ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks

Zelalem Birhanu Aweke,[†] Salessawi Ferede Yitbarek,[†] Rui Qiao,[‡] Reetuparna Das,[†]
Matthew Hicks,[†] Yossi Oren,[*] and Todd Austin[†]

[†] University of Michigan [‡] Stony Brook University [*] Ben-Gurion University of the Negev

Abstract

Ensuring the integrity and security of the memory system is critical. Recent studies have shown serious security concerns due to "rowhammer" attacks, where repeated accesses to a row of memory cause bit flips in adjacent rows. Recent work by Google's Project Zero has shown how to leverage rowhammer-induced bit-flips as the basis for security exploits that include malicious code injection and memory privilege escalation. Being an important security concern, industry has attempted to defend against rowhammer attacks. Deployed defenses employ two strategies: (1) doubling the system DRAM refresh rate and (2) restricting access to the *CLFLUSH* instruction that attackers use to bypass the cache to increase memory access frequency (i.e., the rate of rowhammering).

We demonstrate that such defenses are inadequte: we implement rowhammer attacks that both avoid using the *CLFLUSH* instruction and cause bit flips with a doubled refresh rate. Our next-generation *CLFLUSH-free* rowhammer attack bypasses the cache by manipulating cache replacement state to allow frequent misses out of the last-level cache to DRAM rows of our choosing.

To protect existing systems from more advanced rowhammer attacks, we develop a *software-based* defense, ANVIL, which thwarts all known rowhammer attacks on existing systems. ANVIL detects rowhammer attacks by tracking the locality of DRAM accesses using existing hardware performance counters. Our detector identifies the rows being frequently accessed (i.e., the aggressors), then selectively refreshes the nearby victim rows to prevent hammering. Experiments running on real hardware with the SPEC2006 benchmarks show that ANVIL has less than a 1% false positive rate and an average slowdown of 1%. ANVIL is low-cost and robust, and our experiments indicate that it is an effective approach for protecting existing and future systems from even advanced rowhammer attacks.

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA
 ISBN 978-1-4503-4091-5/16/04$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872390

1. Introduction

In recent years, hardware security has grown in importance for all aspects of hardware design. The emergence of viable hardware security vulnerabilities, such as side-channel attacks [2] and malicious circuits [10, 22], ensures that this trend continues today. Perhaps no recent hardware security vulnerability has garnered more concern than the rowhammer attack [24]. The rowhammer attack allows the manipulation of data in a row of a DRAM (referred to as the victim row) by repeatedly accessing (or "hammering") adjacent rows. Serious security concerns arise when, for example, adjacent DRAM rows are **not** within the same memory protection domain [28].

1.1 Rowhammer Attacks

The rowhammer attack exploits the physical architecture of modern DRAMs, which are composed of numerous rows of densely packed capacitive bit cells. The bit cells hold the charge that represent program data values. Natural leakage of charge out of capacitive bit cells requires that the DRAM rows be refreshed (i.e., their charge restored through a refresh operation) regularly. Current DRAM architectures (e.g., DDR3) require a refresh command to be issued every 7.8us [3], refreshing individual rows once every 64ms. The rowhammer attack exploits an electrical cross-talk property within the dense interconnect of modern DRAMs. Repeated accesses to one row (the aggressor) within a single refresh cycle (e.g., 100's of thousands of accesses) speeds up the discharge of bit cells in adjacent rows (victim rows) [11]. This causes bit-flips in the victim rows most sensitive to hammering.

Recent studies have demonstrated the rowhammer attack on a wide variety of commercial system [24, 28]. These attacks used the *CLFLUSH* x86 instruction to flush specific cache lines, thereby allowing high locality rowhammer accesses to reach the DRAM unhindered by on-chip caches. Shortly after viable rowhammmer demonstrations emerged, work began to weaponize the mechanism, i.e., utilize it to breach a secure system. Of note, Google's Project Zero demonstrates two such weaponizations of a *CLFLUSH*-based rowhammer attack [28]. The first attack uses rowhammering to insert malicious code into Google's Native Client sandbox. The second attack is a privilege escalation attack

that allows a process to gain write access to its own page tables, and hence gain read-write access to all of physical memory.

1.2 Mitigating Rowhammer Attacks

The response from the hardware industry focuses on protecting both legacy and future systems. For legacy systems, industry employs two mitigation strategies. The first mitigation strategy is to double the refresh rate of DRAM systems, from once every 64ms to once every 32ms. Many hardware manufacturers, for example HP [15] and Lenovo [18], deployed firmware updates that double refresh rates as a rowhammer protection. However, this approach is insufficient, as shown by us in Section 2.1 and by others [24]. The state of the art in rowhammer attacks are effective at causing bit-flips within a 32ms refresh window. The second mitigation strategy involves restricting access to the *CLFLUSH* instruction which attacks use to flush cache lines and gain direct access to DRAM rows. For example, Google recently updated the Chrome Native Client sandbox (which allows local execution of remotely downloaded binaries) to prevent the loading of any code containing the *CLFLUSH* instruction [28]. Again, we show that this protection is insufficient by detailing the first CLFLUSH-free rowhammer attack.

An emerging defense, as suggested by some manufacturers, is that increasing ECC scrub rates could be a rowhammer protection mechanism [14]. But, prior work [24] shows multiple bit-flips per word when executing rowhammer attacks, making this approach of questionable value in protecting existing or future systems.

Protections for future DRAM architectures are beginning to appear in industry announcements and the literature. For example, the upcoming LPDDR4 specification and new DDR4 modules include a targeted row refresh (TRR) capability designed to thwart rowhammer attacks [19, 21]. The mechanism tracks the number of row activations within a fixed time window, and selectively refreshes rows neighboring a too-frequently accessed DRAM row. However, Micron DDR4 documentation [19] suggests that this mechanism is an "optional module", thus there may exist future DDR4 systems still susceptible to rowhammer attacks. In fact, bit flips due to rowhammering in DDR4 system have been reported [7]. Intel has partially disclosed the existence of pseudo-targeted row refresh (pTRR) in Xeon-class Ivybridge architectures to help in the mitigation of rowhammer attacks, but Intel has yet to release the details of this mechanism.

Finally, the architecture literature has seen a few rowhammer protection proposals. For example, one proposal, PARA, utilizes probabilistic adjacent row activation to refresh the neighboring rows of any DRAM row access, with low probability. The idea behind this approach is that the many repeated DRAM row accesses required to hammer a victim DRAM row will result in an early refresh of the victim row with extremely high (cumulative) probability [24]. Such solutions require the introduction of new hardware.

1.3 Contributions of This Work

In this work, we show that current rowhammer mitigation techniques for existing systems (i.e., disallowing cache flush instructions and doubling refresh rates) do *not* work. We demonstrate in this paper two attacks that defeat these protections. Specifically, we demonstrate the first *CLFLUSH-free* rowhammer attack, thereby thwarting efforts to deter rowhammering by restricting access to the *CLFLUSH* instruction. The attack manipulates cache replacement state to ensure high-frequency misses out of the last-level cache to DRAM rows of our choosing. The algorithm is able to efficiently implement a successful rowhammer attack within one 64ms DRAM refresh cycle, creating approximately 220,000 DRAM row accesses [1]. A critical implication of the *CLFLUSH*-free attack is that now any program with access to loads and stores can perform a rowhammer attack. In addition, we demonstrate that is it straightforward to rowhammer DRAM under a double-rate 32ms refresh cycle, underscoring the futility of this popular, but costly, protection mechanism.

