Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations

Kevin Loughlin
kevlough@umich.edu
University of Michigan

Stefan Saroiu
ssaroiu@microsoft.com
Microsoft

Alec Wolman
alecw@microsoft.com
Microsoft

Baris Kasikci
barisk@umich.edu
University of Michigan

ABSTRACT

Rowhammer attacks exploit electromagnetic interference among nearby DRAM cells to flip bits, corrupting data and altering system behavior. Unfortunately, DRAM vendors have opted for a blackbox approach to preventing these bit flips, exposing little information about in-DRAM mitigations. Despite vendor claims that their mitigations prevent Rowhammer, recent work bypasses these defenses to corrupt data. Further work shows that the Rowhammer problem is actually *worsening* in emerging DRAM and posits that system-level support is needed to produce adaptable and scalable defenses.

Accordingly, we argue that the systems community can and must drive a fundamental change in Rowhammer mitigation techniques. In the short term, cloud providers and CPU vendors must work together to supplement limited in-DRAM mitigations—ill-equipped to handle rising susceptibility—with their own mitigations. We propose novel hardware primitives in the CPU's integrated memory controller that would enable a variety of efficient software defenses, offering flexible safeguards against future attacks. In the long term, we assert that major consumers of DRAM must persuade DRAM vendors to provide precise information on their defenses, limitations, and necessary supplemental solutions.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Security in hardware**.

HotOS '21, June 1–3, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465295

KEYWORDS

Rowhammer, DRAM Disturbance, Security

ACM Reference Format:
Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. 2021. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *Workshop on Hot Topics in Operating Systems (HotOS '21), June 1–3, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465295

1 INTRODUCTION

DRAM is the most prominent main memory technology, attractive due to its high density and low cost. DRAM *cells* are organized in row-column arrays, accessed by first *activating* a row (i.e., connecting it to a buffer) and then reading from or writing to this buffer. Since cells leak charge over time, rows are periodically *refreshed* to retain their data.

Unfortunately, as DRAM density increases with successive module generations (desirable for cost and efficiency reasons), so too does the electromagnetic interference among physically-proximate DRAM rows. Ultimately, this rising interference increases *DRAM disturbances* [32], wherein bit values in nearby rows of DRAM are flipped .

Concerningly, Rowhammer attacks [12, 14, 15, 19, 20, 27, 28, 30, 32, 35, 40, 43, 45–47, 50–52, 58] show that certain memory access patterns can increase the frequency of DRAM disturbances. In particular, frequently activating one or more *aggressor* rows—prior to the scheduled refreshes of nearby *victim* rows—may cause bit flips in the victim rows.

These hardware-level bit flips manifest as system-level problems, with particularly troubling ramifications in multi-tenant computing environments (e.g., the cloud). For instance, one tenant may corrupt the data of another, leading to data loss or machine shutdown/failure. In other scenarios, flips of security-critical bits (e.g., page table permission bits [47]) can compromise an entire host.

To date, DRAM vendors have shown years of unwillingness (perhaps due to economic reasons) to provide a comprehensive solution to Rowhammer. Despite vendor claims

that their defenses prevent all Rowhammer attacks [38, 42], recent work [14, 15] demonstrates that Rowhammer exploits remain viable. Given the blackbox nature of vendor mitigations, system administrators are left largely powerless to prevent these attacks, just as they were the original exploits.

In fact, recent work [30] argues that optimal defenses should include *software* support. The authors show that proposed hardware mitigations [32, 37, 60] struggle to scale or cannot provide comprehensive protection given increasing DRAM density. Consistent with these findings, state-of-the-art follow-up defenses [44, 59] are limited by worsening performance overhead and a need for increasing SRAM or CAM area (i.e., relatively-expensive memory) as density increases.

However, existing software defenses [4, 7–9, 19, 26, 26, 34, 39, 52, 57, 62] cannot achieve compehensive and practical protection due to the lack of hardware-based Rowhammer management primitives. For instance, ANVIL [4] relies on information from performance counters that do not account for direct memory accesses (DMAs); this leaves the system vulnerable to DMA-based Rowhammer attacks [52, 56], a concerning threat surface for cloud providers.

