ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks

Radhesh Krishnan Konoth, ***Vrije Universiteit Amsterdam;*** **Marco Oliverio,** ***University of Calabria/Vrije Universiteit Amsterdam;*** **Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi,** ***Vrije Universiteit Amsterdam***

https://www.usenix.org/conference/osdi18/presentation/konoth

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks

Radhesh Krishnan Konoth[†], Marco Oliverio[†§], Andrei Tatar[†], Dennis Andriesse[†], Herbert Bos[†], Cristiano Giuffrida[†] and Kaveh Razavi[†]

[†] *Vrije Universiteit Amsterdam, The Netherlands*
[§] *Università della Calabria, Italy*

Abstract

The Rowhammer vulnerability common to many modern DRAM chips allows attackers to trigger bit flips in a row of memory cells by accessing the adjacent rows at high frequencies. As a result, they are able to corrupt sensitive data structures (such as page tables, cryptographic keys, object pointers, or even instructions in a program), and circumvent all existing defenses.

This paper introduces ZebRAM, a novel and comprehensive software-level protection against Rowhammer. ZebRAM isolates every DRAM row that contains data with guard rows that absorb any Rowhammer-induced bit flips; the only known method to protect against all forms of Rowhammer. Rather than leaving guard rows unused, ZebRAM improves performance by using the guard rows as efficient, integrity-checked and optionally compressed swap space. ZebRAM requires no hardware modifications and builds on virtualization extensions in commodity processors to transparently control data placement in DRAM. Our evaluation shows that ZebRAM provides strong security guarantees while utilizing all available memory.

1 Introduction

The Rowhammer vulnerability, a defect in DRAM chips that allows attackers to flip bits in memory at locations to which they should not have access, has evolved from a mere curiosity to a serious and very practical attack vector for compromising PCs [6], VMs in clouds [28, 37], and mobile devices [13, 34]. Rowhammer allows attackers to flip bits in DRAM rows simply by repeatedly reading neighboring rows in rapid succession. Existing software-based defenses have proven ineffective against advanced Rowhammer attacks [4, 7], while hardware defenses are impractical to deploy in the billions of devices already in operation [23]. This paper introduces ZebRAM, a comprehensive software-based defense preventing all Rowhammer attacks by isolating every data row in memory with guard rows that absorb any bit flips that may occur.

Practical Rowhammer attacks Rowhammer attacks can target a variety of data structures, from page table entries [30, 34, 36, 37] to cryptographic keys [28], and from object pointers [6, 13, 32] to opcodes [14]. These target data structures may reside in the kernel [30, 34], other virtual machines [28], the same process address space [6, 13], and even on remote systems [32]. The attacks may originate in native code [30], JavaScript [6, 15], or from co-processors such as GPUs [13] and even DMA devices [32]. The objective of the attacker may be to escalate privileges [6, 34], weaken cryptographic keys [28], compromise remote systems [32], or simply lock down the processor in a denial-of-service attack [18].

Today's defenses are ineffective Existing hardware-based Rowhammer defenses fall into three categories: refresh rate boosting, target row refresh, and error correcting codes. Increasing the refresh rate of DRAM [21] makes it harder for attackers to leak sufficient charge from a row before the refresh occurs, but cannot prevent Rowhammer completely without unacceptable performance loss and power consumption increase. The target row refresh (TRR) defense, proposed in the LPDDR4 standard, uses hardware counters to monitor DRAM row accesses and refreshes specific DRAM rows suspected to be Rowhammer victims. However, TRR is not widely deployed; it is optional even in DDR4 [20]. Moreover, researchers still regularly observe bit flips in memory that is equipped with TRR [29]. As for error correcting codes (ECC), the first Rowhammer publication already argued that even ECC-protected DRAM is susceptible to Rowhammer attacks that flip multiple bits per memory word [21]. While this is complicating attacks, they do not stop fully stop them as shown by the recent ECCploit attack [10]. Furthermore, ECC memory is unavailable on most consumer devices.

Software defenses do not suffer from the same deployment issues as hardware defenses. These solutions can be categorized into primitive weakening, detection, and

isolation.

Primitive weakening makes some of the steps in Rowhammer attacks more difficult, for instance by making it harder to obtain physically contiguous uncached memory [30], or to create the cache eviction sets required to access DRAM in case the memory *is* cached. Research has already shown that these solutions do not fundamentally prevent Rowhammer [13].

Rowhammer detection uses heuristics to detect suspected attacks and refresh victim rows before they succumb to bit flips. For instance, ANVIL uses hardware performance counters to identify likely Rowhammer attacks [4]. Unfortunately, hardware performance counters are not available on all CPUs, and some Rowhammer attacks may not trigger unusual cache behavior or may originate from unmonitored devices [13].

A final, and potentially very powerful defense against Rowhammer is to *isolate* the memory of different security domains in memory with unused *guard rows* that absorb bit flips. For instance, CATT places a guard row between kernel and user memory to prevent Rowhammer attacks against the kernel from user space [7]. Unfortunately, CATT does not prevent Rowhammer attacks between user processes, let alone attacks *within* a process that aim to subvert cryptographic keys [28]. Moreover, the lines between security domains are often blurry, even in seemingly clear-cut cases such as the kernel and user-space, where the shared page cache provides ample opportunity to flip bits in sensitive memory areas and launch devastating attacks [14].

ZebRAM: isolate everything from everything Given the difficulty of correctly delineating security domains, the only *guaranteed* approach to prevent all forms of Rowhammer is to isolate *all* data rows with guard rows that absorb bit flips, rendering them harmless. The guard rows, however, break compatibility: buddy allocation schemes (and certain devices) require physically-contiguous memory regions. Furthermore, the drawback of this approach is obvious—sacrificing 50% of memory to guard rows is extremely costly. This paper introduces ZebRAM, a novel, comprehensive and compatible software protection against Rowhammer attacks that isolates everything from everything else *without* sacrificing memory consumed by guard rows. To preserve compatibility, ZebRAM remaps physical memory using existing CPU virtualization extensions. To utilize guard rows, ZebRAM implements an efficient, integrity-checked and optionally compressed swap space in memory.

As we show in Section 7, ZebRAM incurs an overhead of 5% on the SPEC CPU 2006 benchmarks. While ZebRAM remains expensive in the memory-intensive `redis` instance, our evaluation shows that ZebRAM's in-memory swap space significantly improves performance compared to our basic solution that leaves the guard rows unused, in some cases eliminating over half of the observed performance degradation. In practice, the recent Meltdown/Spectre vulnerabilities show that for a sufficiently serious threat, even expensive fixes are accepted [24]. First and foremost, however, this work investigates an extreme point in the design space of Rowhammer defenses: the first complete protection against all forms of Rowhammer, without sacrificing memory, at a cost that is a function of the workload.

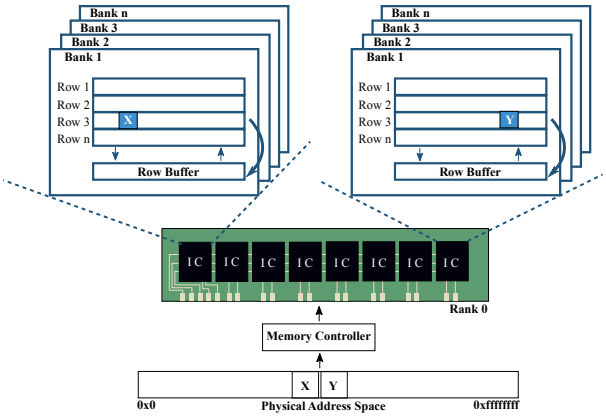

Figure 1: DRAM organization and example mapping of two consecutive addresses.

