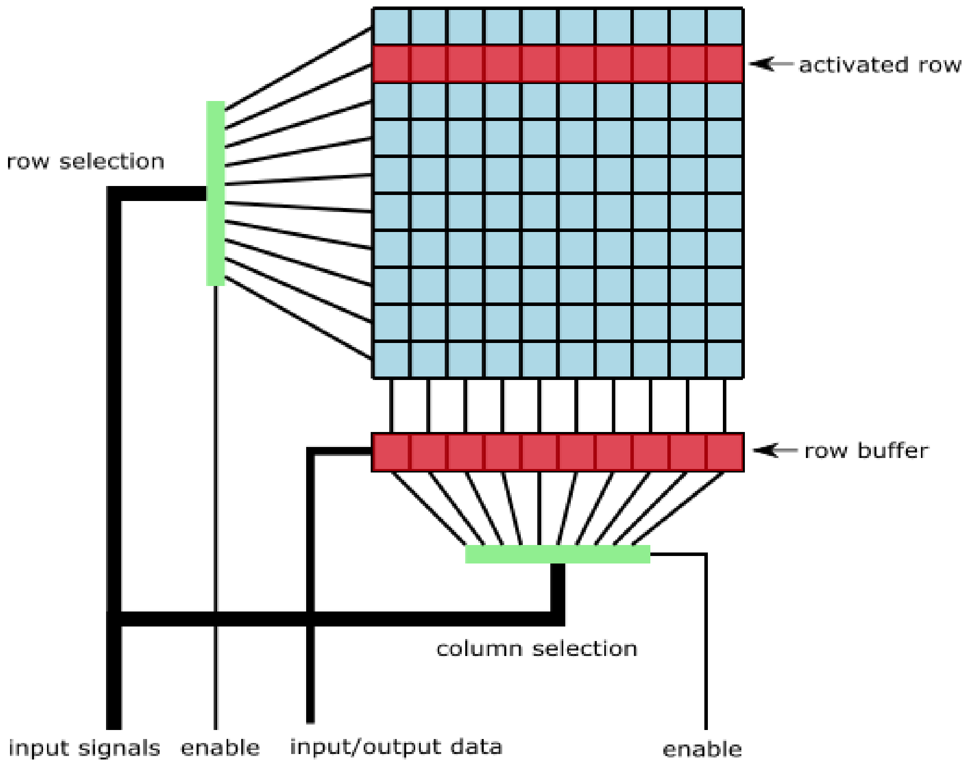

Figure 2.2: DRAM Structure, Source: [2] (Modified)

into the row-buffer. The row buffer in this case essentially acts as a cache to facilitate faster access to data with spatial locality (in the DRAM module) and each bank has its own row-buffer. Finally, after the data gets sensed into the row-buffer, the input signals for the data columns are used to return the requested data. A graphical way to describe the above operation can be observe in Figure 2.2

2.2 Virtual Memory

The detailed description of virtual memory is not within the scope of this thesis. For the purpose of this thesis, we will overview some aspects of virtual memory and its general operation. On a higher level, virtual memory can be considered as an intermediate layer between the user and the physical memory. As such, in this model the user interacts only with the virtual memory. A simple a example of how the model of virtual memory works can be seen in Figure 2.3. As we can see in the figure, a virtual memory location could correspond to a location in physical memory, like the physical frame 9 in this example, but it could also map to the disk or even be invalid for the current user. These virtual memory mappings are stored inside special data structures called Page Tables. Page Tables are composed by Page Table Entries (PTE) where each PTE holds the mapping of a virtual memory region. Now due to how virtual memory operates, each memory access has to be preceded by the calculation of that mapping. This operation is commonly implemented in hard-

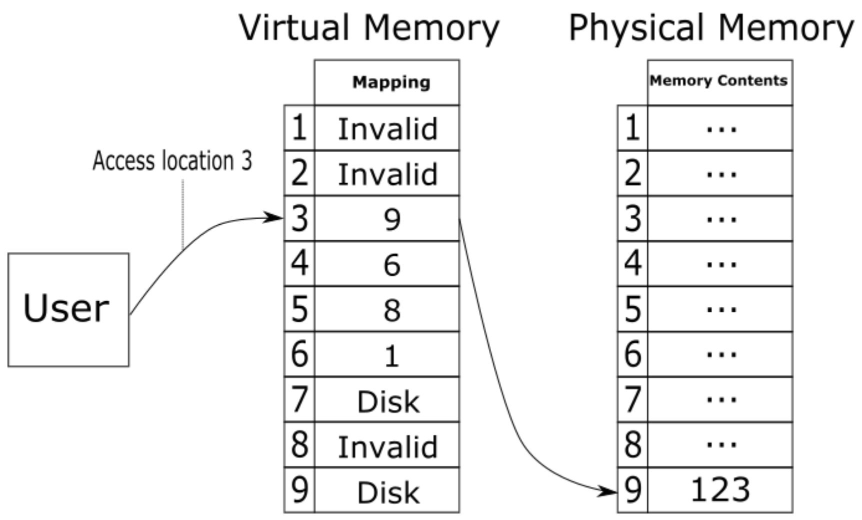

Figure 2.3: Virtual Memory Mapping

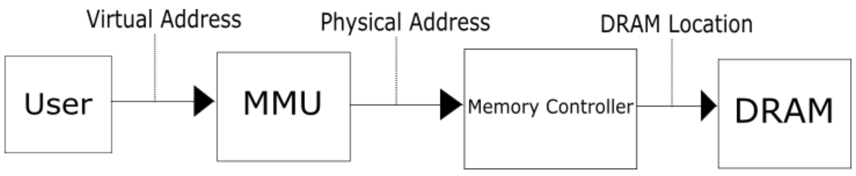

Figure 2.4: Virtual Address to DRAM Dataflow

ware for performance reasons. Modern processors include a Memory Management Unit (MMU) which has as part of its responsibilities the virtual address translation. The units normally engaged when there is a memory access can be observed in Figure 2.4. In this example, we have also included the memory controller, which is normally part of the CPU and as illustrated in the figure is responsible for the translation of a physical address to a DRAM memory location. This is a quick introduction into the virtual memory, for the reader who is interested in learning more about memory in general, Ulrich Drepper has written a detailed paper in [5] and is an excellent source of information.

Chapter 3

The Rowhammmer Side Channel

After defining the basics of DRAM architecture and the quick virtual memory overview, it's time to introduce the Rowhammer phenomenon. The term Rowhammer in the memory context was formally defined and publicized by Kim, Daly, Fallin et al. [4] in 2014. It is described as the process of repeatedly activating specific rows within a bank which had the side-effect of flipping bits in neighboring rows. This behavior, as it was mentioned in the paper, is a direct consequence of the inter-cell coupling effects that accelerate charge leaking in adjacent rows which eventually provokes bit flips in memory locations which potentially belong to another process. The disturbance caused by those effects is amplified by the increased density of modern memory cells e.g. in DDR3 due to poor isolation between neighboring cells. The end result, bit flips in different rows, adjacent to those ”Hammered” by the attacker.

