ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/299466481

A New Approach for Rowhammer Attacks

Conference Paper · May 2016

DOI: 10.1109/HST.2016.7495576

CITATION
1

READS
150

2 authors, including:

Rui Qiao
Stony Brook University
8 PUBLICATIONS **7** CITATIONS

SEE PROFILE

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.

Available from: Rui Qiao
Retrieved on: 25 October 2016

A New Approach for Rowhammer Attacks

Rui Qiao
Stony Brook University
ruqiao@cs.stonybrook.edu

Mark Seaborn
Google
mseaborn@google.com

***Abstract*—Rowhammer is a hardware bug identified in recent commodity DRAMs: repeated row activations can cause bit flips in adjacent rows. Rowhammer has been recognized as both a reliability and security issue. And it is a classic example that layered abstractions and trust (in this case, virtual memory) can be broken from hardware level. Previous rowhammer attacks either rely on rarely used special instructions or complicated memory access patterns. In this paper, we propose a new approach for rowhammer that is based on x86 non-temporal instructions. This approach bypasses existing rowhammer defense and is much less constrained for a more challenging task: *remote* rowhammer attacks, i.e., triggering rowhammer with existing, benign code. Moreover, we extend our approach and identify libc memset and memcpy functions as a new rowhammer primitive. Our discussions on rowhammer protection suggest that it is critical to understand this new threat to be able to defend in depth.**

I. Introduction

Rowhammer is a kind of DRAM disturbance error: repeated DRAM row activations can cause bit flips in adjacent rows [15]. As a DRAM hardware bug, rowhammer is closely related to DRAM organization. DRAM consists of a two dimensional array of cells: the rows and columns are used for addressing a cell. For each DRAM cell, the charged or uncharged state of its capacitor is used to denote a single bit of stored value. In order to access a cell, its row needs to be *activated* in order to be copied to a *row buffer*. However, repeatedly activating a row can cause cells at adjacent rows discharge at an accelerated rate. If a cell state changes from charged to uncharged before it is *refreshed*, the bit has flipped. This phenomenon is widely recognized as the "rowhammer" problem.

The rowhammer problem exists in recent commodity DRAM chips and mostly results from the design/process limitations for sub 40 nm memory technology such as DDR3 [15]: as the chip density increases, the cells become smaller and the capacitors can hold less charge. The cells are also closer. Therefore the voltage fluctuations when activating a row are more likely to affect adjacent rows and finally result in bit flips. It has been shown that 110 out of 129 recent DRAM modules from three major DRAM manufactures are susceptible to the rowhammer problem [15].

Rowhammer was first considered as a reliability issue which may result in data corruption, but later it has been shown to also be a security issue [17]. Two exploits were developed: the first one gains kernel privilege by causing bit flips in page table entries (PTE) of an unprivileged, malicious process. Specifically, the changed PTE may point one of the malicious process's writable page to a physical frame containing page table, and write access to page tables enables attacker to control the whole physical memory. The second exploit is a sandbox escape by causing bit flips in the instructions enforcing control-flow integrity [17].

The key challenge of triggering rowhammer is to reliably and frequently cause DRAM row activations. However, CPU cache can prevent DRAM accesses (and therefore row activations). Previous rowhammer approaches either rely on the x86 *CLFLUSH* instruction [15], or use carefully selected memory access patterns to evict cache lines [9], [10]. Therefore, rowhammer attacks are usually assumed by software level defenses to be based on these approaches.

In this paper, we propose new approaches for triggering rowhammer. We further discuss their security implications under two different threat models. In the first model, the program code is from untrusted sources, and may be sandboxed. The second model only allows benign code to be executed, but the program may be exposed to untrusted data inputs. We show that rowhammer attacks can be performed with our new approaches. Specifically, we make the following contributions:

- *A new rowhammer method without* CLFLUSH. Other than existing methods, we demonstrate that rowhammer can also be triggered with non-temporal store instructions. This method can be less constrained and more suitable for some scenarios.
- *A new rowhammer primitive.* We identify a new rowhammer primitive that extends our non-temporal store based method. Specifically, libc functions memset and memcpy are found capable of rowhammer. This can be convenient because of their wide usage.
- *Exploit with untrusted code.* We develop an exploit that can escape from a sandbox, bypassing existing application layer rowhammer defense. We highlight the differences of the exploit as compared to the *CLFLUSH* based approach, and illustrate how we overcome these challenges.
- *Remote rowhammer attacks.* We describe the possible *remote* rowhammer attacks, i.e., triggering rowhammer with malicious inputs but benign code, based on the new rowhammer methods. Compared with user level *software* based memory corruptions which only affect the current process, one distinguishing feature of rowhammer is that bit flips can be caused in other processes or kernel.