The shortcomings of both hardware and software defenses highlight the need for a hardware-software co-design to mitigate Rowhammer. More specifically, current hardware defenses need software support to adapt and scale to emerging attacks, while software defenses need hardware assistance to effectively mitigate attacks.

Fortunately, despite years of incomplete and blackbox mitigations from DRAM vendors, *CPU* vendors can still provide hardware assistance for defenses. We argue that CPU vendors should add a new set of Rowhammer management primitives to the CPU's integrated memory controller. Compared to DRAM vendors, CPU vendors have shown a willingness to expose a relatively-high number of memory management features to programmers, including a variety of performance counters [23] and BIOS configuration parameters [24].

We motivate our proposed memory controller primitives with key insights about Rowhammer attacks and existing defenses, producing a novel taxonomy of mitigation approaches: *isolation-centric*, *frequency-centric*, and *refresh-centric*. We show that system admins (e.g., cloud providers) can use our primitives to produce scalable and adaptable software defenses according to this taxonomy. Finally, we conclude with a long-term outlook on how major consumers of DRAM can drive the changes in hardware-software co-design needed for a comprehensive solution to Rowhammer.

2 BACKGROUND

In this section, we provide background on DRAM and a novel taxonomy of Rowhammer defenses to understand our new hardware primitives and software defenses.

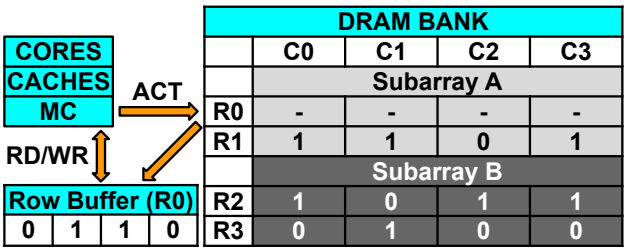

Figure 1: A simplified memory system. Memory controller *MC* activates row *R0* in subarray *A*, connecting it to the bank's row buffer for read/write commands.

2.1 DRAM+Rowhammer: A Crash Course

DRAM modules (e.g., SO-DIMMs in laptops and DIMMs in servers) consist of numerous *banks*, where each bank is a set of row-column *subarrays* of cells. A cell's charge distinguishes a particular bit as either 0 or 1.

Modules are programmed via a memory controller. For instance, the memory controller converts requests targeting CPU physical addresses into commands targeting *DDR logical* addresses (e.g., bank, row, column) according to a fixed mapping determined by BIOS settings [11].

To actually read/write the cells within a particular row, the memory controller must first issue an *activate* (ACT) command to the row containing the cells, thereby connecting this row to its encompassing bank's *row buffer* for processing. Such an ACT is shown in Fig. 1, where a module is depicted as a single bank with two 2×4 subarrays for simplicity. We note that each bank has its own row buffer, and a bank may contain hundreds of subarrays that share its row buffer.

At this point, the memory controller can issue read (RD) or write (WR) commands to cache line-sized column offsets within the activated row, until the row is *precharged* (i.e., deactivated); precharge (PRE) commands are typically issued so that another row in the same bank may occupy the row buffer for RDs/WRs. In line with processor cache behavior, RDs/WRs that hit in the row buffer are faster than those that necessitate an ACT before the data access can proceed.

Because DRAM cells leak their charges over time, the memory controller periodically issues *refresh* (REF) commands such that each row's cells are recharged (i.e., repaired) before losing their bit values. Typically, each 8 KB row must be refreshed within 64 milliseconds of its last refresh, where the module cycles through its rows during this *refresh interval*. We note that an ACT of a row also repairs the row as a side effect; thus, an ACT can essentially be used for row refresh.

Unfortunately, Rowhammer attacks [32] show that frequent ACTs of the same row(s)—induced by certain memory access patterns—can corrupt data in *physically-proximate* rows. In particular, alternating RDs or WRs to a set of *aggressor* rows within a single bank necessitate alternating ACTs

of these aggressors due to bank conflicts (i.e., row buffer contention). In turn, because of the electromagnetic interference among physically-proximate rows, these frequent ACTs may disturb the charges in nearby *victim* rows.

