2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)

BreakHammer: Enhancing RowHammer Mitigations by Carefully Throttling Suspect Threads

Oğuzhan Canpolat[§†] A. Giray Yağlıkçı[§] Ataberk Olgun[§] Ismail Emir Yuksel[§]
Yahya Can Tuğrul[§†] Konstantinos Kanellopoulos[†] Oğuz Ergin[‡§†] Onur Mutlu[§]
[§]*ETH Zürich* [†]*TOBB University of Economics and Technology* [‡]*University of Sharjah*

RowHammer is a major read disturbance mechanism in DRAM where repeatedly accessing (hammering) a row of DRAM cells (DRAM row) induces bitflips in other physically nearby DRAM rows. RowHammer solutions perform preventive actions (e.g., refresh neighbor rows of the hammered row) that mitigate such bitflips to preserve memory isolation*, a fundamental building block of security and privacy in modern computing systems. However, preventive actions induce non-negligible memory request latency and system performance overheads as they interfere with memory requests. As shrinking technology node size over DRAM chip generations exacerbates RowHammer, the overheads of RowHammer solutions become prohibitively expensive. As a result, a malicious program can effectively hog the memory system and deny service to benign applications by causing many RowHammer-preventive actions.*

In this work, we tackle the performance overheads of RowHammer solutions by tracking and throttling the generators of memory accesses that trigger RowHammer solutions. To this end, we propose BreakHammer. BreakHammer 1) observes the time-consuming RowHammer-preventive actions of existing RowHammer mitigation mechanisms, 2) identifies hardware threads that trigger many of these actions, and 3) reduces the memory bandwidth usage of each identified thread. As such, BreakHammer significantly reduces the number of RowHammer-preventive actions performed, thereby improving 1) system performance and DRAM energy, and 2) reducing the maximum slowdown induced on a benign application, with near-zero area overhead. Our extensive evaluations demonstrate that BreakHammer effectively reduces the negative performance, energy, and fairness effects of eight RowHammer mitigation mechanisms. To foster further research we open-source our BreakHammer implementation and scripts at `https://github.com/CMU-SAFARI/BreakHammer`.

1. Introduction

To ensure system robustness (i.e., reliability, security, and safety), it is critical to maintain memory isolation: accessing a memory address should *not* cause unintended side-effects on data stored on other addresses. Unfortunately, with aggressive technology node scaling, dynamic random access memory (DRAM) [1], the prevalent main memory technology, suffers from increased read disturbance: accessing (reading) a DRAM cell degrades the data integrity of other physically close yet *unaccessed* DRAM cells by exacerbating the unaccessed cells' inherent charge leakage and consequently causing bitflips. *RowHammer* [2–5] is a prime example of such DRAM read disturbance phenomena where a DRAM row (i.e., victim row) can experience bitflips when a nearby DRAM row (i.e., aggressor row) is repeatedly opened (i.e., hammered) [2–70].

Many prior works demonstrate attacks on a wide range of systems where they exploit read disturbance to escalate privilege, leak private data, and manipulate critical application outputs [2–4, 6–54, 70–86]. Recent experimental studies [2–4, 37, 38, 62, 87] find that newer DRAM chip generations are more susceptible to read disturbance. For example, chips manufactured in 2018-2020 can experience RowHammer bitflips after an order of magnitude fewer row activations compared to the chips manufactured in 2012-2013 [62]. As RowHammer worsens, ensuring robust (i.e., reliable, secure, and safe) operation causes prohibitively large performance overheads [62, 88–90]. Although RowHammer mitigation mechanisms can mitigate RowHammer bitflips, they aggressively and sometimes excessively perform time- and energy-hungry operations to mitigate RowHammer (i.e., RowHammer-preventive actions) as fewer DRAM row activations can induce RowHammer bitflips in newer DRAM chips. In fact, adversarial memory access patterns can be crafted to trigger RowHammer mitigation mechanisms more frequently. Thus, RowHammer mitigation mechanisms provide data integrity at the cost of DRAM bandwidth availability by incurring large performance overheads. Therefore, reducing the performance overhead of such mechanisms is critical to provide data integrity without reducing DRAM bandwidth availability.

Goal. Our goal is to reduce the performance overhead of RowHammer mitigation mechanisms by carefully reducing the number of performed RowHammer-preventive actions *without* compromising system robustness.

Key Idea. Our key idea is to limit the dynamic memory request count of a hardware thread based on how frequently the thread triggers RowHammer-preventive actions.

Key Mechanism. We propose a new memory controller-based throttling mechanism, BreakHammer.[1] Cooperating with an existing memory controller-based or on-DRAM-die RowHammer mitigation mechanism (e.g., [2, 89, 92–97]) as we show, BreakHammer 1) observes the triggered RowHammer-preventive actions, 2) identifies hardware threads that trigger many RowHammer-preventive actions (we call these threads *suspect threads*), and 3) reduces the dynamic memory re-

[1]BreakHammer is analogous to breakwater, which is a structure that protects docks against tides, waves, and storm surges; and consequently reduces their maintenance costs [91]. Similarly, our mechanism, BreakHammer, protects the memory subsystem against hammering access patterns and, by doing so, reduces the overheads of RowHammer mitigation mechanisms.

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00072

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00072

quest count of each suspect thread. Meanwhile, the existing RowHammer mitigation mechanism operates as usual and maintains security guarantees against RowHammer attacks. As such, BreakHammer significantly reduces the number of necessary RowHammer-preventive actions by throttling the malicious threads' memory requests *without* compromising a RowHammer mitigation mechanism's security guarantees. To achieve this, BreakHammer divides execution timeline into equal periods that we call *throttling windows* and in each throttling window performs three key operations when a RowHammer-preventive action is triggered.

First, BreakHammer attributes a score (i.e., *RowHammer-preventive score*) to hardware threads that cause RowHammer-preventive actions that interfere with other memory accesses, e.g., blocking a DRAM bank due to refreshing victim rows. For this purpose, BreakHammer maintains a *RowHammer-preventive score counter* per hardware thread. To showcase BreakHammer's compatibility with various types of existing RowHammer mitigation mechanisms, we implement BreakHammer with five memory controller-based [2,89,92–94] and three on-DRAM-die [95–97] state-of-the-art RowHammer mitigation mechanisms that we summarize under five categories. 1) PARA [2], Graphene [89], Hydra [93], and TwiCe [98] perform RowHammer-preventive refresh operations, 2) AQUA [92] migrates the contents of DRAM rows to a quarantine area within DRAM, 3) REGA [95] modifies the DRAM chip with a secondary row buffer to refresh multiple rows in parallel, 4) *Periodic Refresh Management* (RFM) [99] requires the memory controller to issue refresh management (RFM) commands when a bank's activation count exceeds a threshold, and 5) *Per Row Activation Counting* (PRAC) [97] uses the *alert_n* signal to request a predetermined number of RFM commands when a row's activation count exceeds a threshold.

Second, BreakHammer identifies suspect threads using outlier analysis. To do so, BreakHammer marks a hardware thread as suspect if the thread's RowHammer-preventive score both 1) exceeds a threshold and 2) largely deviates from the average RowHammer-preventive score across all threads in the system. Third, BreakHammer reduces the memory bandwidth usage of each suspect thread until it is *not* identified as a suspect in a future throttling window. To this end, BreakHammer limits the number of *cache-miss buffers* (MSHRs [100]) that track the last level cache (LLC) misses a suspect thread can allocate.

Security. We mathematically evaluate the impact of a worst-case memory performance attack for BreakHammer, where the attacker operates near BreakHammer's outlier detection threshold *without* being detected as an outlier by BreakHammer. Our security analysis shows that BreakHammer enforces a theoretical upper bound for the RowHammer-preventive score an attacker can gather based on 1) the RowHammer-preventive score of benign threads and 2) the number of hardware threads used by the attacker. For example, we show that an attacker *cannot* trigger twice the RowHammer-preventive action count of concurrently running benign applications unless the attacker uses 90% of all hardware threads.

Hardware Implementation. We model BreakHammer's hardware design (RTL) in Chisel [101] and evaluate its latency and area overhead using Synopsys Design Compiler [102] (65nm process technology) and CACTI [103]. Our hardware complexity evaluation shows that BreakHammer can be implemented off the critical path of memory request scheduling because BreakHammer has a latency of 0.67ns, which is lower than *the minimum time window between two row activation commands targeting different banks* (t_{RRD}) (e.g., 2.5ns in DDR4 [104]) and incurs near-zero area overhead (i.e., $0.00042mm^2$), which consumes 0.0002% of a high-end Intel Xeon processor's chip area.

Key Results. We rigorously evaluate BreakHammer's impact on performance and DRAM energy using Ramulator 2.0 [105, 106], an open-source cycle-level simulator. We execute 1) 90 four-core workloads where one malicious application mounts a memory performance attack by triggering many RowHammer-preventive actions and 2) 90 four-core workloads where all applications are benign. We select a diverse set of benign workloads from SPEC CPU2006 [107], SPEC CPU2017 [108], TPC [109], MediaBench [110], and YCSB [111] benchmark suites. Our evaluation shows that BreakHammer significantly improves system performance and reduces DRAM energy (e.g., by 90.1% and 55.7%, respectively, on average across all tested workload mixes with a malicious application).

We make the following contributions:

- We show that state-of-the-art RowHammer mitigation mechanisms significantly reduce DRAM bandwidth availability.
- We introduce the idea that it is possible to 1) mount memory performance attacks by exploiting the RowHammer-preventive actions of RowHammer mitigation mechanisms and 2) reduce the performance overheads of RowHammer mitigation mechanisms by observing the time-consuming RowHammer-preventive actions and throttling the threads that trigger such actions.
- We propose BreakHammer, a memory controller-based low-cost mechanism that provides both memory controller-based and on-DRAM-die RowHammer mitigation mechanisms with throttling support to 1) reduce their performance and energy overheads and 2) prevent their actions from being exploited to reduce DRAM bandwidth availability.
- We rigorously evaluate BreakHammer and show that it significantly reduces the performance overhead of eight state-of-the-art RowHammer mitigation mechanisms without compromising robustness at near-zero area overhead.
- To foster further research in the design of more robust and high-performance systems, we open-source our implementations at `https://github.com/CMU-SAFARI/BreakHammer`.

2. Background

2.1. DRAM Organization and Operation

Organization. Fig. 1 shows the organization of a DRAM-based memory system. A memory channel connects the processor (CPU) to a set of DRAM chips. This set of DRAM chips forms one or more DRAM ranks where each rank operates in lockstep.

Each chip has multiple DRAM banks, where DRAM cells are organized as a two-dimensional array of DRAM rows and columns. A DRAM cell is connected to the row buffer via a wire called bitline. The DRAM cell stores one bit of data in the form of electrical charge in a capacitor. The access transistor, controlled by the wordline, connects the cell to the bitline.

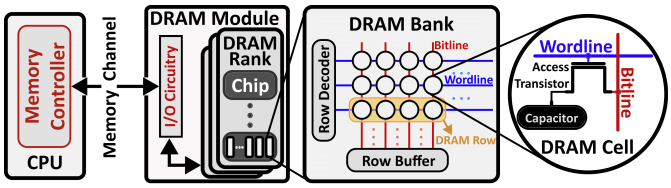

Figure 1: DRAM organization

Operation. DRAM cells are internally accessed at DRAM row granularity. To access a DRAM row, the memory controller issues a set of commands to DRAM over the memory channel. The memory controller sends an activate (ACT) command to activate a DRAM row, which asserts the corresponding wordline and loads the row data into the row buffer. Then, the memory controller can issue read (RD)/write (WR) commands to read from/write into the row buffer. Subsequent accesses to the same DRAM row cause a row buffer hit. Accessing a different row causes a row buffer conflict. Therefore, to access a different DRAM row, the memory controller must first close the open row by issuing a precharge (PRE) command and open the row that contains the accessed data.

DRAM Refresh. A DRAM cell is inherently leaky and loses its charge over time due to charge leakage in the access transistor and the storage capacitor. To maintain data integrity, the memory controller periodically refreshes each row every *refresh window* (t_{REFW}), which is $32ms$ for DDR5 [96] and $64ms$ for DDR4 [104] (under normal operating temperature ranges). To timely refresh all cells, the memory controller issues a refresh (REF) command every *refresh interval* (t_{REFI}), which is $3.9\mu s$ for DDR5 [96] and $7.8\mu s$ for DDR4 [104].

2.2. DRAM Read Disturbance

As DRAM manufacturing technology node size shrinks, interference between cells increases, causing circuit-level read disturbance mechanisms. Two prime examples of such read disturbance mechanisms are RowHammer [2, 3, 5, 112] and RowPress [87, 113], where repeatedly activating (i.e., opening) a DRAM row (i.e., aggressor row) or keeping the aggressor row active for a long time induces bitflips in physically nearby rows (i.e., victim rows), respectively. To induce RowHammer bitflips, an aggressor row needs to be activated more times than a threshold value called *RowHammer threshold* (N_{RH}). Various characterization studies [2–5, 37, 47, 50, 55–69, 87, 114–130] show that as DRAM scales to smaller technology nodes, DRAM chips become more vulnerable to RowHammer (i.e., newer chips have lower N_{RH} values). For example, chips manufactured in 2018-2020 can experience bitflips at an order of magnitude fewer row activations than chips manufactured in 2012-2013 [62]. To make matters worse, RowPress [87, 131] reduces N_{RH} significantly (e.g., by orders of magnitude).

DRAM Read Disturbance Mitigation Mechanisms. Many prior works propose mitigation techniques [2, 5, 25, 31, 37, 43, 58, 83, 86, 88–90, 92, 93, 95, 97, 98, 104, 116, 117, 119, 132–196] to protect DRAM chips against RowHammer bitflips. These techniques usually perform two tasks: 1) execute a trigger algorithm and 2) perform RowHammer-preventive actions. The *trigger algorithm* observes memory access patterns and triggers a *RowHammer-preventive action* based on the result of a probabilistic or a deterministic process. RowHammer-preventive actions include one of 1) preventively refreshing victim row [2, 37, 43, 89, 93, 98, 132, 136, 138, 140–142, 144, 145, 147–149, 154–158, 167, 197], 2) dynamically remapping aggressor rows [92, 150, 177, 179], and 3) throttling unsafe accesses [88, 135]. Existing RowHammer mitigation mechanisms can also prevent RowPress bitflips when their trigger algorithms are configured to be more aggressive by configuring them against relatively lower N_{RH} values [87].