As such, existing systems immediately require a more robust means of protection against rowhammer attacks. To this end, we present ANVIL, a software-based rowhammer detector which protects existing and future commodity DRAMs. We implement ANVIL using an existing hardware performance monitoring infrastructure. ANVIL works by monitoring the locality of DRAM row accesses out of the last-level cache. In the event that row access locality becomes too high (i.e., indicating repeated accesses to the same DRAM row), the detector selectively refreshes neighboring potential victim rows with a read operation (for DRAM, a read operation fully refreshes the accessed DRAM row).

We implement ANVIL as a Linux kernel module that utilizes Intel architectures' performance monitoring capabilities to detect DRAM row access locality out of the last-level cache. The performance monitoring capabilities we rely on are also available on AMD-based systems [6]. The detector uses a multi-staged approach to reduce detector overheads, leading to an average slowdown (for non-malicious programs running on real hardware) of about 1%, and worst-case slowdown of 3.2%. In addition, our detector has a latency of only 12ms. This enables the detection of rowhammer attacks that are more than twice as fast as the fastest known technique today. Finally, the detector is accurate, with no false negatives and less than 1% false positives. Beside their small performance impact, false positives are innocuous in that they incur only a small number of extra DRAM

[1] We were the first to disclose the existence of a *CLFLUSH*-free rowhammer attack. The attack was announced on May 10, 2015 on the Google Project Zero rowhammer forum. The details of the attack are being disclosed for the first time in this paper.

read operations. In summary, this paper makes the following contributions:

- We demonstrate attacks that side-step existing measures to protect systems from rowhammer attacks. Our *CLFLUSH*-free attack does not use the *CLFLUSH* instruction, instead, it cleverly manipulates cache replacement state to allow frequent misses out of the last-level cache to hammer DRAM rows of our choosing. Moreover, we demonstrate the ease with which recent rowhammer attacks can work successfully within double-rate 32ms refresh cycles.
- We propose, implement and analyze a *software-based* protection against rowhammer attacks, ANVIL, which successfully thwarts all of the known rowhammer attacks on commodity systems. ANVIL detects rowhammer attacks by tracking the locality of DRAM row accesses using existing hardware performance counters. When a potential attack is detected, it is thwarted by selectively refreshing identified victim DRAM rows with a simple read operation. Running on real hardware, ANVIL has a less than 1% false positive rate, and it uses a multi-stage design to reduce average slowdowns to only 1% across diverse benchmarks.

2. Breaking Current Mitigation Techniques

Multiple techniques have been proposed to protect existing systems from DRAM rowhammering errors. Currently deployed mitigation techniques include doubling the DRAM refresh rate and disallowing cache flush instructions. In this section, we show that these techniques are insufficient to guarantee protection from rowhammer exploits. First, we show that a refresh period of 32ms is sufficient time to implement a rowhammer attack. Second, we show how to implement a rowhammer attack without using the CLFLUSH instruction.

2.1 Rowhammering under a Double Refresh Rate

After DRAM rowhammering errors and their security implications were widely recognized, a number of vendors published BIOS updates that double the rate at which DRAM refreshes its data [15, 18]. By refreshing the DRAM more frequently, it is believed that there is insufficient time to carry out a rowhammering attack. We perform experiments on a commodity platform that show that this belief is indeed false. Even when refresh intervals are reduced to 32ms, it is still possible for a malicious program to cause bit flips by repeatedly accessing two rows adjacent to a victim row using a rowhammering technique dubbed *double-sided rowhammering* [28]. Table 1 lists our experimental results for three rowhammer attacks. The attacks are analyzed on a real system with an Intel core i5-2540M processor (Sandy Bridge) and a 4GB DDR3 DRAM module while running Ubuntu 14.04 LTS. As shown in the results of Table 1, it is possible to employ double-sided rowhammering using the CLFLUSH instruction to flip bits in only 15ms on our DDR3 module—well below the 32ms window of deployed defenses.

Hammer Technique	Minimum Number of DRAM Row Accesses	Time to first bit flip
Single-Sided with CLFLUSH	400K	58 ms
Double-Sided with CLFLUSH	220K	15 ms
Double-Sided without CLFLUSH	220K	45 ms

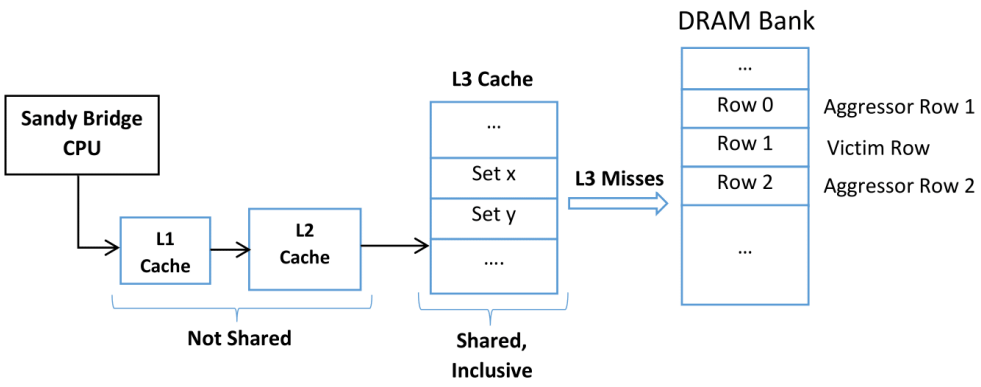
Table 1: Rowhammer Attack Characteristics: The measured performance of three rowhammer techniques, i.e., single and double-sided rowhammering and with/without CLFLUSH to flush the cache. The experiments are run on a Ubuntu-based Sandy Bridge laptop with a 4GB DDR3 DRAM module. The table gives the minimum number of DRAM row accesses required to induce a bit-flip and the time until the first bit-flip.

Sequence (a) in Figure 1 shows the access sequence used to implement our double-sided rowhammer attack using CLFLUSH instructions. The attack involves three rows: Rows 0 and 2 are the aggressor rows, and Row 1 is the victim row. The aggressor rows are repeatedly activated to increase the discharge rate in the victim row. The attack works by first accessing an address in Row 0 (i.e., $A0_{(row0)}$); Then an address in Row 2 ($A1_{(row2)}$) is accessed. After each access, a CLFLUSH instruction is used to flush all levels of cache at addresses $A0_{(row0)}$ and $A1_{(row2)}$ thereby ensuring the next access goes directly to the DRAM. This sequence is repeated N number of times; for our experiments the minimum value of N was equal to 110k to see a bit flip.

Given the results of our experiment, one might suggest further increases in refresh rate. The problem with this approach, in general, is that increasing the refresh rate comes at the cost of increased power and reduced DRAM throughput—as refresh commands compete with software-requested memory accesses. Going from a 64ms refresh period to the 15ms required to protect our DRAM requires over a 4x increase in refresh power and throughput overhead. Also, as DRAM continues to move to smaller feature sizes, the vendors will likely have to lower the refresh rate more to account for increased density (i.e., future DRAM devices may be more susceptible to rowhammering at the cell level).