The success of inducing the Rowhammmer perturbation relies on two capabilities on the attacker's side. First, the attacker has to be capable of bypassing the CPU cache for his memory accesses. This is due to the fact that when a memory access is already in the CPU cache, the DRAM module will not get activated and instead the memory access will be served by the CPU cache. After bypassing the CPU cache, the attacker needs to be able to avoid hitting the target's bank row buffer which is described in §2.1. By accessing memory locations which are already in the row buffer will prevent the corresponding row to get activated and so the attack cannot be performed.

Even though the attack's requirements are straightforward, their implementation in practice can hide various complexities. In the next chapter we will discuss various ways of implementing each of the requirements that could be useful in different scenarios.

3.1 Attack Foundations

3.1.1 Requirement 1: Bypassing the CPU Cache

Full description of cache memory is not within the scope of this thesis. We will only provide the elements essential for the Rowhammer attack. Modern processors utilize multiple levels of cache memory in order to keep to the minimum the required DRAM memory accesses. An important characteristic of each cache level is its cache inclusion policy. We will say that a cache memory is *higher* in the memory hierarchy if it's closer to the CPU. For example, L1 cache will be always the highest ranking type of memory among the rest of the cache levels. An example of the various levels of a modern memory hierarchy can be seen in Figure 3.1.

Most common types of cache inclusion policies are Inclusive, Exclusive and Non-Inclusive/Non-Exclusive (NINE). When a cache level is Inclusive, it is guaranteed to include all the data found in the higher cache levels. When a cache level is Exclusive, then it is guaranteed that it doesn't include any data found in the cache levels above its own. With the NINE, there is no guarantee to be neither Inclusive nor Exclusive. Having this brief description of the cache, we will review various

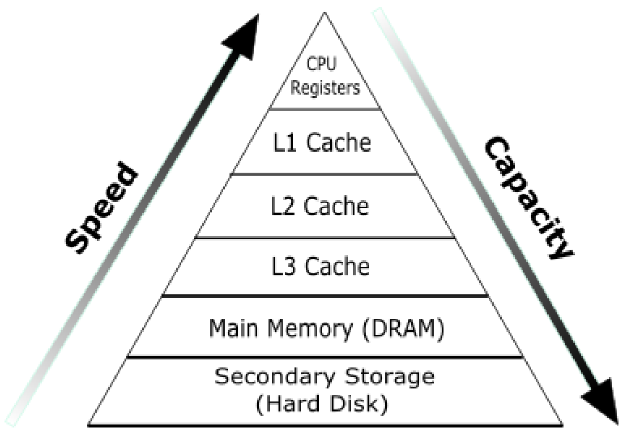

Figure 3.1: Example of the Multiple Levels of Memory Hierarchy

techniques which aim at bypassing the cache.

Architecture Specific Instructions Since the CPU cache plays a critical role in the system's performance optimization, modern CPU architectures are designed to allow fine-grained functionality which allows to control how the cache will be used. One commonly found fine-grained method is the invalidation of a cache entry (from all the cache levels). For example, the x86 processors may include the instruction clflush [1], which takes a memory location as input and removes the associated cache line from all levels in processor's cache hierarchy. Now even though various computing architectures support such functionality, it's not always the case that such functionality will be available in the user space. For example, clflush instruction is x86 is available in all privilege levels, however in ARM processors the equivalent instruction requires elevated privileges. Calling clflush is the fastest and the most effective way of bypassing the CPU cache.

Another functionality often found in CPU architectures is support for non temporal memory access. Primarily focused towards avoiding cache pollution with data that are known not to be used in the near future, modern processors offer instructions that effectively bypass the cache for the specified memory access. The use of Non-Temporal instructions for the Rowhammer attack was first mentioned in [6] and successfully used in [7].

Cache Eviction The most intuitive way to direct a request to DRAM and avoid the whole cache complexity is by filling the whole cache with irrelevant data. This can be achieved by accessing a buffer that is as large as the last level cache. This way, it is possible to evict a target cache line which was e.g. demonstrated in [8]. Even though this technique is effective irrespective to the implemented cache coherence policies, it is particularly ineffective within the context of Rowhammer. If we use it, we need to cause eviction of the whole cache just in order to evict a much smaller quantity of data related with the rowhammer attack. Accordingly, this technique is typically considered totally impractical for Rowhammer.

A better way would be to achieve the eviction of a single block of data which

[1]The clflush instruction was introduced with Streaming SIMD Extensions 2 (SSE2) and its availability can be verified throughout its CPUID feature flag.

includes a target memory location. For example, consider a CPU where Last-Level Cache (LLC) is L3 and which implements any type of Inclusive policy, with a known mapping function from the physical address space to that level cache L3. In this case, one can evict this physical RAM address by just evicting it from the LLC, i.e. L3. The set of addresses that is required to be accessed in order to generate the eviction of the target is called Cache Eviction Set (CES)

This approach was initially taken by Seaborn and Dullien in [9] where they implemented the Rowhammer attack for dual core processors based on Sandy Bridge micro-architecture, which is known to have inclusive LLC. For the attack, they used the mapping function that was reverse engineered in [8]. Later, Gruss, Clementine and Mangard in [10] implemented the Rowhammer attack in Javascript with the same methodology of bypassing the cache.

3.1.2 Requirement 2: Bypassing the Rowbuffer

The techniques described for bypassing the row buffer can be separated into those that rely on first calculating the DRAM mapping from virtual address space all the way to DRAM chip locations, and those that operate without precise knowledge of that mapping. By successfully mapping the memory into the DRAM, the attack can be more efficient, however it will often require special privileges or specific assumptions about the target system which could be restrictive as we will see later. In contrast, the second class of techniques may work with less privileges or target a bigger range of systems.