II. Background

A. DRAM

From a low level, the basic unit of storing information in *dynamic random-access memory (DRAM)* is a *cell*. It consists

code1a:	code1b:
mov (X), %eax	mov (X), %eax
mov (Y), %eax	clflush (X)
clflush (X)	
clflush (Y)	
mfence	mfence
jmp code1a	jmp code1b

Fig. 1. Rowhammer with *CLFLUSH*

of a *capacitor* and a *transistor*. The capacitor can either be charged or uncharged, representing a binary data value. And the transistor is used for accessing the cell.

From a high level, a DRAM module can have one or two *ranks*, where one rank corresponds to one side of the *dual inline memory modules (DIMMs)*. Each rank is divided into banks, which can be accessed concurrently to increase parallelism. In each bank, cells are organized into *rows* and *columns*. To access a particular cell, the corresponding row is first *activated*, and then data is read into the *row buffer* of the bank, and the accesses are served from row buffer.

DRAM cells lose charge gradually. Therefore, they need to be recharged periodically to keep the stored value. This process is called *refresh*. The time period for a cell to lose data because of discharging is called *retention time*, which is specified by DDR3 DRAM specification [12] to be at least 64 ms. A refresh must happen on every row within this time.

In most systems, memory controller uses physical rather than virtual addresses for selecting the underlying rank, bank, row, and column of the DRAM. Although not documented, this mapping of some CPUs can be reverse engineered [16].

B. Rowhammer: The DRAM Bug

As discussed, rowhammer can be triggered if a DRAM row is repeatedly activated. Figure 1 shows the code snippets used for rowhammer from the original paper [15]. X and Y are addresses that belong to the same bank, but different rows. Note that code1a can induce bit flips, while code1b cannot. This is because in code1b, if a single row (which X belongs to) is repeatedly accessed, the data will be served from the row buffer since there is no need to reactivate the row. On the other hand, code1a alternates the accesses to two different rows (but on the same bank). Therefore, the next memory read is not from the same row, and the content of the row buffer cannot be used. This essentially forces the two rows to be activated in turn, and finally there can be bit flips in nearby rows of either of the activated rows. From this we can see that the key for rowhammering is that a row being repeatedly *activated*, rather than *accessed*.

The frequency and number of row activations are important factors for successful rowhammer. We refer to the time period between two activations of the same DRAM row as *activation interval*. When the retention time is the standard 64 ms, the activation interval has to be at most 500 ns to induce bit flips [15], and there has to be at least 139K row activations.

III. Non-temporal Stores: A New Method for Rowhammer

In this section, we introduce the involved instructions, challenges, and implementation details for our new rowhammer method.

A. Non-temporal Instructions

In computation, data references can have different patterns. Some are temporal: data will be accessed again soon in future. Some are spatial: data in adjacent locations will be accessed. And some are non-temporal: data is referenced just once and not again in the near future. As most data accesses exhibit either temporal or spatial locality, caches are introduced to effectively improve performance. However, non-temporal accesses could pollute cache and harm performance.

To address this problem, non-temporal instructions were introduced by CPU vendors for supporting cacheability control. Once non-temporal data references are expected, programmers or compilers can use these non-temporal instructions to minimize cache pollution. On the other hand, because their cache bypass characteristics are desirable, we explore the possibility of using them for rowhammer.

Non-temporal instructions can be loads, stores and prefetches. The only x86 non-temporal load instruction *MOVNTDQA* is not useful for rowhammer because it requires the underlying memory to be *write combining (WC)* type to bypass cache. The only x86 non-temporal prefetch instruction *PREFETCHNTA* is not helpful either, because it may (pre)fetch data into L2 cache from L3 cache, therefore not reliably generating DRAM accesses and hence row activations.

