On Feasibility and Performance of Rowhammmer Attack

Varnavas Papaioannou
University College London
London, UK
varnavas.papaioannou.16@ucl.ac.uk

Nicolas Courtois
University College London
London, UK
n.courtois@cs.ucl.ac.uk

ABSTRACT

In this paper we study the Rowhammer sidechannel 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. This is a very strong requirement which, in a sense, puts ths attack into the theoretical spectrum. The purpose of this report is to explore different techniques that would allow the execution of the Rowhammer attack in userspace. More specifically, we provide two implementations, each of them having different strength of requirements but with one characteristic in common: the capability of executing the Rowhammer attack without elevated privileges. 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.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Operating systems security**;

KEYWORDS

side channel attack, rowhammer, DRAM, DFA, perturbation attacks

ACM Reference Format:
Varnavas Papaioannou and Nicolas Courtois. 2017. On Feasibility and Performance of Rowhammmer Attack. In *Proceedings of ASHES'17*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3139324.3139330

1 BACKGROUND ON COMPUTER MEMORY

1.1 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 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 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 on the same board 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 time 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 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 1

2 THE ROWHAMMMER SIDE CHANNEL

After defining the basics of DRAM architecture, 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. [11] in 2014. It is described as the process of repeatedly activating specific rows within a bank which may have 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

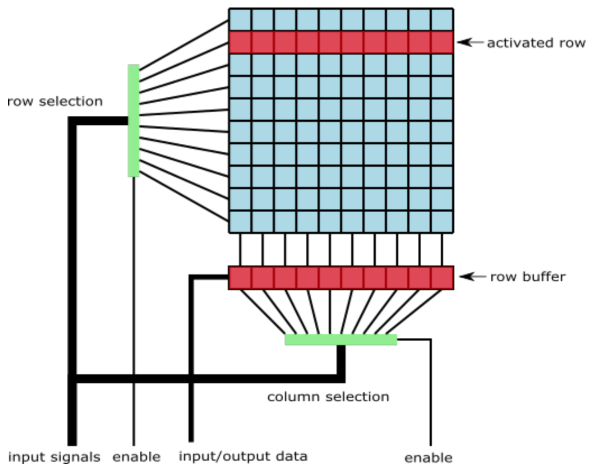

Figure 1: DRAM Structure, Source: [1] (Modified)

coupling effects that accelerate charge leaking in adjacent rows which eventually provokes bit flips in memory locations that potentially belong to another process. The disturbance caused by those effects is amplified by the increased density of modern memory modules 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 at least be capable of avoiding the target's bank row buffer. Accessing memory locations that are already in the row buffer would 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 hides a lot of complexities. In the next chapter we will discuss various ways of implementing each of the requirements that could be useful in different scenarios.

2.1 Attack Foundations

2.1.1 Requirement 1: Bypassing the CPU Cache. Full description of cache memory is not within the scope of this paper. We will only provide the elements essential for the Rowhammmer 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. An example of the various levels of a modern memory hierarchy can be seen in Figure 2.

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 techniques which aim at bypassing the cache.

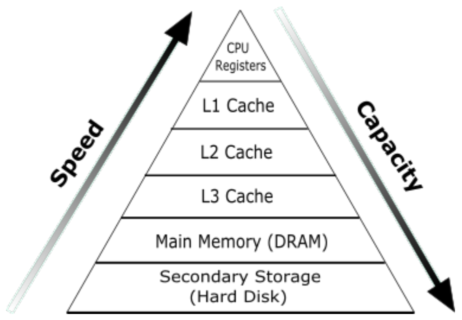

Figure 2: Example of the multiple levels of memory hierarchy

Architecture Specific Instructions. Since the CPU cache plays a critical role in optimizing the system's performance, modern CPU architectures are designed to allow fine-grained functionality over how the cache should be used. One commonly found feature over the cache 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 in 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 accesses. Primarily focused towards avoiding cache pollution with data which are known that are not going 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 [5] and successfully used in [15].

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 with size at least as big as the last level cache. That way, it is possible to evict a target cache line which was e.g. demonstrated in [10]. Even though this technique is effective irrespectively to the implemented cache coherence policies, this technique 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 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 L3 cache. In this case one can evict the contents of a physical address by just evicting it from the LLC, i.e. L3. The set of addresses that are 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 [17] 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 [10]. Later, Gruss, Clementine and Mangard in [9] implemented the Rowhammer attack using javascript with the same methodology of bypassing the cache.

2.1.2 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 the virtual address space all the way to the DRAM chip locations, and those

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

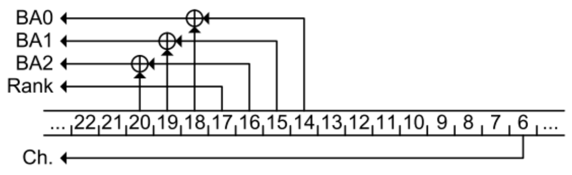

Figure 3: Illustration of Sandy Bridge DRAM Mapping Found in [6]

that operate without precise knowledge of that mapping. By successfully mapping the memory into the DRAM the attack can be more efficient however will often require special privileges or dependency on specific assumptions about the target system which we will study later. In contrast, the second class of techniques has the advantage that may work with less privileges or for a wider range of target systems.