Mapping Memory to DRAM The physical component of/for the CPU responsible for the DRAM mapping is the memory controller. Upon receiving a memory transaction with a given physical address, the memory controller uses some logic to map this address to the DRAM. This logic is documented in AMD CPU microarchitectures but is kept secret by Intel. Since Intel possess the lion's share of the CPU market, this obstacle has limited significantly the ability of inducing the Rowhammer attack in the wide range of systems. Despite those mappings were kept secret, researchers have managed to reverse engineer them [3, 11, 12]. In Figure 3.2 we can see the mapping of the memory controller with a Sandy Bridge microarchitecture,

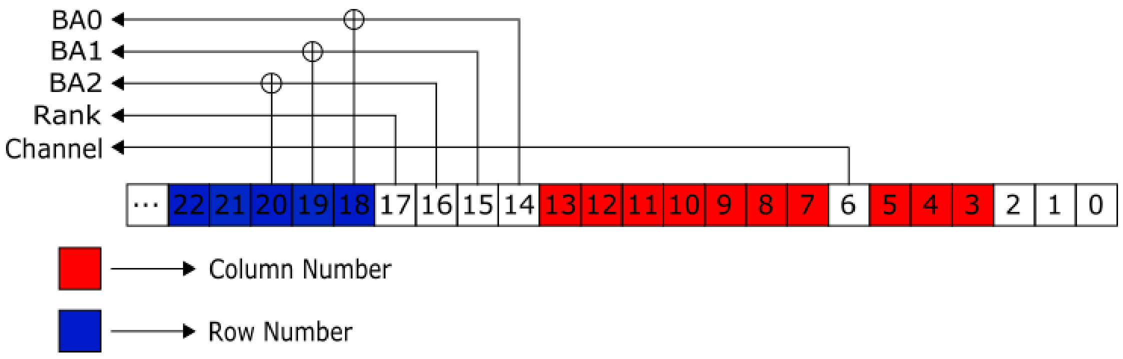

Figure 3.2: Illustration of Sandy Bridge DRAM Mapping Found in [3]

in a system with 2-channels of memory with each channel composed by a single DRAM module. As a result, in many cases the knowledge of the physical addresses allows accurate calculation of the DRAM location on the chip.

The procedure of translating virtual to physical address is standardized and it can be performed on every operating system by writing code that would manually perform the ”Page-Walk[2]”. Even though possible, this is relatively complex and so modern operating systems often offer some kind of interface for translating virtual memory to physical. Linux offers this functionality through the pagemap interface. Up to the Kernel version 4.0, it was allowed for userspace programs to use this functionality. After the release of the first exploit based on the Rowhammer bug [9] by Mark Seaborn and Thomas Dullien, the access to the translation information became available only to privileged users. Windows operating systems do not offer such functionality natively. Here it is possible to find where virtual memory of a program is mapped for users with elevated privileges by using the WinDbg debugger in kernel mode.

Initially, we should say that the ideal is to have a complete physical translation of all attacker’s memory locations. But as explained above, the attacker may not have this capability. In this case, the consideration of current OS page size can be helpful. The page size is used in the physical address translation through the fact that the virtual page offset is the same as the physical page offset (since memory is allocated in page granularity). This allows us to partially map the addresses, with

[2]Page-Walk is a procedure usually executed by the Memory Management Unit to translate a virtual address to a physical one.

the accuracy level entirely dependent on the page size.

The most commonly supported page size among CPU architectures is 4KB. When available, this is the default page size modern operating systems choose for their operations. In that case, the 4KB do provide the 12 Least Significant Bits (LSB) of the physical address which for the purpose of the Rowhammer attack will not be sufficient for efficiency. This can be observed in Figure 3.2, where for the given configuration, the 12 LSB provide only the channel information.

But even though the default page size is 4KB, both CPU and operating systems support bigger sizes as an optimization primarily to reduce the pressure in the Translation Lookaside Buffer (TLB) [3]. To give some perspective on the subject, a 1GB memory allocation with 4KB page would result in 250K entries while for example a 1GB page would result in just one entry. For example, x86-64 CPU architectures support 2MB and potentially 1GB pages. In Windows, this optimization is called large page support and special privileges are required for its use. In Linux from the other hand, a feature called Transparent Huge Pages (THP), allows the automatic promotion (2^9 4KB pages to single 2MB page) and demotion (single 2MB page to 2^9 4KB pages) of pages without requiring special permissions. In Linux build for x86-64 microarchitectures, the typical page size used with THP is 2MB. This particular page size provides essentially the 21 LSB of the physical address which is more than enough to accurately map the memory to DRAM in various configurations, as for example in the one listed in Figure 3.2.

Alternative to DRAM Mapping Here the attacker just tries to identify which addresses lead to Same Bank Different Rows (SBDR), a term defined in [3]. This is expected to be the minimum requirement for launching a Rowhammer attack.

The first method for identifying addresses that map to SBDR is based on a timing channel which was first described in [13] and it's based on the way DRAM works. A slight delay occurs when different rows within the same bank are requested. This delay, even though it's very small, it can be identified by using high

[3]TLB basically acts like a cache for the virtual to physical address mappings. It has a limited space and a TLB miss generally is a very expensive operation.

Table 3.1: Code for Inducing the Rowhammer Vulnerability

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

precision timers[4]. Even without the possession of accurate counters, it is possible to rely just on luck for hitting a particular bank. Given that the total number of banks is often really small, by picking a small set of addresses it's very probable to just pick at least one within the target bank. This technique was used in one test-case implementation by Seaborn and Dullien in [9].

3.2 Inducing the Rowhammer Bug

The basic approach for the Rowhammer attack, is to first choose a pair of addresses with the SBDR property as described in §3.1.2. Then depending on the environment, a technique described in §3.1.1 should be chosen for bypassing the cache for each memory access over the selected address pair. After that, the the aggressor rows and the expected victims are initialized with a specific pattern. Traditionally, the aggressor rows are set to 0x00 and victim rows to 0xff, but different patterns could be used as well.

Then we start the ”hammering” procedure as follows. For a certain number of iterations, the SBDR address pair is repeatedly accessed (reading only), with each access directed to DRAM using one of the cache bypass techniques. After the given number of iterations is completed, the rowhammer attack is succeeded if at least one of the victim rows had its memory contents changed. The number of iterations should be specified (ideally) as an integer multiple of the total number of memory accesses that can be performed within the DRAM refresh period. The code of this process is displayed in Table 3.1. The efficiency of the attack heavily relies

[4] CPU counters can be used for the task of high precision timing. For example, in x86 processors, this can be achieved though the Time Stamp Counter (TSC), which is a 64-bit register that counts the number of cycles since last reset.

on the way the SBDR pair is chosen and this is what we will discuss next.

3.2.1 Single Sided Rowhammer

In single-sided Rowhammer, the only requirement for the selected address pair is to be SBDR. This was the original approach used in [4]. Even though this kind of address selection has the minimum number of requirements, it is the least efficient.

3.2.2 Row Targeted Rowhammer

In the targeted approach, the selection of the SBDR addresses depends on the difference of their corresponding row numbers in DRAM. So for the success of this attack, it is necessary to have a way to obtain the DRAM mapping as was described in §3.1.2.