More precisely, each row can safely withstand a per-module *maximum activation count* (MAC) of ACTs within a refresh interval. However, if one or more aggressors surpass their MACs before a (potential) victim row is refreshed, the victim's data may be corrupted. Victim rows are those found up to b rows away from an aggressor, where b defines an aggressor's *blast radius* (which varies across DRAM technologies).

Notably, attackers with knowledge of DRAM address mappings can target *specific* data for corruption. While DRAM occasionally remaps two logically-adjacent rows to different internal locations [11], these remaps (and thus, internal adjacency) can be revealed via established methods. In particular, prior work [11, 15, 30, 34] uses the success or failure of Rowhammer attacks themselves—which require physically-proximate rows—to infer row adjacency.

2.2 Rowhammer Mitigations: A Taxonomy

At a high level, mounting a Rowhammer attack requires three conditions. First, at least one victim row must be located within the blast radius of at least one aggressor row. Second, one or more of the aggressor rows must be activated greater than MAC times within a refresh interval. Third, the victim row(s) must not themselves be refreshed before the aggressor(s) surpass the MAC.

Thus, Rowhammer defenses should eliminate one of these conditions, yielding our novel taxonomy of viable mitigations: *isolation-centric*, *frequency-centric*, and *refresh-centric*. We note that concurrent work [59] offers a similar taxonomy.

Isolation-Centric. Isolation-centric mitigations (e.g., [7, 8, 34, 57]) aim to physically isolate the rows from two different trust domains such that no cross-domain aggressor-victim relationships exist (e.g., a process cannot hammer another). For instance, ZebRAM [34] places b restricted-use "guard" rows between each potential aggressor-victim pair (where b is equal to the blast radius). Notably, isolation-centric mitigations typically do not prevent intra-domain DRAM disturbances (i.e., where an aggressor-victim relationship exists within a single domain, potentially inadvertently).

Frequency-Centric. Frequency-centric mitigations (e.g., [55, 59]) try to prevent the dangerously-frequent ACTs of aggressor rows needed to disturb nearby victim rows. For instance, BlockHammer [59] throttles (i.e., rate-limits) ACTs of aggressor rows according to a set of proposed memory controller counters, ensuring the number of ACTs to any row during a refresh interval stays below the MAC.

Refresh-Centric. Finally, refresh-centric mitigations (e.g., [4, 15, 32, 37, 48, 60, 62]) seek to refresh potential victim rows before they experience bit flips. More specifically, these defenses use a set of hardware and—in some cases—software mechanisms to identify potential victim rows. The defense systems then proactively refresh these victims before the corresponding aggressor row(s) reach their MACs.

3 D(R)AMIT, I CAN'T DO IT BY MYSELF!

Concerningly, recent work [30] demonstrates that the Rowhammer problem is worsening in successive DRAM generations. Specifically, as emerging DRAM technology nodes become denser, the electromagnetic interference among rows worsens, resulting in greater blast radii and orders-of-magnitude fewer ACTs (i.e., lower MACs) needed to induce charge leakages. Furthermore, lower MACs imply that a greater number of rows can act as aggressors (i.e., bypass the MACs).

Thus, while various hardware defenses have been proposed [15–17, 21, 29–32, 37, 38, 42, 44, 48, 53–55, 59, 60], recent work has concluded that even the state of the art among them either (a) cannot provide comprehensive protection or (b) require significant overheads to scale to denser DRAM technology [30]. Ultimately, **DRAM experts have identified hardware-software cooperative mitigations as a key avenue for addressing the scalability challenges of Rowhammer defenses going forward [30]**.

Unfortunately, sufficient hardware support for Rowhammer defenses is unlikely to come from DRAM vendors in the immediate future. First, DRAM vendors continue to expose little information about their Rowhammer mitigations and potential limitations, possibly due to a desire to maintain trade secrets about their DRAM design.