3. Motivation

Prior works [62, 88, 89, 105] show that existing RowHammer mitigation mechanisms incur prohibitively high performance overheads as the RowHammer threshold (N_{RH}) decreases because these mechanisms perform RowHammer-preventive actions more aggressively. Given that DRAM read disturbance worsens with shrinking technology node size, N_{RH} is expected to reduce even more, e.g., $< 1K$ [87, 113]. Therefore, reducing the performance overhead of existing RowHammer mitigation mechanisms is critical.

We analyze the performance impact of four state-of-the-art RowHammer mitigation mechanisms as N_{RH} decreases from $4K$ to 64.[2] Fig. 2 presents the system performance (in terms of weighted speedup [198, 199]) of different RowHammer mitigation mechanisms. The x- and y-axes show the N_{RH} values and system performance normalized to a baseline with no RowHammer mitigation mechanism, respectively. Different bars identify RowHammer mitigation mechanisms and error bars mark the 100% confidence interval across 90 workloads.

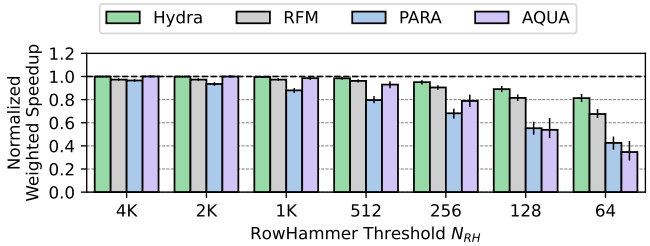

Figure 2: System performance overheads of RowHammer mitigation mechanisms with worsening RowHammer vulnerability

We make three observations from Fig. 2. First, as N_{RH} decreases (i.e., DRAM chips get more vulnerable to read disturbance), system performance significantly reduces for all tested RowHammer mitigation mechanisms ranging from 18.7% (Hydra) to 65.9% (AQUA). Second, at an N_{RH} of 64, Hydra exhibits the least system performance overhead with a performance

[2]We conduct these simulations following the methodology described in §7 for 90 randomly chosen four-core benign workloads from our benchmarks.

degradation of 18.7%. Hydra achieves this at the expense of 1) storing a counter per row in DRAM and 2) increasingly large processor chip area overhead that reaches 56.5KB when configured for a dual rank system with 16 banks at each rank. Third, among the tested RowHammer mitigation mechanisms, PARA is the most lightweight and hardware-scalable RowHammer mitigation mechanism [90], but it has a high system performance degradation of 57.6%. We conclude that RowHammer mitigation mechanisms become more expensive in terms of area and performance as N_{RH} decreases, and reducing their performance overheads at low cost is critical.[3]

4. BreakHammer

BreakHammer is the first mechanism that enhances existing RowHammer mitigation mechanisms with throttling support to reduce their performance overheads.[4]

Overview. BreakHammer is designed to cooperate with RowHammer mitigation mechanisms and reduce their performance overhead by reducing the number of performed RowHammer-preventive actions *without* compromising system robustness. BreakHammer's key idea is to limit the dynamic memory request count of a hardware thread that repeatedly triggers RowHammer-preventive actions. BreakHammer splits the execution timeline into periods of equal length that we call *throttling windows* (similar to a refresh window [104]). During each throttling window, BreakHammer performs three key operations: it 1) observes the time-consuming RowHammer-preventive actions of existing RowHammer mitigation mechanisms, 2) identifies hardware threads that trigger many of these actions (we call such threads *suspect threads*), and 3) reduces the memory bandwidth usage of each suspect thread until it is *not* identified as a suspect in a future throttling window. We implement BreakHammer near the memory controller without requiring proprietary information from or modifications to the DRAM chips. Therefore, BreakHammer is compatible with commodity DRAM chips.

Fig. 3 depicts a high-level overview of BreakHammer in a simplified generic system with multiple processors accessing the main memory through the cache hierarchy or via a direct memory access (DMA) unit [200, 201]. The memory controller implements a memory request scheduler to issue DRAM commands via the DRAM interface (e.g., DDR4 [104]). The memory controller employs a RowHammer mitigation mechanism or issues special DRAM commands (e.g., RFM [96] and PRAC [97]) to provide an on-DRAM-die mitigation mechanism with a time window to perform RowHammer-preventive actions.

BreakHammer divides the execution timeline into throttling windows of length TH_{window} (e.g., 64ms) where it performs three key operations when the RowHammer mitigation mechanism triggers a RowHammer-preventive action.

[3]This is *not* a worst-case evaluation. There are worse workloads and system configurations that lead to higher overheads for *all* mechanisms (§8.1).

[4]BlockHammer [88] proposes throttling as a mechanism to *prevent* RowHammer bitflips. On the other hand, BreakHammer uses throttling to reduce performance overheads of RowHammer mitigation mechanisms.

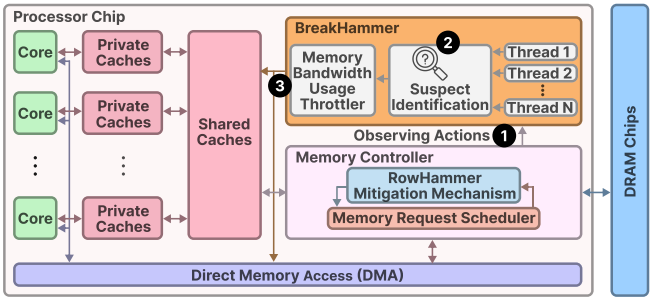

Figure 3: BreakHammer high-level overview

❶ **Observing RowHammer-Preventive Actions.** When a hardware thread causes a RowHammer-preventive action, BreakHammer increments a per-thread counter called *RowHammer-preventive score* for the thread.

❷ **Identifying Suspect Threads.** BreakHammer performs outlier analysis on the RowHammer-preventive scores to identify suspect threads. BreakHammer identifies a hardware thread as a suspect when the thread's RowHammer-preventive score 1) exceeds a threshold and 2) largely surpasses the average RowHammer-preventive score across all threads.

❸ **Throttling Memory Bandwidth Usage of Suspect Threads.** BreakHammer limits the dynamic request count of a suspect thread. To do so, BreakHammer calculates a memory request resource (i.e., last level cache-miss buffers [100]) quota available to each suspect thread based on the duration the thread has been a suspect for. By doing so, BreakHammer reduces the performance degradation the RowHammer mitigation mechanism causes by altering the memory access patterns such that the RowHammer mitigation mechanism performs fewer RowHammer-preventive actions.

Optional Feedback to the System Software. BreakHammer optionally exposes each hardware thread's RowHammer-preventive score counter to the system software. The system software can access these counters similarly to how it accesses thread-specific special registers (e.g., CR3 in x86). By doing so, BreakHammer supports associating scores with software threads, address spaces, processes, users, or other identifiers.

4.1. Observing RowHammer-Preventive Actions

BreakHammer observes a hardware thread's performance overhead due to triggering RowHammer-preventive actions. To do so, BreakHammer maintains a RowHammer-preventive score for each hardware thread, which tracks the number of RowHammer-preventive actions caused by a thread. Since one *cannot* directly attribute a RowHammer-preventive action to a thread, we update the RowHammer-preventive score of each thread proportionally to the fraction of all activations that thread is responsible for a given RowHammer-preventive action (activation count tracking is reset after a given RowHammer-preventive action). The exact method to update RowHammer-preventive scores (i.e., score attribution method) depends on the employed RowHammer mitigation mechanism. We briefly describe the score attribution methods used for the RowHammer mitigation mechanisms we evaluate.

PARA [2]. PARA probabilistically performs a RowHammer-preventive refresh when the memory controller issues a row activation. PARA is *stateless*, i.e., does not keep track of any history or statistics. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive RowHammer-preventive refreshes and 2) when a RowHammer-preventive refresh is performed, attributes a score to each thread proportional to its activation count.

Graphene [89]. Graphene implements the Misra-Gries frequent element finding algorithm [202], which counts the number of activations for the most frequently accessed DRAM rows. Graphene issues a RowHammer-preventive refresh when an aggressor row's activation count exceeds Graphene's refresh threshold. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive RowHammer-preventive refreshes and 2) when a RowHammer-preventive refresh is performed, attributes a score to each thread proportional to its activation count.

Hydra [93]. Hydra implements a two-part tracking algorithm that 1) tracks an aggregated group of rows until the collective activation of the group reaches a threshold (i.e., *group threshold*) and 2) switches to per-row tracking for that group. Hydra stores the per-row tracking table in DRAM and keeps a small cache for the table's entries in the memory controller. Hydra performs a RowHammer-preventive refresh when a per-row table entry exceeds a threshold (i.e., *refresh threshold*). *Score Attribution:* BreakHammer 1) tracks each hardware thread's row activation counts between two consecutive RowHammer-preventive actions (i.e., per-row table cache miss/eviction and RowHammer-preventive refresh) and 2) when a RowHammer-preventive action is performed, attributes a score proportional to its activation count.

TWiCe [98]. TWiCe implements a counter table of frequently accessed rows. Table entries have a lifetime, and infrequently accessed rows are periodically pruned. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive RowHammer-preventive refreshes and 2) when a RowHammer-preventive refresh is performed, attributes a score to each thread proportional to its activation count.

AQUA [92]. AQUA implements the Misra-Gries frequent element finding algorithm (similar to Graphene) and transfers aggressor rows to a quarantine area in DRAM (via *row migration*) to mitigate attacks. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive row migrations and 2) when a row migration is performed, attributes a score to each thread proportional to its activation count.

REGA [95]. REGA modifies the DRAM chip by adding a second row buffer (i.e., *REGA row buffer*) to each subarray. REGA periodically performs refreshes based on a configurable activation frequency (i.e., $REGA_T$) using the REGA row buffer while serving requests using the first row buffer. *Score Attribution:* BreakHammer increments a hardware thread's score by one for every $REGA_T$ activations the thread performs.

Refresh Management (RFM) [96]. The DDR5 standard introduces the new *refresh management* (RFM) command. The memory controller periodically sends RFM commands (e.g., once every 80 row activations to a bank [96]), which provides the DRAM chip with a time window to take necessary RowHammer-preventive actions. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive RFM commands and 2) when the memory controller issues an RFM command, attributes a score to each thread proportional to its activation count.

Per Row Activation Counting (PRAC) [97]. The latest (as of April 2024) JEDEC DDR5 standard introduces a new on-DRAM-die read disturbance mitigation mechanism called *Per Row Activation Counting* (PRAC). PRAC 1) maintains an activation counter per DRAM row [2] and 2) proposes a new *back-off* signal to convey the need for RowHammer-preventive refreshes from the DRAM chip to the memory controller, similar to what prior works propose [148, 154, 167, 184, 203]. *Score Attribution:* BreakHammer 1) tracks the activations of each hardware thread between two consecutive back-off signals and 2) when the DRAM chip triggers a back-off, attributes a score to each thread proportional to its activation count.

4.2. Identifying Suspect Threads

BreakHammer identifies a suspect thread that causes too much performance overhead due to triggering RowHammer-preventive actions. To do so, it can employ different mechanisms. We use a mechanism we call *thresholded deviation from the mean* with two tunable configuration parameters: 1) TH_{threat}: the minimum RowHammer-preventive score to consider a thread as a potential suspect and 2) $TH_{outlier}$: the maximum allowed divergence from the average of all thread RowHammer-preventive scores to mark a thread as suspect. When the RowHammer mitigation mechanism performs a RowHammer-preventive action, BreakHammer uses Alg. 1 to increment RowHammer-preventive scores, detect outliers, and mark suspect threads.[5]

Our BreakHammer implementation attributes score to each thread proportional to its row activation count since the last RowHammer-preventive action (lines 3-7). BreakHammer performs two checks for each thread to detect outliers each time BreakHammer increments scores. First, BreakHammer compares a thread's RowHammer-preventive score against TH_{threat} to determine if the thread has triggered enough RowHammer-preventive actions to be considered as a potential suspect thread (line 11). Second, BreakHammer checks if a thread's RowHammer-preventive score exceeds the mean of all RowHammer-preventive scores by a factor of $TH_{outlier}$ (line 15). If a thread passes both checks, it is marked as a suspect thread (line 16) for the remainder of the throttling window.

Time-interleaving-based continuous monitoring. Indefinitely incrementing scores could eventually cause the coun-

[5]Our score attribution method described in Alg. 1 (lines 3-7) works for all evaluated RowHammer mitigation mechanisms except for REGA. We refer the reader to our open-source repository for the implementation of each mechanism at https://github.com/CMU-SAFARI/BreakHammer.

Algorithm 1: Identifying suspect threads

```
// Threads: All threads in the system
// Scores: RowHammer-preventive score of each thread
// Activations: Row activation count of each thread
// TH_threat: Threat threshold
// TH_outlier: Outlier threshold
1  Function updateScores():
2      // Attribute score to each thread based on its row activation count
3      totalACTs ← sum(Activations)
4      foreach T_i in Threads do
5          Scores(T_i) ← Activations(T_i) / totalACTs
6          Activations(T_i) ← 0
7      end
8      maxDeviation ← (1 + TH_outlier)*mean(Scores)
9      foreach T_i in Threads do
10         // Avoid marking threads with low scores
11         if Scores(T_i) < TH_threat then
12             continue
13         end
14         // Mark threads that exceed the mean by a factor of TH_outlier
15         if Scores(T_i) > maxDeviation then
16             markSuspect(T_i)
17         end
18     end
```

ters to saturate, in which case BreakHammer would mistakenly reduce the memory bandwidth usage of benign workloads. To avoid this issue, BreakHammer periodically resets RowHammer-preventive score counters. To avoid losing information when resetting counters, BreakHammer time interleaves across two sets of counters (similar to what prior works do [88,204]) where each set contains a RowHammer-preventive score counter for each hardware thread, as shown in Fig. 4.

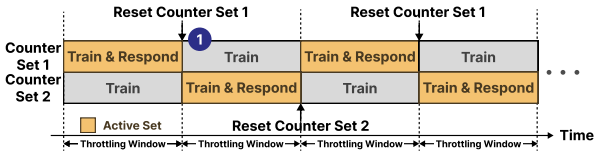

Figure 4: Time interleaving across BreakHammer's counters

During a throttling window, a thread's counters in each set are updated. However, only the counters of one set, called the *active set* (denoted with orange in Fig. 4, e.g., counter set 1 before ❶), responds to queries needed for suspect identification of a thread. At the end of each throttling window, only the counters in the active set are reset (e.g., counter set 1 is reset after ❶). After the reset, the set with retained counter values becomes the active set (e.g., counter set 2 after ❶) for the next throttling window. Because counters in the new active set were already tracking the RowHammer-preventive actions that took place in the previous throttling window, they start responding to queries with already trained values. This method ensures continuous monitoring of each thread's activity across throttling windows.