As was discussed earlier, the crucial parameter in this attack, is the distance between the targeted rows. When the distance is equal to two, the targeted approach is called Double-Sided Rowhammer and it was suggested initially in [9]. Through experimentations, this hammering setup was found to be the most efficient and it is considered to be the ideal method for performing the Rowhammer attack. When the double-sided Rowhammer attack was performed, it was observed that the most affected row (the most bitflips) was the one between the two aggressor rows, even though a limited number of bitflips had been observed in the neighboring rows as well. The diagram of this attack can be seen in Figure 3.3

Researchers in [10] also explored the possibility with row distance equal to 1, with adjacent rows. This attack was named Amplified Rowhammer Attack and even though it is less effective than the double-sided Rowhammer in the ”bug” identification phase, it could be more efficient in the exploitation phase. This is due to the fact that by using adjacent rows on the page boundaries, it is possible to cause bit flips on memory that is not currently allocated to the attacker. That way, on the exploitation phase, the attacker will not have to wait for the victim row to get swapped in order to get it allocated to the target. Of course, this assumes that the page spans over multiple rows and that the attacker does not have into his disposal commands to deallocate some memory selectively. This was true for example in

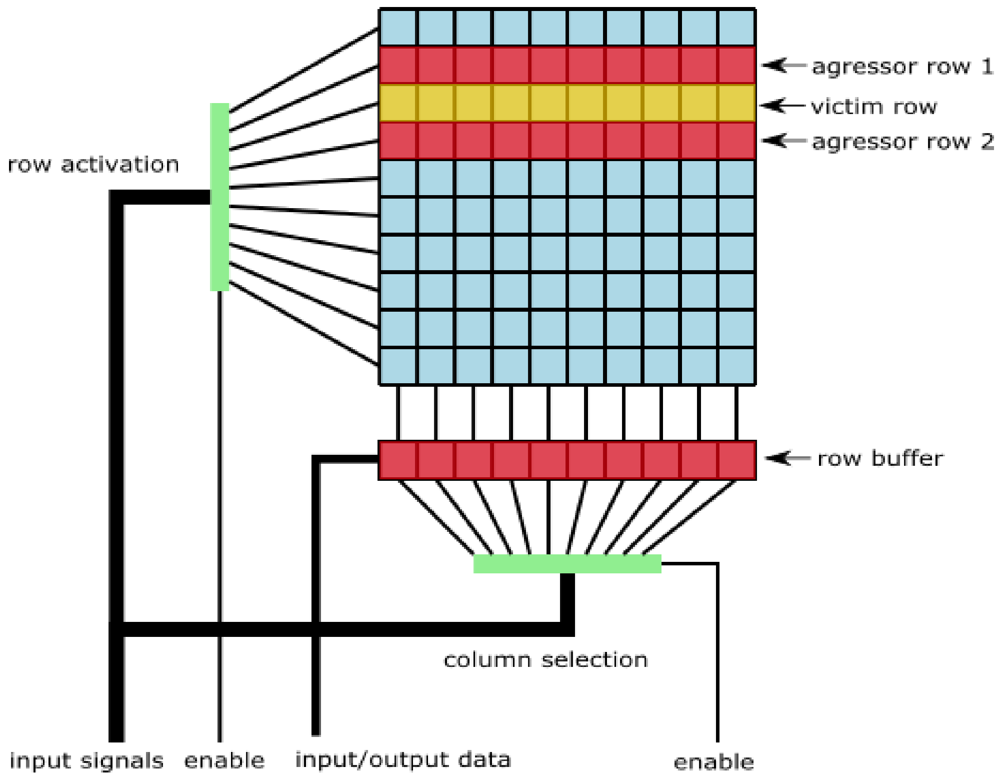

Figure 3.3: Illustration of Double Sided Rowhammer, Source: [2] (Modified)

the Javascript environment where the memory allocated was also backed by Huge Pages which was the case studied in [10]

3.3 Rowhammer Bit Flips Exploitation

The first documented technique for exploiting random bit flips was described originally by Govindavajhala et al. in [14], where they managed to break out of a Java VM by heat induced bit flips. Their exploitation strategy was to target a structure in memory which even with a single bit flip could break the underlying security assumptions of the system. Even though that particular attack was mostly theoretical, since they used a special lamp for the generation of the heat, it became practical with the Rowhammer vulnerability. So, by using similar exploitation principles with the original paper, Mark Seaborn and Thomas Dullien in [9] released the first exploit based on the flipped bits induced by Rowhammer.

One important aspect of the attack is how to make the bit flips occur in the target memory structure. There are two possibilities, either spray the whole memory

Table 3.2: Sandbox Evasion Example

Illustration of the code provided in [9] for sandbox evasion

Original Code:

Opcodes	Instructions	
`83 E0 E1`	`andl $~31, %eax`	// Truncate address to 32 bits
		// and mask to be 32-byte-aligned.
`4C 01 F8`	`addq %r15, %rax`	// Add %r15, the sandbox base address.
`FF E0`	`jmp *%rax`	// Indirect jump.

Code After Bit Flip:

Opcodes	Instructions	
`83 E2 E1`	`andl $~31, %ecx`	// Truncate address to 32 bits
		// and mask to be 32-byte-aligned.
`4C 01 F8`	`addq %r15, %rax`	// Add %r15, the sandbox base address.
`FF E0`	**`jmp *%rax`**	**// Indirect jump to unsanitized register**

with the target structure or the more targeted approach, which requires knowledge of were in memory the bit flips occur but also the capability of mapping into that memory the targeted structure as well. All the exploits presented on Rowhammer can be classified based on these two attack primitives.

3.3.1 Spraying the Memory

One of the two exploits presented in [9] was a sandbox evasion. The idea behind this exploit was to first create a specially crafted piece of code. This piece of code should have the property that even with a single bit flip would yield with high probability a new code that would bypass the security assumptions of the sandbox. The next step was to create multiple instances of that code and wait until it eventually gets mapped into a vulnerable row. In Table 3.2 we can see an illustration of this exploitation technique. As it is shown in the first table, the original code has multiple locations where a bit flip could result in successful exploitation. In the second table we can see one such example were the bit flip in the first instruction caused the truncation of *ecx* instead of *eax*. Since *eax* is directly controlled by the attacker, at the end, there is a transfer of control flow to a location directly influenced by the attacker.

The other exploit presented in [9] targeted the Page Table structure described in §2.2. The first step in this attack was to repeatedly map a shared region in memory.