Second, even *today's* DRAM modules (let alone tomorrow's) are still vulnerable to Rowhammer [14, 15], despite vendors originally claiming the opposite [38, 42]. Prior work [14, 15] has shown that in-DRAM blackbox defenses (Target Row Refresh, or TRR) mitigate attacks by tracking a small number n of aggressor rows (where n varies by module and vendor), but can be bypassed with $> n$ aggressors. Given increasing numbers of aggressors, this a bleak observation.

4 CHANGING THE GAME WITH NEW PRIMITIVES

In contrast to hardware defenses, software defenses tend to require less invasive—if any—changes to memory system hardware, with the added benefit that software implementations allow for adapting to yet-unknown exploit patterns. Unfortunately, as we will show, software defenses presently lack sufficient support from hardware to provide comprehensive and practical protection against Rowhammer attacks.

However, while a solution to the Rowhammer problem would ideally include changes to DRAM, we argue that the

Class	MC Primitive	Corresponding Software Defense(s)	Optional DRAM Assistance
Isolation	Subarray-isolated interleaving	Subarray-aware memory allocation	Internal subarray mappings
Frequency	Precise ACT interrupt event	Aggressor remapping, cache line locking	-
Refresh	CPU REFRESH instruction	Efficient software refresh of victim rows	REF_NEIGHBORS command

Table 1: Summary of proposed memory controller (MC) primitives, corresponding software defense(s), and optional assistance from DRAM by mitigation class.

current limitations of software defenses can still be overcome via minor changes to the CPU's integrated memory controller. Compared to DRAM vendors, CPU vendors have demonstrated a willingness to expose a plethora of information and configuration parameters, including various memory controller performance counters [23] and memory configuration settings in the BIOS [24].

We therefore discuss three key limitations in the context of implementing isolation-, frequency-, and refresh-centric software mitigations on existing CPUs. We then describe how to address each of these limitations with simple extensions to the memory controller (summarized in Table 1), thereby forming the primitives necessary to produce efficient and practical isolation-, frequency-, and refresh-centric defenses in software. We plan to precisely evaluate the benefits/drawbacks of these defenses in future work (e.g., using the gem5 [6, 41] microarchitectural simulator); the RISC-V [3] ecosystem offers a viable alternate evaluation platform.

Notably, we largely assume that the host OS (e.g., the hypervisor) is trusted to implement and enforce these mitigations. We provide considerations for enclave memory (e.g., Intel SGX [13]) at the end of this section.

4.1 Isolation-Centric: Interleave It To Me

Problem: Interleaving Mixes Trust Domains. Implementing isolation-centric defenses in software—wherein aggressor rows from one trust domain d_1 (e.g., a process) cannot disturb victim rows from another domain d_2—requires that the host OS's page-level memory allocator can ensure data from d_1 is outside the blast radius b of data from d_2 in DRAM. Such isolation has been achieved by placing b "guard" rows between trust domains [34], or using a bank-aware memory allocator [7, 8, 61] to place d_1 and d_2 on different banks.

However, prior work [7, 8, 34, 61] fails to sufficiently account for the complexity and performance benefits of *memory interleaving* on modern systems. Because a bank can only process one command at a time, the memory system interleaves (i.e., spreads) consecutive cache lines from the CPU's physical address space across the system's numerous banks [63]. Such interleaving achieves bank-level parallelism when accessing physically-consecutive cache lines (i.e., consecutive lines can be accessed simultaneously for efficiency).

Unfortunately, such interleaving also distributes lines from different pages (i.e., potentially different trust domains) into the same bank. While interleaving can be disabled in the BIOS (allowing the memory allocator to isolate pages from different domains to specific banks), this eliminates the performance benefits of bank-level parallelism [10, 33, 36, 49, 63] (i.e., parallelism measured to reduce execution time for certain applications by over 18% [49]) and is thus an undesirable solution for production environments.

Primitive: Subarray-Isolated Interleaving. We argue that there is a middle ground, where existing interleaving (and its performance benefits) can remain fully-enabled, while pages from different trust domains can be isolated. Namely, rather than BIOS support for *bank*-aware memory allocation, we propose support for *subarray*-aware memory allocation.