4.3. Throttling Memory Bandwidth Usage of Suspect Threads

BreakHammer attributes a hardware thread i with two properties: the dynamic request quota (Q_i) and the $recent\_suspect_i$ flag which indicates whether the thread is identified as a suspect thread (see Alg. 1) in the previous throttling window. In the very first throttling window (i.e., when the system boots), no dynamic request quota is enforced on any thread (i.e., Q_i of each thread is equal to the number of all cache-miss buffers) and *no* hardware thread is identified as a suspect thread (i.e., $recent\_suspect_i$ is $false$ for each thread). When a hardware thread i is identified as a suspect (line 16 in Alg. 1), BreakHammer uses Expression 1 to reduce the thread's dynamic memory access quota (Q_i). If the thread is identified as a suspect thread in the previous throttling window (i.e., $recent\_suspect_i$ is $true$), the thread's dynamic request quota is reduced by a constant amount ($P_{oldsuspect}$) such that Q_i becomes $Q_i - P_{oldsuspect}$. If the thread is *not* identified as a suspect thread in the previous throttling window (i.e., $recent\_suspect_i$ is $false$), the thread's dynamic request quota is divided by a constant number ($P_{newsuspect}$) such that Q_i becomes $Q_i / P_{newsuspect}$.

$$Q_i = \begin{cases} max(Q_i - P_{oldsuspect}, 0) & \text{if } recent\_suspect_i \\ Q_i / P_{newsuspect} & \text{else} \end{cases} \quad (1)$$

Resetting Reduced Quotas. A suspect thread executes with reduced dynamic request quota as long as it continues to be identified as a suspect (i.e., $recent\_suspect_i$ stays $true$ across throttling windows). If a suspect thread stays benign (i.e., does *not* get marked as a suspect in Alg. 1) for the full duration of a throttling window (i.e., $recent\_suspect_i$ becomes $false$ at the beginning of a new throttling window), BreakHammer restores the thread's full dynamic quota (i.e., Q_i becomes equal to the number of all cache-miss buffers again).

Throttling with Cache-Miss Buffers. BreakHammer throttles a suspect thread by limiting the number of cache-miss buffers the thread can allocate in the last level cache (LLC) due to two reasons. First, by limiting cache-miss buffers, BreakHammer allows a suspect thread to continue accessing memory locations that result in cache-miss buffer hits. By doing so, a suspect can access the data 1) that already exists in or 2) being brought to caches. Second, BreakHammer is implemented near the memory controller, where it can easily communicate with caches (similar to prior work [88,205–212]).

4.4. DMA and Systems without Caches

BreakHammer throttles memory bandwidth usage by limiting the number of cache-miss buffers that track LLC misses each hardware thread can allocate. However, 1) the memory request serving unit (e.g., DMA) may lack resources to track memory request status (e.g., cache-miss buffers) or 2) processors in a system may lack caches. BreakHammer's memory throttling support can be implemented in such systems by extending: 1) the memory request serving unit with a relatively small counter table that tracks the unresolved memory request count of each hardware thread or 2) the processor's load store unit (LSU) to limit the unresolved memory instruction count of each hardware thread. We do *not* recommend throttling hardware threads at the memory controller because the unresolved memory requests of a malicious application can potentially reduce performance by blocking the memory request resources of the memory hierarchy (e.g., cache-miss buffers) or the processor (e.g., LSU) [213].

5. Security Analysis

We analyze BreakHammer's security against 1) RowHammer attacks, 2) attacks that exploit RowHammer-preventive actions to hog DRAM bandwidth availability (i.e., memory performance attacks [211]), and 3) attacks that try to manipulate score attribution by causing benign threads to perform the access that triggers a RowHammer-preventive action.

5.1. Security Against RowHammer Attacks

BreakHammer is *not* a RowHammer defense by itself, but it is a mechanism that reduces the performance overheads incurred by existing RowHammer defenses. BreakHammer does *not* interfere with any triggering algorithm and the RowHammer-preventive actions that an existing RowHammer mitigation mechanism performs. Therefore, BreakHammer preserves the security guarantees of the RowHammer mitigation mechanism it is attached to. When BreakHammer observes that the RowHammer mitigation mechanism works aggressively (i.e., performs many RowHammer-preventive actions) as a result of the memory accesses generated by one or few threads, BreakHammer effectively reduces the memory requests coming from those threads. Meanwhile, the RowHammer mitigation mechanism continues to execute its triggering algorithm on the memory accesses that contain fewer requests from the suspect threads. As such, the RowHammer mitigation mechanism performs RowHammer-preventive actions less aggressively *without* compromising its security guarantees.

5.2. Security Against Memory Performance Attacks

BreakHammer increments a hardware thread's RowHammer-preventive score when a thread causes a RowHammer-preventive action. Due to time-interleaving-based continuous monitoring across two sets of counters, similar to what prior work does [88], BreakHammer successfully tracks *all* RowHammer-preventive actions. Therefore, BreakHammer 1) detects a suspect thread responsible for too many RowHammer-preventive actions that reduce DRAM bandwidth availability and 2) limits the suspect's dynamic memory access count. By doing so, BreakHammer reduces the interference in the memory subsystem and makes it harder for an attacker to mount a memory performance attack by exploiting RowHammer-preventive actions.

Multi-Threaded Attacks. There are two ways that an attacker can try to defeat BreakHammer using a group of threads: 1) increasing the average RowHammer-preventive score across all threads where mounting a memory performance attack becomes the norm (i.e., *rigging suspect identification*) or 2) intelligently utilizing threads such that BreakHammer works as intended against each thread, but the coordinated group mounts a memory performance attack (i.e., circumventing suspect identification).

Rigging Suspect Identification. An attacker can utilize many hardware threads and maliciously increase the average RowHammer-preventive score across all hardware threads so that the attacker's behavior becomes the norm instead of an outlier. As such, a thread used by the attacker (denoted as an *attack thread*) can trigger many RowHammer-preventive actions *without* getting identified as a suspect. During the attack, an attacker can control the *RowHammer-preventive score of an attack thread* i ($RS_{\text{atk}i}$) but *cannot* control the *RowHammer-preventive score of a benign thread* i ($RS_{\text{ben}i}$). To avoid suspect identification, the *attack thread with maximum RowHammer-preventive score* ($RS^{\text{max}}_{\text{atk}}$) should not exceed the average RowHammer-preventive score across all threads by a factor of $TH_{outlier}$. Expression 2 calculates $RS^{\text{max}}_{\text{atk}}$ before an attack thread is identified as a suspect, where the number of attack and benign threads in the system are N_{atk} and N_{ben}, respectively.

$$RS^{\text{max}}_{\text{atk}} < \frac{\sum_{i=1}^{N_{\text{atk}}} RS_{\text{atk}i} + \sum_{i=1}^{N_{\text{ben}}} RS_{\text{ben}i}}{N_{\text{atk}} + N_{\text{ben}}} (1 + TH_{outlier}) \quad (2)$$

Fig. 5 presents a visualization of Expression 2. The x- and y-axes show the percentage of attack threads in the system ($N_{\text{atk}}/(N_{\text{atk}} + N_{\text{ben}})$) and $RS^{\text{max}}_{\text{atk}}$ before an attacker thread is identified as suspect normalized to the *average RowHammer-preventive score of benign threads* ($RS^{\text{avg}}_{\text{ben}}$), respectively. Different lines identify $TH_{outlier}$ configurations.

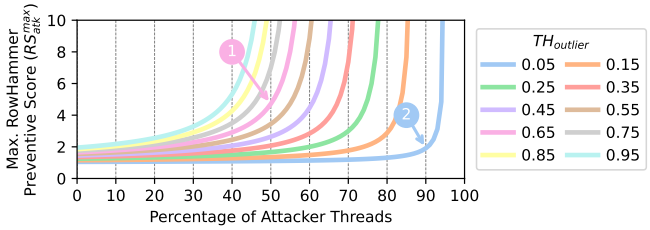

Figure 5: $RS^{\text{max}}_{\text{atk}}$ normalized to $RS^{\text{avg}}_{\text{ben}}$ before an attack thread is identified as a suspect

We make two observations from Fig. 5. First, at a $TH_{outlier}$ value of 0.65 (1), if the attacker concurrently uses 50% of all hardware threads, an attack thread can trigger $4.71\times$ the number of RowHammer-preventive actions that benign threads trigger on average. Second, at a $TH_{outlier}$ value of 0.05 (2), if the attacker concurrently uses 90% of all hardware threads, an attack thread can trigger $1.90\times$ the number of RowHammer-preventive actions that benign threads trigger on average. We conclude that an attacker needs to concurrently use an overwhelming fraction (i.e., $> 90\%$) portion of all hardware threads to defeat BreakHammer. The system software can determine such an attack by monitoring system-wide resource consumption statistics. We leave the design of such system-level detection techniques for future research.[6]

Circumventing Suspect Identification. An attacker can try to circumvent suspect identification with a large group of attack threads by switching to a new attack thread each time the primary attack thread used to trigger RowHammer-preventive action is identified as a suspect. Doing so allows the attacker to avoid the limited memory bandwidth and continue triggering RowHammer-preventive actions. To prevent the attack,

[6]Rigging suspect identification problem can also be solved by developing other suspect identification mechanisms. For example, one can devise a technique that is sensitive to the fraction of aggressive threads in the system. We leave such techniques to future work.

the system software can track RowHammer-preventive scores within system software structures (e.g., thread context). Doing so allows tracking RowHammer-preventive scores at different granularities than a single hardware thread. For example, the system software can 1) associate a process, address space, or user as the owner of its threads' RowHammer-preventive scores and 2) slow down or stop owners with high cumulative RowHammer-preventive scores. We leave such system integration of BreakHammer to future work.

5.3. Manipulating Score Attribution

Multiple applications can share a DRAM row. An attacker can try to trick BreakHammer's score attribution method into increasing the RowHammer-preventive score of a benign application by: 1) activating a row many times *without* causing a RowHammer-preventive action and then 2) waiting for a benign thread to trigger the RowHammer-preventive action. BreakHammer attributes a score to each hardware thread proportional to its activation count for a given RowHammer-preventive action. Therefore, BreakHammer is secure against score attribution manipulation because it tracks each hardware thread's contribution to a RowHammer-preventive action.

6. Hardware Complexity

BreakHammer is implemented near the memory controller and does *not* introduce any changes to the DRAM chip or its interfaces. We evaluate BreakHammer's hardware complexity using CACTI [103] and Synopsys Design Compiler [102]. We implement BreakHammer in Chisel HDL [101] and synthesize the emitted Verilog HDL design using Synopsys Design Compiler [102] with a 65nm process technology.

Area Analysis. BreakHammer uses two 32-bit RowHammer-preventive score counters, one 16-bit activation counter, and two 1-bit suspect flags per hardware thread, which consume $0.000105mm^2$ per memory channel at a 65nm process technology, consuming an overall area overhead of 0.0002% of a state-of-the-art Intel Xeon processor's chip area [214] (which is implemented in an Intel 10 nm technology node).

Latency Analysis. Our BreakHammer implementation is fully pipelined: it can observe new actions (e.g., memory access or RowHammer-preventive refresh) and make a throttling decision every cycle using an 8-stage pipeline.[7] According to our Chisel HDL model, BreakHammer can be clocked at 1.5GHz ($\sim$0.67ns). This latency is faster than the latency of regular memory controller operations as it is smaller than t_{RRD} (e.g., 2.5ns in DDR4 [104] and 5ns in DDR5 [96]).

7. Experimental Methodology

We evaluate BreakHammer's effect on system performance using cycle-accurate simulations. We use Ramulator2 [105,106] for our simulations where 1) BreakHammer is integrated with PARA [2], Graphene [89], Hydra [93], TWiCe [98], AQUA [92], REGA [95], RFM [97], and PRAC [97] and 2) BlockHammer [88] is implemented. We evaluate system performance using the weighted speedup metric [198, 199] and unfairness using the maximum slowdown on a benign application [206,208,213,215].

Table 1 shows our system configuration. We assume a realistic quad-core system, connected to a dual-rank memory with eight bank groups, each containing two banks (32 banks in total). The memory controller employs the FR-FCFS memory scheduler [216, 217] with a Cap on Column-Over-Row Reordering (FR-FCFS+Cap) of four [211].

Table 1: Simulated System Configuration

Processor	4.2 GHz, 4 core, 4-wide issue, 128-entry instr. window
Last-Level Cache	64-byte cache line, 8-way set-associative, 8 MB
Memory Controller	64-entry read/write request queues; FR-FCFS+Cap with Cap=4 [211]; Address mapping: MOP [218]
Main Memory	DDR5, 1 channel, 2 ranks, 8 bank groups, 2 banks/bank group, 64K rows/bank

Comparison Points. We pair BreakHammer with eight different state-of-the-art RowHammer mitigation mechanisms that provide RowHammer-safe operation: one is a probabilistic mechanism [2], five are deterministic mechanisms [89, 92, 93, 95, 98], and two use the DDR5 RFM command [97] and PRAC [97]. We configure all RowHammer mitigation mechanisms except RFM and PRAC according to the methodology described by their studies and scale them to the N_{RH} values used in our evaluation. For RFM and PRAC, we assume that the DRAM chip maintains an activation counter for each DRAM row [97,184,219] and use mathematically-proven RowHammer-secure configurations from prior work [220].

We compare BreakHammer-paired versions of these eight RowHammer mitigation mechanisms to their baseline implementations without BreakHammer. We also compare these eight BreakHammer-paired mechanisms to BlockHammer, the state-of-the-art throttling-based RowHammer mitigation mechanism. We configure the duration of a throttling window as 64ms to match the refresh window [104], similar to counter reset periods of prior work [88, 89, 94]. We empirically configure TH_{threat} and $TH_{outlier}$ on a smaller set of workloads separate from our evaluations. Table 2 summarizes BreakHammer's configuration parameters for our evaluation.