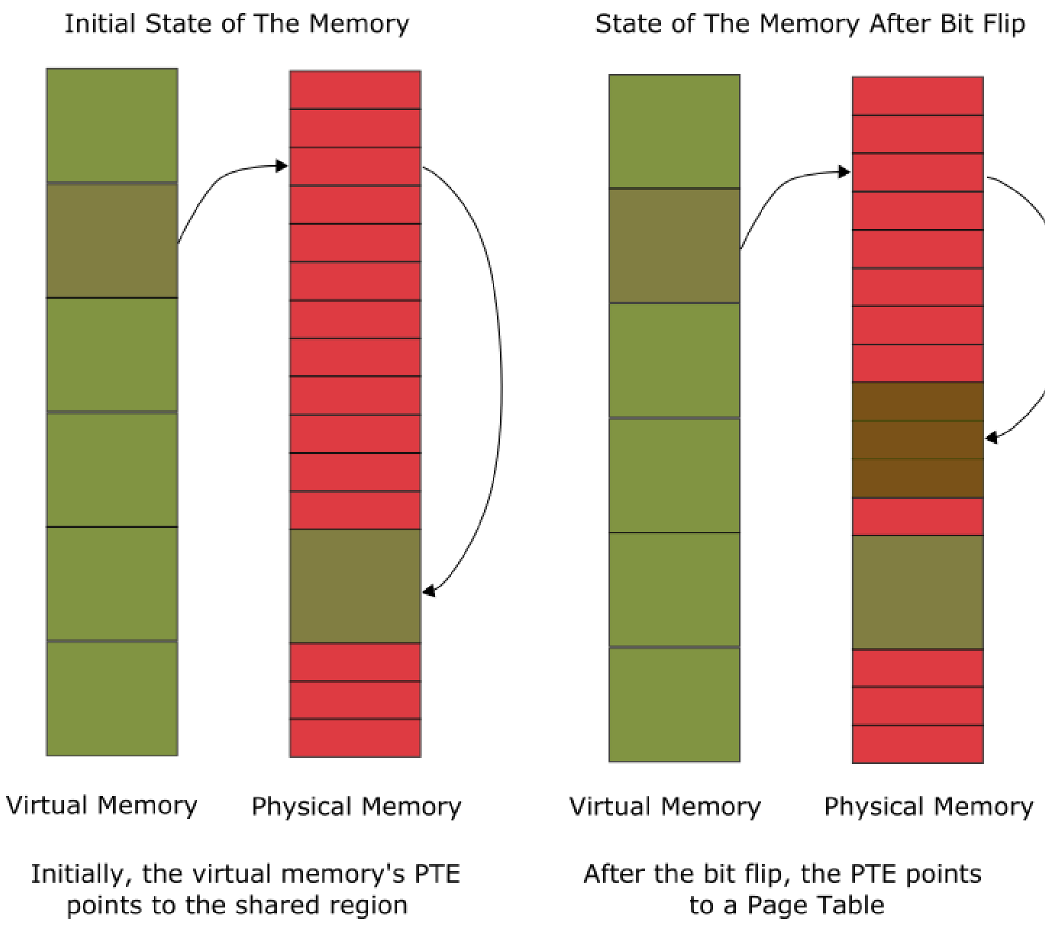
Initial State of The Memory
State of The Memory After Bit Flip
Virtual Memory
Physical Memory
Virtual Memory
Physical Memory
Initially, the virtual memory's PTE points to the shared region
After the bit flip, the PTE points to a Page Table

Figure 3.4: Page Table Exploitation

This repeated action caused the memory to reach a state where it's mainly filled with Page Tables and the single shared region where the Page Table Entries (PTEs) point onto. After reaching that state, the rowhammer attack is utilized to alter a translation entry of a PTE and gain access to an attacker's Page Table, something that is equivalent to full memory access. We can see the graphical representation of the exploitation process in Figure 3.4. As we can see in the diagram, the memory is filled with Page Table Entries (red regions) that point to the shared memory. After a bit flip in one PTE, the attacker will most likely get access to a Page Table. Now the attacker equipped with access to a Page Table, he can modify his own PTE which essentially allows him to arbitrarily read/write anything into the memory. This primitive can be then used for privilege escalation, as it was illustrated in [9].

3.3.2 Targeted Approach

Memory deduplication is a commonly abused feature in the context of Rowhammer. This is due to the fact that it could allow the placement of potentially sensitive data in rows controlled by the attacker. This has of course the prerequisite of the attacker already knowing the content of that sensitive data. Based on this technique, researchers in [15] managed to take control of a Microsoft Edge Browser running on Windows 10 without depending on any software vulnerabilities. This technique was also utilized in [16], where researchers abused the memory deduplication feature to map RSA public keys and update urls on vulnerable to Rowhammer rows, which ultimately allowed them to exploit both SSH as well as the update mechanism of debian based distributions. Even though memory deduplication can be used very efficiently for the Rowhammer attack, it should be noted that is not enabled by default in modern operating systems[5].

Another targeted approach was shown in [12]. The researchers abused the memory management conducted in Xen paravirtualization environments which ultimately allowed them to replace a whole page table with a forged one. With controlled access to a page table structure they essentially managed to gain control of the whole memory of the system, similarly to the attack presented in [9].

A more calculated approach was followed in [17]. They studied the behavior of the physical memory allocator in android phones and through specific allocation patterns they managed to force it to allocate memory from predictable regions of memory. By using this predictable allocation patterns, they managed to launch the Page Table attack and perform privilege escalation in ARM devices without relying in system-dependent features.

[5]In Windows 10, memory deduplication is disabled by default as a mitigation measure against the attack described in [15]

Chapter 4

User-Space Rowhammer Testing Implementations

For the purpose of this thesis we have implemented three Rowhammer testing tools that differentiate from the rest of the tools with regards to the fact that they do not depend on the pagemap interface. The first tool is based on the Transparent Huge Pages (THP) feature and the other two on the SBDR timing channel. At the end, we provide information on some of our experimentations that did not result in fruitful outcomes.

4.1 THP Based Implementation

4.1.1 hprh

Our first tool is called *hprh (huge page rowhammer)*[18] and its operation depends on having allocated 2MB chunks of contiguous memory (THP feature). Now, as we can see for example in Figure 3.2, this requirement essentially allows us to map addresses within that region in their exact bank within the DRAM. It does also provides as with the three least significant bits of row mapping within that specific bank. Now taking into consideration the fact that all addresses within that 2MB region map to the same Page Frame Number or PFN[1] (since they are all part of the same (huge) page), having the 3 least significant bits of the row means we have 8

[1]PFN are the remaining bits when the page offset bits are removed. For 2MB pages, `PhysicalAddress=PFN`$\cdot 2^{21}$`+PageOffset`

consecutive rows within each bank. From that point on, we have all the necessary information to launch the double-sided rowhammer attack and this is exactly what we do in our implementation. Since the double-sided attack is considered to be the best way of identifying vulnerable rows, comparing our tool with the respective implementation that required the use of pagemap yielded about the same number of identified bit flips. In our implementation, we do also take into account the fact that regular sized pages (4KB) may be promoted in our case to 2MB pages only if their address is 2MB aligned. So, in order to optimize our implementation for smaller areas of allocated memory, we request additionally 2MB to the original buffer, and we set manually the returned buffer address to the next 2MB aligned address[2]. That way, we can start generating bit flips from the beginning of the buffer. Initially for the 2MB aligned address we used the *posix_memalign*, but through our experimentations we noticed that with this method we got a slight performance hit on the number of induced bit flips and that's why we worked around it. In addition to our implementation, we provide patches for the rowhammer-test[19] and hammertime[20] that extend their functionality to support THP.