Recall from Fig. 1 that a DRAM bank consists of a set of row-column subarrays. Notably, each subarray within a bank is electromagnetically-isolated from the others (i.e., they do not share bit lines) [10, 33], meaning data from different trust domains can be placed on different subarrays to prevent inter-domain aggressor-victim relationships. For instance, in a cloud environment, subarray isolation can be used to prevent inter-VM Rowhammer attacks.

Therefore, we posit that CPU vendors should provide a BIOS configuration option to enable *subarray-isolated interleaving* in each memory controller, as shown in Fig. 2. This option provides the host OS with two key features. First, cache lines from the same page will map to the same *subarray group* (i.e., set of specific subarrays across banks). Second, the host OS will be able to specify—either directly or indirectly—the trust domain of each page, such that the memory controller enforces that pages from the same trust domain map to the same subarray group.

From a software convenience standpoint, a direct specification would allow the host OS and memory controller to coordinate trust domains via an address space ID (ASID) tag per domain, akin to those already used in the TLB. However, if CPU vendors are unwilling to add hardware to track the mappings for a set of ASIDs, the memory controller already provides indirect support via its known set of CPU physical to DDR logical address mappings [11]; knowledge of these mappings has been used to allocate specific physical pages on specific banks [7, 8, 34, 61] and can similarly be used to map specific physical pages to specific subarray groups.

In an ideal world, DRAM vendors would also facilitate subarray-isolated interleaving by exposing DRAM-internal

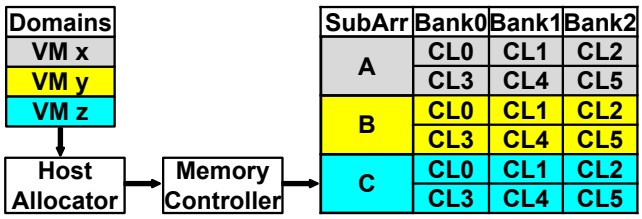

Figure 2: An example of subarray-isolated interleaving. The host memory allocator and memory controller cooperate to ensure that different trust domains (VMs x, y, and z) can reap the performance benefits of interleaving their consecutive cache lines CL0-CL5 across banks 0, 1 and 2. For security, the lines from each domain are restricted to per-domain, Rowhammer-isolated subarray(s): in this case, single subarray mappings of $x \rightarrow A$, $y \rightarrow B$, and $z \rightarrow C$.

subarray mappings in the DDR logical address space, akin to what is already done with bank mappings. However, even without this information, internal subarray mappings can be inferred via the same methods used to infer internal row adjacency/remappings (§2.1); that is, the success or failure of Rowhammer attacks within a bank can be used to infer subarray boundaries from software.

Notably, DRAM could still remap a row from its logical subarray to a different internal subarray, posing a threat to subarray isolation. Thankfully, the host OS can use the inferred DRAM subarray mappings—coupled with knowledge of CPU to DDR logical address mappings—to ensure that all rows are allocated with their *internal* subarrays.

4.2 Frequency-Centric: Context Welcome

Problem: ACT Interrupts Lack Context. Implementing frequency-centric defenses in software necessitates the ability to identify potential aggressor rows (i.e., rows experiencing frequent ACTs). While modern Intel memory controllers can count ACTs per channel [22]—as well as interrupt system software after a host OS-configurable number of ACTs—they do not provide any information about the specific row being activated, nor the specific RD/WR command causing the ACT. Thus, system software is powerless to determine which address(es) to take action on in response to an ACT interrupt.

Primitive: Precise ACT Interrupt Events. To support identifying potential aggressors, CPU vendors should augment the existing ACT_COUNT overflow event [22] to report the physical address causing the latest ACT. Doing so would allow the host OS to probabilistically identify and react to potential aggressor rows/hot cache lines within. We note that this address information would be consistent with that already reported by various cache events on Intel [23].

More precisely, recall from §2.1 that ACT commands are issued when the cache line needed for a RD/WR is not in a row buffer. Thus, we propose that upon overflow, the ACT_COUNT interrupt event should report the physical (cache line) address of the most recent RD/WR to have triggered an ACT of the row. The host OS can then reset the counter to an arbitrary value, probabilistically identifying aggressors according to specific MACs. By also including a degree of randomness in counter reset values, the host OS can prevent attackers from avoiding detection.