We can use non-temporal stores for rowhammer because they *treat* target memory as uncached "write combining (WC)" type [3][1]. There are a handful of x86 non-temporal store instructions, some examples are:

- *MOVNTI*: this instruction moves a 32-bit or 64-bit integer from source register operand to destination memory operand, with a non-temporal hint.
- *MOVNTDQ*: similarly, this instruction stores a 128-bit value from a SSE register to memory using a non-temporal hint.

B. Write-Combining Buffers

Although non-temporal stores serve as a basis for our rowhammer method, a straight-forward implementation would not work due to write-combining buffers.

As discussed, the memory used by non-temporal stores are treated as "write combining (WC)" type [3]. Therefore, writes may be delayed and combined in the write-combining buffer (WC buffer) to reduce DRAM accesses (Figure 2). Specifically, non-temporal writes to the same address are usually combined at WC buffer, and only the last write goes through to the DRAM chip. As a result, the row activation frequency has been greatly reduced and the rate is not sufficient for rowhammer. This is what happened when a straight-forward implementation, such as code2a of Figure 3 is used.

In order to make sure each non-temporal store goes through to the DRAM chip, we need a way to flush the WC buffer.

[1]This means even if the underlying memory is cached, it is treated as WC type and therefore uncached. This is different from *MOVNTDQA* where the underlying memory has to be WC type.

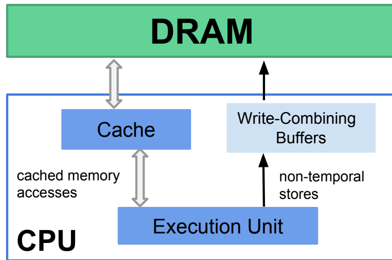

Fig. 2. Cached and non-temporal memory accesses

code2a:	code2b:
movnti %eax, (X)	movnti %eax, (X)
movnti %eax, (Y)	movnti %eax, (Y)
	mov %eax, (X)
	mov %eax, (Y)
jmp code2a	jmp code2b

Fig. 3. Rowhammer with non-temporal stores

We've found out this can be achieved by a following cached memory access (either read or write) to the same address where the non-temporal store instruction writes to. Code2b of Figure 3 implements this and is effective for rowhammer.

C. Triggering Bit Flips

Another requirement for a successful rowhammer is to pick addresses for memory accesses (hammering). As discussed in Section II-B, the two (physical) addresses have to be mapped to different rows but the same bank. To achieve this, we reused the probabilistic approach adopted in rowhammer-test [6]: (virtual) addresses are picked at random. Since for typical DRAMs there are 8 banks, the probability of satisfying the "same-bank-different-row" property is roughly 1/8. [2]

To check for bit flips, a large portion of memory is mapped and initialized. After the rowhammer attempts, they are checked against the initial value. Using our new method, we have repeatedly produced bit flips on a set of vulnerable machines.

Note that because of their special property of bypassing cache, non-temporal instructions have been conjectured as a viable means for rowhammer [17]. However, none of the previous works have shown their effectiveness or the key details for triggering rowhammer [17], [7].

IV. Exploit with Untrusted Code

As discussed, non-temporal stores can be used for rowhammering. In this section, we further investigate whether non-temporal stores can induce bit flips that are useful for exploits.

We performed our case study on NaCl sandbox escape exploit. The Google Native Client (NaCl) is a sandbox used in Chrome browser to securely and efficiently run untrusted native code in the context of a web application [18]. Escaping from NaCl sandbox could lead to arbitrary code execution and is therefore a serious exploit. We chose Chrome NaCl sandbox because it validates untrusted software (before their execution) with strict rules therefore representing a difficult case, and the

[2]Because each bank has many (e.g., 32K) rows, the probability that the two addresses are in the same row of same bank is negligible.

```
andl $0xffffffe0, %eax // make eax 32-byte aligned
addq %r15, %rax        // add base register r15
jmp *%rax    // after a single bit flip => jmp *%rcx
```

Fig. 4. Sandbox escape example: originally the first two instructions constrain the value of rax, so that the indirect jump at the third instruction is confined. However, rowhammer induced bit flips may change the third instruction to a different one: an indirect jump using an unrestricted register rcx, therefore bypassing control flow restrictions. Note that the hardware layer bit flip has changed *validated, read-only* code.

convenience to compare with the original exploit which is also based on NaCl.