4.2 SBDR Timing Channel Based Implementations

4.2.1 tcrh

The next tool we implemented is called *tcrh[18] (timing channel rowhammer)*. This tool is based on the timing channel in DRAM described in §3.1.2. For its operation, it initially builds a set of addresses that map to the same bank. That way, the total number of memory locations that have to be scanned for vulnerable rows is immediately reduced to 1/16 (assuming uniform distribution of the data across a single DRAM module with 16 banks). It then proceeds by testing each memory location in that set with the rest in that set. One problem we had with this approach, even though the number of the locations we had to test was significantly lower than the number we started with, it was still very slow. In order to speed up the procedure, we have essentially reduced the number of DRAM refresh intervals on which

[2] $buf = (buf + 2^{21} - 1) \& {\sim}(2^{21} - 1)$.

the hammering function spans over. Through our experiments, we discovered that even though the number of bit flips per row were decreased, the vulnerable rows were still identified. Next, after identifying vulnerable rows, we hammered again the target addresses with bigger number of iterations in order to uncover the vulnerable memory cells that didn't flip because of the low number of memory access iterations.

Another optimization that we included in this testing program is after finishing the stage of identifying memory locations that map to the same bank, to try and identify memory locations that map to the same row. In this case we used the same timing channel as before, but instead of selecting memory locations that have big access delay, we gathered the ones that have small access delay, since addresses that map to the same row will be served from the row buffer. With that optimization in place, we managed to decrease the number pages we had to test by about 25%, a percentage that ultimately corresponds to the performance increase.

After implementing and experimenting with *tcrh* we came up with two observations. The first thing was that sometimes the mapping to the banks was not very precise. After spending some time tweaking our timing parameters, we discovered that the problem was inside the reverse engineered mapping function. As it turned out the rank selection included the next bit after the bank selection as well, which after we included in our calculations the accuracy of our results increased significantly. The new mapping function can be seen on Figure 4.1. The second observation came after analyzing the logs generated by the program. What we found out is

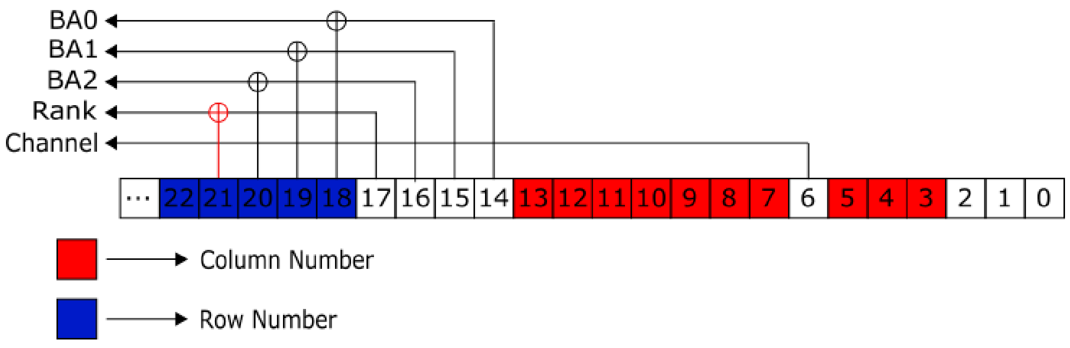

Figure 4.1: The New DRAM Mapping Function

that the difference between the addresses used for hammering was divided in some constant value groups, commonly within the range of 100KB. We speculated that this behavior was related with the fact that the Linux kernel was trying to allocate contiguous physical frames when possible. After experimenting and monitoring the kernel allocation patterns we managed to verify our theory. Based on the second observation, we further optimized the tool to ”hammer” rows that their distance is within a predefined range. This optimization provides a significant increase in the efficiency of the tool with just a small sacrifice of missing a small percentage of vulnerable rows that their distance is out of our range, something that is negligible in the overall results.

4.2.2 thrh

Based on our findings through the *tcrh*’s second observation, we created a new tool, called *thrh*. What makes *thrh* different from *tcrh* is its more targeted approach in the tested rows. The main operation of this tool is to determine whether a region in memory is contiguously mapped and only then start hammering the identified region. This was achieved by scanning the whole memory for regions where we could predict addresses inside them that mapped to the same bank but different rows, based on the assumption that the given regions were contiguously mapped. Normally, the probability of hitting such a pair is at least $\frac{1}{8}$ (number of banks), so its success signifies that at least the region between those two addresses could potentially be contiguous. Even though the development of this tool came at a late state of the whole project and so we didn’t have enough time to optimize it as we wanted, we were able to verify that our current implementation was capable of identifying contiguous regions with high probability.

4.3 Further Experimentation

With the completion of the implementations that ultimately dependented on the *clflush* instruction for the cache eviction, we tried experimenting with Cache Eviction Sets. We have tried every tool publicly available that was based on CES and we extended our own tools to include such functionality as well. Unfortunately, we didn't manage to induce any bit flips even though it was verified through the performance counters that our implementation had 95% eviction rate. We speculate that the reason behind this result could be the memory bandwidth of our processor and that's why we believe that with a different processor we could have achieved better results.

After our experimentations in Linux we transitioned to Windows platform where interestingly enough we didn't manage to induce any bit flips. Our first attempt was to port our *hprh* to Windows. The application was relatively portable, we only had to replace the Linux memory allocation function with the Windows equivalent. The final step in the transition was to allocate 2MB pages for our application. Windows does offer such page sizes through their allocation function and so we managed to take care and of that final obstacle. Before running the program we verified that we were getting a buffer which was indeed appropriately aligned for 2Mbyte pages. Everything looked like they were in place, but after running the program no bit flips were observed. The code was compiled in Visual Studio 2017 and we have tried building the program with different compiler parameters without any results. We have also tried the *tcrh*, which even though it looked like it managed to map the addresses to the correct bank, it failed to induce any bit flips.

Chapter 5

Experiments

For our experiments we used a DELL NH6K945 laptop, with Intel Core i7-2760QM CPU @ 2.40GHz and a single 8GB DRAM module with model number M471B1G73QH0-YK0. The operating system for our experiments is Ubuntu 16.04.2 LTS with kernel version 4.10.0-28. For the experiments run within VMware Virtual Machine, the guest operating system was Ubuntu as well with the same kernel version with the host machine. The tools assessed in our experiments are *rowhammer-test*, *rowhammerjs*, *hammertime* and the tools provided as part of this thesis *hprh*, *tcrh*, *thrh* as well as extensions for *rowhammer-test* and *hammertime*. Before moving into the description of the experiments, we will provide some implementation details of *rowhammer-test*, *rowhammerjs* and *hammertime* for the better interpretation of the upcoming results.