2.2 Rowhammering without the CLFLUSH instruction

Modern processors include multiple levels of cache for faster access of frequently used data. It is common to have three levels of cache with the last-level cache capable of



a) CLFLUSH Attack: $(A0_{(row0)}, CLFLUSH(row0), A1_{(row2)}, CLFLUSH(row2))^{N=110K}$

b) CLFLUSH-free Attack: $(A0_{(row0,\ setx)}, X1_{(setx)}, X2_{(setx)}, \ldots, X9_{(setx)}, X10_{(setx)}, X11_{(setx)}, X1_{(setx)}, X2_{(setx)}, \ldots, X9_{(setx)}, X12_{(setx)}, A1_{(row2,\ sety)}, Y1_{(sety)}, Y2_{(sety)}, \ldots, Y9_{(sety)}, Y10_{(sety)}, Y11_{(sety)}, Y1_{(sety)}, Y2_{(sety)}, \ldots, Y9_{(sety)}, Y12_{(sety)})^{N=110K}$

Figure 1: Memory Access Patterns for CLFLUSH-based & CLFLUSH-free Double-sided Rowhammer Attacks: In (a), a CLFLUSH instruction is used to flush caches after accessing aggressor DRAM rows Row 0 and Row 2. This sequence of operations is repeated N times. In our experiments the minimum value of N observed was 110K. In (b) the CLFLUSH instructions are replaced with sequences of memory accesses that force misses in the L3 cache at addresses that map to aggressor Row 0 and aggressor Row 2. This is done by accessing conflicting data that belong to the same cache set as the aggressor row addresses. *A0:* address in aggressor row 1 and maps to set X; *A1:* address in aggressor row 2 and maps to set Y; *X1, X2, ... X12:* addresses that map to cache set X and evict A0;*Y1, Y2, ... Y12:* addresses that map to cache set Y and evict A1.

storing megabytes of data. In order to repetitively access a DRAM row, memory access to that DRAM row must miss on all cache levels and the DRAM row buffer. One way to achieve this is to use cache flushing instructions like CLFLUSH on the x86 architecture. Previous works on exploiting the rowhammer problem all used the CLFLUSH instruction to bypass caches. One counter measure that has been taken to thwart CLFLUSH-based attacks is to disallow the CLFLUSH instruction [28]. Such measures thwart rowhammmer attacks based on cache flushing, but we show that it is possible to implement a rowhammer attack without using cache flush instructions.

Rowhammering in the presence of caches: Caches are limited in size; given varied enough accesses, eventually a cache will fill and memory accesses will miss in the cache. One way to force a miss from a cache is to evict previously accessed data by accessing conflicting data that belongs to the same cache set. To accomplish this, an *eviction set* that contains addresses that belong to the same cache set is created. Then the addresses in the eviction set are accessed one after the other to force eviction of a particular data element from the cache. If the access sequence is cleverly designed to manipulate the cache eviction policy, it is possible to precisely control which addresses hit in the cache and which addresses miss the cache and make it to main memory. This minimizes the delay between subsequent target misses. By repetitively evicting data from a target address and then re-accessing it, corresponding DRAM rows can be accessed.

Nonetheless, there are significant challenges in devising efficient address reference streams that can implement a rowhammer attack. First, last-level caches in modern processors have high associativity, usually 8-way to 16-way. Because each way can hold the address of the aggressor row, we must generate at least as many conflicting memory accesses that hit the cache as there are ways. Therefore, many memory accesses are required to evict a cache block, which slows down the rowhammering process. In addition, replacement policies used on real hardware are not true LRU, and vendors often do *not* publicly disclose their replacement algorithms. This means that access patterns that assume true LRU replacement policy often do not result in misses on the required target addresses. Missing on the exact target addresses is important as creating extraneous memory acceses dramatically decreases the rate of rowhammering. Finally, creating eviction sets can be a challenge in situations where the mapping of an address to a cache set is via physical address, as the mapping algorithm is not publicly disclosed by manufacturers and user-level application may not have access to the virtual memory mapping.

Demonstration of the attack: In this section we describe how we were able to overcome the challenges mentioned above to do CLFLUSH-free rowhammering. For our demon-

stration, we use a processor with an Intel Sandy Bridge microarchitecture. The processor has three levels of cache. The last-level cache is an inclusive, shared, physically indexed 12-way cache. It is an inclusive cache, therefore it is enough to evict a word from the last-level cache to bypass the whole cache hierarchy.

One way to create an eviction set is to directly use physical addresses and select memory addresses with the same set-index bits. Previous work in this area has revealed the mapping for an Intel Sandy Bridge microarchitecture with four CPU cores [12]. We discover that our Sandy Bridge microarchitecture based processor with two CPU cores uses a slightly modified version of this mapping. In our eviction set, we have one address that belongs to a row (which we call an aggressor address). Since our cache is a 12-way cache, we need 13 addresses in the eviction set, 12 conflicting address and the aggressor address. We create an eviction set by first picking the aggressor address and then using its physical address to find 12 more addresses with matching cache set mappings. On our Intel Sandy Bridge machine, bits 6 to 16 of the physical addresses are used to map to last-level cache sets. Furthermore, the last-level cache is organized into *slices* [16], with one slice per processor core. Conflicting addresses will have the same cache slice and cache set bits.

The next step is to create an efficient memory access pattern that has a high probability of misses on the aggressor address. Creating such a pattern requires knowing the cache replacement policy of the microarchitecture. We did this by generating a high miss-rate pattern that cyclically accesses the 13 addresses in the eviction set, and using performance counters (particularly the last-level cache miss counter) to determine whether each access was a cache hit or a cache miss. Then we correlate the performance counter results with results from different cache replacement policy simulators that we built. Our results show that one of the replacement algorithms Sandy Bridge favors (it uses more than one) is Bit Pseudo-LRU (Bit-PLRU) which is similar to the Not Recently Used (NRU) replacement policy [20]. In Bit-PLRU, each cache line in a set has a single MRU (Most Recently Used) bit. Every time a cache line is accessed, its MRU bit is set. The least-recently used cache line is the line with the lowest index whose MRU bit is cleared. When the last MRU bit is set, the other MRU bits in the set are cleared.

A time efficient memory access pattern misses the last-level cache only on the aggressor address and one additional conflicting address, and hits on the rest of addresses in the eviction set. This works by always driving the aggressor address to the least recently used position in the replacement state. Sequence (b) in Figure 1 outlines the access pattern we used for our CLFLUSH-free double-sided rowhammer attack. This attack is similar to the CLFLUSH-based attack except here the CLFLUSH instructions are replaced with memory accesses that drive the two aggressor DRAM row addresses to the least recently used (LRU) position in the L3 cache and subsequently evict them, thereby ensuring their next access goes to the aggressor DRAM rows. In Figure 1b, address $A0_{(row0,setx)}$ belongs to Row 0 in the DRAM, and *Set X* in the L3 cache. Address $A1_{(row2,sety)}$ belongs to Row 2 in the DRAM, and *Set Y* in the L3 cache. The two addresses constitute the aggressor addresses. First, data at address $A0_{(row0,setx)}$ is accessed. Then 10 addresses ($X1_{(setx)}$ to $X10_{(setx)}$) that belong to *Set X* are accessed to put $A0_{(row0,setx)}$ to the LRU position of the L3 cache. Then, when data from address $X11_{(setx)}$ is accessed, data at address $A0_{(row0,setx)}$ is evicted from the L3 cache. The next 9 accesses ($X1_{(setx)}$ to $X9_{(setx)}$) hit in the L3. Then, after data at address $X12_{(setx)}$ is accessed, address $X11_{(setx)}$ is put to the LRU position and subsequently replaced by data at address $A0_{(row0,setx)}$. This access sequence is repeated N times, with only two addresses ($A0_{(row0,setx)}$ and $X11_{(setx)}$) missing for each iteration. In *Set Y*, a similar access pattern is used to miss only from addresses $A1_{(row2,sety)}$ and $Y11_{(sety)}$. Using this technique, accesses to Row 0 and Row 2 will always access the DRAM.