A. NaCl Sandbox Escape

Currently, *CLFLUSH* instruction used by most rowhammering has been forbidden by NaCl validator. However, non-temporal store instructions are still allowed and can be used in an exploit.[3] The major difference between *CLFLUSH* and non-temporal store based schemes is that the former performs repeated memory reads, while the latter needs repeated writes. If we call the DRAM row where bit flip happens a *victim* row, and the row where we perform memory accesses (to trigger bit flips) an *aggressor* row, then in the case of non-temporal store based rowhammer, we need to be able to write to the aggressor row. In other words, the corresponding page should have write permission, i.e., it should be a data page.

In the original NaCl sandbox escape exploit [17], repeated reads of an aggressor row that contains code cause bit flips in an adjacent victim row, which also contains code. If the flipped bit happens to be one of the sandbox instructions that constrain control flow, the instruction has therefore changed to a different one, and the sandbox can be escaped. Figure 4 shows such an example. Although the bit flips are random and may not result in the same instruction change as in Figure 4, it has been shown that 13% of the possible changes are exploitable [17]. And because the untrusted process is "sprayed" with all such code, the probability that the bit flips happen to these code is high.

In our new exploit, we want to use the same exploit technique by triggering bit flips in sandboxing *code*. The key is that the aggressor row should now contain *data* instead of code (because obviously code is read-only and cannot be written to by non-temporal stores), and it should be *physically adjacent* to victim rows which contain *code*. This is especially challenging when we want to reuse the probabilistic approach: aggressor address pairs are picked at random. To ensure a high probability of bit flips to happen in code, this requires the aggressor data rows and victim code rows to be sufficiently interleaved. Based on former understanding of DRAM physical address mapping [16], what we need is to influence the OS so that large, physically contiguous memory are assigned to interleaved code and data pages.

To get the desired mapping, a series of steps are taken:

1) *Create high memory pressure and reserve most remaining memory.* We create high system memory pressure by

[3]We have reported our new rowhammer method and exploit to Google. As a response, non-temporal instructions are also forbidden (or rewritten) in latest NaCl implementation.

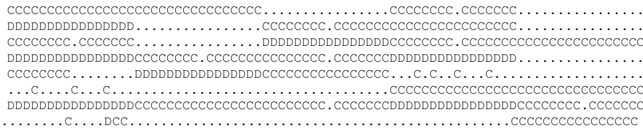

Fig. 5. Contiguous physical page frames and their assignment to code and data pages

launching another process which maps a large chunk of memory (and forces physical frame allocation). When the "memory eater" program is running, start our NaCl process, which first reserves most of the remaining physical frames.

2) *Allocate code with random physical frames.* The NaCl process then allocates memory for code in the following fashion: right before a code chunk allocation request is sent to the OS, it unmaps a random chunk from reserved memory, i.e., release a random chunk of physical frames. Because of the high memory pressure, it is very likely that the following code allocation request will be served with the just released page frames. And because the page frames are random, once we repeat this process until half of the reserved memory is allocated for code, they are physically quite evenly distributed.
3) *Allocate data with random physical frames.* Finally we allocate data from the remaining (half) reserved memory using the same approach.

Before using this strategy on bare metal, we experimented in a VM, and printed out whether each physical page frame is assigned to code ("C") or data ("D"). A very small portion of this mapping is shown in Figure 5. We can see that code pages and data pages are sufficiently interleaved in physical address space.

Note that in our threat model (a malicious web application), attackers do not have the freedom to run another normal process on the target machine to create memory pressure. However, this could still be achieved by embedding one or more extra memory consuming NaCl modules on the same web page (or JavaScript ArrayBuffers in multiple pages). Since each NaCl module is allowed to use up to 4G of memory for x86-64, sufficient memory pressure could be created.