Mapping Memory to DRAM. The physical component of 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. Regardless of that fact, various researchers have managed to reverse engineer a significant amount of DRAM mapping functions used by Intel in [6, 14, 19]. In Figure 3 we can see the mapping of the memory controller on Sandy Bridge microarchitecture, on 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 [17] 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. On Windows, 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, the ideal is to have a complete physical translation of all attacker's memory locations. As explained above, the attacker may be unable to obtain these. In that case, the consideration of current OS page size can be useful. 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. This allows us to partially map the addresses, with the accuracy level which depends 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. Now considering the 4KB page for the purpose of the Rowhammer attack, the provided 12 least significant bits of the physical address will not provide us with enough information to successfully launch the attack. This can be observed in Figure 3, where the first 12 bits 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 instance, 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 and demotion of page sizes without requiring special permissions. In Linux build for x86-64 architectures, the typical page size used with THP is 2MB. This particular page size is more than enough to accurately map the memory to DRAM in various configurations, as for example in the one listed in Figure 3.

Alternative to DRAM Mapping. Here the attacker just tries to identify which addresses lead to Same Bank Different Rows (SBDR), a term defined in [6]. 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 [12] 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 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 have a pair that satisfies the given requirements. This technique was used in one test-case implementation by Seaborn and Dullien in [17].

3 INDUCING THE ROWHAMMMER BUG

The basic approach for the Rowhammer attack, is to first choose a pair of addresses with the SBDR property as described in §2.1.2. Then depending on the environment, a technique described in §2.1.1 should be chosen for bypassing the cache for each memory access over the selected address pair. After that, the aggressor rows and the expected victims are initialized with specific patterns. Traditionally, the contents of aggressor rows are filled with 0 and the victim rows with 255.

3.1 Hammmering Procedure

With everything in place, we start "hammering" by repeatedly accessing each SBDR pair. The process is depicted in Figure 1. The efficiency of the attack heavily relies on the way the SBDR pair is chosen and this is what we will discuss next.

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

[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.

[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.

Table 1: Code Snippet for Inducing the Rowhammer Vulnerability

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

Single Sided Rowhammmer. In single-sided Rowhammer, the only requirement for the selected address pair is to be SBDR. This was the approach originally used in [11]. Even though this kind of address selection has the minimum number of requirements, it is the least efficient.

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 §2.1.2.

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 [17]. Through experimentations, this hammering setup was found to be the most efficient and it is considered to be the best 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 4

Researchers in [9] also explored the possibility of setting the row distance equal to 1, or simply put, setting the aggressor rows to adjacent rows. This attack was named Amplified Rowhammmer Attack and even though it is less efficient than the double-sided Rowhammer in the "bug" identification phase, it could be more effective 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 in memory that is not currently allocated to the attacker. That way,

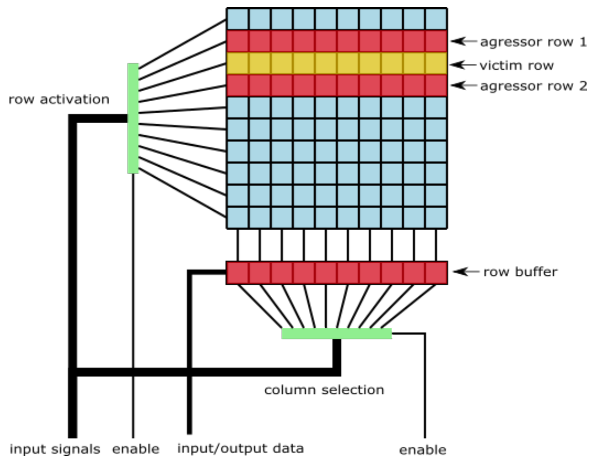

Figure 4: Double Sided Rowhammmer, Source: [1] (Modified)

in the exploitation phase, the attacker will not have to wait for the victim row to get swapped in order to get it allocated to another process. 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 is true for example in the Javascript environment which was the case studied in [9].

3.1.1 Rowhammmer Bit Flips Exploitation. The first documented technique for exploiting bitflips was described originally by Govindavajhala et al. in [7], 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, it became practical with the Rowhammer vulnerability. So, by using similar exploitation principles as the original paper, Mark Seaborn and Thomas Dullien in [17] 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 a location where a vulnerable memory structure is mapped. There are two possibilities, either spray the whole memory with that memory structure or the more targeted approach, in which the attacker has the capability of mapping memory into specific locations in DRAM. All the exploits presented on Rowhammer can be classified based on these two attack primitives.

Spraying the Memory. One of the two exploits presented in [17] 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 2 we can see an illustration of this exploitation technique.

The other exploit presented in [17] aimed the Page Table structure. The first step in this attack was to repeatedly map a shared region in memory. This repeated action caused the memory to reach a state where it was 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 was 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 read/write access.