Access to the last-level cache on Sandy Bridge takes 26 to 31 cycles [16]. Considering a DRAM access latency of 150 cycles, the access pattern in sequence (b) in Figure 1 takes an estimated (29*20) + (2*150) = 880 cycles. On our test machine, which runs at a nominal frequency of 2.6GHz, this access pattern takes approximately 338 nanoseconds. This allows up to 190K double-sided hammers with-in a 64ms refresh period. This is enough to produce a flip on our test DRAM module—which only requires 110k accesses to produce a bit flip.

In addition to rowhammering, the technique used in the CLFLUSH-free rowhammering attack can be used in other attacks that need to flush the cache at specific addresses. For example the Flush + Reload cache side-channel attack [29] relies on the CLFLUSH instruction. Our CLFLUSH-free cache flushing method can extend this attack to situations where the CLFLUSH instruction is not available (e.g., JavaScript) – a similar approach was explored in [27].

2.3 Attack Implementations

The CLFLUSH-based and CLFLUSH-free attacks were implemented as native C++ applications. The implementations are based on the double-sided rowhammering attack implementation [1]. The CLFLUSH-free rowhammering attack uses the Linux /proc/pagemap utility to convert virtual addresses to physical addresses in order to create conflicting LLC access patterns (eviction set). We demonstrated the attacks on a laptop with an Intel core i5-2540M processor (Sandy Bridge) with a 4GB DDR3 DRAM module.

Table 1 compares the minimum number of DRAM row accesses and the corresponding time required to produce a bit flip for the CLFLUSH-based and CLFLUSH-free attacks for our test DRAM module. Double-sided, CLFLUSH-based rowhammering is the most aggressive of the three.

It is also worth noting that a double-sided CLFLUSH-free rowhammering can produce bit flips faster than single-sided CLFLUSH-based rowhammering.

It is interesting to note that if both of the protection mechanisms detailed in this section are used in tandem (i.e., double refresh plus restricted access to CLFLUSH), such a system would still today have a measure of protection against rowhammer attacks, including those detailed in this paper. As shown in Table 1, we are unable to yet rowhammer memory in less than 32ms without use of the CLFLUSH instruction. While we are unaware of any systems that combine these two protection measures, one that did would likely only acquire a temporary measure of protection against novel rowhammer attacks. We continue to optimize the performance of our CLFLUSH-free attack, and if we are able to reduce its time-to-first bit flip by an additional 13ms, the combined protections will no longer work. Recognizing the tenuous nature of today's rowhammer protections, we feel a better approach to protect systems is to provide in-situ mechanisms that detect and subsequently defeat rowhammer attacks.

In summary, current techniques used to protect systems from rowhammer attacks are insufficient. We show that reducing the DRAM refresh period to 32ms is not sufficient as faster rowhammer attacks are possible using double-sided rowhammering in as little at 15ms. Moreover, by manipulating the LRU chain of the last-level cache, enough DRAM row activations can be performed in a single refresh cycle to flip bits, without using the CLFLUSH instruction.

3. Software-Based Rowhammer Detection and Protection

As Section 2 shows, currently deployed rowhammer defenses are insufficient. What is needed is a more robust solution that can detect rowhammering activity in time to protect any potential victim rows. In this section, we introduce a software technique that uses existing hardware performance counters in commercial processors to detect rowhammering activity and perform selective refresh on potential victim rows.

3.1 Detecting Rowhammer Attacks

Rowhammering relies on repetitively accessing an aggressor DRAM row within a single refresh cycle. We make the observation that this fundamentally requires accesses to the aggressor rows to miss on all cache levels. This reveals two identifying characteristics of rowhammering: *high cache miss rate* and *high spatial locality of DRAM row accesses*. This is in contrast to general memory access patterns where high locality results in high cache hit rates. As such it is straightforward to discriminate between rowhammer attacks and non-malcious programs by looking at DRAM access patterns and rate.

Another property of rowhammer attacks is high *bank locality*. DRAM disturbance errors occur due to repeated opening and closing of a DRAM row. When a row is accessed, it is opened and its data is transfered to a row-buffer. Subsequent accesses to the same row are served by the row-buffer. In order to close the row, a different row located in the same bank must be accessed. Therefore, a rowhammer attack involves repeatedly accessing at least two rows within the same bank—otherwise the row buffer would prevent the rowhammering. This bank locality property can be used to differentiate between "real" rowhammering and false positives that are caused by thrashing access patterns observed in some applications.

To minimize the performance impact of rowhammer detection, we propose a two-stage detection mechanism. In the first stage, we monitor the last-level cache miss rate. If this rate is high enough to successfully implement a rowhammer attack, the second stage samples the physical addresses of the memory accesses that miss from the last-level cache. If the samples reveal DRAM row accesses with high temporal locality, then the detector signals this as a potential rowhammer attack. To reduce the possibility of false positives, the detector also verifies that the samples have high bank locality. If there is enough bank locality among samples, then a protection phase follows.

3.2 Protecting Potential Rowhammer Victims

When the detector identifies potential rowhammering activity, it identifies the potential victim DRAM rows. Victim rows are adjacent to (preceding and following) identified aggressor rows. To protect the victim rows we refresh them by reading a word from them. Reading from a row opens that row which has the effect of refreshing cells in the row [24]. This approach does not incur significant performance penalties even in the case of false positives.

3.3 ANVIL: A Linux-Based Rowhammer Protection Mechanism

To demonstrate the protection mechanism, we built ANVIL, a Linux kernel module that prevents all known forms of rowhammer attacks. The module uses hardware performance counters found in modern processors to get memory access information, such as the addresses of loads and stores and the miss rate of the last-level cache. Specifically, we used performance counters found in Intel microprocessors with Sandy Bridge and later microarchitectures. AMD also provides similar capabilities required for our implementation [6]. In this section we provide details of our implementation. We start by reviewing the performance counter features used in our implementation.