With above setting, we have used the non-temporal stores to trigger bit flips in code, and escaped the sandbox.

V. Bit Flips with Benign Code

We discuss rowhammer under a different threat model in this section. The "benign-code-untrusted-input" model is first described in detail, and then the feasibility of different rowhammer methods are compared. We continue by illustrating the more wide usage of non-temporal instructions and how memset/memcpy functions can be used as a primitive to simplify our rowhammer attempts. Finally, possible remote rowhammer attacks are discussed.

A. Threat Model

Our threat model is as follows: We assume the DRAM of the system is susceptible to the rowhammer bug. We assume attackers do *not* have local access to a system, i.e., they cannot run processes. We assume there is benign software running on the system which does not deliberately compromise system security. These software can be, but are not limited to multimedia players, PDF file readers, Internet service servers, or file compression utilities. We assume the benign software do not contain *software-based* memory corruption vulnerabilities, but if they do present, more sophisticated and powerful attacks could be launched.

B. Rowhammer Methods

Several different approaches have been shown to be effective for rowhammer. Based on the memory access properties, they can be roughly classified as cached or uncached. The original paper [15] uses uncached memory accesses and relies on *CLFLUSH* instruction. In this paper, we have shown a new uncached approach based on non-temporal store instructions.

Recently, cached memory accesses have also been shown to have the capability of triggering rowhammer [9], [10]. The basic idea is to use a memory access pattern that can effectively fill a cache set and evict cache lines therefore resulting in DRAM access. While Aweke et al [9] used a static pattern with native code, Daniel Gruss et al developed an adaptive algorithm to work on more CPU microarchitectures, and generated the pattern both with native code and JavaScript [10].

Although not requiring special instructions, cached rowhammer methods need a complicated memory access pattern to evict cache lines. While this is possible when attackers can execute his own code, it is very difficult to induce the access pattern only by feeding malicious inputs to existing (benign) code, especially in a non-scripting environment. Moreover, cached rowhammer methods require special system settings/interfaces such as huge pages, or virtual to physical address mapping (e.g., /proc/pid/pagemap of Linux) which are not always available. And finally, it is not clear whether the cache eviction technique can be applied to exclusive cache schemes that are adopted, e.g., by AMD CPUs.

As a comparison, uncached approaches only need a simple access pattern, do not require special system setting, and are cache scheme independent. Therefore, they are much less constrained. However, it is not possible without special instructions. We therefore perform a study of availability of rowhammer-capable instructions in real world software.

C. Rowhammer-Capable Instruction Availability

We performed a search of *CLFLUSH* and non-temporal store instructions in Debian source code repository. We have found that non-temporal stores are more widely used: in our data set, 21 software packages contain non-temporal stores, while only 7 packages contain *CLFLUSH*. Non-temporal stores also present in a more diverse set of software: they exist not only in OS kernels and compilers, but also in window managers, boot loaders, and especially (10 different) multimedia players. As multimedia player are usually well exposed to untrusted inputs, they can be a convenient target.

libc implementation	Newlib	uClibc	Bionic (Android)	Glibc	musl	dietlibc
used in memset /memcpy	Y	Y	Y	Y	N	N
non-temporal store execution threshold size	256 bytes	120K bytes	128K bytes	∼700K bytes	N/A	N/A
rowhammer-ready	Y	N	N	N	N/A	N/A

Fig. 6. Non-temporal stores in libc implementations

Non-temporal stores present in many software probably because they have great potential for memory access optimizations. The ability of avoiding cache pollution offers new opportunities for improving performance of some applications, especially data-intensive ones such as multimedia players.

D. A New Rowhammer Primitive: memset/memcpy

Non-temporal stores also exist in an important piece of software: the C library. Specifically, they are used in libc's memset(3) and memcpy(3) functions: when the filled/copied data are not expected to be accessed soon, they are not stored to cache.

This finding has important security implications. Since libc is linked by virtually all software, if non-temporal stores are used in the memset/memcpy implementations, and if rowhammer can be triggered by just calling memset or memcpy, almost all software has the potential of rowhammering.