The host OS can use the address information provided by the interrupt to limit near-future ACTs of the encompassing row (i.e., within the refresh interval) in a variety of ways. For instance, software could implement a form of ACT wear-leveling by remapping and moving the row's data to a new physical location, either in DRAM or another storage device. To improve the efficiency of this data transfer, the CPU could support an uncore (i.e., off-core but on the CPU chip) move instruction using buffers in the memory controller, thereby avoiding the need to transfer data to and from on-core registers to relocate a line.

Alternatively, with the addition of cache line locking support (i.e., instructions or other mechanisms that temporarily pin a line to the processor cache, already available on many ARM processors [2, 18]), system software could use cache line locking for the duration of a refresh interval as a first line of defense. In particular, one or more ways in the LLC could be used for locked lines; data remapping and movement would then only be used as a fallback if the way(s) become full. Ultimately, such locking could improve access time for the line in question, prevent its continued use in ACT generation, and avoid a potentially costly data transfer.

4.3 Refresh-Centric: A Refreshing Take

Problem: SW Can't Directly Refresh Rows. Refresh-centric defenses must proactively refresh potential victim rows (i.e., rows that are near aggressor rows). Given methods for determining aggressors (§4.2) and row adjacency (§2.1), we focus on how to refresh potential victims.

Unfortunately, software has at best inefficient and potentially unreliable mechanisms to refresh rows. In particular, the REF command does not include a row address argument, meaning the memory controller/software cannot use it to refresh specific rows; instead, the ACT command (which takes a row address argument) must be used (§2.1). However, software still lacks the ability to directly issue ACTs to DRAM, with this decision left up to the memory controller. Thus, software defenses cannot directly refresh rows.

In fact, software can only potentially refresh a specific row via a series of convoluted memory instructions (e.g., loads/stores). To reach the memory controller, the load/store

must first miss in the cache, generally requiring software cache manipulation (e.g., a preceding flush instruction, if available) and strict ordering (e.g., memory fences) to reliably occur. At this point, the memory controller then converts the load/store into a set of RD/WR, ACT, and PRE commands; the specific set of commands issued depends on the state of the row buffers—information which is not directly exposed to software. Ultimately, this indirection introduces inefficiency and imprecision to defenses, especially in the presence of noise (e.g., memory operations from other cores/devices).

Primitive: A Refresh Instruction. To address software's lack of control over which rows are refreshed, we propose that the CPU should expose an instruction REFRESH whose effect is to refresh a specific row of DRAM. We liken this to Intel's patented mechanism for targeted refreshes via the memory controller [5]. However, in our case, the REFRESH instruction would be exposed in the ISA and—crucially—not require additional support from DRAM. Because the ACT side effect of the REFRESH instruction could be abused to carry out a Rowhammer attack, REFRESH should be a host-privileged instruction (i.e., only executable by the host OS).

The REFRESH instruction will take as an argument a virtual address VA—which maps to a particular DRAM row—and a bit AP indicating whether to auto-precharge the row after activation to prevent bank conflicts. The instruction will be implemented as follows. First, the TLB will translate VA to its corresponding physical address, which the memory controller will convert to a row address. Second, the memory controller will issue a PRE command to the row's encompassing bank to clear its row buffer. Third, the memory controller will issue an ACT command to the desired row, thereby effectively performing the refresh. Fourth, if the AP bit is set, the memory controller will issue another PRE command to the bank to clear the row buffer for subsequent accesses.

Finally, in an ideal world with support from DRAM, the DDR standard would include a REF_NEIGHBORS command, similar to that proposed in prior work [37, 44]. However, in addition to taking an aggressor row address as an argument, we propose that the command should also accept a blast radius b for adaptability to emerging threats. DRAM would then automatically refresh the potential victims of the provided aggressor row up to b rows away.

4.4 What About Enclave Memory?

The preceding discussion assumes that the host OS is trusted to implement and enforce the described mitigations. Notably, in certain enclave execution contexts (e.g., Intel SGX [13], Intel TDX [25], and AMD SEV [1]), only the enclave itself and hardware are trusted. Thus, these scenarios require additional considerations for software Rowhammer defenses.