Load Latency Performance Monitoring Facility: The Load Latency performance monitoring facility is part of Intel's Performance Event Base Sampling (PEBS) feature. PEBS uses a debug store mechanism and a performance

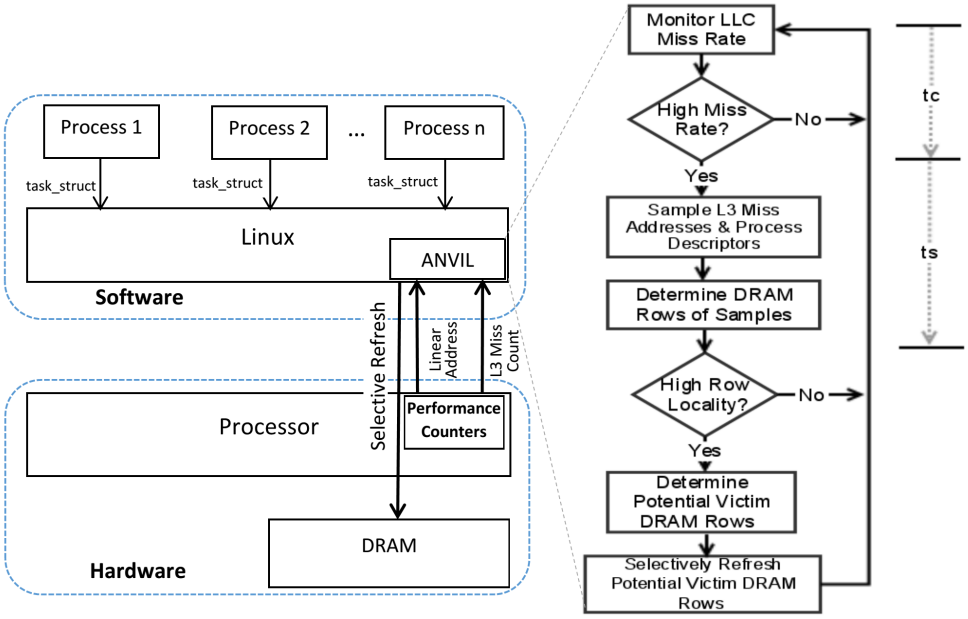


Figure 2: Software-Based Rowhammer Attack Detector: ANVIL is as a kernel module. It gets last-level cache miss count and memory access samples from hardware performance counters. By combining sampled virtual address information and process descriptor structures, samples of DRAM row accesses are obtained. ANVIL then checks the samples for high locality that suggests potential rowhammer activity. Upon detection of potential rowhammer activity, ANVIL performs selective read operations to refresh potential victim DRAM rows.

monitoring interrupt to store a set of architectural states [17]. The load latency facility measures latency of a load operation from the load's first dispatch until final data writeback from the memory subsystem. The load operation is sampled probablistically by hardware. If the latency of the sampled load operation exceeds a latency value specified by a dedicated programmable register, the operation is tagged to carry the following information:

- Load data virtual address
- Data source
- Latency value

When the next event categorized as a precise event (e.g., "load retired", "store retired") occurs, the last update of the load information is written to a PEBS record which then can be read by software. By setting the latency threshold to match last-level cache miss latency, it is possible to sample last-level cache misses. The data source information confirms the source of the load operation.

Precise Store Facility: The Precise Store facility complements the Load Latency facility by providing additional information about sampled store operations. When a precise event occurs, hardware samples the virtual address and data source of the next store that retires. Similar to the Load Latency event, data source information can be used to determine if the store was a miss. The precise store facility is replaced with the Data Address Profiling facility on Intel's Haswell and later microarchitectures [17]. This facility profiles load and store memory events similar to the other facilities, but has support for more events like DRAM access events. While we could implement ANVIL with either performance counter, we use the Precise Store facility for our implementation since it allows our rowhammer detection mechanism to support older micro-architectures.

In addition to the previously mentioned facilities, we utilize the last-level cache miss counter facility that generates an interrupt after N misses. The count is set such that if the miss interrupt arrives before the sample window timer interrupt, we know that the miss threshold has been breached.

Rowhammer Detection: Figure 2 shows the process of detecting a rowhammer activity in ANVIL. In the first stage of the detection phase, the last-level cache miss count event (LONGEST_LAT_CACHE.MISS) is used to measure the last-level cache miss rate. The miss rate is calculated by reading the last-level cache miss count for a time duration of t_c. If this rate is beyond a last-level cache miss threshold, the second stage of the detector is triggered. The last-level cache miss threshold (LLC_MISS_THRESHOLD) is set by considering the minimum cache miss rate that is enough to cause bit-flips within a single refresh period. As will be described in the next section, we set this value based on our

empirical observations, but it is adaptable to other systems and attack scenarios.

In the second stage, ANVIL samples virtual addresses for a time duration of t_s using Load Latency (MEM_TRANS_RETIRED.LOAD_LATENCY) and Precise Store (MEM_TRANS_RETIRED.PRECISE_STORE) events. The load latency facility allows sampling of loads that have latency beyond a preset clock cycle value. We set the clock cycle value to match last-level cache miss latency so that we only sample loads that miss in the L3 cache. The counter also provides information about the source of the sample, therefore we can ensure the load is accessing DRAM. The precise store facility is used to sample stores. It also provides information about the source of a store operation. Which facility to use for sampling is selected based on a count of retired memory load operations that missed from the last-level cache (MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS) for a time duration of t_c. ANVIL compares this value with the total number of last-level cache misses for that duration. If load operations account for more than 90% of all misses then only loads are sampled. On the other hand, if load operations account for less than 10% of all misses, only stores are sampled. For the remaining cases, both stores and loads are sampled. A sampling rate of 5000 samples per second is used which gives an average of 30 samples for a sampling duration of 6ms. The second stage also samples the process descriptor (task_struct) of the process that generated the memory access. This structure is used to determine the physical address and DRAM row of the memory access in combination with the sampled virtual address.

At the end of sampling, sampled DRAM row accesses are sorted and the sample distribution is analyzed to identify high DRAM row locality. DRAM row locality is determined by considering the number of samples, the number of last-level cache misses for the sampling duration and the required last-level cache miss rate for a successful rowhammr attack. For each row that has high DRAM locality, a check is made to see if there are other row access samples from the same DRAM bank. If the cumulative of samples of the other row accesses from the same DRAM bank is high enough, then there is a potential rowhammer attack occurring.

Rowhammer Protection: Once ANVIL detects potential rowhammering activity, it uses the physical addresses of the identified aggressor rows to determine potential victim rows. The kernel module was pre-configured using a reverse engineered physical address to DRAM row and bank mapping scheme. We also make the assumption that sequentially numbered rows are physically adjacent. Two potential victim rows are considered for each potential aggressor row: rows that are directly above and below each potential aggressor row (our approach easily extends to N adjacent rows). ANVIL performs a single read operation per victim row to refresh its value. The number of selective read operations performed on a potential victim row is low enough (once every $t_c + t_s$ in the worst case) that it has little effect on performance of non-rowhammering applications, even if they experience a high incidence of false positive detection. Also, it is not possible for an attacker to use the selective refresh mechanism to rowhammer DRAM rows adjacent to the potential victim row since the selective read rate is well below the minimum access rate for a rowhammer attack. After performing a selective refresh, ANVIL starts the detection process again.

4. Experimental Evaluation

In this section we evaluate the accuracy and performance of our software-based detection mechanism. All tests are conducted on a real physical system with an Intel Core i5-2540M processor and Ubuntu 14.04 LTS with Linux kernel version 4.0.0.