Table 2: BreakHammer Configuration

Component	Parameters
Suspect Identification	TH_{window} : 64ms TH_{threat} : 32 $TH_{outlier}$: 0.65
Memory Throttling	$P_{oldsuspect}$: 1 $P_{newsuspect}$: 10

Workloads. We evaluate BreakHammer with applications from five benchmark suites: SPEC CPU2006 [107], SPEC CPU2017 [108], TPC [109], MediaBench [110], and YCSB [111]. We group applications into three memory intensity groups based on their row buffer misses-per-kilo-instructions (RBMPKIs). These groups are High (H), Medium (M), and Low (L) for the lowest RBMPKI values of 20, 10, and 0, respectively. We

[7] We note that having a latency in the order of a few nanoseconds is unnecessary because BreakHammer 1) slows down *future* memory requests of a thread and 2) does *not* stop online memory requests. As such, BreakHammer *can* allow a few more memory requests from a thread before throttling it as long as RowHammer-preventive action tracking is done correctly.

create six four-core workload mixes (HHHH, HHMM, MMMM, HHLL, MMLL, and LLLL)[8] where each mix contains 15 four-core workloads (90 in total). We simulate these workloads until each benign core completes 100M instructions.[9] Table 3 summarizes RBMPKI and the average number of rows with more than 512, 128, and 64 activations per 64ms time window for the 8 most memory-intensive workloads. From this table, we observe that as N_{RH} decreases, many benign applications become capable of triggering RowHammer-preventive actions.

Table 3: Workload Characteristics: RBMPKI and Average Number of Rows with More Than 512+, 128+, and 64+ Activations per 64ms Time Window

Workload	RBMPKI	ACT-512+	ACT-128+	ACT-64+
429.mcf	68.27	2564	2564	2564
470.lbm	28.09	664	6596	7089
462.libquantum	25.95	0	0	1
549.fotonik3d	25.28	0	88	10065
459.GemsFDTD	24.93	0	218	10572
519.lbm	24.37	2482	5455	5824
434.zeusmp	22.24	292	4825	11085
510.parest	17.79	94	185	803
Average	29.615	762	2491	6000

8. Performance Evaluation

We evaluate BreakHammer's performance and energy impact in four parts. First (§8.1), we experimentally demonstrate that 1) a malicious hardware thread can incur significant performance and energy overheads on concurrently running multi-programmed workloads, and 2) BreakHammer prevents such performance and energy overheads by significantly reducing the number of RowHammer-preventive actions and increasing DRAM bandwidth availability. Second, we show that BreakHammer does *not* incur significant performance and DRAM energy overheads when there is *no* malicious thread in the system (§8.2). Third, we compare BreakHammer against the state-of-the-art memory access throttling-based RowHammer mitigation mechanism, BlockHammer [88] (§8.3). Our analysis shows that BreakHammer 1) outperforms BlockHammer across *all* evaluated N_{RH} values and 2) maintains a minimal area overhead with only two counters per hardware thread whereas BlockHammer requires a significantly growing history buffer as N_{RH} decreases. Fourth, we analyze BreakHammer's sensitivity to configuration parameters (§8.4).

8.1. Under RowHammer Attack

System Performance. Fig. 6 presents the average performance impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, at an N_{RH} of $1K$. The x-axis shows the memory intensity of applications in the mix (15 workloads per mix) where H, M, L, and A denote High, Medium, Low, and Attacker, respectively, alongside the geometric-mean (geomean) of performance across *all* 90 workloads. Different bars identify RowHammer mitigation mechanisms paired with BreakHammer (e.g., PARA+BH). The y-axis shows the performance of benign workloads normalized to each baseline RowHammer mitigation mechanism *without* BreakHammer.

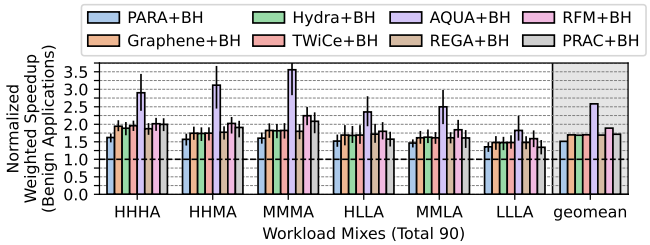

Figure 6: BreakHammer's impact on performance for existing RowHammer mitigation mechanisms with an attacker present

From Fig. 6, we observe that BreakHammer increases the system performance of benign workloads by an average (maximum) of 84.6% (170.0%) when the system is under RowHammer attack from one malicious thread. We note that BreakHammer detects and throttles the attacker in *all* 90 workloads.

Unfairness. Fig. 7 presents the average unfairness impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, at an N_{RH} of $1K$. The x- and y-axes show the memory intensity of the applications in the mix and unfairness on benign applications normalized to each baseline RowHammer mitigation mechanism *without* BreakHammer.

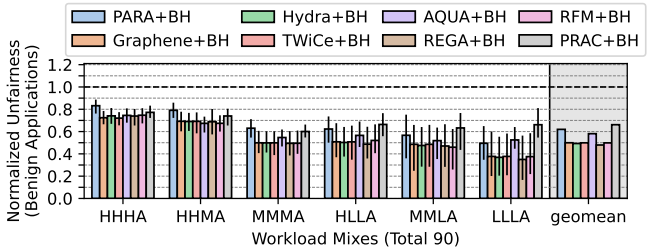

Figure 7: BreakHammer's impact on unfairness for existing RowHammer mitigation mechanisms with an attacker present

We make two observations from Fig. 7. First, BreakHammer reduces unfairness on benign workloads by an average (maximum) of 45.8% (90.6%). Second, BreakHammer reduces unfairness the most (55.9% on average) for LLLA and the least (24.7% on average) for HHHA mixes. We attribute this trend to high memory intensity applications causing more memory interference against the attacker and reducing the attacker's effectiveness at mounting a memory performance attack.

System Performance with Future DRAM Chips. Fig. 8 presents the average performance impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, as N_{RH} decreases from $4K$ to 64. The x- and y-axes show the N_{RH} values and system performance of benign workloads normalized to a baseline with *no* RowHammer mitigation mechanism, respectively.

[8] Each letter in a mix denotes the memory intensity of an application.

[9] We do *not* wait for attacker cores to complete 100M instructions because BreakHammer significantly slows down the attacker's progression and attacker performance is not important (nor evaluated).

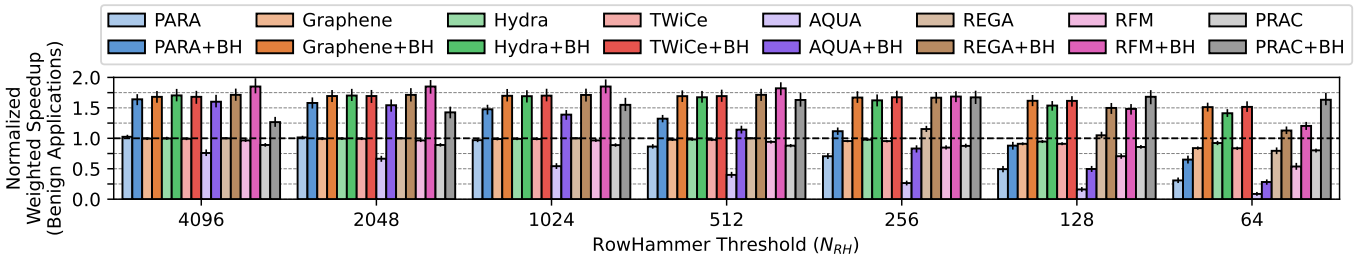

Figure 8: BreakHammer's performance scaling for existing RowHammer mitigation mechanisms with an attacker present

We make three observations from Fig. 8. First, across *all* evaluated N_{RH} values, BreakHammer increases the system performance of benign workloads by an average (maximum) of 90.1% (162.4%). Second, as N_{RH} decreases, BreakHammer maintains a speedup above the no defense baseline for *all* mechanisms except for PARA at $N_{RH} < 256$ and AQUA at $N_{RH} < 512$. At low N_{RH} values (e.g., $< 1K$), PARA and AQUA significantly degrade system performance even after the attacker is throttled. PARA is *stateless*, where it probabilistically performs a RowHammer-preventive refresh *even* for a benign application that does *not* activate a row many times. On the other hand, AQUA uses row migration, which is a costly RowHammer-preventive action in itself. Third, RFM and PRAC induce high system performance overheads *even* with *per row activation counters* in DRAM. However, BreakHammer still reduces the performance overheads of RFM and PRAC *without* any online information about the on-DRAM-die mechanisms.

Unfairness with Future DRAM Chips. Fig. 9 presents the unfairness impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, as N_{RH} decreases from $4K$ to 64. The x- and y-axes show the N_{RH} values and unfairness on benign workloads normalized to a baseline with *no* RowHammer mitigation mechanism, respectively.

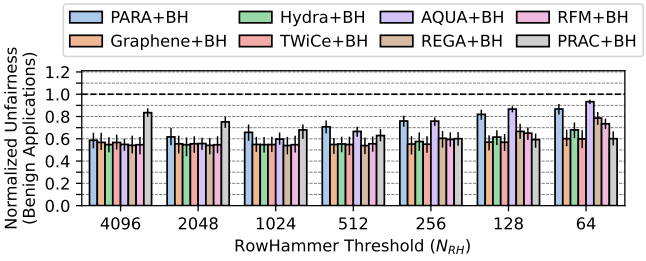

Figure 9: BreakHammer's impact on unfairness for existing RowHammer mitigation mechanisms with an attacker present

We make two observations from Fig. 9. First, across *all* evaluated N_{RH} values, BreakHammer reduces the unfairness on benign workloads by an average (maximum) of 31.5% (99.1%). Second, as N_{RH} decreases, BreakHammer's unfairness benefits for 1) PRAC increases while 2) PARA and AQUA decreases. We attribute PRAC's unfairness benefit improvements to it accurately tracking each row's activation count, where PRAC does *not* perform many RowHammer-preventive refreshes at relatively high N_{RH} values (i.e., $> 1K$). As such, when N_{RH} decreases, the attacker triggers RowHammer-preventive refreshes frequently and deviates from the average RowHammer-preventive score of benign threads faster. On the other hand, PARA and AQUA's unfairness benefits decrease. This is because PARA and AQUA increase memory interference for benign applications and make it harder to detect the attacker as 1) PARA's starts performing many RowHammer-preventive refreshes for the benign applications and 2) AQUA uses time-consuming row migration operations.

Effect on Performed RowHammer-Preventive Actions. Fig. 10 presents the performed RowHammer-preventive action counts of PARA, Graphene, Hydra, TWiCe, AQUA, RFM, and PRAC 1) by themselves and 2) when paired with BreakHammer, as N_{RH} decreases from $4K$ to 64.[10] Each subplot displays a different mechanism where the x-axis shows the different N_{RH} values and the y-axis shows the number of RowHammer-preventive actions taken with BreakHammer normalized to without BreakHammer at an N_{RH} of $4K$. The orange and blue lines respectively depict the 1) BreakHammer-paired and 2) baseline mechanisms. The band shade around each line marks the 100% confidence interval across *all* 90 workloads.

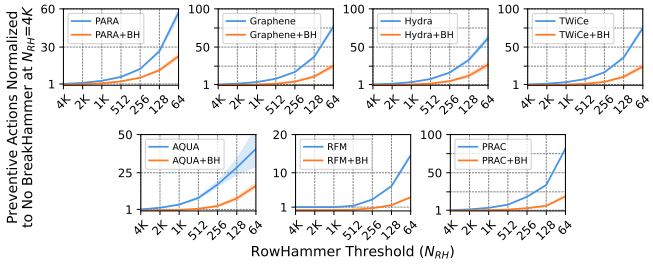

Figure 10: BreakHammer's impact on the number of RowHammer-preventive actions performed by existing RowHammer mitigation mechanisms with an attacker present

We make two observations from Fig. 10. First, as N_{RH} decreases, RowHammer-preventive actions of *all* mitigation mechanisms increase. Second, across all evaluated N_{RH} values, BreakHammer decreases the RowHammer-preventive actions of *all* mechanisms by an average (maximum) of 71.6% (91.8%).

Memory Latency. Fig. 11 presents the memory latency percentiles (P_N) of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, at an N_{RH} of 64. Each subplot depicts the memory latency results of a different RowHammer mitigation mechanism. In subplots, the x- and

[10] We do *not* include REGA in this analysis because it performs refreshes in parallel to row activations. However, we still evaluate REGA's performance and energy overheads based on its impact on DRAM timing constraints.

y-axes show the percentile values (e.g., 90th percentile shows that 90% of memory latencies are below this value) and the memory latency in nanoseconds, respectively. Different curves identify the baseline with no RowHammer mitigation and a RowHammer mitigation mechanism 1) by itself and 2) paired with BreakHammer. The background color indicates the mechanism with lower memory latency for a given percentile of accesses, e.g., orange means that the tested mechanism results in a lower memory latency when used with BreakHammer, compared to without BreakHammer.[11]

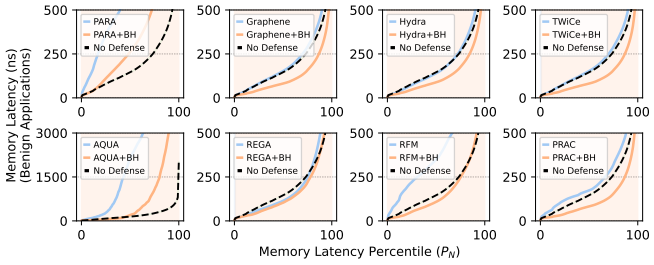

Figure 11: BreakHammer's impact on memory latency for existing RowHammer mitigations with an attacker present

From Fig. 11, we observe that at an N_{RH} of 64, BreakHammer reduces the memory latency benign applications experience for *all* RowHammer mitigation mechanisms, sometimes to values *even* lower than the no defense baseline. This is because BreakHammer *significantly* reduces the memory interference caused by the attacker at multiple levels of the memory hierarchy, e.g., memory request scheduler and caches.

DRAM Energy. Fig. 12 presents the DRAM energy impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, with an attacker present, as N_{RH} decreases from $4K$ to 64. The x- and y-axes shows the N_{RH} values and DRAM energy of benign workloads normalized to a baseline with *no* RowHammer mitigation mechanism, respectively.