To understand the impact, we first analyzed the usage of non-temporal stores in memset/memcpy in different implementations of libc, and under what scenarios they can be executed. Figure 6 shows our results.

We have found that 4 out of 6 popular libc implementations use non-temporal stores in memset/memcpy, including Glibc. However, the non-temporal store instructions may not get executed for all invocations of the functions. This is because there can be alternative implementations inside the same memset/memcpy function body, and the executed implementation is usually based on the requested filling/copying size. This is reasonable because preventing data from storing to cache is most useful for large chunk of data movement: otherwise a large portion of cache is polluted. When the moved data size is small, on the other hand, there is no need to prevent data store to cache because not much is polluted. Storing to cache may even be beneficial: the small chunk of moved data may turn out to be temporal and accessed soon.

As a result, there is usually a threshold value for each implementation: if the requested size is larger, the non-temporal store based version is executed. From Figure 6 we can see the threshold varies from 256 bytes to around 700K bytes.

In order to use memset/memcpy for rowhammer, we need a large size of data movement to execute the non-temporal instructions within. However, a large data movement may take longer time to complete and result in a lower hammering rate. We have experimented with straight-forward memset/memcpy hammering for different implementations, and successfully triggered bit flips using the Newlib version. The code for memset is shown in Figure 7, and using memcpy is very similar. The dereferences of X and Y serve as normal cached accesses after non-temporal stores to the same addresses.

```
code3:
   memset(X, -1, 256);
   *(volatile char *)X;
   memset(Y, -1, 256);
   *(volatile char *)Y;
   goto code3;
```

Fig. 7. Rowhammer with memset

Due to a much larger latency, the straight-forward implementation for other libc implementations do not work. To have an estimation of what is the largest size of data copy that is still fast enough for rowhammer, we modified code3 of Figure 7 by changing the size argument of memset, and see if it can induce bit flips. After a "manual" binary search, we found that 2KB is the magic number for our test machine.

To overcome the latency problem of large data movement, one idea is to interrupt and cancel large data movement before it is finished, and start new memset/memcpy. However, our initial investigations of using signals and/or multi-threading provided negative results: the hammering rate is still not high enough.

We note that although it seems challenging and may take more effort to trigger rowhammer using memset/memcpy from other C libraries, at least all software using Newlib should beware of memset/memcpy based rowhammer. These include various embedded systems, the Red Hat GCC distribution, Cygwin, and Google's Native Client[4]. Moreover, memcpy/memset-like functions using non-temporal stores also appear in other critical software such as OS kernels. For example, the `pagezero` routine of FreeBSD fills zero to page-sized memory using non-temporal stores [2], and the `__copy_user_nocache` routine of Linux requires only 64 bytes to execute non-temporal stores [4].

E. Remote Rowhammer Attacks

Any program that can exhibit a memory access pattern such as code2b in Figure 3 or code3 in Figure 7 may be vulnerable to *remote rowhammer attacks*. In addition, other potentially existing software vulnerabilities can be utilized by attackers to alter the control flow to induce such pattern. As a result, the following attacks could be launched.

a) Data Corruption Attacks. Rowhammer can be used to induce bit flips in data pages, and critical user data can be corrupted. This is especially the case for data intensive applications, which pull massive data into memory.

The uniqueness of rowhammer is that it breaks memory protection from hardware level. Therefore, it is possible that the flipped bits belong to physical pages of another process. In this case, attackers can corrupt data of a process even if it is not directly exposed to malicious inputs. Therefore, this is a new threat needs to be considered for server consolidation.

b) Denial of Service Attacks. Another possible attack is denial of service. If bit flips happen at code, the underlying instruction will change, and could result in various faults when it is executed, and the process may terminate. On the

[4]NaCl has stopped using non-temporal instructions in their port of Newlib as a mitigation to rowhammer.

other hand, it is also possible that bit flips occur at control flow decision-making data, therefore program would follow a different execution path.

Similarly, denial of service can happen to another process or even the whole system, if one process is attacked.

VI. Possible Defenses

Since the introduction of the rowhammer problem, there are different approaches proposed for prevention or mitigation. These approaches target different layers of a computing system. In this section, we will discuss the techniques and their effectiveness against rowhammer threats, including our new method based on non-temporal stores.