4.1 Benchmark Applications

We use several benchmark programs for our evaluations. To evaluate rowhammer detection accuracy, we use two rowhammer attacks. The first is a CLFLUSH-based double-sided rowhammer attack, *CLFLUSH_hammer*, adapted from the original weaponization [1]. The second application is CLFLUSH-free double-sided rowhammer attack, *CLFLUSH-free_hammer*, used to demonstrate our CLFLUSH-free attack in Section 2[1]. We measure the slowdown incurred on non-malicious programs using SPEC2006 integer benchmarks [9].

4.2 Rowhammer Detection Characteristics

We first evaluate ANVIL's ability to detect rowhammer activity. The evaluation is done for scenarios where the test machine is heavily and lightly loaded. To emulate heavy load, we run the rowhammering applications along with memory-intensive applications (*mcf*, *libquantum* and *omnetpp* running at the same time) from the SPEC2006 integer benchmark suite. The detector parameters used for our evaluations are given on Table 2. The time values are selected to be low enough so that any rowhammering activity can be detected with enough time to deploy protection. With this setting, hammering activity can be detected within 12 milliseconds. The last-level cache miss threshold value was experimentally found by considering the minimum number of memory accesses required to cause a DRAM bit flip. In our experiments the minimum number of memory accesses that caused a flip was 220K for CLFLUSH-based double-sided rowhammering attack. In order to achieve this many activations within a refresh period of 64ms, a minimum of 20.6K activations must occur within 6ms. Therefore, we will use 20K misses in 6ms as a threshold value for the first stage of detection.

[1] Code for the *CLFLUSH-free_hammer* program and ANVIL can be found at `https://github.com/zaweke/rowhammer`

Parameter	Value
LLC_MISS_THRESHOLD	20K
Miss Count Duration (t_c)	6ms
Sampling Duration (t_s)	6ms

Table 2: Rowhammer Detector Parameters to Evaluate Accuracy of Rowhammer Detection

Benchmark	Average Time to Detect	Refreshes per 64ms	Total Bit Flips
CLFLUSH *(Heavy Load)*	12.8 ms	12.35	0
CLFLUSH *(Light Load)*	12.3 ms	10.3	0
CLFLUSH-free *(Heavy Load)*	35.3 ms	4.53	0
CLFLUSH-free *(Light Loaded)*	22.85 ms	5.10	0

Table 3: Rowhammer Detection Result for Rowhammering Programs: The table shows the average time before rowhammer activity is detected and the rate of selective refreshes performed.

Table 3 shows the result of rowhammering detection for applications *CLFLUSH_hammer* and *CLFLUSH-free_hammer* under heavy and light load. For both attacks, the table shows the average time to detect a rowhammer attack within a 64ms refresh cycle in which rowhammering was occurring. This time includes the time to identify and selectively refresh potential victim rows. The table also lists the average selective refresh rate, which are refreshes that occur when the rowhammer detector identifies potential DRAM victim rows. As seen in these results, ANVIL is quite responsive, with response times well within a single refresh cycle, and with only slight increases in response time due to a heavy loaded system. In addition, the selective refresh rates are low, but sufficient for multiple refreshes within a single refresh cycle for any detected victim row. The low selective refresh rate ensures that a clever attacker cannot use the selective refresh to hammer other DRAM rows. Finally, it is good to note that our detector stopped all rowhammering, resulting in zero bit flips for all of the attacks.

4.3 Performance Evaluation

We evaluated the slowdown incurred by ANVIL by analyzing the execution of non-malicious applications from the SPEC2006 integer benchmark suite. We used the parameters listed on Table 2 for the evaluation. In addition to this experiment, we compare the performance overhead of ANVIL with that incurred by doubling DRAM refresh rate. For these evaluations our baseline is an unprotected system with a refresh period of 64ms.

Benchmark	Refreshes/sec
astar	0.10
bzip2	1.05
gcc	0.71
gobmk	0.19
h264ref	0.00
hmmer	0.00
libquantum	0.06
mcf	0.01
omnetpp	0.02
perlbench	0.00
sjeng	0.00
xalancbmk	0.05

Table 4: Rate of False Positive Refreshes: The table shows rate of superfluous refreshes for SPEC2006 integer benchmarks while running under ANVIL.

Figure 3 shows relative execution times for ANVIL-protected system relative to our baseline. The ANVIL-protected system has peak and average overheads of 3.18% and 1.17%, respectively. Most of the performance overhead by ANVIL is attributed to the low last-level cache miss rate threshold. *libquantum*, *omnetpp*, *mcf* and *Xalancbmk* crossed the last-level cache miss threshold 95% to 99% of the time. On the other extreme, *h264ref*, *gobmk*, *sjeng* and *hmmer* crossed the threshold less than 10% of the time. This indicates that sampling of addresses in the second stage of the detection phase contributes to almost all of the performance overhead. As can be observed from the results, the overheads of continuously running ANVIL's rowhammer detection are very low. Low enough to protect existing systems from rowhammer attacks, and likely low enough to obviate the need for dedicated hardware-based rowhammer protection mechanisms in future systems.

Table 4 shows the false positive rate for the SPEC2006 integer benchmarks. The rate is measured as the average number of superfluous selective refreshes per second. The number of false positives is low enough that selective refresh of rows has negligible effect on performance.

4.4 Comparison with Double Refresh Rate

As we have shown in Section 2, doubling DRAM refresh rate is not sufficient to prevent all rowhammering attacks. Equally important is the execution time and power overhead incurred by the increased refresh rate. Previous studies have shown that increasing refresh rate reduces parallelism in the

memory subsystem, affecting overall system performance [5, 26]. Figure 3 shows performance overhead of doubling DRAM refresh rate as compared with our software-based protection mechanism. ANVIL's performance overheads are only marginally larger (on average) than doubling the refresh rate, while providing a significantly higher level of protection against rowhammer attacks, as demonstrated in Section 2. As can be observed, memory intensive applications like *mcf* suffer most from doubling DRAM refresh rate thus, their performance benefits greatly from the use of ANVIL's protection.

4.5 Robustness to Potential Future Rowhammer Attacks

As the density of DRAM devices increases, DRAM cells become more susceptible to disturbance errors. It is then expected that for future DRAM devices, rowhammer attacks will be possible with less DRAM row activations. An attacker might take advantage of this to evade detection by our software-based protection mechanism by: 1) Activating DRAM aggressor rows at a high rate such that rowhammer attacks will be faster than they can be detected by the protection mechanism. 2) Spreading out fewer DRAM row activations over a refresh period such that the last-level cache miss rate stays below the last-level cache miss threshold. Our detection mechanism can cope with both situations by adjusting the detector parameters listed on Table 2. To evaluate the effect that more nimble future attacks have on the performance of non-malicious programs, we consider a future scenario where bit flips can occur with 110K DRAM row accesses (i.e., half the number of accesses that produced flips on our experiments).

Figure 4 examines the performance impact on a subset of the SPEC2006 benchmarks for three cases. The benchmarks are selected to be representatives of the memory access characteristics of SPEC2006 benchmark suit. *ANVIL-baseline* is our baseline detector with parameters as given on Table 2. *ANVIL-heavy* considers the case where the 110K DRAM row accesses can occur within 7.5ms (i.e., half the time we observed for our experiments). For this case values of t_c and t_s are set to 2ms while the value of the last-level cache miss threshold remains unchanged at 20K. The third case, *ANVIL-light*, considers a situation where the 110K DRAM row accesses are spread out across a refresh period of 64ms (i.e., half the number of accesses purposely spread out maximally). For this case values of t_s and t_c are set to 6ms, and the last-level cache miss threshold is halved to 10K. As seen in Figure 4, ANVIL has room to grow if future rowhammer attacks become more aggressive. Overheads do grow to detect these more nimble attacks, but only slightly. Decreasing the last-level miss sample period to 2ms has the larger performance impact, which is expected as the sampling overheads are experienced continuously.