5.1 Implementation Details of Already Existing Tools

rowhammer-test Mark Seaborn and Thomas Dullien released a number of tools for the purpose of testing the rowhammer vulnerability along with their bug exploitation research in [9]. The only tool capable on inducing bit flips in our configuration was their double-sided rowhammer implementation, and from now on this is the tool that we will refer to with the *rowhammer-test*. Moving to the implementation details, this tool has two major weaknesses. The first one is that is designed to test for the rowhammer vulnerability in Sandy Bridge processors with two channels of memory, each consisted with a single DRAM module with each module composed by two ranks. The second weakness is on the implementation of the DRAM mapping function, where they take into account only the row numbers. As such, time is wasted testing for rows that belong to different channels/ranks/banks, which in the context of rowhammer is useless as we have discussed in §3. None of these two points are mentioned in the tool's description which was the reason for a lot of wasted time and frustration when we initially started experimenting with rowhammer.

The other tools released by the researchers were a single-sided version for testing the rowhammer bug as well as a version that was based on Cache Eviction Sets (CES). The single-sided version implemented the Random Pick technique described in §3.1.2, by creating a set of eight random addresses which they hammered simultaneously in each iteration. This approach turned out to be inappropriate for our configuration. We have also experimented with the other tool that was based on CES, which even after modification to support our 4-core Sandy Bridge processor, we did not manage to induce any bit flips. That's why none of the two tools are included in our experiments.

rowhammerjs Daniel Gruss, Clementine Maurice and Stefan Mangard along with the research in [10], released a new rowhammer testing tool. It is a fork of *rowhammer-test* with the addition of a custom DRAM mapping function. In their implementation, they set up the DRAM mapping function to take as input data produced by *DRAMA Reverse Engineering Tool*, a tool released by the same researchers to dynamically reverse engineer the DRAM mapping functions and it was part of [11]. At this point it should be noted that the output of *DRAMA Reverse Engineering Tool* was further processed to completely reverse engineer the mappings in the original research. The fact that this tool uses this output directly may prove to be inappropriate but nevertheless this tool is more portable than *rowhammer-test* since it could theoretically function across different CPU microarchitectures and different DRAM configurations.

rowhammerjs also provides functionality for using Cache Eviction Sets (CES) instead of CLFLUSH, but even after customization of the code to work on our

Table 5.1: The operational characteristics of the various tools on Rowhammer testing. The ✔ denotes that the underlying operation can be completed in user space while the ✔ denotes that elevated permissions are required. The + signifies that the given functionality is offered as an extension throughout our repository. The tools marked with * are implemented within the scope of this thesis.

	DRAM Mapping			Cache Eviction	
	pagemap	THP	TC	CLFLUSH	CES
rowhammer-test[19]	✔	+✔	-	✔	✔
rowhammerjs[21]	✔	-	-	✔	✔
hammertime[20]	✔	+✔	-	✔	-
hprh[18]*	-	✔	-	✔	-
tcrh[18]*	-	-	✔	✔	-
thrh[18]*	-	-	✔	✔	-

configuration (Sandy Bridge), it was not possible to induce any bit flips. That's why it's not included in our tests. It should also be noted that the cache eviction loop implemented in the program targets the Haswell microarchitectures (something that is not mentioned in the tool's description.)

hammertime This tool was created by Andrei Tatar and was released as part of the research conducted in [17]. *hammertime*, at the time of writing, is the most comprehensive tool for testing the rowhammer vulnerability. For the DRAM mapping function, they included primarily the results produced by the research conducted by the authors of the previously discussed tools, with some extensions of their own to support even more CPU microarchitectures and DRAM configurations. Unlike *rowhammerjs*, the program dynamically identifies the system information and accordingly selects the appropriate function out of a list of predefined functions. The list of predefined functions targets multiple microarchitectures so this cannot be considered as a limitation. Finally, this tool also implements the eviction of the cache before checking for potential bit flips, which theoretically should provide some boost over its performance.

Summary In Table 5.1 we provide an overview of the tools and their features that they implement for testing the Rowahammer attack.

5.2 Measurements

For the measurements, we initially group the tools into 4 categories based on the underlying technique used for mapping the memory to the DRAM. In each category we use different test configurations to either allow the tools to function optimally or to emphasize on their weaknesses. The actual parameters used for the tests are the buffer size as well as the time each tool will have at its disposal to induce the maximum number of bit flips.

For the tools based on pagemap and THP, the first test configuration is set to assess their efficiency in inducing the rowhammer vulnerability while remaining stealthy, by utilizing a small buffer (2MB) for a limited amount of time (one minute). For the second test, there is a 256MB buffer which the tools have at their disposal to "rowhammer" for ten minutes. This test aim to provide the information of which tool is best for inducing the bug in general.

The testing parameters for the *tcrh* and *thrh* were set separately since they operate on a completely different premises. For *tcrh*, the buffer size was set to 16MB and 32MB for total run time of 20 minutes. The buffer size was decided after observing that 16MB was the minimum buffer size for which there were pages allocated that mapped to three consecutive rows.

The *thrh*'s parameters were set to 512MB and 1GB for 1 minute and 10 minutes respectively. The buffer size was chosen heuristically, after running multiple experiments and studying the memory allocation patterns. Through that experiments, it was observed that the bigger the memory allocation the more like it was for the kernel to allocate big chunks in memory contiguously. The chosen sizes were a compromise between the ideal and realistic scenarios.

The results can be observed in Table 5.2

5.3 Discussion

Based on pagemap First thing that we notice in this table is the inefficiency of the tools to induce bit flips when tested in a small buffer size. This is primarily attributed to the fact that with only 2MB of buffer, the programs cannot find pages

Table 5.2: Performance Results (Total Number of Induced Bit Flips)

Based on pagemap				
	2MB_1MIN		256MB_10MIN	
	Native	*VM*	*Native*	*VM*
rowhammer-test	0	0	8	0
rowhammer-js	0	0	1322	66
hammertime	0	0	25983	1177

Based on THP				
	2MB_1MIN		256MB_10MIN	
	Native	*VM*	*Native*	*VM*
rowhammer-ext	932	0	6016	5
hammertime-ext	1911	0	25965	46
hprh	2301	0	25003	63

Based on the Timing Channel				
	16MB_20MIN		32MB_20MIN	
	Native	*VM*	*Native*	*VM*
tcrh	256	346	187	150

Based on Kernel Allocation Patterns				
	512MB_1MIN		1GB_10MIN	
	Native	*VM*	*Native*	*VM*
thrh	894	613	4649	4322

that map to three consecutive rows to perform the double-sided rowhammer test.