We make three observations from Fig. 12. First, across *all* evaluated N_{RH} values, BreakHammer significantly reduces energy by an average (maximum) of 55.4% (67.4%). Second, as N_{RH} decreases from $4K$ to 64, *all* baseline RowHammer mitigation mechanisms consume more energy (4.4x on average). Third, at an N_{RH} of 64, AQUA and RFM incur the highest energy overhead by 22.3x and 3.7x on avearge, respectively.

We conclude that 1) RowHammer mitigation mechanisms incur increasing performance and energy overheads due to performing many RowHammer-preventive actions as N_{RH} decreases and 2) BreakHammer significantly improves performance, unfairness, memory latency, and energy when an attacker is present, at *all* evaluated N_{RH} values.

[11] AQUA's subplot uses a different scale because AQUA incurs relatively large memory latencies due to costly row migration operations.

8.2. No RowHammer Attack

System Performance. Fig. 13 presents the performance impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, where all applications are benign, at an N_{RH} of 64. The x- and y-axes show the workload mixes and the system performance normalized to each baseline RowHammer mitigation mechanism *without* BreakHammer.

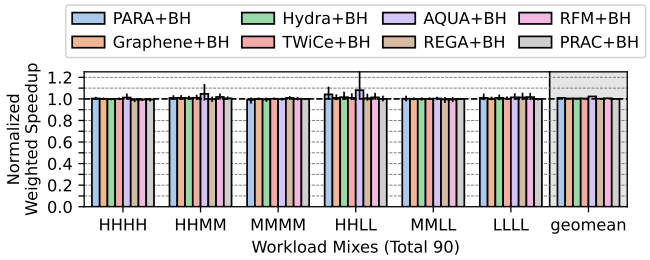

Figure 13: BreakHammer's impact on performance for existing RowHammer mitigation mechanisms with no attacker

From Fig. 13, we observe that BreakHammer increases the system performance of benign application mixes by an average (maximum) of 0.7% (2.4%).

Unfairness. Fig. 14 presents the unfairness impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, where all applications are benign, at an N_{RH} of $1K$. The x- and y-axes show the memory intensity of the applications in the mix and unfairness normalized to each baseline RowHammer mitigation mechanism *without* BreakHammer.

From Fig. 14, we observe that BreakHammer *slightly* increases unfairness by 0.9% on average when compared to the baseline mechanisms *without* BreakHammer. Our results at an N_{RH} of 1K, across eight mitigation mechanisms, and 90 workloads show that BreakHammer identifies a benign application as suspect in 2.2% of the simulations (not shown in Fig. 14).

System Performance with Future DRAM Chips. Fig. 15 presents the performance impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, where all applications are

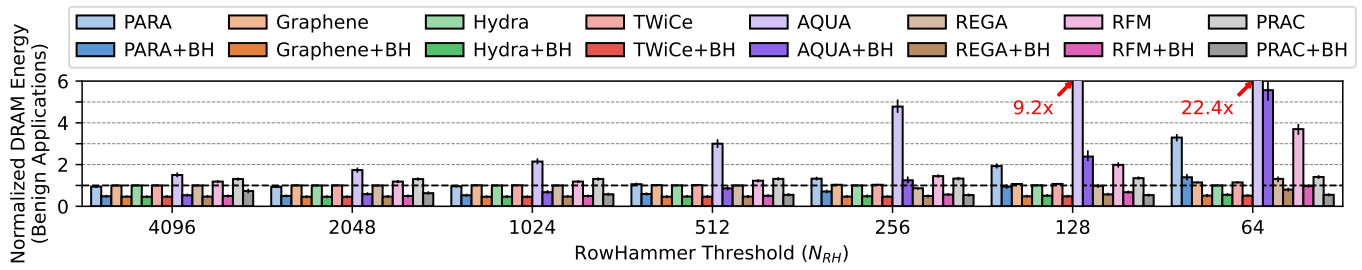

Figure 12: BreakHammer's energy scaling for existing RowHammer mitigation mechanisms with an attacker present

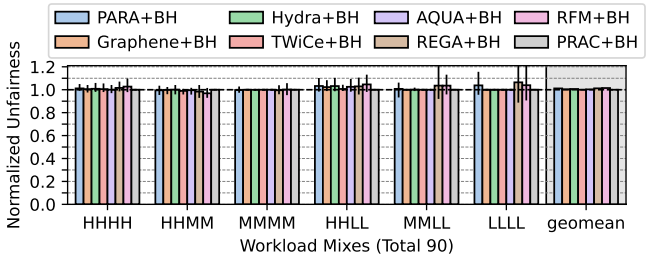

Figure 14: BreakHammer's impact on unfairness for existing RowHammer mitigation mechanisms with no attacker

benign, as N_{RH} decreases from $4K$ to 64. The x- and y-axes show the N_{RH} values and system performance normalized to the baseline RowHammer mitigation mechanism *without* BreakHammer, respectively.

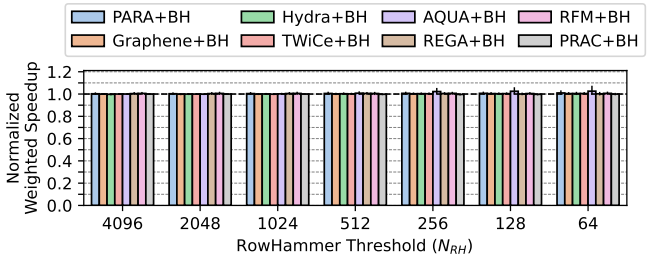

Figure 15: BreakHammer's impact on performance for existing RowHammer mitigation mechanisms at various N_{RH} values

From Fig. 15, we observe that below an N_{RH} of 1024, BreakHammer slightly improves the average (maximum) performance of *all* tested RowHammer mitigation mechanisms.

Unfairness with Future DRAM Chips. Fig. 9 presents the average unfairness impact of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, where all applications are benign, as N_{RH} decreases from $4K$ to 64. The x- and y-axes show the N_{RH} values and unfairness normalized to a baseline with *no* RowHammer mitigation mechanism, respectively.

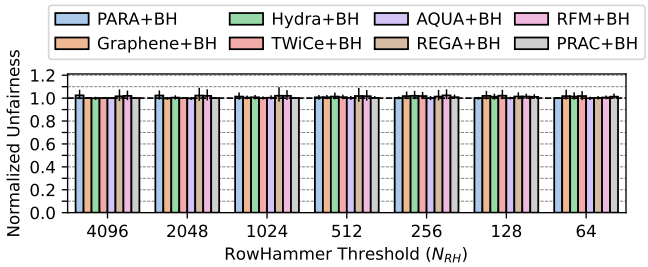

Figure 16: BreakHammer's impact on unfairness for existing RowHammer mitigation mechanisms at various N_{RH} values

We make two observations from Fig. 16. First, as N_{RH} decreases, BreakHammer *slightly* increases unfairness by 0.9% on average compared to the baseline mechanisms *without* BreakHammer. Second, BreakHammer has a best-case unfairness reduction of 29.1% and a worst-case unfairness increase of 36.4%. Our results across *all* N_{RH} values, eight mitigation mechanisms, and 90 workloads show that BreakHammer marks a benign application as suspect in 18.7% of the simulations.

Memory Latency. Fig. 17 presents the memory latency percentiles (P_N) of BreakHammer when paired with eight state-of-the-art RowHammer mitigation mechanisms on four-core workloads, where all applications are benign, at an N_{RH} of 64. Each subplot depicts the memory latency results of a different RowHammer mitigation mechanism. In subplots, the x- and y-axes show the percentile values (e.g., 90th percentile shows that 90% of memory latencies are below this value) and the memory latency in nanoseconds, respectively.

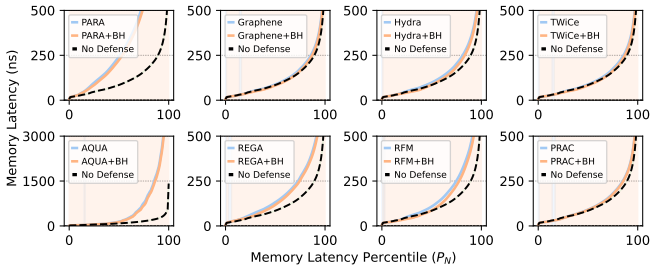

Figure 17: BreakHammer's impact on memory latency for existing RowHammer mitigations with no attacker

From Fig. 17, we observe that at an N_{RH} of 64, BreakHammer induces no overhead and improves the memory latency at all P_N values across all RowHammer mitigation mechanisms.

We conclude that BreakHammer either *slightly* improves or does *not* degrade average performance and energy for benign applications for *all* evaluated RowHammer mitigation mechanisms.

8.3. Comparison to BlockHammer

BlockHammer [88, 221] is the state-of-the-art throttling based RowHammer mitigation mechanism. It works by blacklisting rows that are frequently activated. To understand BreakHammer's performance benefits compared to BlockHammer we study four-core workloads with an attacker present at seven N_{RH} values (from $4K$ to 64).

Fig. 18 presents the performance impact of 1) BreakHammer when paired with eight RowHammer mitigation mechanisms and 2) BlockHammer on four-core workloads, with an attacker present, as N_{RH} decreases from $4K$ to 64. The x- and y-axes show N_{RH} values and system performance normalized to a baseline with no RowHammer mitigation, respectively.

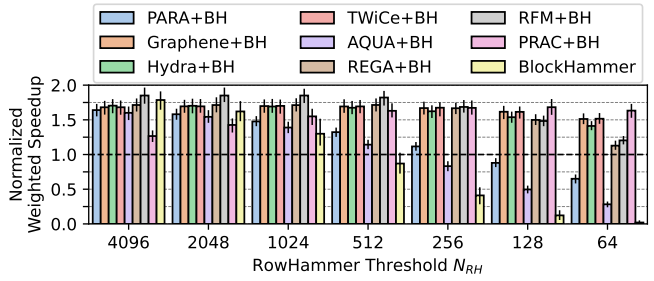

Figure 18: BreakHammer's impact on performance compared to BlockHammer [88], the state-of-the-art throttling-based RowHammer mitigation mechanism

We make two observations from Fig. 18. First, across *all* evaluated N_{RH} values, a mechanism paired with BreakHammer

outperforms BlockHammer. Second, as N_{RH} decreases from $4K$ to 64, BlockHammer's average system performance impact drastically drops from 78.6% improvement to 98.0% *degradation*. This is because BlockHammer blocks accesses to a row that is close to causing RowHammer bitflips until the row's victims are periodically refreshed. As N_{RH} decreases, even a benign application activates DRAM rows many times before they are periodically refreshed (see Table 3). Therefore, at low N_{RH} values, BlockHammer blocks access to increasingly many rows and significantly degrades performance. BreakHammer does *not* suffer from such a weakness because it allows refreshing victim rows when necessary and only throttles suspect applications. Based on these observations, we conclude that BreakHammer outperforms BlockHammer, the previous state-of-the-art throttling based mechanism at *all* N_{RH} values even when combined with the *worst*-scaling RowHammer mitigations, i.e., AQUA and PARA.

8.4. Sensitivity to Configuration Parameters

BreakHammer has three tunable configuration parameters: 1) TH_{window}: the length of a throttling window, 2) TH_{threat}: the minimum RowHammer-preventive score to consider a thread as a potential suspect, and 3) $TH_{outlier}$: the maximum allowed divergence from the average of all thread RowHammer-preventive scores to mark a thread as suspect. We set TH_{window} as 64ms to match with the refresh interval [104], similar to prior work [88, 98]. We set $TH_{outlier}$ as 0.65 to limit the RowHammer-preventive score of attackers (e.g., $\approx$ 5x the total score of benign threads) when the fraction of attack threads in a system is near 50% (see §5.2).

To determine TH_{threat} we sweep our parameters from $4K$ to 32. Fig. 19 shows the performance impact of BreakHammer for varying TH_{threat} configurations, as N_{RH} decreases from $4K$ to 64, in a box-and-whisker plot.[12] The rows of subplots show the workloads with an attacker present (top) and where all applications are benign (bottom). The columns of subplots show N_{RH} values and for each subplot, the x-axis shows the TH_{threat} values and the y-axis shows the weighted speedup normalized to a baseline system with TH_{threat} of $4K$.

We make two observations from Fig. 19. First, as $TH_{outlier}$ decreases, BreakHammer's average system performance benefit increases for both RowHammer attack (top) and no RowHammer attack (bottom) workloads. Second, as N_{RH} decreases, more all-benign workload mixes observe reduced system performance compared to a baseline system with TH_{threat} of $4K$. This is because as N_{RH} decreases, many benign applications become capable of triggering RowHammer-preventive actions (see Table 3). Therefore, more applications are identified as suspects when $TH_{outlier}$ and N_{RH} are low enough.

We choose a $TH_{outlier}$ value of 32 because such configuration 1) provides high performance benefits with an attacker present and 2) induces either little or *no* performance overheads when all applications are benign.

[12]The box is lower-bounded by the first quartile (i.e., the median of the first half of the ordered set of data points) and upper-bounded by the third quartile (i.e., the median of the second half of the ordered set of data points). The interquartile range (IQR) is the distance between the first and third quartiles (i.e., box size). Whiskers mark the central $1.5IQR$ range.

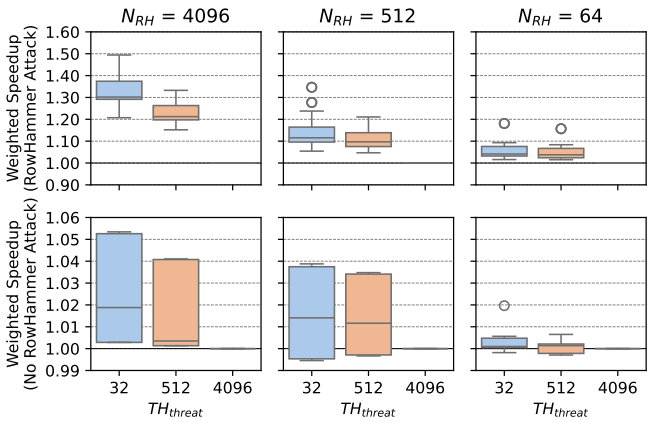

Figure 19: BreakHammer's sensitivity to TH_{threat}

9. Related Work

To our knowledge, this is the first work that provides throttling support to reduce the performance and energy overheads of RowHammer mitigation mechanisms. §8 qualitatively and quantitatively compares BreakHammer with the most relevant prior mechanisms. This section discusses other related works.