Strictly-speaking, if enclave memory is checked for integrity upon access, then Rowhammer attacks can only cause system-wide denial-of-service (as opposed to arbitrary behavior changes stemming from data corruption). More specifically, upon a failed integrity check, the system will lock up and require a reset [27]. Because the host OS is untrusted and can already tamper with the integrity of enclave pages (i.e., without Rowhammer), such denial-of-service attacks are typically not considered in enclave threat models.

However, if enclave memory is *not* integrity-checked upon access, then the system must prevent (or at least, detect and gracefully shutdown upon) bit flips to ensure security. For isolation-centric defenses, the CPU could report the subarray(s) upon which the enclave resides in terms of physical addresses, such that the enclave may simply verify its virtual to physical address mappings (as is already done [13]). For frequency-centric defenses, the CPU could report ACT interrupts directly to enclaves, such that they might infer they are under attack and either (a) require a remap to a new location or (b) peacefully exit where permissible. Finally, for refresh-centric defenses, we posit that in the presence of subarray-isolated memory, an enclave could be permitted to issue REFRESH instructions to addresses mapped within its address space. We leave the exploration of these solutions to future work.

5 OUTLOOK: OPTIMAL FIXES

In this paper, we have described the limitations of and disconnect between existing hardware and software Rowhammer mitigations. To address these issues, we have proposed a variety of CPU-based primitives that would enable effective and practical hardware-software co-design defenses. Nonetheless, while a combination of CPU and software mitigations may prove more immediately-viable than in-DRAM support (e.g., in terms of scalability, adaptability, and economics), the root of the Rowhammer problem lies in DRAM.

Thus, in the long-term, we argue that optimal implementations of our defenses would include collaboration with both CPU *and* DRAM vendors. To achieve cooperation from DRAM vendors, cloud providers, CPU manufacturers, and other major consumers of DRAM must convince DRAM vendors to expose the details—and limitations—of their mitigations. Doing so would allow software, CPU, and in-DRAM mitigations to work in tandem to efficiently and scalably solve the Rowhammer problem once and for all.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedback, as well as Lucian Cojocar, Ishwar Agarwal, Daniel Berger, Tanj Bennett, Tim Cowles, Brett Dodds, Todd Farrell, Terry Grunzke, and Todd Merritt for many helpful discussions. Kevin Loughlin has been supported by an NSF Graduate Research Fellowship (award DGE 1256260).

REFERENCES

[1] AMD. 2020. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[2] ARM. 2018. CP15 c9, cache lockdown support. https://developer.arm.com/documentation/ddi0406/cb/Appendixes/ARMv4-and-ARMv5-Differences/System-Control-coprocessor--CP15-support/CP15-c9--cache-lockdown-support.

[3] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[4] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *ASPLOS*.

[5] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. 2015. Row hammer refresh command. US Patent 9,117,544.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH CAN* (2011).

[7] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks. In *Asia CCS*.

[8] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security*.

[9] A. Chakraborty, M. Alam, and D. Mukhopadhyay. 2019. Deep Learning Based Diagnostics for Rowhammer Protection of DRAM Chips. In *ATS*.

[10] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *HPCA*. https://doi.org/10.1109/HPCA.2016.7446095

[11] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. 2020. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *S & P*.

[12] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *S & P*.

[13] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016).

[14] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security*.

[15] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S & P*.

[16] Mohsen Ghasempour, Mikel Lujan, and Jim Garside. 2015. Armor: A run-time memory hot-row detector.

[17] Hector Gomez, Andres Amaya, and Elkim Roa. 2016. DRAM row-hammer attack reduction using dummy cells. In *NORCAS*.

[18] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *USENIX Security*.

[19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of Rowhammer defenses. In *S & P*.

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in javascript. In *DIMVA*.

[21] Hasan Hassan, Minesh Patel, Jeremie S Kim, A Giray Yaglikci, Nandita Vijaykumar, Nika Mansouri Ghiasi, Saugata Ghose, and Onur Mutlu. 2019. CROW: A low-cost substrate for improving DRAM performance, energy efficiency, and reliability. In *ISCA*.