Nevertheless, we can see that for the 256MB buffer size, all the programs managed to induce at least couple of bit flips. As expected, *hammertime* achieved the highest number of induced bit flips since as it was discussed, it's the tool that has the most accurate DRAM mapping functions. Its optimization with the cache eviction gave it an extra boost of about 25% more bit flips. *rowhammer-js* even though we can see that it performed significantly better than *rowhammer-test*, its inaccuracy in the DRAM mapping function implementation created a significant gap between its performance and the performance of *hammertime*.

The results within the virtualized environment as expected were significantly lower. This is primarily due to the fact that within the guest virtual machine, there is another layer of indirection for the virtual address translation. The guest machine has its own view of physical memory, the guest physical memory which is then translated to the machine's physical memory. Since the pagemap interface has access only to the guest's physical memory, there is no accurate physical address translation. Nevertheless, when a chunk in both guest's physical memory and in machine's physical memory is allocated contiguously, the addresses within that chunk are are mapped accurately relatively with each other. It's the same concept on which *hprh* depends upon and was described at the beginning of this chapter. As such, the decreased performance of every tool is justifiable with *hammertime* still on the lead.

Based on THP For this category of tools, we can observe that each and every one of them was capable of inducing bit flips in the small buffer setting within the native environment. This is mainly due to the fact that with THP enabled, the tools are very likely to have a physically contiguous space allocated (assuming the memory is not under pressure) which as we have discussed in the beginning of the chapter allows them to efficiently perform the double sided rowhammer. We can also observe here that our implementation *hprh* has an advantage of about 20% over the second best performance of *hammertime* and since both have the same core operation, we attribute this difference to the simplicity of our implementation.

On the *256MB* buffer test, we can instantly observe the difference in the *rowhammer-test*'s performance, after the implementation of the DRAM mapping function. Still, the *rowhammer-test* was in a significant disadvantage when compared with the other two tools, but here we display how much of a difference a properly implemented mapping function can have on the results. Moving on to *hammertime-ext* and *hprh* we can observe that both have identical performance with *hammertime* having a slight edge. At this point we should note that through a slight modifications of the *hprh*, it was possible to induce more bit flips for this particular scenario. For example, by adding a delay of one second every 30 seconds, it was possible to reach higher numbers than *hammertime* consistently. Nevertheless, we chose to leave it as it is due to the fact that all those optimizations were considered minor with some of them potentially applicable to the rest of the tools as well.

Based on Timing Channel On this table we observe that *tcrh* managed to induce a significant number of bit flips when considering the fact that it didn't depend on neither pagemap nor THP. The fact that it depends on the timing channel allows it to operate properly in both native and virtualized environments, something that can be observed from the results. Nevertheless, due to its nature of blindly testing all the available rows for the vulnerability causes a significant drop to the number of induced bit flips when compared with the tools that perform a more targeted approach.

Another interesting observation on the results is the fact that *tcrh* performed better on the smaller buffer size, something that comes in contrast with the behavior of the previously tested tools. Before we run our experiments, we knew that the smaller buffer size would result in more rows getting tested. What we didn't know is if that smaller buffer would include memory regions mapping in rows that are capable of inducing the rowhammer vulnerability. As the results has shown, in the limited time frame, the 16MB of buffer is more suitable for inducing the maximum number of bit flips and for this result

Based on Kernel Allocation Patterns In the final table, we can observe that *thrh* performed with the same efficiency in both native and virtualized environment. Regarding its performance in the two configurations, through the results we can see that it is linearly dependent on time. This of course depends on its ability to find a contiguous chunk big enough to allow the test to run for the given time frame. For example, a 2MB buffer can be exhaustively tested in one minute, so if there is a need for statistics for a two minute time frame, then a buffer of at least 4MB has to be identified. Finally, we can observe that *thrh* can performed similarly in both native and virtualized environments, with the total number of bit flips being slightly higher in the native environment. We speculate that this slight difference is linked with the fact that the identified region in memory has to be contiguous in both guest's physical memory as well as the machine's physical memory. Of course the buffer don't have to be aligned and with even a small overlap between the two we can efficiently perform our attack. For example, in the test configuration, the overlapping region would have to be at least 2MB since we are only testing it for a minute. So, with the kernel allocating contiguous chunks of even 256MB, we can see that this 2MB is very likely to overlap. All in all, *thrh* can be viewed as the combination of *tcrh* and *hprh*, it has the requirements of the former while having the potential of achieving the performance of the latter.

Chapter 6

Conclusion and Future Work

The Rowhammer attack revolves around a lot of complicated concepts. In this report we aim to shed some light into those concepts, study the internals of the Rowhammer Attack and clarify the requirements on which a successful attack really depends on. We then provide three implementations that are capable of inducing the Rowhammer attack in userspace as well as their performance evaluation. Through the results, it becomes obvious that the pagemap interface can be efficiently substituted by using the described techniques. Since those techniques can be utilized in userspace, we open the way for further research into the actual exploitation of the vulnerability.

Bibliography

[1] David Tawei Wang. *Modern dram memory systems: performance analysis and scheduling algorithm*. PhD thesis, University of Maryland, College Park, 2005.

[2] Wikipedia - Row hammer. `https://en.wikipedia.org/wiki/Row_hammer`. [Accessed 22-July-2017].

[3] Thomas Dullien and Mark Seaborn. How physical addresses map to rows and banks in DRAM. `http://lackingrhoticity.blogspot.co.uk/2015/05/how-physical-addresses-map-to-rows-and-banks.html`, 2015. [Accessed 28-July-2017].

[4] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 361–372. IEEE Press, 2014.

[5] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[6] Thomas Dullien and Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. `https://googleprojectzero.blogspot.co.uk/2015/03/`

`exploiting-dram-rowhammer-bug-to-gain.html`, 2015. [Accessed 28-July-2017].

[7] Rui Qiao and Mark Seaborn. A new approach for rowhammer attacks. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, pages 161–166. IEEE, 2016.

[8] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[9] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 2015.

[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[11] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[12] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and MR Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[13] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 18. USENIX Association, 2007.

[14] Sudhakar Govindavajhala and Andrew W Appel. Using memory errors to attack a virtual machine. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 154–165. IEEE, 2003.

[15] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 987–1004. IEEE, 2016.

[16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, pages 1–18, 2016.

[17] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[18] Varnavas Papaioannou. User-Space Rowhammer Testing Tools. `https://github.com/vp777/Rowhammer`, 2017. [Accessed 28-July-2017].

[19] Thomas Dullien and Mark Seaborn. Program for testing for the DRAM ”rowhammer” problem. `https://github.com/google/rowhammer-test`, 2015.

[20] Tatar Andrei. Hammertime: a software suite for testing, profiling and simulating the rowhammer DRAM defect. `https://github.com/vusec/hammertime/`, 2016. [Accessed 28-July-2017].

[21] Daniel Gruss and Clémentine Maurice. Program for testing for the DRAM ”rowhammer” problem using eviction. `https://github.com/IAIK/rowhammerjs`, 2015. [Accessed 28-July-2017].