Table 5 shows false positive refresh rates due to false positives for *ANVIL-light* and *ANVIL-heavy*. Though both configurations show an increase in false positive rates than *ANVIL-baseline*, they do not incur significant overheads.

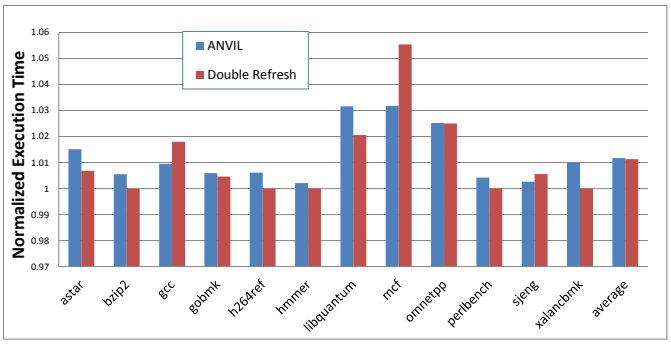


Figure 3: ANVIL's Impact on Non-Malicious Programs: The Figure shows execution times for selected benchmarks on ANVIL-enabled system and a system with doubled DRAM refresh rate. The values shown are normalized to execution time without ANVIL and at a single refresh period.

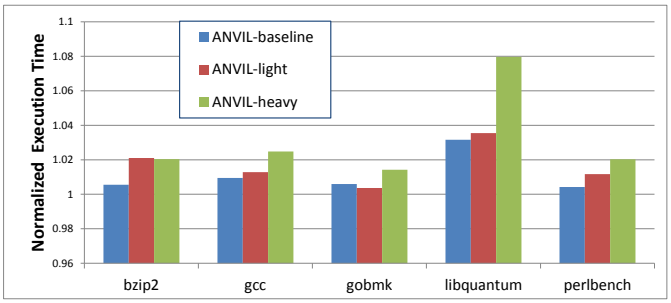


Figure 4: Sensitivity of Execution Overheads to Potential Future Attacks: The figure compares normalized execution times for selected benchmarks running on ANVIL-enabled system with three configurations. *ANVIL-Heavy* is configured to have the highest sampling rate while *ANVIL-Light* has the lowest last-level cache miss threshold.

5. Related Work

Previous works have studied exploitation and prevention of the rowhammer vulnerability. In this section, we detail currently known attacks and proposed mitigations.

5.1 Rowhammer Vulnerability and Its Exploitation

Even if the rowhammer vulnerability on modern DRAMs has been known by manufacturers since at least 2012 [24], the first detailed experimental study was published in 2014 by Yoongu Kim, et. al [24]. Their study shows that bits in a DRAM row (the victim row) can be flipped by repeatedly accessing adjacent rows in the same bank (the aggressor rows). The authors used x86's CLFLUSH instruction to bypass the cache and enable frequent references directly to DRAM.

Leveraging the early attack demonstrations, Seaborn and Dullien [28] demonstrated two security exploits that take advantage of rowhammer-induced bit flips. Their first attack bypasses Google's Native Client (NaCl) sandboxing system. NaCl is a software sandbox, integrated with the Chrome browser, that allows secure execution of untrusted client side applications and plug-ins. NaCl works by carefully scanning code for illegal code operations at load time (e.g., system calls or arbitrary indirect jumps), and by funneling all I/O operations through a security analyzer. The NaCl rowhammer attack works by having a securely loaded NaCl application hammer its own code segment until an illegal arbitrary code jump sequence is formed, then the application jumps to the middle of an instruction where illegal operations can be formed from validated code. Note that the attack is changing code that has been verified and deemed safe. Since the instructions are modified at the hardware level, the sandbox will not be aware of any of these changes. The authors' current proof-of-concept implementation can take advantage of 13% of the possible bit flips within an instruction.

Seaborn and Dullien's second attack takes advantage of the bit flips to bypass the memory page protection mechanism of a Linux system running on x86-64 [28]. The attack works by filling physical memory with page tables for a single process, by repeatedly mmap()'ing a file into its memory. This repeated file mapping sprays the memory with page table entries (PTEs), that are used to translate the newly mmap()'ed virtual addresses. By rowhammering the memory with page tables, there is a non-trivial probability that a PTE will be changed to point to a physical page containing a page table, thereby giving the application access to its own page tables. This will give the attacker full R/W permission to a page table entry, which in effect results in access to all of physical memory.

Even if all of the exploits mentioned above rely on CLFLUSH instruction in x86, the attack we presented in Section 2 demonstrates how rowhammer attacks can be launched without the use of any cache line flush instruction. A concurrent effort similarly demonstrated a CLFLUSH-free rowhammer attack [8]. This attack, which was disclosed two months after our attack, is similar in that it uses cache manipulation to avoid CLFLUSH instruction, but unlike our attack, it is implemented in Javascript. This earlier effort is only an attack and does not detail protections like we describe in this work.

5.2 Rowhammer Mitigation

In this section, we detail both software and hardware techniques that have been proposed and deployed to protect against data corruption and security exploits due to rowhammering.

5.2.1 Protections for Legacy Systems

Software Patches: To date, two open source projects have released patches in response to the security vulnerabilities explained above. Google's NaCl sandbox was patched to disallow the use of CLFLUSH instruction by applications running inside it. The attack mechanism we present in Section 2 defeats this protection and enables malicious applications to effectively hammer rows without using CLFLUSH (or any other explicit cache flush instructions).

Benchmark	Refreshes/sec (ANVIL-light)	Refreshes/sec (ANVIL-heavy)
bzip2	1.61	1.09
gcc	7.12	1.88
gobmk	0.28	0.84
libquantum	0.13	0.08
perlbench	0.06	0.00

Table 5: Rate of False Positive Refreshes for *ANVIL-Heavy* and *ANVIL-Light*: The table shows false positive rates for selected SPEC2006 integer benchmarks while running under two ANVIL configurations. *ANVIL-Heavy* has a relatively small sampling period which reduces the probability of misses with high address locality on non-malicious applications. On the other hand, *ANVIL-light* allows more samples for a longer sampling period thus resulting in a relatively larger false positive rate.

Recently, the Linux kernel was updated to disallow the use of the pagemap interface from the user space, as a measure to make it more difficult to do double-sided rowhammering in Linux-based systems. This change prevents malicious applications from analyzing the physical address space to launch targeted attacks. However, this attack still leaves room for potential attacks that rely on side-channel information to make inferences about the physical memory layout. Furthermore, certain attacks such as the NaCl sandbox escape attack can be implemented by repeatedly picking two random addresses without having any knowledge of the physical address mapping.

Doubling Refresh Rate: Some vendors published BIOS updates that double DRAM refresh rates(i.e. halving the refresh interval from 64ms to 32ms) [13, 15, 18]. Doubling the refresh rate reduces the amount of time an attacker has to mount an attack, since the discharging of a rowhammer'ed bit must be completed within one refresh cycle. However, our empirical studies show that it is still possible to induce bit flips through double-sided hammering even when the refresh period is as low as 16ms. Further increases in refresh rates would have significant effects on system performance and energy consumption [24].