Contributions Our contributions are the followings:

- We describe ZebRAM, the first comprehensive software protection against all forms of Rowhammer.
- We introduce a novel technique to utilize guard rows as fast, memory-based swap space, significantly improving performance compared to solutions that leave guard rows unused.
- We implement ZebRAM and show that it achieves both practical performance and effective security in a variety of benchmark suites and workloads.
- ZebRAM is open source to support future work.

2 Background

This section discusses background on DRAM organization, the Rowhammer bug, and existing defenses.

2.1 DRAM Organization

We now discuss how DRAM chips are organized internally, which is important knowledge for launching an effective Rowhammer attack. Figure 1 illustrates the DRAM organization.

The most basic unit of DRAM storage is a *cell* that can hold a single bit of information. Each DRAM cell consists of two components: a capacitor and a transistor. The capacitor stores a bit by retaining electrical charge. Because this charge leaks away over time, the memory controller periodically (typically every 64 ms) reads each cell and rewrites it, restoring the charge on the capacitor. This process is known as *refreshing*.

DRAM cells are grouped into *rows* that are typically 1024 cells (or *columns*) wide. Memory accesses happen at row granularity. When a row is accessed, the contents of that row are put in a special buffer, called the *row buffer*, and the row is said to be *activated*. After the access, the activated row is written back (i.e., recharged) with the contents of the row buffer.

Multiple rows are stacked together to form *banks*, with multiple banks on a DRAM *integrated circuit (IC)* and a separate row buffer per bank. In turn, DRAM ICs are grouped into *ranks*. DRAM ICs are accessed in parallel; for example, in a DIMM that has eight ICs of 8 bits wide each, all eight ICs are accessed in parallel to form a 64 bit *memory word*.

To address a memory word within a DRAM rank, the system memory controller uses three addresses for the bank, row and column, respectively. Note that the mapping between a physical memory address and the corresponding rank-index, bank-index and row-index on the hardware module is nonlinear. Consequently, two consecutive physical memory addresses can be mapped to memory cells that are located on different ranks, banks, or rows (see Figure 1). As explained next, knowledge of the address mapping is vital to effective Rowhammer.

2.2 The Rowhammer Bug

As DRAM chips become denser, the capacitor charge reduces, allowing for increased DRAM capacity and lower energy consumption. Unfortunately, this increases the possibility of memory errors owing to the smaller difference in charge between a "0" bit and a "1" bit.

Research shows that it is possible to force memory errors in DDR3 memory by activating a row many times in quick succession, causing capacitors in neighboring *victim* rows to leak their charge before the memory controller has a chance to refresh them [21]. This rapid activation of memory rows to flip bits in neighboring rows is known as the *Rowhammer attack*. Subsequent research has shown that bit flips induced by Rowhammer are highly reproducible and can be exploited in a multitude of ways, including privilege escalation attacks and attacks against co-hosted VMs in cloud environments [6, 15, 27, 28, 30, 34, 37].

The original Rowhammer attack [30] is now known as *single-sided* Rowhammer. As Figure 2 shows, it uses many rapid-fire memory accesses in one *aggressor* row $k-1$ to induce bit flips in a neighboring victim row k. A newer variant called *double-sided* Rowhammer hammers rows $k-1$ and $k+1$ on both sides of the victim row k, increasing the likelihood of a bit flip (see Figure 2). Recent research shows that bit flips can also be induced by hammering only one memory address [14] (*one-location* hammering). Regardless of the type of hammering, Rowhammer can only induce bit flips on directly neighboring DRAM rows.

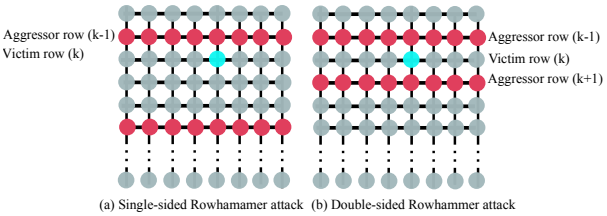

Figure 2: Flipping a bit in a neighboring DRAM row through single-sided (a) and double-sided (b) Rowhammer attacks.

In contrast to single-sided Rowhammer, the double-sided variant requires knowledge of the mapping of virtual and physical addresses to memory rows. Since DRAM manufacturers do not publish this information, this necessitates reverse engineering the DRAM organization.

2.3 Rowhammer Defenses

Research has produced both hardware- and software-based Rowhammer defenses.

The original hardware defense proposed by Kim et al. [21] doubles the refresh rate. Unfortunately, this has been proven insufficient to defend against Rowhammer [4]. Other hardware defenses include error-correcting DRAM chips (ECC memory), which can detect and correct a 1-bit error per ECC word (64-bit data). Unfortunately, ECC memory cannot correct multi-bit errors [3, 23] and is not readily available in consumer hardware. The new LPDDR4 standard [19] specifies two features which together defend against Rowhammer: *Target Row Refresh (TRR)* enables the memory controller to refresh rows adjacent to a certain row, and *Maximum Activation Count (MAC)* specifies a maximum row activation count before adjacent rows are refreshed. Despite these defenses, Gruss et al. [29] still report bit flips in TRR memory.

ANVIL [4], a software defense, uses Intel's performance monitoring unit (PMU) to detect physical addresses that cause many cache misses indicative of Rowhammer.[1] It then recharges suspected victim rows

[1] Rowhammer attacks repeatedly clear hammered rows from the CPU cache to ensure that they hammer DRAM memory, not the cache.

by accessing them. Unfortunately, the PMU does not accurately capture memory accesses through DMA, and not all CPUs feature PMUs. Moreover, the current implementation of ANVIL does not accurately take into account DRAM address mapping and has been reported to be ineffective because of it [31].

Another software-based defense, B-CATT [8], implements a bootloader extension to blacklist all the locations vulnerable to Rowhammer, thus wasting the memory. However, Gruss et al. [14] show that this approach is not practical as it may blacklist over 95% of memory locations; similar results were reported by Tatar et al. [31] showing DIMMs with 99+% vulnerable memory locations. In addition, in our experiments, we have observed different bit flip patterns over time for the same module, making B-CATT incomplete.

Yet another software-based defense called CATT [7] proposes an alternative memory allocator for the Linux kernel that isolates user and kernel space in physical memory, thus ensuring that user-space attackers cannot flip bits in kernel memory. However, CATT does not defend against attacks between user-space processes, and previous work [14] shows that CATT can be bypassed by flipping bits in the code of the `sudo` program.

3 Threat Model

The Rowhammer attacks found in prior research aim for privilege escalation [6, 27, 28, 30, 34, 37, 15], compromising co-hosted virtual machines [28, 37] or even attacks over the network [32]. Our approach, ZebRAM, addresses all these attacks through its principle of isolating memory rows from each other. Our prototype implementation of ZebRAM focuses only on virtual machines, stopping all of the aforementioned attacks launched from or at a victim virtual machine, assuming the hypervisor is trusted. We discuss possible alternative implementations (e.g., native) in Section 9.2.

4 Design

To build a comprehensive solution against Rowhammer attacks, we should consider Rowhammer's fault model: bit flips only happen in adjacent rows when a target row is hammered as shown in Figure 3. Given that any row can potentially be hammered by an attacker, all rows in the system can be abused. To protect against Rowhammer in software, we can follow two approaches: we either need to protect the entire memory against Rowhammer or we need to limit the rows that the attacker can access. Protecting the entire memory is not secure even in hardware [23, 34] and software attempts have so far been shown to be insecure [14]. Instead, we aim to design a

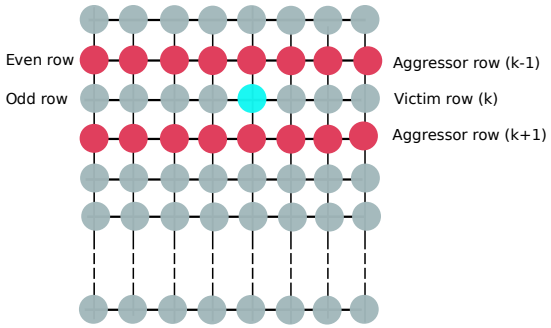

Figure 3: Hammering even-numbered rows can only induce bit flips in odd-numbered rows and vice versa.