RowHammer Mitigation Mechanisms. There have been various mitigation mechanisms [2, 5, 25, 31, 37, 43, 58, 83, 86, 88–90, 92, 93, 95, 97, 98, 104, 116, 117, 119, 132–196] proposed (both by academia and industry) against RowHammer [2]. DRAM manufacturers implement RowHammer mitigation mechanisms, also known as *Target Row Refresh* (TRR) [37, 43, 96, 104], in commercial DRAM chips, but they do *not* openly disclose specific designs. Recent research shows that custom attack patterns can bypass these mechanisms [37, 42–44]. With worsening RowHammer vulnerability, TRR mechanisms need to perform more preventive refresh operations. To provide TRR mechanisms with the necessary time window to perform preventive refreshes, the JEDEC DDR5 standard [99] introduces the *Refresh Management* (RFM) command and (as of April 2024) the *Per Row Activation Counting* (PRAC) mechanism [97]. RFM and PRAC are secure for future DRAM chips with worse RowHammer vulnerabilities, at the cost of high performance overheads [220]. Our goal in this paper is *not* to propose a new RowHammer mitigation mechanism, but rather to reduce the performance and energy overheads of RowHammer mitigation mechanisms by carefully reducing the number of RowHammer-preventive actions they perform, *without* compromising system robustness. As we show in §8, BreakHammer *significantly* reduces the performance and energy overheads of eight state-of-the-art RowHammer mitigation mechanisms

Memory Performance Attacks. Several prior works propose techniques for mounting [211, 222] and preventing [207, 208, 212] memory performance attacks. These works tackle memory performance attacks from the memory request scheduling algorithms' point of view. However, they 1) do *not* consider RowHammer-preventive actions and 2) are *unaware* of the

DRAM bandwidth usage caused by RowHammer mitigation mechanisms. In contrast, BreakHammer tackles the performance and energy overheads induced by RowHammer mitigation mechanisms. Therefore, memory performance attack mitigation proposals (e.g., [207, 208, 212]) can be combined with BreakHammer to achieve better security against memory performance attacks.

Other Uses of Throttling. Prior works on quality-of-service and fairness-oriented architectures propose selectively throttling main memory accesses to provide latency guarantees and/or improve fairness across applications (e.g., [206, 208, 210–213, 222–228]). These mechanisms are 1) *unaware* of RowHammer-preventive actions, 2) *not* designed to reduce the overheads of RowHammer mitigation mechanisms, and 3) *not* designed to prevent RowHammer attacks. Thus, these mechanisms do *not* interfere with memory performance attacks that arise from the overheads of the RowHammer mitigation mechanisms. As such, BreakHammer is complementary to these mechanisms and can work together with them.

10. Conclusion

We introduced BreakHammer, the first mechanism to reduce performance and energy overheads of existing RowHammer solutions. The key idea of BreakHammer is to limit the dynamic memory request count of a hardware thread based on how frequently the thread triggers RowHammer-preventive actions. We show that BreakHammer is compatible with both memory controller-based and on-DRAM-die RowHammer mitigation mechanisms by implementing BreakHammer with eight state-of-the-art RowHammer mitigation mechanisms. Our rigorous experimental evaluations show that BreakHammer significantly 1) reduces the number of performed RowHammer-preventive actions and thereby 2) improves system performance and DRAM energy. BreakHammer's benefits increase as DRAM chips become more vulnerable to RowHammer. We hope and expect that BreakHammer will help researchers and engineers to build higher-performance and lower energy read disturbance solutions that will be increasingly needed as DRAM technology scaling continues.

Acknowledgments

We thank the anonymous reviewers of S&P 2024 and MICRO 2024 (both main submission and artifact evaluation) for the encouraging feedback. We thank the SAFARI Research Group members for valuable feedback and the stimulating scientific and intellectual environment. We acknowledge the generous gift funding provided by our industrial partners (especially Google, Huawei, Intel, Microsoft, VMware), which has been instrumental in enabling the research we have been conducting on read disturbance in DRAM since 2011 [229]. This work was also in part supported by the Google Security and Privacy Research Award, the Microsoft Swiss Joint Research Center, and the ETH Future Computing Laboratory (EFCL).

References

[1] Robert H. Dennard. Field-Effect Transistor Memory, 1968. U.S. Patent 3,387,286.
[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.
[3] Onur Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. In *DATE*, 2017.
[4] Onur Mutlu and Jeremie S Kim. RowHammer: A Retrospective. *TCAD*, 2019.
[5] Onur Mutlu, Ataberk Olgun, and A. Giray Yaglikci. Fundamentally Understanding and Solving RowHammer. In *ASP-DAC*, 2023.
[6] Apostolos P Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks. *Electronics*, 2017.
[7] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking Deterministic Signature Schemes using Fault Attacks. In *EuroS&P*, 2018.
[8] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer Attacks Over the Network and Defenses. In *USENIX ATC*, 2018.
[9] Sebastien Carre, Matthieu Desjardins, Adrien Facon, and Sylvain Guilley. OpenSSL Bellcore's Protection Helps Fault Attack. In *DSD*, 2018.
[10] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks. In *IVSW*, 2018.
[11] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. Triggering Rowhammer Hardware Faults on ARM: A Revisit. In *ASHES*, 2018.
[12] Sarani Bhattacharya and Debdeep Mukhopadhyay. Advanced Fault Attacks in Software: Exploiting the Rowhammer Bug. In *Fault Tolerant Architectures for Cryptography and Hardware Security*, pages 111–135. Springer, 2018.
[13] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. http://googleprojectzero.blogspot.com.tr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.
[14] SAFARI Research Group. RowHammer — GitHub Repository. https://github.com/CMU-SAFARI/rowhammer, 2014.
[15] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.
[16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.
[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript. arXiv:1507.06955 [cs.CR], 2016.
[18] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*, 2016.
[19] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.
[20] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security*, 2016.
[21] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as An Advanced Exploitation Vector. In *S&P*, 2016.
[22] Sarani Bhattacharya and Debdeep Mukhopadhyay. Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis. In *CHES*, 2016.
[23] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Invited: Who is the Major Threat to Tomorrow's Security? You, the Hardware Designer. In *DAC*, 2016.
[24] Rui Qiao and Mark Seaborn. A New Approach for RowHammer Attacks. In *HOST*, 2016.
[25] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory. In *USENIX Security*, 2017.
[26] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SOSP*, 2017.
[27] Misiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks. In *HOST*, 2017.
[28] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID*, 2018.
[29] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.
[30] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing Rowhammer Faults Through Network Requests. arXiv:1805.04956 [cs.CR], 2018.
[31] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *DIMVA*, 2018.
[32] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning

Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, 2018.
[33] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*, 2019.
[34] Sangwoo Ji, Youngjoo Ko, Saeyoung Oh, and Jong Kim. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In *ASIACCS*, 2019.
[35] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *USENIX Security*, 2019.
[36] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.
[37] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, 2020.
[38] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *S&P*, 2020.
[39] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms. arXiv:1912.11523 [cs.CR], 2020.
[40] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *MICRO*, 2020.
[41] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips. In *USENIX Security*, 2020.
[42] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-Sided Rowhammer Attacks from JavaScript. In *USENIX Security*, 2021.
[43] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO*, 2021.
[44] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *S&P*, 2022.
[45] M Caner Tol, Saad Islam, Berk Sunar, and Ziming Zhang. Toward Realistic Backdoor Injection Attacks on DNNs using RowHammer. arXiv:2110.07683, 2022.
[46] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *USENIX Security*, 2022.
[47] Lois Orosa, Ulrich Rührmair, A Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. SpyHammer: Using RowHammer to Remotely Spy on Temperature. arXiv:2210.04084, 2022.
[48] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *IEEE TDSC*, 2022.
[49] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating Robust DNN with Resistance to Bit-Flip based Adversarial Weight Attack. *IEEE TC*, 2022.
[50] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In *CCS*, 2022.
[51] Mengxin Zheng, Qian Lou, and Lei Jiang. TrojViT: Trojan Insertion in Vision Transformers. arXiv:2208.13049, 2022.
[52] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. *CCS*, 2022.
[53] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *S&P*, 2022.
[54] Adnan Siraj Rakin, Md Hafizul Islam Chowdhuryy, Fan Yao, and Deliang Fan. DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories. In *S&P*, 2022.
[55] Kyungbae Park, Donghyuk Yun, and Sanghyeon Baeg. Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components. *Microelectronics Reliability*, 2016.
[56] Kyungbae Park, Chulseung Lim, Donghyuk Yun, and Sanghyeon Baeg. Experiments and Root Cause Analysis for Active-Precharge Hammering Fault in DDR3 SDRAM under 3xnm Technology. *Microelectronics Reliability*, 2016.
[57] Chulseung Lim, Kyungbae Park, and Sanghyeon Baeg. Active Precharge Hammering to Monitor Displacement Damage Using High-Energy Protons in 3x-nm SDRAM. *TNS*, 2017.
[58] Seong-Wan Ryu, Kyungkyu Min, Jungho Shin, Heimi Kwon, Donghoon Nam, Taekyung Oh, Tae-Su Jang, Minsoo Yoo, Yongtaik Kim, and Sungjoo Hong. Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing. In *IEDM*, 2017.
[59] Donghyuk Yun, Myungsang Park, Chulseung Lim, and Sanghyeon Baeg. Study of TID Effects on One Row Hammering using Gamma in DDR4 SDRAMs. In *IRPS*, 2018.
[60] Thomas Yang and Xi-Wei Lin. Trap-Assisted DRAM Row Hammer Effect. *EDL*, 2019.
[61] Andrew J. Walker, Sungkwon Lee, and Dafna Beery. On DRAM RowHammer and the Physics on Insecurity. *IEEE TED*, 2021.
[62] Jeremie S. Kim, Minesh Patel, Abdullah Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques. In *ISCA*, 2020.
[63] Lois Orosa, A Giray Yağlıkçı, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses. In *MICRO*, 2021.
[64] A. Giray Yağlıkcı, Haocong Luo, Geraldo F De Oliviera, Ataberk Olgun, Minesh Patel, Jisung Park, Hasan Hassan, Jeremie S Kim, Lois Orosa, and Onur Mutlu. Understanding RowHammer Under Reduced Wordline Voltage: An Experimental Study Using Real DRAM Devices. In *DSN*, 2022.
[65] Mohammad Nasim Imtiaz Khan and Swaroop Ghosh. Analysis of Row Hammer Attack on STTRAM. In *ICCD*, 2018.
[66] S. Agarwal, H. Dixit, D. Datta, M. Tran, D. Houssameddine, D. Shum, and F. Benistant. Rowhammer for Spin Torque based Memory: Problem or not? In *INTERMAG*, 2018.
[67] Haitong Li, Hong-Yu Chen, Zhe Chen, Bing Chen, Rui Liu, Gang Qiu, Peng Huang, Feifei Zhang, Zizhen Jiang, Bin Gao, Lifeng Liu, Xiaoyan Liu, Shimeng Yu, H.-S. Philip Wong, and Jinfeng Kang. Write Disturb Analyses on Half-Selected Cells of Cross-Point RRAM Arrays. In *IRPS*, 2014.
[68] Kai Ni, Xueqing Li, Jeffrey A. Smith, Matthew Jerry, and Suman Datta. Write Disturb in Ferroelectric FETs and Its Implication for 1T-FeFET AND Memory Arrays. *IEEE EDL*, 2018.
[69] Paul R. Genssler, Victor M. van Santen, Jörg Henkel, and Hussam Amrouch. On the Reliability of FeFET On-Chip Memory. *TC*, 2022.
[70] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't Knock! Rowhammer at the Backdoor of DNN Models. In *DSN*, 2023.
[71] Hakan Aydin and Ahmet Sertbaş. Cyber Security in Industrial Control Systems (ICS): A Survey of RowHammer Vulnerability. *Applied Computer Science*, 2022.
[72] Koksal Mus, Yarkın Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS Signing Keys via Rowhammer Faults. *Cryptology ePrint Archive*, 2022.
[73] Jianxin Wang, Hongke Xu, Chaoen Xiao, Lei Zhang, and Yuzheng Zheng. Research and Implementation of Rowhammer Attack Method based on Domestic NeoKylin Operating System. In *ICFTIC*, 2022.
[74] Sam Lefforge. Reverse Engineering Post-Quantum Cryptography Schemes to Find Rowhammer Exploits. Master's thesis, University of Arkansas, 2023.
[75] Michael J Fahr. *The Effects of Side-Channel Attacks on Post-Quantum Cryptography: Influencing FrodoKEM Key Generation Using the Rowhammer Exploit.* PhD thesis, University of Arkansas, 2022.
[76] Anandpreet Kaur, Pravin Srivastav, and Bibhas Ghoshal. Work-in-Progress: DRAMaUT: DRAM Address Mapping Unveiling Tool for ARM Devices. In *CASES*, 2022.
[77] Kunbei Cai, Zhenkai Zhang, and Fan Yao. On the Feasibility of Training-time Trojan Attacks through Hardware-based Faults in Memory. In *HOST*, 2022.
[78] Dawei Li, Di Liu, Yangkun Ren, Ziyi Wang, Yu Sun, Zhenyu Guan, Qianhong Wu, and Jianwei Liu. CyberRadar: A PUF-based Detecting and Mapping Framework for Physical Devices. arXiv:2201.07597, 2022.
[79] Arman Roohi and Shaahin Angizi. Efficient Targeted Bit-Flip Attack Against the Local Binary Pattern Network. In *HOST*, 2022.
[80] Felix Staudigl, Hazem Al Indari, Daniel Schön, Dominik Sisejkovic, Farhad Merchant, Jan Moritz Joseph, Vikas Rana, Stephan Menzel, and Rainer Leupers. NeuroHammer: Inducing Bit-Flips in Memristive Crossbar Memories. In *DATE*, 2022.
[81] Li-Hsing Yang, Shin-Shan Huang, Tsai-Ling Cheng, Yi-Ching Kuo, and Jian-Jhih Kuo. Socially-Aware Collaborative Defense System against Bit-Flip Attack in Social Internet of Things and Its Online Assignment Optimization. In *ICCCN*, 2022.
[82] Saad Islam, Koksal Mus, Richa Singh, Patrick Schaumont, and Berk Sunar. Signature Correction Attack on Dilithium Signature Scheme. In *Euro S&P*, 2022.
[83] Loïc France, Florent Bruguier, Maria Mushtaq, David Novo, and Pascal Benoit. Modeling Rowhammer in the gem5 Simulator. In *CHES 2022-Conference on Cryptographic Hardware and Embedded Systems*, 2022.
[84] Anil Kurmus, Nikolas Ioannou, Matthias Neugschwandtner, Nikolaos Papandreou, and Thomas Parnell. From Random Block Corruption to Privilege Escalation: A Filesystem Attack Vector for RowHammer-like Attacks. In *USENIX WOOT*, 2017.
[85] Dawei Li, Di Liu, Yangkun Ren, Ziyi Wang, Yu Sun, Zhenyu Guan, Qianhong Wu, and Jianwei Liu. FPHammer: A Device Identification Framework based on DRAM Fingerprinting. In *TrustCom*, 2023.
[86] Chihiro Tomita, Makoto Takita, Kazuhide Fukushima, Yuto Nakano, Yoshiaki Shiraishi, and Masakatu Morii. Extracting the Secrets of OpenSSL with RAMBleed. *Sensors*, 2022.
[87] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *ISCA*, 2023.
[88] A Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizibarzoki, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoullos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA*, 2021.
[89] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.
[90] A Giray Yağlikci, Ataberk Olgun, Minesh Patel, Haocong Luo, Hasan Hassan, Lois Orosa, Oğuz Ergin, and Onur Mutlu. HiRA: Hidden Row Activation for Reducing Refresh Latency of Off-the-Shelf DRAM Chips. In *MICRO*, 2022.