Table 2: Sandbox Evasion Example

Illustration of the code provided in [17] 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**

Targeted Approach. Memory deduplication is a commonly abused feature in the context of Rowhammmer. This is due to the fact that it could allow the placement of potentially sensitive data in rows controlled by the attacker. Based on this technique, researchers in [3] 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 [19]. 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 [17].

A more calculated approach was followed in [18]. 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 to system-dependent features.

3.2 Our Contributions

For the purpose of this paper we have implemented two 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 one on the SBDR timing channel.

The first tool is called *hprh (**h**uge **p**age **r**ow**h**ammer)*[13] and its operation depends on having allocated big chunks of contiguous physical memory (THP feature). Now, as we can see for example in Figure 3, this requirement essentially allows us to map addresses within that region in their exact bank. It does also provides us with the three least significant bits of the row mapping within that specific bank, something that essentially allow us to know the relative position of eight consecutive rows. 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 interface 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 when a page-fault occurs in a 2MB-aligned address. This is achieved by either explicitly requesting a 2MB aligned buffer or by manually aligning the buffer to the boundary. That way, we can generate bit flips even within smaller buffer sizes. In addition to our implementation, we provide patches for the *rowhammer-test*[4] and *hammertime*[2] that extend their functionality to utilize the THP feature.

The next tool we implemented is called *tcrh[13] (**t**iming **c**hannel **r**ow**h**ammer)*. This tool is based on the timing channel in DRAM described in §2.1.2. 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). Next, we further reduced the number of entries in that set by identifying memory locations that mapped to the same rows. This was achieved by utilizing the same timing channel that was used originally, with the difference that for this case we pick address pairs that have small access delay, since that signifies that the requests are served by the row buffer and as such the address pairs must belong to the same row. By using that procedure, we managed to reduce the number of entries in the set by about 25%.

The original approach after getting done with the "sieving" phase was to test the previously found set of memory location with each other. This decision was based on the assumption that physical frames were randomly allocated to the user. As such, we expected that the more memory locations we got tested the higher would the probability be to hit memory locations that mapped to nearby rows and bruteforce in a sense the double-sided technique. As expected, it required a significant amount of time for its operation even for small buffers but even so, it was capable of generating bit flips. After analysis of the logs generated by the program, it was observed that a lot of target and victim rows were having the same distance between them. This behavior could have only be explained in the case where large chunks of contiguous memory were allocated to our program. Since the THP feature was disabled for our tests, this observation is speculated to be related to the way Linux allocates memory (buddy allocator). With that in mind, we modified the program to be efficient only when allocated chunks of contiguous memory. We achieved that by testing only memory locations that were positioned within a predefined distance from the currently tested memory locations. That way, we lower the probability of finding vulnerable rows in the case that the currently tested region is not mapped contiguously. Nevertheless, in the other case that the region is contiguously mapped, then we always manage to identify vulnerable rows something that turned out to be much more efficient than our original approach. At the end, using our final implementation, we managed to induce a significant number of bit flips not only in native but in virtualized environments as well. Before closing this section we should also mention that for our 4-core Sandy Bridge configuration, we have discovered that the bit 21 shown in Figure 1 was also part of the rank selection function.

3.3 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. Our operating system was Ubuntu 16.04.2 LTS with Kernel version 4.10.0-28. For the experiments run within Virtual Machine we used VMware Workstation and for the guest operating system we used Ubuntu with the same Kernel version as the host machine. In Figure 3 we provide an

[5]In Windows 10, memory deduplication is disabled by default as a mitigation measure to the attack described in [3]

Table 3: 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 paper.

	DRAM Mapping			Cache Eviction	
	pagemap	THP	TC	CLFLUSH	CES
rowhammer-test[4]	✔	+✔	-	✔	✔
rowhammerjs[8]	✔	-	-	✔	✔
hammertime[2]	✔	+✔	-	✔	-
hprh[13]*	-	✔	-	✔	-
tcrh[13]*	-	-	✔	✔	-

overview of the tools and their features which they implement for testing the Rowahammer attack.

3.4 Measurements

For the measurements, we assess all the tools under two different settings. In the first one, we assess their efficiency of inducing the rowhammer vulnerability while remaining stealthy, by utilizing a small buffer (2MB) for a limited amount of time (one minute). For the second one, there is a 256MB buffer which the tools have at their disposal to "rowhammer" for ten minutes. This test aims to provide the information of which tool is best for inducing the bug in general.

4 CONCLUSION

The Rowhammer attack revolves around a lot of concepts that get complicated due to the lack of proper documentation. 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 relies upon. We then provide two 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.

Table 4: 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				
	2MB_1MIN		256MB_10MIN	
	Native	*VM*	*Native*	*VM*
tcrh	62	0	832	169

REFERENCES

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

[2] Tatar Andrei. Hammertime: a software suite for testing, profiling and simulating the rowhammer DRAM defect. https://github.com/vusec/hammertime/, 2016.

[3] 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.

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

[5] 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].

[6] 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.

[7] 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.

[8] 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].

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[18] 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.

[19] 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.