A. Hardware Layer Assurance

At hardware level, measures can be taken at DRAM chips, CPU memory controllers, BIOS which configures memory controllers, or a combination of these components. The benefits of hardware-based approaches are that they are generic and agnostic of the specific rowhammer triggering methods.

Error-correcting code (ECC) can help mitigate rowhammer. However, it is mostly used by servers due to a larger cost. Moreover, ECC only provides limited detection/correction ability when there are multiple bit flips [15].

The LPDDR4 standard describes a technique called *target row refresh (TRR)* which identifies hot (frequently activated) rows and refreshes their neighbors [14]. However, TRR support is only optional for LPDDR4, and it is not part of the DDR4 standard [13]. Therefore, we cannot expect its broad implementation in the next generation of DRAM. Another related approach called *probabilistic adjacent row activation (PARA)* was proposed, but has not yet seen adoption [15].

The current mitigation by PC vendors for rowhammer is to double the refresh rate [8], [1]. Although this can reduce the possibility of rowhammer, there is no guarantee [15]. In addition, the patches require BIOS update, which are unlikely to be performed by most of the end users.

B. System Layer Defenses

Based on the observation that rowhammer attempts result in large number of last level cache misses, hardware performance counters can be leveraged to detect and prevent rowhammer attacks with a low overhead [11], [9]. Additionally, some critical system interfaces such as `pagemap` [5] and huge pages can be disabled to significantly increase the difficulty for cache eviction based rowhammer.

C. Application Layer Mitigation

Since lower layer defenses may not be widely deployed, to defend in depth, application layer mitigations should be applied whenever possible.

Perhaps the easiest mitigation at application layer is to forbid the presence or execution of rowhammer-capable instructions. Benign code should consider alternative ways of implementation and release patches to replace instructions such as *CLFLUSH* and non-temporal stores. For untrusted code which is sandboxed, the validator needs to make sure these instructions are not present. Although not a complete solution, this will force attackers to use eviction-based rowhammer, which is more complicated and sometimes more constrained.

VII. Conclusion

In this paper, we have demonstrated a new approach for rowhammer based on x86 non-temporal store instructions. We extended our findings and identified libc's memset and memcpy functions as a new rowhammer primitive. We developed an exploit that bypasses existing application layer defense and showed that our approach is practical. Moreover, unlike previous approaches, our approach does not require complicated memory access patterns, special system interfaces, or rarely used instructions. We introduced remote rowhammer threats that is more feasible with our approach, and finally discussed and analyzed the benefits and limitations of defenses deployed at different hardware and software layers.

References

[1] About the security content of Mac EFI security update 2015-001. https://support.apple.com/en-us/HT204934.

[2] FreeBSD source file. http://fxr.watson.org/fxr/source/amd64/amd64/support.S.

[3] Intel 64 and IA-32 architectures software developer's manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[4] Linux source file. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/arch/x86/lib/copy_user_64.S.

[5] pagemap: do not leak physical addresses to non-privileged userspace. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce.

[6] Program for testing for the DRAM rowhammer problem. https://github.com/google/rowhammer-test.

[7] Research report on using JIT to trigger rowhammer. http://xlab.tencent.com/en/2015/06/09/Research-report-on-using-JIT-to-trigger-RowHammer/.

[8] Row hammer privilege escalation. https://support.lenovo.com/us/en/product_security/row_hammer.

[9] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *ASPLOS*, 2016.

[10] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. *arXiv preprint arXiv:1507.06955*, 2015.

[11] N. Herath and A. Fogh. These are not your grand daddy's CPU performance counters: CPU hardware performance counters for security. In *Black Hat*, 2015.

[12] JEDEC. Standard No. 79-3F. DDR3 SDRAM Specification. July 2012.

[13] JEDEC. Standard No. 79-4A. DDR4 SDRAM Specification. Nov. 2013.

[14] JEDEC. Standard No. 209-4A. LPDDR4 Specification. Nov. 2015.

[15] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[16] M. Seaborn. How physical addresses map to rows and banks in DRAM. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

[17] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[18] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.