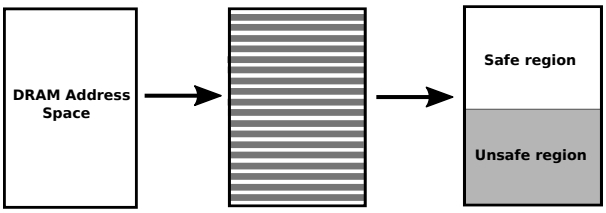

Figure 4: Splitting the memory into safe and unsafe regions using even and odd rows in a zebra pattern.

system where an attacker can only hammer a subset of rows directly.

Basic ZebRAM In order to make sure that Rowhammer bit flips cannot target any data, we should enforce the invariant that all *adjacent rows are unused.* This can be done by making sure that either all odd or all even rows are unused by the system. Assuming odd rows are unused, all even rows will create a *safe region* in memory; it is not possible for an attacker to flip bits in this safe regions simply because all the odd rows are inaccessible to the attacker. The attacker can, however, flip bits in the odd rows by hammering the even rows in the safe region. Hence, we call the odd rows the *unsafe region* in memory. Given that the unsafe region is unused, the attacker cannot flip bits in the data used by the system. This simple design with its zebra pattern shown in Figure 4 already stops all Rowhammer attacks. It however has an obvious downside: it wastes half of the memory that makes up the unsafe region. We address this problem later when we explain our complete ZebRAM design.

A more subtle downside in this design is incompatibility with the Buddy page allocation scheme used in commodity operating systems such as Linux. Buddy allocation requires contiguous regions of physical memory in order to operate efficiently and forcing the system not to use odd rows does not satisfy this requirement. Ideally, our design should utilize the unsafe region while providing (the illusion of) a contiguous physical address

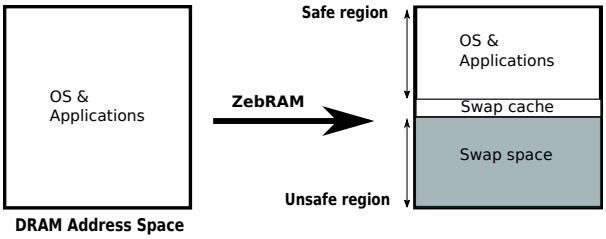

Figure 5: ZebRAM logically divides system memory into a safe region for normal use, a swap space made from the unsafe region, and a swap cache to protect the safe region from accesses made to the unsafe region.

space for efficient buddy allocation as shown on the right side of Figure 4. To address this downside, our design should provide a translation mechanism that creates a linear physical address space out of the safe region.

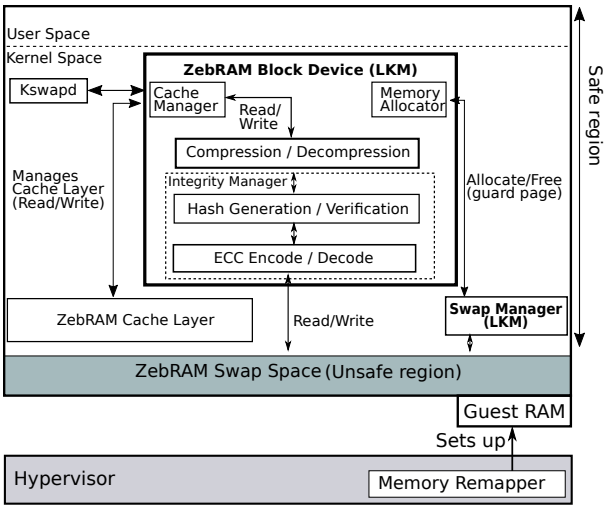

Figure 6: ZebRAM Components.

ZebRAM If we can find a way to securely use the unsafe region, then we can gain back the memory wasted in the basic ZebRAM design. We need to enforce two invariants if we want to make use of the unsafe region for storing data. First, we need to make sure that we properly handle potential bit flips in the unsafe region. Second, we need to ensure that accessing the unsafe region does not trigger bit flips in the safe region. Our proposed design, ZebRAM, shown in Figure 5 satisfies all these requirements. To handle bit flips in the unsafe region, ZebRAM performs software integrity checks and error correction whenever data in the unsafe region is accessed. To protect the safe region from accesses to the unsafe region, ZebRAM uses a cache in front of the unsafe region. This cache is allocated from the safe region and ZebRAM is free to choose its size and replacement policy in a way that protects the safe region. Finally, to provide backward-compatibility with memory management in commodity systems, ZebRAM can employ translation mechanisms provided by hardware (e.g., virtualization extensions in commodity processors) to translate even rows into a contiguous physical address space for the guest.

To maintain good performance, ZebRAM ensures that accesses to the safe region proceed without interposition. As mentioned earlier, this can potentially cause bit flips in the unsafe region. Hence, all accesses to the unsafe region should be interposed for bit flip detection and correction. To this end, ZebRAM exposes the unsafe region as a swap device to the protected operating system. With this design, ZebRAM reuses existing page replacement policies of the operating system to decide which memory pages should be evicted to the swap (i.e., unsafe region). Given that most operating systems use some form of Least Recently Used (LRU), the working set of the system remains in the safe region, preserving performance. Once the system needs to access a page from the unsafe region, the operating system selects a page from the safe region (e.g., based on LRU) and creates necessary meta data for bit flip detection (and/or correction) using the contents of the page and writes it to the unsafe region. At this point, the system can bring the page to the safe region from the unsafe region. But before that, it uses the previously calculated meta data to perform bit flip detection and correction. Note that the swap cache (for protecting the safe region) is essentially part of the safe region and is treated as such by ZebRAM.

Next, we discuss our implementation of ZebRAM's design before analyzing its security guarantees and evaluating its performance.

5 Implementation

In this section, we describe a prototype implementation of ZebRAM on top of the Linux kernel. Our prototype protects virtual machines against Rowhammer attacks and consists of the following four components: the *Memory Remapper*, the *Integrity Manager*, the *Swap Manager*, and the *Cache Manager*, as shown in Figure 6. Our prototype implements Memory Remapper in the hypervisor and the other three components in the guest OS. It is possible to implement all the components in the host to make ZebRAM guest-transparent. We discuss alternative implementations and their associated trade-offs in Section 9.2. We now discuss these components as implemented in our prototype.

5.1 ZebRAM Prototype Components

Memory Remapper implements the split of physical memory into a safe and unsafe region. One region con-

tains all the even-numbered rows, while the other contains all the odd-numbered rows. Note that because hardware vendors do not publish the mapping of physical addresses to DRAM addresses, we need to reverse engineer this mapping following the methodology established in prior work [26, 37, 31].

Because Rowhammer attacks only affect directly neighboring rows, a Rowhammer attack in one region can only incur bit flips in the other region, as shown in Figure 3. In addition, ZebRAM supports the conservative option of increasing the number of guard rows to defend against Rowhammer attacks that target a victim row not directly adjacent to the aggressor row. However, our experience with a large number of vulnerable DRAM modules shows that with the correct translation of memory pages to DRAM locations, bit flips trigger exclusively in rows adjacent to a row that is hammered.