5.2.2 Protections for Future Systems

Currently available hardware-based reliability features are not capable of mitigating DRAM disturbance errors. Error Correcting Codes (ECC) protection, aside from being ex-

pensive, is only capable of repairing single-bit flips. Furthermore, ECC will turn the problem of bit-flips into denial of service if the system has to deal with machine check exceptions every time a flip is detected [28].

Due to the inability of current memory controllers and memory modules to deal with rowhammer attacks, multiple hardware enhancements have been proposed. The possibility of having an activation counter for each row in a DRAM module has been considered in literature [23, 24]. However, due to the high overhead of maintaining and updating per-row counters, other alternatives have been recommended.

Probabilistic row refreshing (e.g. PARA) has been proposed as an alternative to per-row counters [23, 24]. In this technique, when an activation command is sent to a row, a random number generator is used to decide if adjacent row has to be refreshed. Since requests to rows that are being hammered will be encountered very frequently, there is a high probability that it will trigger a refresh. Compared to ANVIL, PARA can have lower overhead but requires a modification to the memory controller, therefore it can not be deployed on existing systems.

Project Armor [25] introduces an extra buffer that will cache data from rows with repeated activation commands. By servicing requests to hammered rows from the extra buffer, Armor prevents rows from being accessed repeatedly.

Processor and memory manufacturers are also deploying products with capabilities to perform targeted row refreshes. The current LPDDR4 standard and recent DDR4 modules support targeted refresh of potential victim rows [19, 21]. Intel has published patents on memory controllers that support targeted row refresh [4]. The memory controllers are designed to identify repeated reads to a row. However, the actual physical placement of rows can differ among different manufacturers. Hence, the controller only transmits the row that is being repeatedly accessed, and the memory module is responsible for refreshing the victim rows based on its internal structure.

It is important to note that the mitigation techniques for existing systems (i.e., doubling refresh rate and removing access to CLFLUSH) are shown to be ineffective in this work. We are able to implement the rowhammer attack in a 32ms double-rate refresh cycle, and we can also rowhammer DRAMs without access to the CLFLUSH instruction. While newly proposed hardware enhancements can protect future systems from rowhammer attacks, a software solution is still necessary to protect current hardware. As such, in this paper we detail a low-cost software-based rowhammer detector that thwarts attacks with little performance impact. It is our claim that these protections are appropriate both for existing and future designs.

6. Conclusion

Securing computing systems is a critical design goal. In this paper, we systematically analyze a security vulnerability found in commodity DRAM chips referred to as rowhammer. Rowhammer attacks use the CLFLUSH instruction to accomplish hammering by bypassing processor caches and repeatedly accessing memory.

We demonstrate that existing mitigation techniques such as doubling refresh rates and disallowing CLFLUSH instructions are not sufficient—we show that it possible to rowhammer in as little as 15ms. We also provide the first CLFLUSH-free rowhammer attack that does not require special cache flushing instructions, thereby expanding the rowhammering attack surface.

As an alternative protection mechanism to rowhammering attacks, we design, implement and evaluate ANVIL, the first software-based defense that protects against all known rowhammer attacks. Our defense leverages the insight that rowhammer memory access patterns are fundamentally different from those of normal applications. Compared to prior approaches, ANVIL is more effective, has lower cost, is readily deployable and is adaptable due to its software-based approach. Experiments with a diverse set of benchmarks on a real system show that ANVIL has an average slowdown of 1% and less than 1% false detections, while protecting against all tested rowhammer attacks. We feel that these results show it is viable to protect current and future systems against rowhammer attacks.

Acknowledgments

We extend special thanks to Mark Seaborn of Google for his many valuable inputs to this work. This work was supported in part by C-FAR, one of the six STARnet Centers, sponsored by MARCO and DARPA.

References

[1] Program for Testing for the DRAM "rowhammer" Problem. `https://github.com/mseaborn/rowhammer-test`. Accessed: 2015-08-11.

[2] National Security Agency. TEMPEST: A Signal Problem. `https://www.nsa.gov/public_info/_files/cryptologic_spectrum/tempest.pdf`. Accessed: 2015-08-11.

[3] JEDEC Solid State Technology Association. DDR3 SDRAM Specification, 2010.

[4] K. Bains, J.B. Halbert, C.P. Mozak, T.Z. Schoenborn, and Z. Greenfield. Row Hammer Refresh Command, 2014.

[5] Ishwar Bhati, Mu-Tien Chang, Zeshan Chishti, Shih-Lien Lu, and Bruce Jacob. DRAM Refresh Mechanisms, Penalties, and Trade-Offs. In *IEEE Transactions on Computers, VOL. 64*, 2015.

[6] Paul J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. 2007.

[7] D. Gruss. `https://twitter.com/lavados/status/685618703413698562`. Accessed: 2016-01-21.

[8] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *ArXiv e-prints*, July 2015.

[9] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[10] M. Hicks, M. Finnicum, S.T. King, M. Martin, and J.M. Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 159–172, May 2010.

[11] Rei-Fu Huang, Hao-Yu Yang, M.C. Chao, and Shih-Chin Lin. Alternate Hammering Test for Application-Specific DRAMs and an Industrial Case Study. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1012–1017, June 2012.

[12] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205, May 2013.

[13] Apple Inc. About the Security Content of Mac EFI Security Update 2015-001 . `https://support.apple.com/en-us/HT204934`. Accessed: 2015-08-11.

[14] CISCO Inc. Mitigations Available for the DRAM Row Hammer Vulnerability. `http://blogs.cisco.com/security/mitigations-available-for-the-dram-row-hammer-vulnerability`.

[15] HP Inc. HP Moonshot Component Pack Version 2015.05.0. `http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx`. Accessed: 2015-08-11.

[16] Intel Inc. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. September 2014.

[17] Intel Inc. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3 (3A, 3B & 3C): System Programming Guide*. June 2015.

[18] Lenovo Inc. Row Hammer Privilege Escalation Lenovo Security Advisory: LEN-2015-009. `https://support.lenovo.com/us/en/product_security/row_hammer`. Accessed: 2015-08-11.

[19] Micron Inc. *DDR4 SDRAM MT40A2G4, MT40A1G8, MT40A512M16 Data sheet*. 2015.

[20] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[21] JEDEC Solid State Technology Association . Low Power Double Data Rate 4 (LPDDR4), 2015.

[22] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '09, pages 50–57, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Dae-Hyun Kim, P.J. Nair, and M.K. Qureshi. Architectural support for mitigating row hammering in dram memories. *Computer Architecture Letters*, 14(1):9–12, Jan 2015.

[24] Yoongu Kim, R. Daly, J. Kim, C. Fallin, Ji Hye Lee, Donghyuk Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 361–372, June 2014.

[25] Mohsen Ghasempour, Mikel Lujan and Jim Garside. Armor: A Run-Time Memory Hot-Row Detector. `http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/index.html`. Accessed: 2015-08-11.

[26] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 48–59, New York, NY, USA, 2013. ACM.

[27] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1406–1418. ACM, 2015.

[28] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. March 2015.

[29] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.