[91] Arthur De Graauw. Ancient Port Structures. Parallels between the Ancient and the Modern. *Méditerranée. Revue géographique des pays méditerranéens/Journal of Mediterranean geography*, 2022.

[92] Anish Saxena, Gururaj Saileshwar, Prashant J. Nair, and Moinuddin Qureshi. AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime. In *MICRO*, 2022.

[93] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J Nair. Hydra: Enabling Low-Overhead Mitigation of Row-Hammer at Ultra-Low Thresholds via Hybrid Tracking. In *ISCA*, 2022.

[94] Eojin Lee, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: Time Window Counter Based Row Refresh to Prevent Row-Hammering. *IEEE CAL*, 2018.

[95] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *S&P*, 2023.

[96] JEDEC. *JESD79-5: DDR5 SDRAM Standard*, 2020.

[97] JEDEC. *JESD79-5c: DDR5 SDRAM Standard*, 2024.

[98] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters. In *ISCA*, 2019.

[99] JEDEC. *JESD79-5: DDR5 SDRAM Standard*, 2020.

[100] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *ISCA*, 1981.

[101] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC*, pages 1216–1225, 2012.

[102] Synopsys, Inc. Synopsys Design Compiler. https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[103] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. https://www.hpl.hp.com/research/cacti/, 2017.

[104] JEDEC. *JESD79-4C: DDR4 SDRAM Standard*, 2020.

[105] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, , and Onur Mutlu. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator, 2023.

[106] SAFARI Research Group. Ramulator V2.0. https://github.com/CMU-SAFARI/ramulator2.

[107] Standard Performance Evaluation Corp. SPEC CPU 2006. http://www.spec.org/cpu2006/.

[108] Standard Performance Evaluation Corp. SPEC CPU2017 Benchmarks. http://www.spec.org/cpu2017/.

[109] *Transaction Processing Performance Council.*

[110] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. MediaBench II Video: Expediting the next Generation of Video Systems Research. *Microprocess. Microsyst.*, 2009.

[111] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[112] Onur Mutlu and Jeremie Kim. RowHammer: A Retrospective. *IEEE TCAD Special Issue on Top Picks in Hardware and Embedded Security*, 2019.

[113] Haocong Luo, Ataberk Olgun, A Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. RowPress Vulnerability in Modern DRAM Chips. *IEEE Micro*, 2024.

[114] Michael Redeker, Bruce F Cockburn, and Duncan G Elliott. An Investigation into Crosstalk Noise in DRAM Structures. In *MTDT*, 2002.

[115] Kyungbae Park, Sanghyeon Baeg, ShiJie Wen, and Richard Wong. Active-Precharge Hammering on a Row-Induced Failure in DDR3 SDRAMs Under 3x nm Technology. In *IIRW*, 2014.

[116] Chia Yang, Chen Kang Wei, Yu Jing Chang, Tieh Chiang Wu, Hsiu Pin Chen, and Chao Sung Lai. Suppression of RowHammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology. *TDMR*, 2016.

[117] Chia-Ming Yang, Chen-Kang Wei, Hsiu-Pin Chen, Jian-Shing Luo, Yu Jing Chang, Tieh-Chiang Wu, and Chao-Sung Lai. Scanning Spreading Resistance Microscopy for Doping Profile in Saddle-Fin Devices. *IEEE Transactions on Nanotechnology*, 2017.

[118] Chulseung Lim, Kyungbae Park, Geunyong Bak, Donghyuk Yun, Myungsang Park, Sanghyeon Baeg, Shi-Jie Wen, and Richard Wong. Study of Proton Radiation Effect to Row Hammer Fault in DDR4 SDRAMs. *Microelectronics Reliability*, 2018.

[119] S. K. Gautam, S. K. Manhas, Arvind Kumar, Mahendra Pakala, and Yiehm Ellie. Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM. *IEEE TED*, 2019.

[120] Yichen Jiang, Huifeng Zhu, Dean Sullivan, Xiaolong Guo, Xuan Zhang, and Yier Jin. Quantifying RowHammer Vulnerability for DRAM Security. In *DAC*, 2021.

[121] Wei He, Zhi Zhang, Yueqiang Cheng, Wenhao Wang, Wei Song, Yansong Gao, Qifei Zhang, Kang Li, Dongxi Liu, and Surya Nepal. WhistleBlower: A System-level Empirical Study on RowHammer. *IEEE Transactions on Computers*, 2023.

[122] Sanghyeon Baeg, Donghyuk Yun, Myungsun Chun, and Shi-Jie Wen. Estimation of the Trap Energy Characteristics of Row Hammer-Affected Cells in Gamma-Irradiated DDR4 DRAM. *IEEE Transactions on Nuclear Science*, 2022.

[123] Onur Mutlu. RowHammer. https://people.inf.ethz.ch/omutlu/pub/onur-Rowhammer-TopPicksinHardwareEmbeddedSecurity-November-8-2018.pdf, 2018.

[124] Ataberk Olgun, Majd Osseiran, Abdullah Giray Yaglikci, Yahya Can Tugrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. An Experimental Analysis of RowHammer in HBM2 DRAM Chips. In *DSN Disrupt*, 2023.

[125] Ataberk Olgun, Hasan Hassan, A. Giray Yağlıkcı, Yahya Can Tuğrul, Lois Orosa, Haocong Luo, Minesh Patel, Ergin Oğuz, and Onur Mutlu. DRAM Bender: An Extensible and Versatile FPGA-based Infrastructure to Easily Test State-of-the-art DRAM Chips. *TCAD*, 2023.

[126] Longda Zhou, Jie Li, Zheng Qiao, Pengpeng Ren, Zixuan Sun, Jianping Wang, Blacksmith Wu, Zhigang Ji, Runsheng Wang, Kanyu Cao, and Ru Huang. Double-sided Row Hammer Effect in Sub-20 nm DRAM: Physical Mechanism, Key Features and Mitigation. In *IRPS*, 2023.

[127] Zhenrong Lang, Patrick Jattke, Michele Marazzi, and Kaveh Razavi. Blaster: Characterizing the blast radius of rowhammer. In *3rd Workshop on DRAM Security (DRAMSec) co-located with ISCA 2023*. ETH Zurich, 2023.

[128] Jie Li, Longda Zhou, Sheng Ye, Zheng Qiao, and Zhigang Ji. Understanding the Competitive Interaction in Leakage Mechanisms for Effective Row Hammer Mitigation in Sub-20nm DRAM. *IEEE EDL*, 2024.

[129] Longda Zhou, Jie Li, Pengpeng Ren, Sheng Ye, Da Wang, Zheng Qiao, and Zhigang Ji. Understanding the Physical Mechanism of RowPress at the Device-Level in Sub-20 nm DRAM. In *IRPS*, 2024.

[130] Longda Zhou, Sheng Ye, Runsheng Wang, and Zhigang Ji. Unveiling RowPress in Sub-20 nm DRAM Through Comparative Analysis With Row Hammer: From Leakage Mechanisms to Key Features. *IEEE TED*, 2024.

[131] Haocong Luo, Ismail Emir Yüksel, Ataberk Olgun, A Giray Yağlıkçı, Mohammad Sadrosadati, and Onur Mutlu. An Experimental Characterization of Combined RowHammer and RowPress Read Disturbance in Modern DRAM Chips. *DSN Disrupt*, 2024.

[132] Apple Inc. About the Security Content of Mac EFI Security Update 2015-001. https://support.apple.com/en-us/HT204934, 2015.

[133] Hewlett-Packard Enterprise. HP Moonshot Component Pack Version 2015.05.0. http://h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/component-pack/index.aspx, 2015.

[134] Lenovo. Row Hammer Privilege Escalation. https://support.lenovo.com/us/en/product_security/row_hammer, 2015.

[135] Zvika Greenfield and Tomer Levy. Throttling Support for Row-Hammer Counters, 2016.

[136] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural Support for Mitigating Row Hammering in DRAM Memories. *CAL*, 2014.

[137] Barbara Aichinger. DDR Memory Errors Caused by Row Hammer. In *HPEC*, 2015.

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

[139] K. Bains et al. Row Hammer Refresh Command. US Patents: 9,117,544 9,236,110 10,210,925, 2015.

[140] Kuljit Bains, John Halbert, Christopher Mozak, Theodore Schoenborn, and Zvika Greenfield. Row Hammer Refresh Command, 2015.

[141] Kuljit S Bains and John B Halbert. Distributed Row Hammer Tracking, 2016.

[142] Kuljit S Bains and John B Halbert. Row Hammer Monitoring Based on Stored Row Hammer Threshold Value. US Patent: 10,083,737, 2016.

[143] H. Gomez, A. Amaya, and E. Roa. DRAM row-hammer attack reduction using dummy cells. In *NORCAS*, 2016.

[144] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. Making DRAM Stronger Against Row Hammering. In *DAC*, 2017.

[145] S. M. Seyedzadeh, A. K. Jones, and R. Melhem. Mitigating Wordline Crosstalk Using Adaptive Trees of Counters. In *ISCA*, 2018.

[146] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Stopping Microarchitectural Attacks Before Execution. *IACR Cryptology*, 2016.

[147] Jung Min You and Joon-Sung Yang. MRLoc: Mitigating Row-Hammering Based on Memory Locality. In *DAC*, 2019.

[148] A. Giray Yağlıkçı, Jeremie S. Kim, Fabrice Devaux, and Onur Mutlu. Security Analysis of the Silver Bullet Technique for RowHammer Prevention. arXiv:2106.07084, 2021.

[149] Ingab Kang, Eojin Lee, and Jung Ho Ahn. CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention. *IEEE Access*, 2020.

[150] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation Between Aggressor and Victim Rows. In *ASPLOS*, 2022.

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

[152] Saru Vig, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Siew-Kei Lam. Rapid Detection of Rowhammer Attacks Using Dynamic Skewed Hash Tree. In *HASP*, 2018.

[153] H. Hassan, M. Patel, J. S. Kim, A. G. Yağlıkçı, N. Vijaykumar, N. Mansouri Ghiasi, S. Ghose, and O. Mutlu. CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In *ISCA*, 2019.

[154] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In *HPCA*, 2022.

[155] Gyu-Hyeon Lee, Seongmin Na, Ilkwon Byun, Dongmoon Min, and Jangwoo Kim. CryoGuard: A Near Refresh-Free Robust DRAM Design for Cryogenic Computing. In *ISCA*, 2021.

[156] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *S&P*, 2022.