Integrity Manager protects the integrity of the unsafe region. Our software design allows for a flexible choice for error detection and correction. For error correction, we use a commonly-used Single-Error Correction and Double-Error Detection (SECDED) code. As shown in recent work [10], SECDED and other similar BCH codes can still be exploited on DIMMs with large number of bit flips. Our database of Rowhammer bit flips from 14 vulnerable DIMMs [31] shows that only 0.00015% of all memory words with bit flips can bypass our SECDED code (found in 2 of the 14 vulnerable DIMMs) and 0.13% of them can cause a detectable corruption (found in 7 of the 14 vulnerable DIMMs). To provide strong detection guarantees, while providing correction possibilities, ZebRAM provides the possibility to mix SECDED with collision resistant hash functions such as SHA-256 at the cost of extra performance overhead.

Swap Manager uses the unsafe region to implement an efficient swap disk in memory, protected by the Integrity Manager and accessible only by the OS. Using the unsafe region as a swap space has the advantage that the OS will only access the slow, integrity-checked unsafe region when it runs out of fast safe memory. As with any swap disk, the OS uses efficient page replacement techniques to minimize access to it. To maximize utilization of the available memory, the Swap Manager also implements a compression engine that optionally compresses pages stored in the swap space.

Note that ZebRAM also supports configurations with a dedicated swap disk (such as a hard disk or SSD) in addition to the memory-based swap space. In this case, ZebRAM swap is prioritized above any other swap disks to maximize efficiency.

Cache Manager implements a fully associative cache that speeds up access to the swap space while simultaneously preventing Rowhammer attacks against safe rows by reducing the access frequency on memory rows in the unsafe region. The swap cache is faster than the swap disk because it is located in the safe region and does not require integrity checks or compression. Because attackers must clear the swap cache to be able to directly access rows in the unsafe region, the cache prevents attackers from efficiently hammering guard rows to induce bit flips in safe rows.

Because the cache layer sits in front of the swap space, pages swapped out by the OS are first stored in the cache, in uncompressed format. Only if the cache is full does the Cache Manager flush the *least-recently-added (LRA)* entry to the swap disk. The LRA strategy is important, because it ensures that attackers must clear the *entire* cache after every row access in the unsafe region.

5.2 Implementation Details

We implemented ZebRAM in C on an Intel Haswell machine running Ubuntu 16.04 with kernel v4.4 on top a Qemu-KVM v2.11 hypervisor. Next we provide further details on the implementation various components in the ZebRAM prototype.

Memory Remapper To efficiently partition memory into guard rows and safe rows, we use *Second Level Address Translation (SLAT)*, a hardware virtualization extension commonly available in commodity processors. To implement the Memory Remapper component, we patched Qemu-KVM's `mmap` function to expose the unsafe memory rows to the guest machine as a contiguous memory block starting at physical address `0x3ffe0000`. We use a translation library similar to that of Throwhammer [32] for assigning memory pages to odd and even rows in the Memory Remapper component.

Integrity Manager The Integrity Manager and Cache Manager are implemented as part of the ZebRAM block device, and comprise 369 and 192 LoC, respectively. The Integrity Manager uses `SHA-256` algorithm for error detection, implemented in mainline Linux, to hash swap pages, and keeps the hashes in a linear array stored in safe memory. Additionally, the Integrity Manager by default uses an ECC derived from the extended Hamming(63,57) code [16], expurgated to have a message size an integer multiple of bytes. The obtained ECC is a $[64, 56, 4]_2$ block code, providing single error correction and double error detection (SECDED) for each individual (64-bit) memory word—functionally on par with hardware SEC-DED implementations.

Swap Manager The Swap Manager is implemented as a Loadable Kernel Module (LKM) for the guest OS that maintains a stack containing the Page Frame Numbers (PFNs) of free pages in the swap space. It exposes the RAM-based swap disk as a readable and writable block device that we implemented by extending the `zram` compressed RAM block device commonly available in Linux distributions. We changed `zram`'s `zsmalloc` slab memory allocator to only use pages from the Swap Manager's stack of unsafe memory pages. To compress swap pages, we use the LZO algorithm also used by `zram` [1]. The Swap Manager LKM contains 456 LoC while our modifications to `zram` and `zsmalloc` comprise 437 LoC.

Cache Manager The Cache Manager implements the swap cache using a linear array to store cache entries and a radix tree that maps ZebRAM block device page indices to cache entries. By default, ZebRAM uses 2% of the safe region for the swap cache.

Guest Modifications The guest OS is unchanged except for a minor modification that uses Linux's boot memory allocator API (`alloc_bootmem_low_pages`) to reserve the unsafe memory block as swap space at boot time. Our changes to Qemu-KVM comprise 2.6K lines of code (LoC), while the changes to the guest OS comprise only 4 LoC. Furthermore, the Linux kernel may eagerly write dirty pages into the swap device based on its `swappiness` tunable. In ZebRAM, we use a swappiness of 10 instead of the default value of 60 to reduce the number of unnecessary writes to the unsafe region.

6 Security Evaluation

This section evaluates ZebRAM's effectiveness in defending against traditional Rowhammer exploits. Additionally, we show that ZebRAM successfully defends even against more advanced ZebRAM-aware Rowhammer exploits. We evaluated all attacks on a Haswell i7-4790 host machine with 16GB RAM running our ZebRAM-based Qemu-KVM hypervisor on Ubuntu 16.04 64-bit. The hypervisor runs a guest machine with 4GB RAM and Ubuntu 16.04 64-bit with kernel v4.4, containing all necessary ZebRAM patches and LKMs.

6.1 Traditional Rowhammer Exploits

Under ZebRAM's memory model, traditional Rowhammer exploits on system memory only hammer the safe region, and can therefore trigger bit flips only in the integrity-checked unsafe region by construction. We tested the most popular real-world Rowhammer exploit variants to confirm that ZebRAM correctly detects these integrity violations.

In particular, we ran the single-sided Rowhammer exploit published by Google's Project Zero,[2] as well as the one-location[3] and double-sided[4] exploits published by Gruss et al. on our testbed for a period of 24 hours. During this period the single-sided Rowhammer exploit induced two bit flips in the unsafe region, while the one-location and double-sided exploits failed to produce any bit flips. ZebRAM successfully detected and corrected all of the induced bit flips.

The double-sided Rowhammer exploit failed due to ZebRAM's changes in the DRAM geometry, alternating safe rows with unsafe rows. Conventional double-sided exploits attempt to exploit a victim row k by hammering the rows $k-1$ and $k+1$ below and above it, respectively. Under ZebRAM, this fails because the hammered rows are not really adjacent to the victim row, but remapped to be separated from it by unsafe rows. Unaware of ZebRAM, the exploit thinks otherwise based on the information gathered from the Linux' `pagemap`—due to the virtualization-based remapping layer—and essentially behaves like an unoptimized single-sided exploit. Fixing this requires a ZebRAM-aware exploit that hammers two consecutive rows in the safe region to induce a bit flip in the unsafe region. As described next, we developed such an exploit and tested ZebRAM's ability to thwart it.

6.2 ZebRAM-aware Exploits

To further demonstrate the effectiveness of ZebRAM, we developed a ZebRAM-aware double-sided Rowhammer exploit. This section explains how the exploit attempts to hammer both the safe and unsafe regions, showing that ZebRAM detects and corrects all the induced bit flips.

6.2.1 Attacking the Unsafe Region

To induce bit flips in the unsafe region (where the swap space is kept), we modified the double-sided Rowhammer exploit published by Gruss et al. [15] to hammer every pair of two consecutive DRAM rows in the safe region (assuming the attacker is armed with an ideal ZebRAM-aware memory layout oracle) and ran the exploit five times, each time for 6 hours. As Table 1 shows, the first exploit run induced a total of 4,702 bit flips in the swap space, with 4,698 occurrences of a single bit flip in a 64-bit data word and 2 occurrences of a double bit flip in a 64-bit word. ZebRAM successfully corrected all 4,698 single bit flips and detected the double bit flips. As shown

[2] `https://github.com/google/rowhammer-test`

[3] `https://github.com/IAIK/flipfloyd`

[4] `https://github.com/IAIK/rowhammerjs/tree/master/native`

Run no.	1 bit flip in 64 bits	2 bit flips in 64 bits	Total bit flips	ZebRAM detection performance	
				Detected bit flips	Corrected bit flips
1	4,698	2	4,702	4,702	4,698
2	5,132	0	5,132	5,132	5,132
3	2,790	0	2,790	2,790	2,790
4	4,216	1	4,218	4,218	4,216
5	3,554	0	3,554	3,554	3,554

Table 1: ZebRAM's effectiveness defending against a ZebRAM-aware Rowhammer exploit.

in Table 1, the other exploit runs produced similar results, with no bit flips going undetected. Note that ZebRAM can also detect more than two errors per 64-bit word due to its combined use of ECC and hashing, although our experiments produced no such cases.

6.2.2 Attacking the Safe Region

In addition to hammering safe rows, attackers may also attempt to hammer unsafe rows to induce bit flips in the safe region. To achieve this, an attacker must trigger rapid writes or reads of pages in the swap space. We modified the double-sided Rowhammer exploit to attempt this by opening the swap space with the *open* system call with the *O_DIRECT* flag, followed by repeated *preadv* system calls to directly read from the ZebRAM swap disk (bypassing the Linux page cache).

Because the swap disk only supports page-granular reads, the exploit must read an entire page on each access. Reading a ZebRAM swap page results in at least two memory copies; first to the kernel block I/O buffer, and next to user space. The exploit evicts the ZebRAM swap cache before each swap disk read to ensure that it accesses rows in the swap disk rather than in the cache (which is in the safe region). After each page read, we use a `clflush` instruction to evict the cacheline we use for hammering purposes. Note that this makes the exploit's job easier than it would be in a real attack scenario, where the exploit cannot use `clflush` because the attacker does not own the swap memory. A real attack would require walking an entire cache eviction buffer after each read from the swap disk.

We ran the exploit for 24 hours, during which time the exploit failed to trigger any bit flips. This demonstrates that the slow access frequency of the swap space—due to its page granularity, integrity checking, and the swap cache layer—successfully prevents Rowhammer attacks against the safe region.

To further verify the reliability of our approach, we re-tested our exploit with the swap disk's cache layer, compression engine, and integrity checking modules disabled, thus providing overly optimistic access speeds (and security guarantees) to the swap space for the Rowhammer exploit. Even in this scenario, the page-granular read enforcement of the swap device alone proved sufficient to prevent any bit flips. Our time measurements using `rdtsc` show that even in this optimistic scenario, memory dereferences in the swap space take 2,435 CPU cycles, as opposed to 200 CPU cycles in the safe region, removing any possibility of a successful Rowhammer attack against the safe region.

7 Performance Evaluation

This section measures ZebRAM's performance in different configurations compared to an unprotected system under varying workloads. We test several different kinds of applications, commonly considered for evaluation by existing systems security defenses. First, we test ZebRAM on the SPEC CPU2006 benchmark suite [17] to measure its performance for CPU-intensive applications. We also benchmark ZebRAM the popular `nginx` and Apache web servers, as well as the `redis` in-memory key-value store. Additionally, we present microbenchmark results to better understand the contributing factors to ZebRAM's overhead.

Testbed Similar to our security evaluation, we conduct our performance evaluation on a Haswell i7-4790 machine with 16GB RAM running Ubuntu 16.04 64-bit with our modified Qemu-KVM hypervisor. We run the ZebRAM modules and the benchmarked applications in an Ubuntu 16.04 guest VM with kernel v4.4 and 4GB of memory using a split of 2GB for the safe region and 2GB for the unsafe region. To establish a baseline, we use the same guest VM with an unmodified kernel and 4GB of memory. In the baseline measurements, the guest VM has direct access to all its memory, while in the ZebRAM performance measurements half of the memory is dedicated to the ZebRAM swap space. In all reported memory usage figures we include memory used by the Integrity Manager and Cache Manager components of ZebRAM. For our tests of server applications, we use a separate Skylake i7-6700K machine as the client. This machine has 16GB RAM and is linked to the ZebRAM machine via a 100Gbit/s link. We repeat all our experiments multiple times and observe marginal deviations across runs.

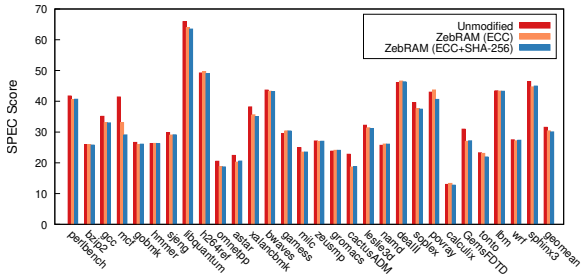

Figure 7: SPEC CPU 2006 performance results.

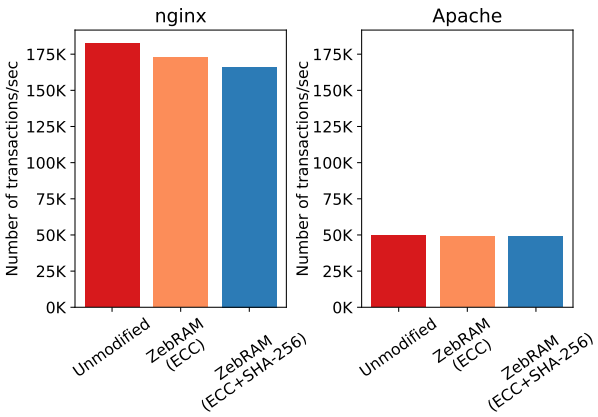

Figure 8: Nginx and Apache throughput at saturation.

SPEC 2006 We compare performance scores of the SPEC 2006 benchmark suite in three different setups: (i) unmodified, (ii) ZebRAM configured to use only ECC, and (iii) ZebRAM configured to use ECC and SHA-256. The ZebRAM (ECC) and ZebRAM (ECC and SHA-256) show a performance overhead over the unmodified baseline of 4% and 5%, respectively (see Figure 7). The reason behind this performance overhead is that as the ZebRAM splits the memory in a zebra pattern, the OS can no longer benefit from huge pages. Also, note that certain benchmarks, such as `mcf`, exhibits more than 5% overhead because they use ZebRAM's swap memory as their working set do not fit in the safe region.

Web servers We evaluate two popular web servers: nginx (1.10.3) and Apache (2.4.18). We configure the virtual machine to use 4 VCPUs. To generate load to the web servers we use the wrk2 [2] benchmarking tool, retrieving a default static HTML page of 240 characters. We set up nginx to use 4 workers, while we set up Apache with the prefork module, spawning a new worker process for every new connection. We also increase the maximum number of clients allowed by Apache from 150 to 500. We configured the wrk2 tool to use 32 parallel keep-alive connections across 8 threads. When measuring the throughput we check that CPU is saturated in the server VM. We discard the results of 3 warmup rounds, repeat a one-minute run 11 times, and report the median across runs. Figure 8 shows the throughput of ZebRAM under two different configurations: (i) ZebRAM configured to use only ECC, and (ii) ZebRAM configured to use ECC and SHA-256. Besides throughput, we want to measure ZebRAM's latency impact. We use wrk2 to throttle the load on the server (using the rate parameter) and report the 99th percentile latency as a function of the client request rate in Figure 9.

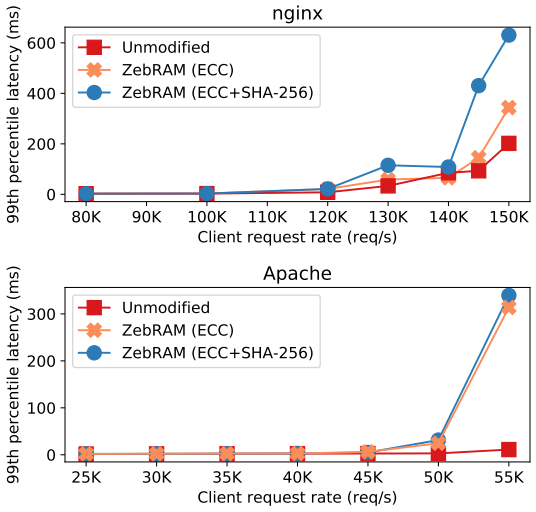

Figure 9: Nginx and Apache latency (99th percentile).

The baseline achieves 182k and 50k requests per second on Nginx and Apache respectively. The ZebRAM's first configuration (only ECC) reaches 172k and 49k while the second configuration reaches 166k and 49k.

Before saturation, the results show that ZebRAM imposes no overhead on the 99th percentile latency. After then, both configurations of ZebRAM show a similar trend with linearly higher 99th percentile response time.

Overall, ZebRAM's performance impact on both web servers and SPEC benchmarks is low and mostly due to the inability to efficiently use Linux' THP support. This is expected, since as long as the working set can comfortably fit in the safe region (e.g., around 400MB for our web server experiments) the unsafe memory management overhead is completely masked. We isolate and study such overhead in more detail in the following.

Microbenchmarks To drill down the overhead of each single feature of ZebRAM, we measure the latency of

swapping in a page from the ZebRAM device under different configurations. To measure the latency, we use a small binary that sequentially writes on every page of a large eviction buffer in a loop. This ensures that, between two accesses to the same page, we touch the entire buffer, evicting that page from memory. To be sure that Linux swaps in just one page for every access, we set the page-cluster configuration parameter to 0. In this experiment, two components interact with ZebRAM: our binary triggers swap-in events from the ZebRAM device while the `kswapd` kernel thread swaps pages to the ZebRAM device to free memory. The interaction between them is completely different if the binary uses exclusively loads to stress the memory. This is because the kernel would optimize out unnecessary flushes to swap and batch together TLB invalidations. Hence, we choose to focus on stores to study the performance in the worst-case scenario and because read-only workloads are less common than mixed workloads.

We reserve a core exclusively for the binary so that `kswapd` does not (directly) steal CPU cycles from it. We measure 1,000,000 accesses for each different configuration. Table 2 presents our results.We also run the binary in a loop and profile its execution with the `perf` Linux tool to measure the time spent in different functions. Due to function inlining, it is not always trivial to map a symbol to a particular feature. Nevertheless, `perf` can provide insights into the overhead at a fine granularity. In the first configuration, we disable the all features of ZebRAM and perform only memory copies into the ZebRAM device. As the copy operation is fast, the `perf` tool reports that just 4% percent of CPU cycles are spent copying. Interestingly, 47% of CPU cycles are spent serving Inter Process Interrupts from other cores. This is because, while we are swapping in, `kswapd` on another core is busy freeing memory. For this purpose, `kswapd` needs to unmap pages that are on their way to be swapped out from the process's page tables. This introduces TLB shootdowns (and IPIs) to invalidate other cores' TLB stale entries. It is important to notice that the faster we swap in pages, the faster `kswapd` needs to free memory. This unfortunately results in a negative feedback loop that represent one of the major sources of overhead when the large number of swap-in events continuously force `kswapd` to wake up.

Adding hashing (SHA-256) on top of the previous configuration shows an increase in latency, which is also reflected in the CPU cycles breakdown. The `perf` tool reports that 55% of CPU cycles are spent swapping in pages (copy + hashing), while serving IPIs accounts for 29%. Adding cache and compression on top of SHA-256 decreases the latency median and increases the 99th percentile. This is because, on a cache hit, the ZebRAM only needs to copy the page to userspace; however, on a cache miss, it has to verify the hash of the page and decompress the page too. The `perf` tool reports 42% of CPU cycles are spent in the decompression routine and 26% in serving IPI requests for other cores and less than 5% in hashing and copying. This confirms the presence of the swap-in/swap-out feedback loop under high memory pressure. Adding ECC marginally increases the latency, the `perf` tool reports similar CPU usage breakdown for the version without ECC.

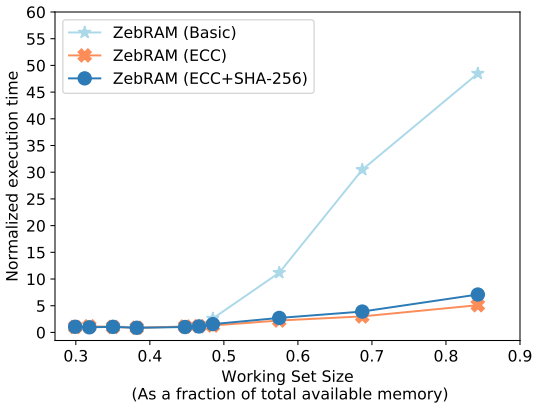

Figure 10: Redis throughput at saturation.

Larger working sets As expected, ZebRAM's overheads are mostly associated to swap-in/swap-out operations, which are masked when the working set can fit in the safe region but naturally become more prominent as we grow the working set. In this section, we want to evaluate the impact of supporting increasingly larger working sets compared to a more traditional swap implementation. For this purpose, we evaluate the performance of a key-value store in four different setups: (i) unmodifed system, (ii) the basic version of ZebRAM (iii) ZebRAM configured with ECC, and (iv) ZebRAM configured with ECC and SHA-256. The basic version of ZebRAM uses just one of the two domains in which ZebRAM splits the RAM and swaps to a fast SSD disk when the memory used by the OS does not fit into it. We use YCSB[11] to generate load and induce a target working set size against a redis (4.0.8) key-value store. We setup YCSB to use 1KB objects and perform a 90/10 read/write operations ratio. Each test runs for 20 seconds and, for each configuration, we discard the results of 3 warmup rounds and report the median across 11 runs. We configure YCSB to access the dataset key space uniformly and we measure the throughput at saturation for different data set sizes.

Figure 10 depicts the reported normalized execution time as a function of the working set size (in percentage compared to the total RAM size). As shown in the figure, when the working set size is small enough (e.g.,

Configuration	median (ns)	90th (ns)	99th (ns)
copy	2,362.0	4,107.0	8,167.0
SHA-256	13,552.0	14,209.0	17,092.0
cache + comp + SHA-256	8,633.0	13,191.0	18,678.0
cache + comp + SHA-256 + ECC	9,862.0	15,118.0	20,794.0

Table 2: Page swap-in latency from the ZebRAM device.

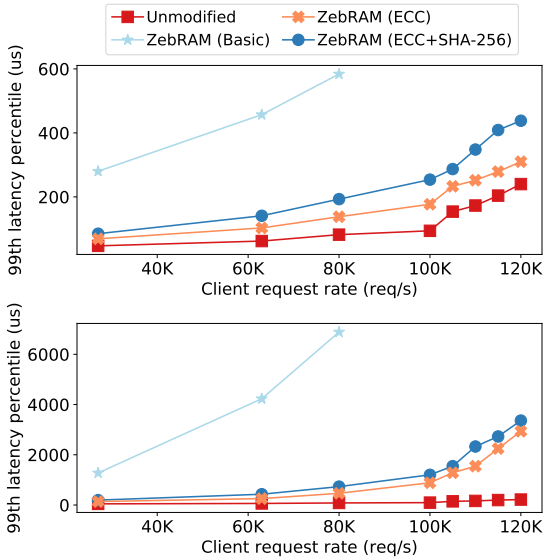

Figure 11: Redis latency (99th percentile). The working set size is 50% of RAM (top) and 70% of RAM (bottom).

44%) the OS hardly reclaims any memory, hence the unsafe region remains unutilized and the normalized execution time is only 1.08x for the basic version of ZebRAM while the normalized execution time is between 1.04x and 1.10x for all other configurations of ZebRAM. As we increase the working set size, the OS starts reclaiming pages and the normalized execution time increases accordingly. However, the increase is much more gentle for ZebRAM compared to the basic version of ZebRAM and the gap becomes more significant for larger working set sizes. For instance, for a fairly large working set size (e.g., 70%), ZebRAM (ECC) has 3.00x normalized execution time, and ZebRAM (ECC and SHA-256) has 3.90x, compared to the basic version of ZebRAM at 30.47x.

To study the impact of ZebRAM on latency, we fix the working set size to 50% and 70% of the total RAM and repeat the same experiment while varying the load on the server. Figure 11 presents our results for the 99th latency percentile. At 50%, results of (i) the ZebRAM configured with ECC, (ii) the ZebRAM configured with ECC and SHA-256, and (iii) baseline (unmodified) follow the same trend. The ZebRAM's first configuration (only ECC) reports a 99th latency percentile of 138us for client request rates below 80,000, compared to 584us for ZebRAM (basic). At 70%, the gap is again more prominent, with ZebRAM reporting a 99th latency percentile of 466us and ZebRAM (basic) reporting 6,887us.

Overall, ZebRAM can more gracefully reduce performance for larger working sets compared to a traditional (basic ZebRAM) swap implementation, thanks to its ability to use an in-memory cache and despite the integrity checks required to mitigate Rowhammer. As our experiments demonstrate, given a target performance budget, ZebRAM can support much larger working sets compared to the ZebRAM's basic implementation, while providing a strong defense against arbitrary Rowhammer attacks. This is unlike the basic ZebRAM implementation, which optimistically provides no protection against similar bit flip-based attacks. Unfortunately, such attacks, which have been long-known for DRAM [21], have recently started to target flash memory as well [9, 22].

8 Related work

This section summarizes related work on Rowhammer attacks and defenses.

Attacks In 2014, Kim et al. [21] were the first to show that it is possible to flip bits in DDR3 memory on x86 CPUs simply by accessing other parts of memory. Since then, many studies have demonstrated the effectiveness of Rowhammer as a real-world exploit in many systems.

The first practical Rowhammer-based privilege escalation attack, by Seaborn and Dullien [30], targeted the x86 architecture and DDR3 memory, hammering the memory rows by means of the native x86 `clflush` instruction that would flush the cache and allow high-frequency access to DRAM. By flipping bits in page table entries, the attack obtained access to privileged pages.

Not long after these earliest attacks, researchers greatly increased the threat of Rowhammer attacks by showing that is possible to launch them from JavaScript also, allowing attackers to gain arbitrary read/write access to the browser address space from a malicious web page [6, 15].

Moreover, newer attacks started flipping bits in memory areas other than page table entries, such as object pointers (to craft counterfeit objects [6]), opcodes [14], and even application-level sensitive data [28].

For instance, Flip Feng Shui demonstrated a new attack on VMs in cloud environments that flipped bits in RSA keys in victim VMs to make them easy to factorize, by massaging the physical memory of the co-located VMs to land the keys on a page that was hammerable by the attacker. Around the same time, other researchers independently also targeted RSA keys with Rowhammer but now for fault analysis [5]. Concurrently, also, Xiao et al. [37] presented another cross-VM attack that manipulates page table entries in Xen.

Where the attacks initially focused on PCs with DDR3 configurations, later research showed that ARM processors and DDR4 memory chips are also vulnerable [34]. While this opened the way for Rowhammer attacks on smartphones, the threat was narrower than on PCs, as the authors were not yet able to launch such attacks from JavaScript. This changed recently, when research described a new way to launch Rowhammer attacks from JavaScript on mobile phones and PC, by making use of the GPU. Hammering directly from the GPU by way of WebGL, the authors managed to compromise a modern smart phone browser in under two minutes. Moreover, this time the targeted data structures are doubles and pointers: by flipping a bit in the most significant bytes, the attack can turn pointers into doubles (making them readable) and doubles into pointers (yielding arbitrary read/write access).

All Rowhammer attacks until that point required local code execution. Recently, however, researchers demonstrated that even remote attacks on servers are possible [32], by sending network traffic over high-speed network to a victim process, using RDMA NICs. As the server that is receiving the network packets is using DMA to write to its memory, the remote attacker is able to flip bits in the server. By carefully manipulating the data in a key-value store, they show that it is possible to completely compromise the server process.

It should be clear that Rowhammer exploits have spread from a narrow and arcane threat to target two of the most popular architectures, in all common computing environments, different types of memory (and arguably flash [9]), while covering most common threat models (local privilege escalation, hosted JavaScript, and even remote attacks). ZebRAM protects against all of the above attacks.

Defenses Kim et al. [21] propose hardware changes to mitigate Rowhammer by increasing row refresh rates or using ECC. These defenses have proven insufficient [4] and infeasible to deploy on the required massive scale. The new LPDDR4 standard [19] specifies two features which together defend against Rowhammer: TRR and MAC. Despite these defenses, van der Veen et al. still report bit flips on a Google pixel phone with LPDDR4 memory [35] and Gruss et al. [29] report bit flips in TRR memory. While nobody has demonstrated Rowhammer attacks against ECC memory yet, the real problem with such hardware solutions is that most systems in use today do not have ECC, and replacing all DRAM in current devices is simply infeasible.

In order to protect from Rowhammer attacks, many vendors simply disabled features in their products to make life harder for attackers. For instance, Linux disabled unprivileged access to the *pagemap* [30], Microsoft disabled memory deduplication [12] to defend from the Dedup Est Machina attack [6], and Google disabled [33] the ION contiguous heap in response to the Drammer attack [34] on mobile ARM devices. Worryingly, not a single defence is currently deployed to protect from the recent GPU-based Rowhammer attack on mobile ARM devices (and PCs), even though it offers attackers a huge number of vulnerable devices.

Finally, researchers have proposed targeted software-based solutions against Rowhammer. ANVIL [4] relies on Intel's performance monitoring unit (PMU) to detect and refresh likely Rowhammer victim rows. An improved version of ANVIL requires specialized Intel PMUs with a fine-grained physical to DRAM address translation. Unfortunately, Intel's (and AMD's) PMUs do not capture precise address information when memory accesses bypass the CPU cache through DMA. Hence, this version of ANVIL is vulnerable to off-CPU Rowhammer attacks. Unlike ANVIL, ZebRAM is secure against off-CPU attacks, since device drivers transparently allocate memory from the safe region.

CATT [7] isolates (only) user and kernel space in physical memory so that user-space attackers cannot trigger bit flips in kernel memory. However, research [14] shows CATT to be bypassable by flipping opcode bits in the `sudo` program code. Moreover, CATT does not defend against attacks that target co-hosted VMs at all [7]. In contrast, ZebRAM protects against co-hosted VM attacks, attacks against the kernel, attacks between (and even within) user-space processes and attacks from co-processors such as GPUs.

Other recent software-based solutions have targeted specific Rowhammer attack variants. GuardION isolates DMA buffers to protect mobile devices against DMA-based Rowhammer attacks [36]. ALIS isolates RDMA buffers to protect RDMA-enabled systems against Throwhammer [32]. Finally, VUSion randomizes page frame allocation to protect memory deduplication-enabled systems against Flip Feng Shui [25].

9 Discussion

This section discusses feature and performance tradeoffs between our ZebRAM prototype and alternative ZebRAM implementations.

9.1 Prototype

Because the ZebRAM prototype relies on the hypervisor to implement safe/unsafe memory separation, and on a cooperating guest kernel for swap management, both host and guest need modifications. In addition, the guest physical address space will map highly non-contiguously to the host address space, preventing the use of huge pages. The guest modifications, however, are small and self-contained, do not touch the core memory management implementation and are therefore highly compatible with mainline and third party LKMs.

9.2 Alternative Implementations

In addition to our implementation presented in Section 5, several alternative ZebRAM implementations are possible. Here, we compare our ZebRAM implementation to alternative hardware-based, OS-based, and guest-transparent virtualization-based implementations.

Hardware-based Implementing ZebRAM at the hardware level would require a physical-to-DRAM address mapping where sets of odd and even rows are mapped to convenient physical address ranges, for instance an even lower-half and an odd upper-half. This can be achieved with by a fully programmable memory controller, or implemented as a configurable feature in existing designs. With such a mapping in place, the OS can trivially separate memory into safe and unsafe regions. In this model, the Swap Manager, Cache Manager and Integrity Manager are implemented as LKMs just as in the implementation from Section 5. In contrast to other implementations, a hardware implementation requires no hypervisor, allows the OS to make use of (transparent) huge pages and requires minimal modifications to the memory management subsystem. While a hardware-supported ZebRAM implementation has obvious performance benefits, it is currently infeasible to implement because memory controllers lack the required features.

OS-based Our current ZebRAM prototype implements the Memory Remapper as part of a hypervisor. Alternatively, the Memory Remapper can be implemented as part of the bootloader, using Linux' boot memory allocator to reserve the unsafe region for use as swap space. While this solution obviates the use of a hypervisor, it also results in a non-contiguous physical address space that precludes the use of huge pages and breaks DMA in older devices. In addition, it is likely that this approach requires invasive changes to the memory management subsystem due to the very fragmented physical address space.

Transparent Virtualization-based While our current ZebRAM implementation requires minor changes to the guest OS, it is also possible to implement a virtualization-based variant of ZebRAM that is completely transparent to the guest. This entails implementing the ZebRAM swap disk device in the host and then exposing the disk to the guest OS as a normal block device to which it can swap out. The drawback of this approach is that it degrades performance by having the hypervisor interposed between the guest OS and unsafe memory for each access to the swap device, a problem which does not occur in our current implementation. The clear advantage to this approach is that it is completely guest-agnostic: guest kernels other than Linux, including legacy and proprietary ones are equally well protected, enabling existing VM deployments to be near-seamlessly transitioned over to a Rowhammer-safe environment.

10 Conclusion

We have introduced ZebRAM, the first comprehensive software defense against all forms of Rowhammer. ZebRAM uses guard rows to isolate all memory rows containing user or kernel data, protecting these from Rowhammer-induced bit flips. Moreover, ZebRAM implements an efficient integrity-checked memory-based swap disk to utilize the memory sacrificed to the guard rows. Our evaluation shows ZebRAM to be a strong defense able to use all available memory at a cost that is a function of the workload. To aid future work, we release ZebRAM as open source.

Acknowledgements

We would like to thank our shepherd, Xi Wang, and the anonymous reviewers for their valuable feedback. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and the UNICORE project, by the MALPAY project, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", NWO 016.Veni.192.262, and NWO 629.002.204 "Parallax". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] LZO. http://www.oberhumer.com/opensource/lzo/, Retrieved 09.09.2018.

[2] WRK2 - a HTTP Benchmarking Tool. https://github.com/giltene/wrk2, Retrieved 09.09.2018.

[3] AICHINGER, B. DDR Memory Errors caused by Row Hammer. HPEC'15.

[4] AWEKE, Z. B., YITBAREK, S. F., QIAO, R., DAS, R., HICKS, M., OREN, Y., AND AUSTIN, T. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. ASPLOS'16.

[5] BHATTACHARYA, S., AND MUKHOPADHYAY, D. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. CHESS'16.

[6] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. S&P'16.

[7] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., AND SADEGHI, A.-R. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. SEC'17.

[8] BRASSER, F., DAVI, L., GENS, D., LIEBCHEN, C., AND SADEGHI, A.-R. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. *arXiv preprint arXiv:1611.08396* (2016).

[9] CAI, Y., GHOSE, S., LUO, Y., MAI, K., MUTLU, O., AND HARATSCH, E. F. Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques. HPCA '17.

[10] COJOCAR, L., RAZAVI, K., GIUFFRIDA, C., AND BOS, H. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. S&P '19.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. SoCC'10.

[12] CVE-2016-3272. Microsoft Security Bulletin MS16-092 - Important. *https://technet.microsoft.com/en-us/library/security/ms16-092.aspx* (2016).

[13] FRIGO, P., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. S&P'18.

[14] GRUSS, D., LIPP, M., SCHWARZ, M., GENKIN, D., JUFFINGER, J., OCONNELL, S., SCHOECHL, W., , AND YAROM, Y. Another Flip in the Wall of Rowhammer Defenses. *arXiv preprint arXiv:1710.00551* (2017).

[15] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16.

[16] HAMMING, R. W. Error detecting and error correcting codes. *Bell Labs Technical Journal 29*, 2 (1950), 147–160.

[17] HENNING, J. L. SPEC CPU2006 memory footprint. ACM SIGARCH Computer Architecture'07.

[18] JANG, Y., LEE, J., LEE, S., AND KIM, T. Sgx-bomb: Locking down the processor via rowhammer attack. SysTEX'17.

[19] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. Low Power Double Data 4 (LPDDR4). *JESD209-4A* (2015).

[20] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. DDR4 SDRAM Specification. *JESD79-4B* (2017).

[21] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTU, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14.

[22] KURMUS, A., IOANNOU, N., PAPANDREOU, N., AND PARNELL, T. From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks. WOOT'17.

[23] LANTEIGNE, M. *How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware* (2016).

[24] NEWMAN, L. H. The hidden toll of fixing meltdown and spectre. *WIRED* (2018).

[25] OLIVERIO, M., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Secure page fusion with vusion. SOSP'17.

[26] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16.

[27] QIAO, R., AND SEABORN, M. A New Approach for Rowhammer Attacks. HOST'16.

[28] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip Feng Shui: Hammering a Needle in the Software Stack. SEC'16.

[29] SCHWARZ, M., GRUSS, D., AND LIPP, M. Another Flip in the Row. BHUS'18. https://gruss.cc/files/us-18-Gruss-Another-Flip-In-The-Row.pdf Retrieved 09.09.2018.

[30] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. BHUS'15.

[31] TATAR, A., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Defeating software mitigations against Rowhammer: A surgical precision hammer. RAID'18.

[32] TATAR, A., KRISHNAN, R., ATHANASOPOULOS, E., GIUFFRIDA, C., BOS, H., AND RAZAVI, K. Throwhammer: Rowhammer Attacks over the Network and Defenses. ATC'18.

[33] TJIN, P. android-7.1.0_r7 (Disable ION_HEAP_TYPE_SYSTEM_CONTIG). *https://android.googlesource.com/device/google/marlin-kernel/+/android-7.1.0_r7* (2016).

[34] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. CCS'16.

[35] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. http://vvdveen.com/publications/drammer.slides.pdf, Retrieved 09.09.2018.

[36] VAN DER VEEN, V., LINDORFER, M., FRATANTONIO, Y., PILLAI, H. P., VIGNA, G., KRUEGEL, C., BOS, H., AND RAZAVI, K. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. DIMVA'18.

[37] XIAO, Y., ZHANG, X., ZHANG, Y., AND TEODORESCU, R. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. SEC'16.