[22] Intel. 2014. Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Unit Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/xeon-e5-2600-v2-uncore-manual.pdf.

[23] Intel. 2017. Intel 64 and IA32 Architectures Performance Monitoring Events. https://software.intel.com/sites/default/files/managed/8b/6e/335279_performance_monitoring_events_guide.pdf.

[24] Intel. 2019. Intel Server Board S2600 Family BIOS Setup User Guide. https://www.intel.com/content/dam/support/us/en/documents/server-products/Intel_Xeon_Processor_Scalable_Family_BIOS_User_Guide.pdf.

[25] Intel. 2020. Intel Trust Domain Extensions (Intel TDX). https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2018. MASCAT: Preventing Microarchitectural Attacks Before Distribution. In *CODASPY*.

[27] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *SysTEX*.

[28] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. 2019. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In *Asia CCS*.

[29] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2014. Architectural support for mitigating row hammering in DRAM memories. *CAL* (2014).

[30] Jeremie S Kim, Minesh Patel, A Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. 2020. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA*.

[31] M. Kim, J. Choi, H. Kim, and H. Lee. 2019. An Effective DRAM Address Remapping for Mitigating Rowhammer Errors. *IEEE Trans. Comput.* (2019).

[32] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*.

[33] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. 2012. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*.

[34] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *OSDI*.

[35] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading bits in memory without accessing them. In *S & P*.

[36] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N Patt. 2009. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO*.

[37] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *ISCA*.

[38] Jung-Bae Lee. 2014. Green Memory Solution. In *Samsung Electronics, Investor's Forum*.

[39] C. Li and J. Gaudiot. 2019. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *COMPSAC*.

[40] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. 2020. Nethammer: Inducing Rowhammer faults through network requests.

In *Euro S & P Workshop*.

[41] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[42] Micron. 2014. DDR4 SDRAM EDY4016A - 256Mb x 16. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf.

[43] Onur Mutlu. 2017. The RowHammer problem and other issues we may face as memory becomes denser. In *DATE*.

[44] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*.

[45] Rui Qiao and Mark Seaborn. 2016. A new approach for Rowhammer attacks. In *HOST*.

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

[47] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM Rowhammer bug to gain kernel privileges. *Black Hat* (2015). See also http://googleprojectzero.blogspot.co/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[48] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM stronger against row hammering. In *DAC*.

[49] Xulong Tang, Mahmut Kandemir, Praveen Yedlapalli, and Jagadish Kotra. 2016. Improving bank-level parallelism for irregular applications. In *MICRO*.

[50] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Defeating software mitigations against Rowhammer: a surgical precision hammer. In *RAID*.

[51] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *CCS*.

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM. In *DIMVA*.

[53] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. 2019. Detect DRAM disturbance error by using disturbance bin counters. *CAL* (2019).

[54] Yicheng Wang, Yang Liu, Peiyun Wu, and Zhao Zhang. 2019. Reinforce Memory Error Protection by Breaking DRAM Disturbance Correlation Within ECC Words. In *ICCD*.

[55] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. S. Kim, J. G. Luna, M. Sadrosadati, N. M. Ghiasi, and O. Mutlu. 2020. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *MICRO*. https://doi.org/10.1109/MICRO50266.2020.00036

[56] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. 2019. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *arXiv preprint arXiv:1912.11523* (2019).

[57] Xin-Chuan Wu, Timothy Sherwood, Frederic T Chong, and Yanjing Li. 2019. Protecting page tables from Rowhammer attacks using monotonic pointers in DRAM true-cells. In *ASPLOS*.

[58] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation. In *USENIX Security*.

[59] A. Giray Yağlikçi, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA 2021*.

[60] Jung Min You and Joon-Sung Yang. 2019. MRLoc: Mitigating Row-hammering based on memory Locality. In *DAC*.

[61] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*.

[62] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Nepal Surya, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. 2021. SoftTRR: Protect Page Tables Against RowHammer Attacks using Software-only Target Row Refresh. *arXiv preprint arXiv:2102.10269* (2021).

[63] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO*.