[157] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal,

Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. SoftTRR: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh. In *USENIX ATC*, 2022.
[158] Biresh Kumar Joardar, Tyler K Bletsch, and Krishnendu Chakrabarty. Learning to Mitigate RowHammer Attacks. In *DATE*, 2022.
[159] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI: Rowhammer–Cryptographic Security and Integrity against Rowhammer (to appear). In *S&P*, 2023.
[160] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. Revisiting Residue Codes for Modern Memories. In *MICRO*, 2022.
[161] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. EVAX: Towards a Practical, Pro-active & Adaptive Architecture for High Performance & Security. In *MICRO*, 2022.
[162] Amir Naseredini, Martin Berger, Matteo Sammartino, and Shale Xiong. ALARM: Active LeArning of Rowhammer Mitigations. https://users.sussex.ac.uk/~mfb21/rh-draft.pdf, 2022.
[163] Biresh Kumar Joardar, Tyler K. Bletsch, and Krishnendu Chakrabarty. Machine Learning-based Rowhammer Mitigation. *TCAD*, 2022.
[164] Hasan Hassan, Ataberk Olgun, A Giray Yaglikci, Haocong Luo, and Onur Mutlu. A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous in-DRAM Maintenance Operations. arXiv:2207.13358, 2022.
[165] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. Leveraging EM Side-Channel Information to Detect Rowhammer Attacks. In *S&P*, 2020.
[166] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *HotOS*, 2021.
[167] Fabrice Devaux and Renaud Ayrignac. Method and Circuit for Protecting a DRAM Memory Device from the Row Hammer Effect. US Patent: 10,885,966, 2021.
[168] Jin Han, Jungsik Kim, Dafna Beery, K Deniz Bozdag, Peter Cuevas, Amitay Levi, Irwin Tain, Khai Tran, Andrew J Walker, Senthil Vadakupudhu Palayam, et al. Surround Gate Transistor With Epitaxially Grown Si Pillar and Simulation Study on Soft Error and Rowhammer Tolerance for DRAM. *TED*, 2021.
[169] Ali Fakhrzadehgan, Yale N. Patt, Prashant J. Nair, and Moinuddin K. Qureshi. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA*, 2022.
[170] Stefan Saroiu, Alec Wolman, and Lucian Cojocar. The Price of Secrecy: How Hiding Internal DRAM Topologies Hurts Rowhammer Defenses. In *IRPS*, 2022.
[171] Stefan Saroiu and Alec Wolman. How to Configure Row-Sampling-Based Rowhammer Defenses. *DRAMSec*, 2022.
[172] Kevin Loughlin, Stefan Saroiu, Alec Wolman, Yatin A. Manerkar, and Baris Kasikci. MOESI-Prime: Preventing Coherence-Induced Hammering in Commodity Workloads. In *ISCA*, 2022.
[173] Ranyang Zhou, Sepehr Tabrizchi, Arman Roohi, and Shaahin Angizi. LT-PIM: An LUT-Based Processing-in-DRAM Architecture With RowHammer Self-Tracking. *IEEE CAL*, 2022.
[174] Seungki Hong, Dongha Kim, Jaehyung Lee, Reum Oh, Changsik Yoo, Sangjoon Hwang, and Jooyoung Lee. DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm. arXiv:2302.03591, 2023.
[175] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In *NDSS*, 2023.
[176] Sonia Sharma, Debdeep Sanyal, Arpit Mukhopadhyay, and Ramij Hasan Shaik. A Review on Study of Defects of DRAM-RowHammer and Its Mitigation. *Journal For Basic Sciences*, 2022.
[177] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J Nair. Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems. In *HPCA*, 2023.
[178] Jin Hyo Park, Su Yeon Kim, Dong Young Kim, Geon Kim, Je Won Park, Sunyong Yoo, Young-Woo Lee, and Myoung Jin Lee. RowHammer Reduction Using a Buried Insulator in a Buried Channel Array Transistor. *IEEE Transactions on Electron Devices*, 2022.
[179] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *HPCA*, 2023.
[180] Woongrae Kim, Chulmoon Jung, Seongnyuh Yoo, Duckhwa Hong, Jeongjin Hwang, Jungmin Yoon, Ohyong Jung, Joonwoo Choi, Sanga Hyun, Mankeun Kang, Sangho Lee, Dohong Kim, Sanghyun Ku, Donhyun Choi, Nogeun Joo, Sangwoo Yoon, Junseok Noh, Byeongyong Go, Cheolhoe Kim, Sunil Hwang, Mihyun Hwang, Min Seol-Yi, Hyungmin Kim, Sanghyuk Heo, Yeonsu Jang, Kyoungchul Jang, Shinho Chu, Yoonna Oh, Kwidong Kim, Junghyun Kim, Soohwan Kim, Jeongtae Hwang, Sangil Park, Junphyo Lee, Inchul Jeong, Joohwan Cho, and Jonghwan Kim. A 1.1 V 16Gb DDR5 DRAM with Probabilistic-Aggressor Tracking, Refresh-Management Functionality, Per-Row Hammer Tracking, a Multi-Step Precharge, and Core-Bias Modulation for Security and Reliability Enhancement. In *ISSCC*, 2023.
[181] C Gude Ramarao, K Tejesh Kumar, G Ujjinappa, and B Vasu Deva Naidu. Defending SoCs with FPGAs from Rowhammer Attacks. *Material Science*, 2023.
[182] Krishnendu Guha and Amlan Chakrabarti. Criticality based Reliability from Rowhammer Attacks in Multi-User-Multi-FPGA Platform. In *VLSID*, 2022.
[183] Loïc France, Florent Bruguier, David Novo, Maria Mushtaq, and Pascal Benoit. Reducing the Silicon Area Overhead of Counter-Based Rowhammer Mitigations. In *18th CryptArchi Workshop*, 2022.
[184] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cojocar. Panopticon: A Complete In-DRAM Rowhammer Mitigation. In *DRAMSec*, 2021.
[185] Shuhei Enomoto, Hiroki Kuzuno, and Hiroshi Yamada. Efficient Protection Mechanism for CPU Cache Flush Instruction Based Attacks. *IEICE Transactions on Information and Systems*, 2022.
[186] Kerem Arıkan, Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Salvatore Pontarelli, Giuseppe Bianchi, Oğuz Ergin, and Marco Ottavi. Processor security: Detecting microarchitectural attacks via count-min sketches. *VLSI*, 2022.
[187] Anish Saxena, Gururaj Saileshwar, Jonas Juffinger, Andreas Kogler, Daniel Gruss, and Moinuddin Qureshi. PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In *DSN*, 2023.
[188] Ranyang Zhou, Sabbir Ahmed, Adnan Siraj Rakin, and Shaahin Angizi. DNN-Defender: An in-DRAM Deep Neural Network Defense Mechanism for Adversarial Weight Attack. arXiv:2305.08034, 2023.
[189] F. Nisa Bostancı, Ismail Emir Yüksel, Ataberk Olgun, Konstantinos Kanellopoulos, Yahya Can Tugrul, A. Giray Yaglıkçı, Mohammad Sadrosadati, and Onur Mutlu. CoMeT: Count-Min-Sketch-Based Row Tracking to Mitigate RowHammer at Low Cost. In *HPCA*, 2024.
[190] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In *NDSS*, 2023.
[191] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *IEEE CAL*, 2017.
[192] Seyed Mohammad Seyedzadeh, Donald Kline Jr, Alex K Jones, and Rami Melhem. Mitigating Bitline Crosstalk Noise in DRAM Memories. In *MEMSYS*, 2017.
[193] Victor van der Veen, Pankaj Deshmukh, Behnam Dashtipour, David Hartley, and Mosaddiq Saifuddin. Dynamic Random Access Memory (DRAM) Row Hammering Mitigation. US Patent App. 17/890,022, 2024.
[194] Victor van der Veen, Pankaj Deshmukh, Behnam Dashtipour, and David Hartley. Dynamic Rowhammer Management. US Patent App. 17/940,430, 2024.
[195] Akash Verma, Victor van der Veen, Joona Kannisto, and Marcel Selhorst. Defense Against Row Hammer Attacks. US Patent App. 17/842,606, 2023.
[196] Ataberk Olgun, Yahya Can Tugrul, F. Nisa Bostancı, Ismail Emir Yüksel, Haocong Luo, Steve Rhyner, A. Giray Yaglıkçı, Geraldo F. Oliveira, and Onur Mutlu. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation. In *USENIX Security*, 2024.
[197] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *CAL*, 2017.
[198] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 2008.
[199] Allan Snavely and Dean M Tullsen. Symbiotic Jobscheduling for A Simultaneous Multithreaded Processor. In *ASPLOS*, 2000.
[200] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan kaufmann, 2017.
[201] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In *PACT*, 2015.
[202] Jayadev Misra and David Gries. Finding Repeated Elements. *Science of Computer Programming*, 1982.
[203] Hasan Hassan, Ataberk Olgun, A Giray Yaglikci, Haocong Luo, and Onur Mutlu. A Case for Self-Managing DRAM Chips: Improving Performance, Efficiency, Reliability, and Security via Autonomous in-DRAM Maintenance Operations. arXiv:2207.13358 [cs.AR], 2022.
[204] Zhaogeng Li, Jun Bi, Sen Wang, and Xiaoke Jiang. Compression of Pending Interest Table with Bloom Filter in Content Centric Network. In *CFI*, 2012.
[205] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *TACO*, 2016.
[206] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.
[207] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *ICCD*, 2014.
[208] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.
[209] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *International Symposium on Microarchitecture (MICRO-39)*, 2006.
[210] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.
[211] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.
[212] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.
[213] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS*, 2010.
[214] Intel Inc. 3rd Gen Intel Xeon Scalable Processors. https://www.intel.com/content/dam/www/public/us/en/documents/a1171486-icelake-productbrief-updates-r1v2.pdf.
[215] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. Application-Aware Prioritization Mechanisms for On-Chip Networks. In *MICRO*, 2009.
[216] Scott Rixner et al. Memory Access Scheduling. In *ISCA*, 2000.
[217] William K Zuravleff and Timothy Robinson. Controller for a Synchronous DRAM

That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order, 1997.

[218] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era. In *MICRO*, 2011.

[219] Woongrae Kim, Chulmoon Jung, Seongnyuh Yoo, Duckhwa Hong, Jeongjin Hwang, Jungmin Yoon, Ohyong Jung, Joonwoo Choi, Sanga Hyun, Mankeun Kang, et al. A 1.1 v 16gb ddr5 dram with probabilistic-aggressor tracking, refresh-management functionality, per-row hammer tracking, a multi-step precharge, and core-bias modulation for security and reliability enhancement. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 1–3. IEEE, 2023.

[220] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Oğuz Ergin, and Onur Mutlu. Understanding the Security Benefits and Overheads of Emerging Industry Solutions to DRAM Read Disturbance. *DRAMSec*, 2024.

[221] SAFARI Research Group. BlockHammer — GitHub Repository. https://github.com/CMU-SAFARI/blockhammer, 2021.

[222] Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security*, 2007.

[223] Kevin KaiWei Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu. HAT: Heterogeneous Adaptive Throttling for On-Chip Networks. In *SBAC-PAD*, 2012.

[224] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *ISCA*, 2011.

[225] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A Joao, Onur Mutlu, and Yale N Patt. Parallel Application Memory Scheduling. In *MICRO*, 2011.

[226] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via Source Throttling: A Configurable and High Performance Fairness Substrate for Multi Core Memory Systems. In *ASPLOS*, 2010.

[227] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. Prefetch-Aware DRAM Controllers. In *MICRO*, 2008.

[228] George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu. Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need? In *HOTNETS*, 2010.

[229] Onur Mutlu. Retrospective: Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *Retrospective Issue for ISCA-50*, 2023.

A. Artifact Appendix

A.1. Abstract

Our artifact contains the data, source code, and scripts needed to reproduce our results. We provide: 1) the source code of our simulation infrastructure based on Ramulator2 and 2) all evaluated memory access traces and all major evaluation results. We provide Bash and Python scripts to analyze and plot the results automatically.

A.2. Artifact Check-list (meta-information)

Parameter	Value
Program	C++ program Python3 scripts Shell scripts
Compilation	C++ compiler with c++20 features
Run-time environment	Ubuntu 20.04 (or similar) Linux C++20 build toolchain (tested with GCC 10) Python 3.10+ Podman 4.5+ Git
Metrics	Weighted speedup Maximum slowdown DRAM energy
Experiment workflow	Perform simulations, aggregate results, and run analysis scripts on the result
Experiment customization	Possible. See §A.6
Disk space requirement	≈ 30GiB
Workflow preparation time	≈ 30 minutes
Experiment completion time	≈ 2 days (on a compute cluster with 250 cores)
Publicly available?	Benchmarks (https://zenodo.org/records/13293692) Zenodo (https://zenodo.org/record/13638017) GitHub (https://github.com/CMU-SAFARI/BreakHammer)
Code licences	MIT

A.3. Description

We highly recommend using Slurm with a cluster that can run experiments in bulk.

A.3.1. How To Access. Source code and scripts are available at `https://github.com/CMU-SAFARI/BreakHammer`.

A.3.2. Hardware Dependencies. We recommend using a PC with 32 GiB of main memory. Approximately 30 GiB of disk space is needed to store intermediate and final evaluation results.

A.3.3. Software Dependencies.

- GNU Make, CMake 3.20+
- C++20 build toolchain (tested with GCC 10)
- Python 3.9+
- pip packages: matplotlib, pandas, seaborn, pyyaml, wget, and scipy
- Ubuntu 22.04
- (Optional) Slurm 20+
- (Optional) Podman 4.5+

A.3.4. Benchmarks. We use workload memory traces collected from SPEC2006, SPEC2017, TPC, MediaBench, and YCSB benchmark suites. These traces are available at `https://zenodo.org/records/13293692`. Install scripts will download and extract the traces.

A.4. Installation

The following steps will download and prepare the repository for the main experiments:

1. Clone the git repository.

```
$ git clone git@github.com:CMU-SAFARI/BreakHammer.git
```

2. (Optional) Build the Podman container to run the scripts.

```
$ podman build . -t breakhammer_artifact
```

The following command runs a script using the container:

```
$ podman run --rm -v $PWD:/app \
   breakhammer_artifact <script>
```

3. Install Python dependencies, compile Ramulator2, download workload traces, and run a small sanity check.

```
$ ./run_simple_test.sh
```

A.5. Evaluation and Expected Results

Claim 1 (C1). RowHammer mitigation mechanisms induce increasing performance overheads as the RowHammer threshold decreases. This property is proven by evaluating the performance of the state-of-the-art RowHammer mitigation mechanisms on benign workloads (E1) as described in §3 whose results are illustrated in Fig. 2.

Claim 2 (C2). BreakHammer improves system performance and DRAM energy overheads of state-of-the-art RowHammer mitigation mechanisms by detecting and stopping attackers that trigger many preventive actions. This property is proven by evaluating the performance and DRAM energy of the state-of-the-art RowHammer mitigation mechanisms on workloads that include an attacker 1) when paired with BreakHammer and 2) without BreakHammer (E2) as described in §8.1 whose results are illustrated in Figs. 6, 8, 10, 12, and 18.

Claim 3 (C3). BreakHammer does *not* degrade system performance or DRAM energy when all applications are benign. This property is proven by evaluating the performance and DRAM energy of the state-of-the-art RowHammer mitigation mechanisms on benign workloads 1) when paired with BreakHammer and 2) without BreakHammer (E3) as described in §8.2 whose results are illustrated in Figs. 13, 15, and 17.

Experiments (E1, E2, and E3). [Ramulator2 simulations] [10 human-minutes + 40 compute-hours (assuming ∼600 Ramulator2 simulations run in parallel) + 30GiB disk]

We prove our claims in two steps: 1) Execute Ramulator2 simulations to generate data supporting C1, C2, and C3. 2) Plot all figures that prove C1, C2 and C3.

1. Launch all Ramulator2 simulation jobs.

```
$ ./run_with_personalcomputer.sh
(or ./run_with_slurm.sh if Slurm is available)
```

2. Wait for the simulations to end. The following displays the status and generates scripts to restart failed runs:

```
$ ./check_run_status.sh
```

3. Parse simulation results and collect statistics.

```
$ ./parse_results.sh
```

4. Generate all figures that support C1, C2, and C3.

```
$ ./plot_all_figures.sh
```

A.6. Experiment Customization

Our scripts provide easy configuration of the 1) evaluated RowHammer mitigation mechanisms, 2) tested RowHammer thresholds, 3) simulation duration, and 4) simulated workload combinations. The run parameters are configurable in `scripts/run_config.py` with 1) *mitigation_list*, 2) *tRH_list*, and 3) *NUM_EXPECTED_INSTS* or *NUM_MAX_CYCLES*, respectively. Simulated attacker and benign workload combinations can be updated in `mixes/microattack.mix` and `mixes/microbenign.mix`, respectively.

A.7. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae