DAPPER: A Performance-Attack-Resilient Tracker for RowHammer Defense

Jeonghyun Woo and Prashant J. Nair
Department of Electrical and Computer Engineering
The University of British Columbia
{jhwoo36, prashantnair}@ece.ubc.ca

***Abstract*—RowHammer vulnerabilities pose a significant threat to modern DRAM-based systems, where rapid activation of DRAM rows can induce bit-flips in neighboring rows. To mitigate this, state-of-the-art host-side RowHammer mitigations typically rely on shared counters or tracking structures. While these optimizations benefit benign applications, they are vulnerable to Performance Attacks (*Perf-Attacks*), where adversaries exploit shared structures to reduce DRAM bandwidth for co-running benign applications by increasing DRAM accesses for RowHammer counters or triggering repetitive refreshes required for the early reset of structures, significantly degrading performance.**

In this paper, we propose secure hashing mechanisms to thwart adversarial attempts to capture the mapping of shared structures. We propose DAPPER, a novel low-cost tracker resilient to *Perf-Attacks* even at ultra-low RowHammer thresholds. We first present a secure hashing template in the form of DAPPER-S. We then develop DAPPER-H, an enhanced version of DAPPER-S, incorporating double-hashing, novel reset strategies, and mitigative refresh techniques. Our security analysis demonstrates the effectiveness of DAPPER-H against both RowHammer and *Perf-Attacks*. Experiments with 57 workloads from SPEC2006, SPEC2017, TPC, Hadoop, MediaBench, and YCSB show that, even at an ultra-low RowHammer threshold of 500, DAPPER-H incurs only a 0.9% slowdown in the presence of *Perf-Attacks* while using only 96KB of SRAM per 32GB of DRAM memory.

I. INTRODUCTION

Modern Dynamic Random Access Memories (DRAM) face severe security issues due to technology scaling. DRAM cells store data as charge, and at smaller technology nodes, rapid accesses can induce data leakage from neighboring cells, resulting in bit-flips that adversaries can exploit to manipulate data [16], [22], [30], [62]. This vulnerability, known as RowHammer (RH) [28], occurs when rapidly accessed rows (aggressor rows) cause bit-flips in neighboring rows (victim rows). One approach to addressing RH involves integrating low-cost trackers into the host-side memory controller [4], [45], [50], [60]. These trackers enable targeted mitigative refreshes to victim rows. However, while effective for both RH attacks and benign applications, these trackers remain vulnerable to Performance Attacks (*Perf-Attacks*). This paper aims to develop a host-side low-cost RH tracker that is resilient against *Perf-Attacks* even at ultra-low RH thresholds (N_{RH}).

RH mitigations typically aim to refresh victim rows before adversarial activations reach the RH threshold (N_{RH}), the minimum number of activations required to induce bit-flips. Over the last decade, N_{RH} has sharply declined from around 70K [28] in 2014 to 4.5K [23] in 2020, increasing the number of vulnerable rows and escalating hardware tracking costs. Advanced trackers like Hydra [50], ABACUS [45], and CoMeT [4] reduce these costs by grouping rows and tracking collective activation counts. Alternatively, START [60] leverages the Last-Level Cache (LLC) to store RH counts. These strategies optimize performance at ultra-low N_{RH} (e.g., $N_{RH} \leq 1K$) and prevent RH under malicious attacks.

However, grouping rows or using the LLC for tracking makes these mitigations vulnerable to *Perf-Attacks*. Adversaries can force misses in RH counters or inflate group activation counts, leading to extra DRAM accesses and redundant refreshes that severely degrade the performance of co-running applications. These *Perf-Attacks* exploit RH mitigation mechanisms and are more severe than attacks like cache thrashing. Figure 1 shows that, at N_{RH} of 500, tailored *RH-Tracker-based Perf-Attacks* cause slowdowns of 60% to 90% across 57 workloads from SPEC2006 [11], SPEC2017 [66], TPC [69], Hadoop [13], MediaBench [14], and YCSB [10], whereas cache thrashing attacks result in only a 40% slowdown. Ideally, we aim for a tracker that is resilient against *Perf-Attacks*. This paper proposes DAPPER, a *Perf-Attack*-resilient RH tracker that minimizes performance impact while providing secure defense against *Perf-Attacks*.

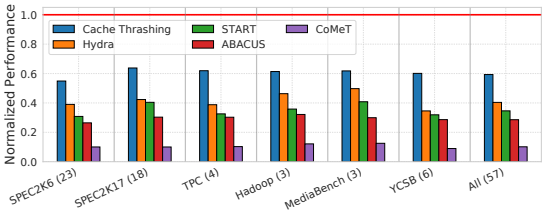


Fig. 1. Normalized performance of state-of-the-art host-side RowHammer (RH) mitigations [4], [45], [50], [60] at the RH threshold of 500, evaluated under *RH-Tracker-based* Performance Attacks (*Perf-Attack*) and cache thrashing attacks on a dual-channel, dual-rank, DDR5 64GB system. Our experiments with 57 workloads demonstrate that a single tailored *RH-Tracker-based Perf-Attack* application can reduce the performance of co-running applications by up to 90%, while cache thrashing attacks degrade them by 40% on average. This paper aims to develop a cost-effective *Perf-Attack*-resistant RH tracker.

To understand the design of DAPPER, it is crucial to decipher the hardware mechanisms exploited by *Perf-Attacks*.

- **Triggering Additional RH Counter Accesses**: Hydra [50] uses shared group counters to track multiple rows with a single counter, reducing storage overhead. Hydra switches to per-row tracking for improved performance when a group counter reaches its thresholds. RH counters are stored in a reserved DRAM area, with a subset cached in the Row

Counter Cache (RCC). When the RCC is full, and a miss occurs, Hydra evicts a stored counter and fetches a new one from DRAM. Similarly, START [60] stores individual RH counters directly in the LLC without grouping. Both Hydra's RCC and START's LLC can be overwhelmed by a tailored access pattern, leading to frequent evictions and counter fetches from DRAM. These repeated operations significantly degrade DRAM bandwidth and introduce substantial performance overhead for co-running applications.

- **Triggering Redundant Refreshes**: CoMeT [4] uses a Count-Min Sketch-based tracker to share counters across rows, reducing storage overhead. However, due to counter sharing, it cannot reset counters after performing a mitigative refresh. CoMeT employs the Recent Aggressor Table (RAT) to track and selectively reset recently mitigated rows to address this. Performance degradation occurs when the number of aggressor rows exceeds the RAT's capacity, and those rows are frequently activated. Similarly, ABACUS [45] uses a Misra-Gries (MG) tracker shared across all banks, with its size designed to handle the maximum number of aggressor rows in a single bank during a refresh interval. However, the MG tracker's spillover counter can overflow under adversarial patterns. CoMeT and ABACUS implement early preventive refreshes that mitigate this problem by refreshing all DRAM rows and resetting the structure, but an attacker can exploit these mechanisms to reduce DRAM bandwidth and cause slowdowns.

To address these concerns, we introduce two versions of DAPPER, each designed to be built on top of the other.

1. DAPPER-S: We propose DAPPER-S as a simple mechanism to prevent RH counter access and mitigate *Perf-Attacks*. DAPPER-S groups rows and assigns dedicated counters, called row group counters (RGCs), to each group. When an RGC reaches the Mitigation threshold (N_M), which is half of N_{RH}, it refreshes victim rows and resets the counter. Unlike previous approaches [50], [60], DAPPER-S stores all RGCs in an SRAM-based table within the memory controller, avoiding main memory storage. Rows are randomly assigned to groups using a secure hash with periodically updated keys, reducing DRAM accesses and preventing attackers from targeting specific rows. This randomization minimizes the risk of overwhelming an RGC. Hash keys are refreshed every 12μs to effectively mitigate *Perf-Attacks* with some overhead. However, DAPPER-S remains vulnerable to two *Mapping-Agnostic* attacks: (1) a 'streaming' attack, where all rows are activated, overwhelming all RGCs and triggering redundant refreshes, and (2) a 'refresh attack' that repeatedly activates one or more rows, causing mitigative refreshes for all rows sharing the same RGC. Despite this, DAPPER-S is a robust foundation for further enhancements to strengthen the tracker.

2. DAPPER-H: We develop DAPPER-H to address the limitations of DAPPER-S. DAPPER-H uses two SRAM tables for storing RGCs, each with a separate secure hash. The mitigation process occurs only when both RGCs reach the N_M, which is also half of N_{RH}, similar to DAPPER-S. This means that an attacker must identify a row's RGCs in both tables to overwhelm the counts. Additionally, DAPPER-H integrates a per-bank bit-vector for one of the RGC tables. This bit-vector acts as a filter, effectively managing scenarios where rows within the same group are activated across different banks. If the corresponding bit is unset, indicating that no row within this bank has been previously activated, DAPPER-H sets the bit without incrementing the RGC. Combined with the periodic rehashing of RGCs at each refresh window (tREFW), the design effectively thwarts *Mapping-Capturing* attacks, which aim to capture even a single pair of row group mappings and exploit it, thereby avoiding performance degradation.

Contributions: This paper makes four key contributions

1) It demonstrates tailored *RH-Tracker-based Perf-Attacks* for state-of-the-art low-cost trackers. These attacks exploit shared structures like tracking counters or the LLC, which is used to minimize slowdowns in benign workloads.
2) It identifies the causes of *Perf-Attacks* in scalable RH trackers: additional accesses to in-DRAM RH counters and additional repetitive refreshes to DRAM for early structure resets. Proposes DAPPER to counter this issue.
3) It introduces a naive solution, DAPPER-S, which randomizes rows mapped into shared counters for tracking using a single secure hash function. DAPPER-S makes it challenging for adversaries to learn the mapping and orchestrate *Perf-Attacks*. Although vulnerable to mapping-agnostic streaming and refresh attacks, DAPPER-S provides a foundation to strengthen the tracker.
4) It proposes DAPPER-H, an enhanced version of DAPPER-S, which uses double-hashing, a novel reset method, and a per-bank bit-vector. This enhancement makes it extremely challenging for adversaries to understand the mappings into the shared counters. Regardless of the access pattern, DAPPER-H is generally resilient against *Perf-Attack* in 99.99% of the refresh intervals and effectively mitigates mapping-agnostic *Perf-Attacks*.

Our experiments show that even at N_{RH} of 500, DAPPER-H incurs only a 0.9% slowdown even under active *Perf-Attacks* while using only 96KB of SRAM per 32GB of memory.

II. BACKGROUND

A. Memory Organization and Timing Parameters

A DRAM-based memory system is organized hierarchically. A Memory controller (MC) manages one or more channels, with each channel containing ranks composed of banks that operate in parallel. Each bank consists of a two-dimensional array of DRAM cells arranged in rows and columns. To access data, the MC issues an ACTIVATE (ACT) command to load data into the row buffer. Switching to a different row within the same bank requires a PRECHARGE (PRE) command to close the current row, followed by another ACT. Due to charge leakage, DRAM performs periodic refreshes (REFs) within a refresh window (tREFW), typically 32ms in DDR5 systems [38], [39]. During tREFW, the MC issues 8K auto-refresh commands at 3.9μs interval (tREFI), refreshing multiple rows in each bank during refresh cycle time (tRFC) [54].

The row cycle time (tRC) sets the minimum interval between ACTs to different rows within the same bank. With tRC of 48ns, a bank can undergo up to 616K activations within tREFW. Similarly, tRRD defines the minimum time between ACTs to different banks, with tRRD_L applying to the same bank group and tRRD_S to different groups. For DDR5-6400 devices, the MC can issue up to 11.8M *ACT*s per rank (32 banks) with tRRD_S latency. Finally, the write recovery time (tWR) defines the required wait time before issuing a PRE after a write operation.

B. The RowHammer Vulnerability

RowHammer (RH) is a read-disturbance phenomenon where rapid activation of aggressor rows causes bit-flips in adjacent victim rows [28]. The RH threshold (N_{RH}) is the minimum number of activations required to trigger bit-flips, which has declined significantly with DRAM technology scaling—from 70K in 2014 [28] to 4.5K by 2020 [23]. Projections suggest N_{RH} could drop below 1K, with recent studies focusing on N_{RH} of 500 or lower [4], [19], [49], [50]. While RH has been a critical security threat [9], [15], [20], [21], [30], [31], [62], [76], it also poses a reliability risk [34], [43], [44] at low N_{RH}.

C. Threat Model

We consider a DRAM-based system vulnerable to RH attacks. Attackers can execute malicious applications with *user* privileges, such as Blacksmith [20], to induce bit-flips in critical data structures such as page tables [62]. An attack succeeds *any* DRAM row exceeds the RH threshold (N_{RH}) within tREFW, causing bit-flips in adjacent rows [12].

Additionally, attackers can degrade the performance of co-running applications by launching Performance Attacks (*Perf-Attacks*). In *Perf-Attacks*, adversaries execute malicious workloads on one or more cores while benign workloads run on others. The RowPress [35] attack is out of scope as its effects are orthogonal. RowPress can be mitigated by limiting row open time [35] or incrementing activation counters based on the row open time [58].

III. Motivation

State-of-the-art scalable RH mitigations use shared counters or tracking structures to reduce storage overhead. While these mitigations are effective on benign applications, their shared nature makes them vulnerable to Performance Attacks (*Perf-Attacks*). This section describes each scalable mitigation, then introduces targeted *RH-Tracker-based Perf-Attacks*. We demonstrate their vulnerability even at high N_{RH} and conclude by showing that increasing cache size or memory channels alone is insufficient to prevent these attacks.

A. Scalable RowHammer Mitigations

Hydra [50]: Hydra consists of three main components: the Group Counter Table (GCT), Row Counter Cache (RCC), and Row Counter Table (RCT). The GCT contains group counters (GCs), each shared among multiple rows, to track activations until they reach the predefined group counter threshold (N_{GC}). This threshold is set to 80% of the mitigation threshold ($N_M = \frac{N_{RH}}{2}$). Once a GC reaches N_{GC}, Hydra transitions to per-row tracking for precise tracking. To minimize storage overhead in the memory controller, Hydra stores per-row RH counters in a reserved DRAM region (RCT) and caches a subset of these counters in the RCC, a small cache within the memory controller. When an RCC miss occurs, Hydra performs additional DRAM accesses to fetch and update RH counters, which can result in performance penalties. In our evaluation, we configure Hydra based on its original design [50], with a GC size of 128, 4K RCC entries per rank, and a 32-way set-associative cache with a random eviction policy.

START [60]: START dynamically allocates per-row RH counters within a reserved portion of the Last-Level Cache (LLC), eliminating the need for additional counter structures. The reserved LLC region, configured as half of the LLC, stores these counters. If this reserved region cannot accommodate all RH counters, START stores these counters in a reserved DRAM space, using the LLC region as a cache, similar to Hydra. This design can result in significant performance overhead due to two factors: 1) reducing the effective LLC capacity by dedicating part of the LLC to RH counter storage and 2) introducing additional DRAM accesses to fetch and update RH counters when counters are not cached in the LLC. Our evaluated system has 8 million RH counters, exceeding the 4 million counters that can fit in the reserved LLC region. Thus, we configure START to store RH counters in DRAM and use the reserved LLC region as a cache for these counters. smallskip **CoMeT [4]**: CoMeT leverages a Count-Min Sketch technique for its counter table (CT), enabling efficient counter sharing across multiple rows and reducing storage overhead. However, counter-sharing prevents individual counters from being reset until the next periodic reset. To address this, CoMeT integrates a Recent Aggressor Table (RAT) to track rows recently involved in mitigations. The RAT uses per-row counters to reduce unnecessary mitigations caused by saturated counters. Despite these optimizations, frequent RAT misses can lead to excessive refreshes as CoMeT resets its structures by refreshing all DRAM rows in the rank, resulting in significant slowdowns. In our evaluation, we configure CoMeT following the original design [4]. The CT uses four hash functions, each managing 512 counters, while the Mitigation threshold is set to $\frac{N_{RH}}{4}$. The RAT is sized at 128 entries. CoMeT resets its structures every $\frac{tREFW}{3}$ by refreshing all DRAM rows. A 256-entry miss history is also employed, and extra resets are triggered when the RAT miss rate exceeds 25%.

ABACUS [45]: ABACUS uses a single Misra-Gries (MG) tracker shared across all banks in the channel. To prevent counter overestimation and unnecessary mitigative refreshes, ABACUS introduces a per-bank bit-vector. The size of the MG tracker is determined by the maximum possible number of aggressor rows within a single bank during tREFW at a given N_{RH}. For instance, at N_{RH} of 500, the MG tracker is configured with 2466 entries. However, since the MG tracker is shared across all banks, the counter can become saturated and remain

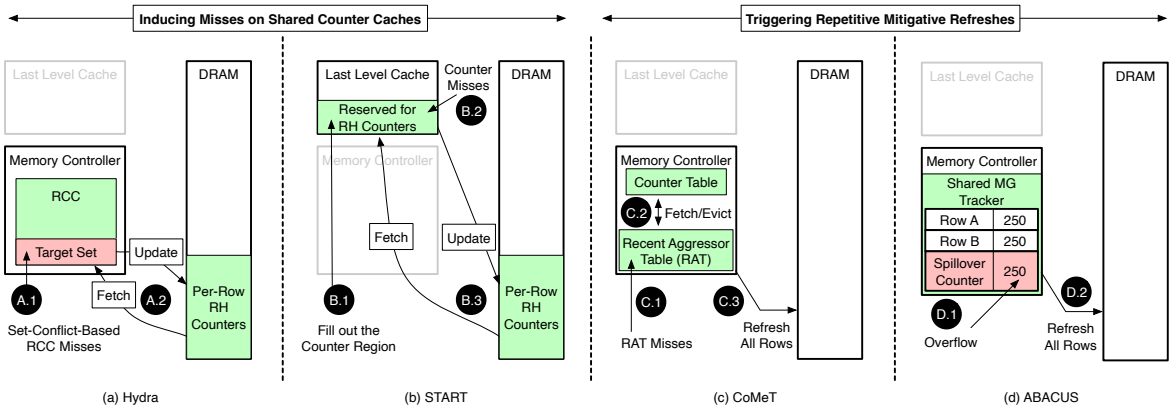


Fig. 2. An overview of the *RH-Tracker-based* Performance Attacks (*Perf-Attack*) tailored for state-of-the-art host-side low-cost RowHammer tracking mechanisms: Hydra [50], CoMeT [4], START [60], and ABACUS [45]. These attacks induce additional memory accesses or repetitive mitigative refreshes.

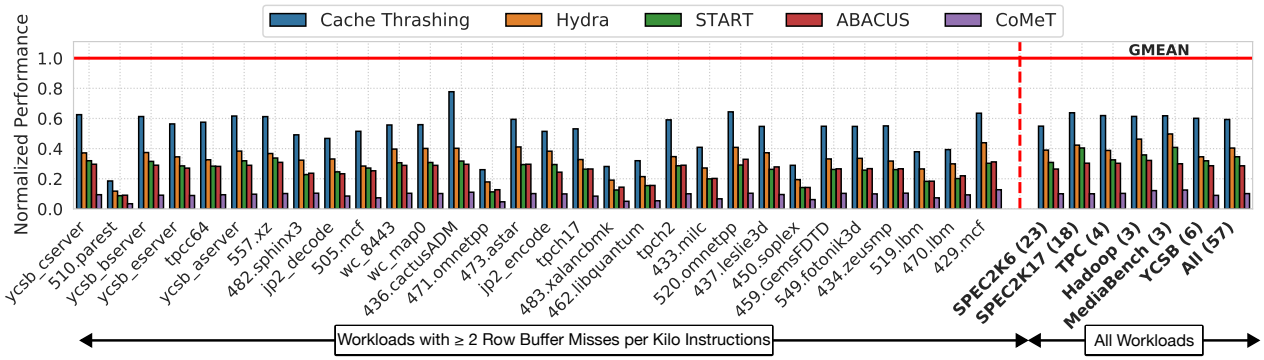


Fig. 3. The performance impact of state-of-the-art RowHammer (RH) trakcers: Hydra [50], CoMeT [4], START [60], and ABACUS [45] under cache thrashing and tailored *RH-Tracker-based Perf-Attacks*. The performance of three benign applications is normalized to a baseline with no RH mitigation. On average, we notice a performance drop of 60% to 90% performance loss under *Perf-Attacks* and a 40% performance drop under the cache thrashing attack.

unchanged until the periodic reset (every tREFW). ABACUS addresses this by refreshing all DRAM rows in the channel and resetting the counters upon overflow, which introduces significant overhead. In our evaluation, we configure MG tracker sizes 309, 617, 1233, 2466, 4931, and 9783 entries for N_{RH} of 4K, 2K, 1K, 500, 250, and 125, respectively.

B. Performance Attacks on Scalable RH Mitigations

To better understand the vulnerability of scalable RH mitigations to Performance Attacks (*Perf-Attacks*), we develop tailored attack patterns that generate excessive DRAM accesses for updating RH counters or trigger numerous additional refreshes to reset tracking structures. Figure 2 provides an overview of these attacks. Unlike cache thrashing attacks, which degrade the performance of co-running applications by inducing frequent cache misses, *RH-Tracker-based Perf-Attacks* specifically exploit RH mitigation mechanisms. These attacks reduce the effective DRAM bandwidth for co-running benign applications by increasing access to in-DRAM RH counters—read and read-modify-write operations—or inducing frequent mitigative refreshes. As shown in Figure 3, *RH-Tracker-based Perf-Attacks* can cause 16% to 49% more significant slowdowns compared to cache thrashing attacks.

1. Inducing Misses on Shared Counter Caches: Figure 2(a) illustrates our proposed attack on Hydra, designed to trigger frequent counter updates by causing misses in the Row Counter Cache (RCC). By exploiting the RCC's set-associative nature, we target over 32 rows that map to the same RCC set. Activating 64 rows across different banks simultaneously leads to an 87% miss probability in the RCC[1] (A.1). Each activation incurs two additional DRAM requests: one read to fetch the RH counter and one write to update the evicted counter (A.2). As shown in Figure 3, this attack drops performance by 61% on average, with a maximum of 88% for *510.parest*.

Figure 2(b) depicts our attack against START, which reduces the LLC's effective capacity for benign applications. It also increases DRAM requests for RH counters, reducing DRAM bandwidth for co-executing benign workloads. By streaming access across all DRAM rows, the attack initially fills the LLC's reserved RH counter regions (B.1). As the reserved LLC region becomes full, counter misses occur more frequently (B.2), leading to additional reads and writes for counter fetch and updates (B.3). On average, this attack drops performance by 65%. START suffers significantly in memory-intensive workloads due to diminished LLC capacity and reduced DRAM bandwidth for benign applications caused by frequent RH counter updates. For example, *510.parest* experiences a 91.2% slowdown.

[1]With a 32-way RCC and random eviction, each set has a miss probability of $\frac{1}{32}$. Targeting T rows results in a miss probability of $1-(1-\frac{1}{32})^T$.

2. Triggering Repetitive Mitigative Refreshes: Figure 2(c) illustrates our *Perf-Attack* on CoMeT, targeting the limited 128-entry capacity of the Recent Aggressor Table (RAT). By rapidly activating more rows than the RAT can handle, such as 192 rows, the attack disrupts CoMeT's operations in two ways. First, it causes frequent evictions between the Counter Table (CT) and the RAT (C.1), leading to counter overestimation due to the CT's lack of a reset feature. This results in premature mitigative refreshes (C.2). Second, repeated RAT evictions force CoMeT to refresh all rows (C.3) to reset its structures, each lasting approximately 2.4 ms and imposing significant overhead. Our attack significantly increases the number of resets, leading to an average performance loss of 90%. CoMeT is particularly vulnerable to *Perf-Attack*, as the reset operation can occur every 1 ms, blocking access for 2.4 ms each time.

Figure 2(d) shows our *Perf-Attack* on ABACUS, which targets the shared N-entry Misra-Gries tracker. By sequentially activating rows across different banks (e.g., row 0 in bank 0, row 1 in bank 1), we overflow the spillover counter every $N \times \frac{T_{RH}}{2}$ activations (D.1). For example, with N_{RH} of 500 and 2466 tracker entries, this strategy can force a reset and induce refreshes for all DRAM rows within 2 ms (D.2). This Attack reduces performance by an average of 72%, with up to a 91% slowdown.

C. Attack Sensitivity to RowHammer Thresholds

Figure 4 shows the normalized performance of scalable RH mitigations under cache thrashing and *RH-Tracker-based Perf-Attacks* as N_{RH} varies. Even at N_{RH} of 4K, scalable mitigations experience significant slowdowns from 46% to 71%, compared to a 41% slowdown caused by cache thrashing attacks. Hydra and CoMeT experience higher slowdowns as N_{RH} decreases because fewer activations are needed to cause additional DRAM accesses or trigger early resets with repetitive refreshes. In contrast, START and ABACUS show consistent slowdowns across evaluated N_{RH}. This is because *Perf-Attacks* on START and ABACUS involve streaming activations across multiple rows, consistently triggering additional DRAM accesses for counter updates and early reset with repetitive refreshes, which are inherently independent of N_{RH}. These findings underscore the persistent vulnerability of scalable RH mitigations to *Perf-Attacks*, even at high N_{RH}.

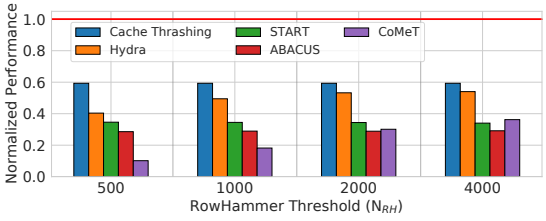


Fig. 4. Normalized performance of scalable RowHammer (RH) mitigations under cache thrashing and *RH-Tracker-based Perf-Attacks* as the RH threshold (N_{RH}) varies. Even at N_{RH} of 4K, scalable mitigations exhibit significant slowdowns of 46% to 71% under *RH-Tracker-based Perf-Attacks*, which is 5% to 30% higher than the slowdowns caused by cache thrashing attacks.

D. Attack Sensitivity to LLC Capacity and Memory Channels

We use a four-core system with a 2MB per-core Last-Level Cache (LLC) and two memory channels, each with 32GB DDR5 memory per channel (64GB total), as our baseline configuration. Modern systems often feature larger memory capacities, additional channels, and larger LLC sizes per core. For instance, recent Intel Xeon processors support up to 12 memory channels with four ranks per channel, while AMD EPYC processors include LLC sizes of 2MB or more per core. To evaluate the *Perf-Attacks* on these system configurations, we simulate a system with eight memory channels with 64GB per-channel memory (512GB total), and vary the per-core LLC size from 2MB to 5MB (8MB to 20MB total). Figure 5 shows that *Perf-Attacks* drops performance by 30% to 79%, compared to a 20% slwdown caused by cache thrashing attacks. These results highlight that, even with large LLC capacities and increased memory channels, current scalable RH mitigations remain highly susceptible to *Perf-Attacks*.

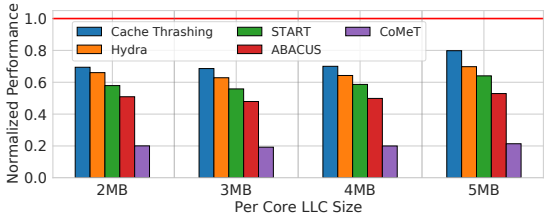


Fig. 5. Normalized performance of scalable RowHammer (RH) mitigations under cache thrashing and *RH-Tracker-based Perf-Attacks* with eight memory channels and the RH threshold (N_{RH}) of 500, as the per-core LLC size varies. Even with a 5MB per-core LLC, scalable mitigations exhibit slowdowns of 30% to 79%, which is 10% to 59% more severe than cache thrashing attacks.

E. Attack Potency Without Internal Knowledge

We have assumed that an attacker knows internal details, such as the structure sizes of each RH mitigation. However, the attacker can maintain attack potency even without this information. The proposed *Perf-Attacks* for START and ABACUS do not require internal details, as they only need to activate different rows with each attempt. Although the attack for Hydra requires knowledge of the Row Counter Cache (RCC) mapping to cause set conflicts and incur redundant DRAM accesses for counter updates, randomly selecting N rows and repeatedly activating them can achieve similar attack potency. Due to the limited capacity of the RCC, this attack will eventually fill the RCC, resulting in capacity misses instead of set-conflict misses. We validate that this attack causes slowdowns comparable to the originally proposed attack. For CoMeT, the attacker can randomly select N rows, repeatedly activate them, and monitor for early resets, which are easily detectable as they block DRAM access for 2.4 ms. Repeating this process several times, the attacker can deduce the Recent Aggressor Table (RAT) sizes and the number of aggressor rows needed to trigger *Perf-Attacks*. This step is required only once, after which the attacker can continuously launch *Perf-Attacks*. Thus, scalable RH mitigations remain highly susceptible, even in environments with limited attack knowledge.

IV. Evaluation Methodology

Simulation Framework: We evaluate designs using the publicly available cycle-accurate memory simulator Ramulator [29], [36], [55]. Our simulated system configuration is detailed in Table I. Our baseline setup consists of four out-of-order cores with a shared Last-Level cache (LLC). We use an out-of-order core model implemented in Ramulator similar to prior research [7], [45], [73], [77]. The LLC is a 16-way set-associative cache with an 8MB capacity. Our baseline system has two memory channels, each with a 32GB DDR5 DIMM. We simulate DDR5 6400MT/s DRAM [39], which has four banks per bank group and eight bank groups per rank. Each bank contains 64K rows that are 8KB in size. Within a 32ms refresh window (tREFW), a single bank can experience up to approximately 616K activations, while a memory channel can encounter up to 11.8M activations.

TABLE I
System Configuration

Processor	4 cores (OoO), 4GHz, 4-wide, 128 entry ROB
Last Level Cache (Shared)	8MB, 16-Way, 64B lines
Memory size	64 GB – DDR5
Memory bus speed	3.2 GHz (6.4GHz DDR)
tRCD-tRP-tCL	16-16-16 ns
tRC, tRFC, tREFI	48ns, 295 ns, 3.9μs
DRAM Organization	4 Banks x 8 Groups x 2 Ranks x 2 Channels
Rows per bank, Size	64K, 8KB

Workloads and Configuration: We use 57 open-sourced applications [55] from SPEC2006 [11], SPEC2017 [66], TPC [69], Hadoop [13], MediaBench [14], and YCSB [10] benchmark suites. We run four homogeneous workloads until each core completes 500 million instructions. For the RowHammer threshold (N_{RH}), we use a default value of 500 and conduct a sensitivity study with N_{RH} values from 125 to 4K. For mitigative actions (refreshes), we assume the memory controller uses commands such as Victim Row Refresh (VRR) to refresh victim rows adjacent to the aggressor row on a per-bank basis, following prior work [4], [28], [45], [50]. By default, each VRR command refreshes one victim row on each side of the aggressor row (i.e., blast radius (BR) of 1). Additionally, we evaluate our design with BR of 2 and the existing Same-Bank Directed Refresh Management ($DRFM_{sb}$) command.

V. Designing the DAPPER Tracker

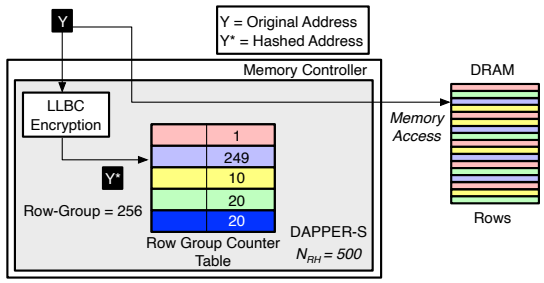
A. The Need for a Secure Hash-Based Mapping

DAPPER employs a shared Row Group Counter (RGC) for low-cost tracking. For example, grouping 256 rows requires only 16KB of SRAM for a 32GB memory. However, relying solely on shared counters causes vulnerabilities to adversarial patterns (See Section III-B). Attackers can uniformly activate rows to overestimate the RGC, triggering unnecessary mitigative refreshes even if individual rows have not reached the Mitigation threshold (N_M), which is half of N_{RH}. For example, with N_M of 250 and a group size of 256 rows (our default), activating each row in the group once will quickly reach N_M and falsely trigger the RH mitigations for RGC, leading to issue mitigative refreshes for all 256 rows in the group.

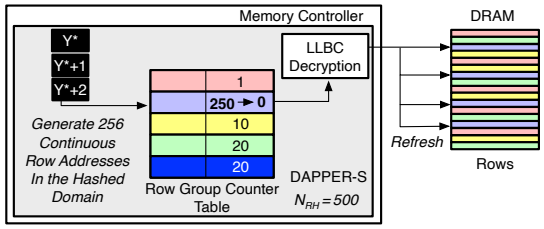
B. DAPPER-S: Using a Dynamic Secure Hash

DAPPER-S mitigates the risk of overestimating Row Group Counters (RGCs) by employing a secure hash to randomize row group mappings. Specifically, it randomizes rows within a defined level, such as per rank (e.g., 2M rows in our baseline), ensuring an even distribution of counter updates. This randomization prevents rapid counter increments in specific RGCs, thus avoiding unnecessary mitigative refreshes, particularly in workloads with high spatial locality. For the remainder of this paper, we will use the per-rank (2M rows) level mapping as our default configuration.

The hash function for this randomization must be reversible to support mitigation operations. DAPPER-S uses a four-round Low-Latency Block Cipher (LLBC)[2], similar to CEASER [51] and CUBE [26], to encrypt n-bit row addresses (e.g., 21-bit for 2M rows). The LLBC takes the original row address as input and uses one key per round, generated at boot time and updated every tREFW (32ms) via a Pseudo-Random Number Generator (PRNG) or True Random Number Generator (TRNG). The keys, stored in four 16-bit registers, ensure unique indices for each row, enhancing security and performance.



(a) Before Row-Group Counter Overflow



(b) After Row-Group Counter Overflows — Refresh and Reset

Fig. 6. Overview of DAPPER-S. The key operation involves updating the Row Group Counter (RGC) during each memory access. (a) Rows are securely hashed to ensure even distribution across RGCs, with each counter tracking 256 rows. (b) When an RGC reaches the mitigation threshold ($\frac{N_{RH}}{2}$), DAPPER-S decrypts the hashed addresses to their original addresses, performing mitigative refreshes to all rows in the group, and resets the RGC to zero.

C. Design of DAPPER-S

Figure 6 shows the design of DAPPER-S, which incorporates both secure hashing and mitigation mechanisms. To prevent attacks that exploit repetitive DRAM accesses for counter updates (similar to attacks for Hydra and START described in Section III-B), all RGCs are stored in the memory controller.

[2]Any low-latency lightweight block cipher, such as SCARF [5], can be used.

N_M is set to half of the RowHammer threshold (N_{RH}) to securely reset RGCs every tREFW, in line with previous studies [4], [50], [74]. The hashing process, shown in Figure 6(a), randomizes each DRAM row access using the LLBC engine, which transforms the original address (Y) into a hashed address (Y*). The hashed address is then divided by the row group size (e.g., 256) to determine the corresponding RGC. The mitigation process is triggered when an RGC reaches N_M. As depicted in Figure 6(b), DAPPER-S decrypts the hashed addresses back to their original address, performs mitigative refreshes to all rows in the group, and resets the RGC to zero.

D. Mapping-Capturing Attacks on DAPPER-S

Figure 7 illustrates a *Mapping-Capturing* attack that targets the static hash mapping of DAPPER-S to identify rows mapped to the same row group and overflow the Row Group Counter (RGC). By inflating RGCs, this attack triggers unnecessary mitigative refreshes with fewer activations than N_M. It unfolds in two stages: (1) repeatedly activating a single row (or two rows under the open-page policy) just below N_M ($N_M - 1$) (❶) and (2) sequentially activating consecutive rows in a different bank until a mitigative refresh occurs[3](❷). Capturing even a single pair of mappings enables effective attacks, such as simultaneously activating multiple rows within the same group across different banks to accelerate RGC overflows.

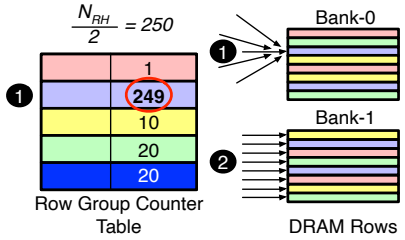


Fig. 7. *Mapping-Capturing* attacks, which targets the static hash mapping of DAPPER-S. The attack begins by activating a single row in a bank (e.g., bank-0) to increment the RGC count just below the mitigation threshold ($N_M - 1$). It then moves to the next bank (e.g., bank-1) and sequentially activates the rows while monitoring the issuance of mitigative refreshes. Upon detection of a refresh, the activated row is identified as belonging to the same RGC. This process is repeated across all banks to uncover the static hash mapping.

DAPPER-S counters this by resetting the RGC table and updating hash functions during each t_{reset} period to disrupt attack patterns. For the attack to succeed, attackers must identify at least one mapping pair before t_{reset} expires. After activating the target row by N_M - 1 times, the remaining attack time (t_{left}) is calculated using Equation (1).

$$t_{left} = t_{reset} - tRC \times (N_M - 1) \tag{1}$$

With a 48ns tRC and N_M of 250, the attack time is approximately 12μs[4]. Now, the maximum number of activations for each channel (ACT_{MAX}) that can be issued during t_{left} can be calculated by Equation (2).

$$ACT_{MAX} = \frac{t_{left}}{tRRD\_S} \tag{2}$$

The total number of row groups (N_{RG}) is determined by the total number of rows (R) in the randomized space and the row group (RG) size (S), calculated as $N_{RG} = \frac{R}{S}$. With a default RG size of 256 and 2M rows in the randomized space, there are 8K row groups. Each row group can be selected with a probability of $p = \frac{1}{N_{RG}}$ for each activation. The attack success probability (P_S) represents the likelihood of selecting the target row group at least once within ACT_{MAX} attempts, as described by Equation (3).

$$P_S = 1 - (1 - p)^{ACT_{MAX}} \tag{3}$$

Based on this, the expected number of attack iterations (AT_{iter}) is represented by Equation (4).

$$AT_{iter} = \frac{1}{P_S} \tag{4}$$

Finally, the time (AT_{time}) for one successful attack can be calculated using Equation (5).

$$AT_{time} = t_{reset} \times AT_{iter} \tag{5}$$

Table II shows the required attack iterations and time to decipher at least one pair of row mappings in DAPPER-S. Even with an impractically short reset period of 12μs, relying on a single secure hash can be broken every 7.6 ms. This highlights the need for significantly stronger security to protect the mapping information. As a result, we are motivated to enhance DAPPER-S to better defend against *Mapping-Capturing* attacks. Our enhanced design targets a 99.99% prevention rate against *Mapping-Capturing* attacks within tREFW (32 ms).

TABLE II
VULNERABILITY OF DAPPER-S TO *Mapping-Capturing* ATTACKS

Reset Period (t_{reset})	Attack Iterations (AT_{iter})	Attack Time (AT_{time})
36μs	1.8	64μs
24μs	3	71μs
12μs	630.6	7.6ms

E. Mapping-Agnostic Attacks on DAPPER-S

While DAPPER-S employs randomized mapping to mitigate *Mapping-Capturing* attacks, it remains vulnerable to *Mapping-Agnostic* attacks. One notable example is the *streaming attack*, which sequentially activates all DRAM rows within a rank, frequently triggering mitigative refreshes for each row. For instance, in our baseline system, this attack results in mitigative refreshes across all 2M rows per rank every 6 ms. With a row group size of 256 and N_{RH} of 500, the streaming attack reduces overall performance by 13%, as illustrated in Figure 9. Another example is the *refresh attack*, which repeatedly activates single or multiple rows per bank to prompt mitigative refreshes. This attack exploits DAPPER-S's requirement to refresh all member rows in an RGC (e.g., 256 rows in our default configuration) when its counter reaches N_M. Consequently, it causes significant slowdowns of 20%, as shown in Figure 9. Addressing these *Mapping-Agnostic* attacks is another key goal of our enhanced design.

[3]We assume attackers have exclusive access and can detect timing differences caused by mitigative refreshes.

[4]To simplify our analysis, we do not consider the impact of periodic DRAM refresh operations. The required attack time will be longer if we account for the refresh operations.

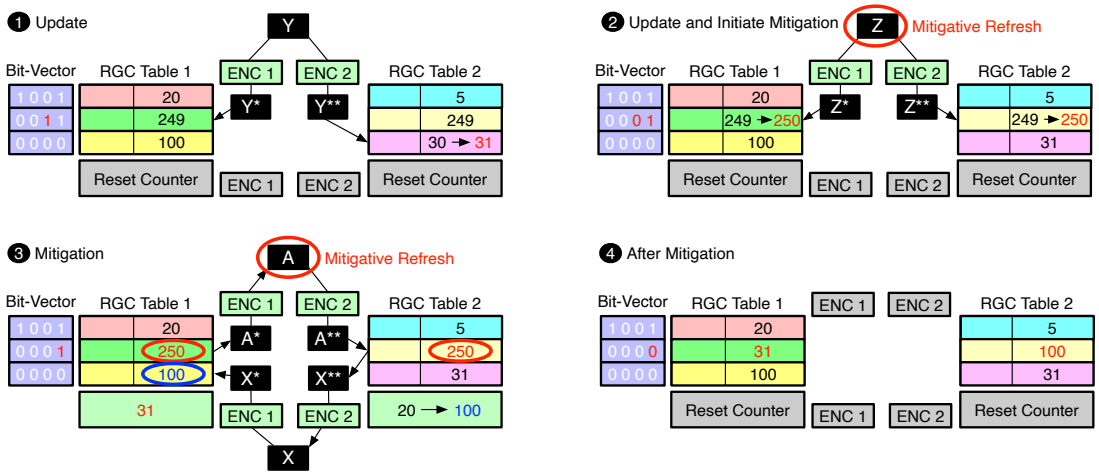


Fig. 8. Overview of DAPPER-H. It uses two hashes to filter out accesses that overwhelm the Row Group Counter (RGC). DAPPER-H performs mitigations only when both RGCs are equal to the Mitigation threshold (N_M).

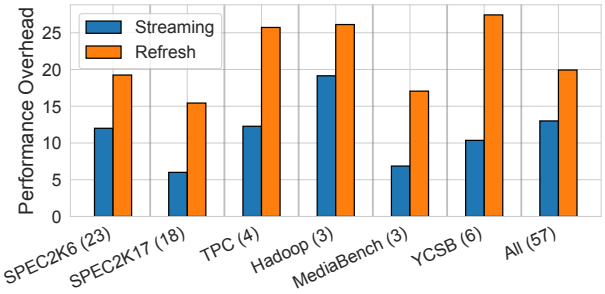


Fig. 9. Performance impact of two *Mapping-Agnostic* attacks, streaming and refresh attacks, on DAPPER-S. The streaming attack drops performance by 13%, while the refresh attack leads to a 20% slowdown.

VI. Enhancing the DAPPER Tracker

We introduce DAPPER-H, an enhanced tracker designed to defend against both *Mapping-Capturing* and *Mapping-Agnostic* attacks while preserving the cost efficiency of DAPPER-S. DAPPER-H employs double-hashing for row group mapping, making it significantly difficult for attackers to capture mapping information. Double-hashing also mitigates the refresh attack, which we will describe in Section VI-D.

A. Overview of DAPPER-H

Unlike DAPPER-S, which uses a single Row Group Counter (RGC) table and performs mitigative refreshes to all rows in a group during mitigation, DAPPER-H tracks activations using two distinct RGC tables and performs mitigative refreshes only for the shared rows of the accessed groups. Each table uses a unique hashing function to have different row groupings. Mitigative refreshes are triggered only when the RGCs in both tables reach the Mitigation threshold (N_M). This approach enhances resilience against Performance Attacks (*Perf-Attacks*) but introduces additional challenges, such as managing RGC resets during mitigations. We will discuss how these issues are addressed in subsequent sections.

B. Design and Operations of DAPPER-H

Figure 8 illustrates the overall design of DAPPER-H. It uses two distinct row group counter (RGC) tables, each integrated with a Low-Latency Block Cipher (LLBC) to randomize addresses upon every activation. DAPPER-H incorporates a bit-vector for each entry in RGC Table 1 to further mitigate the streaming attack, where each bit represents a bank. The RGC of Table 1 is incremented only if the corresponding bit in the bit-vector is already set. The following section (Section VI-D) details how the bit-vector effectively counters the streaming attack. Here, we describe the operations of DAPPER-H.

1) Initialization and Reset: Like DAPPER-S, DAPPER-H sets N_M to half of the RowHammer threshold (N_{RH}), ensuring periodic resets of its tracking structures every tREFW. During each reset, both RGC tables and bit-vectors are cleared. These resets occur at the end of each tREFW interval or during system initialization, such as at boot time.

2) Update Operation with Bit-Vector: DAPPER-H uses the per-table LLBC for each activation to compute randomized addresses and accesses the corresponding entries in both RGC tables. For RGC Table 1, DAPPER-H first checks the bit-vector associated with the bank of the activated row. Suppose the bit for the bank is not set; it is updated to 1, and only the RGC in Table 2 is incremented, thereby preventing overestimation of counters due to activations from different banks. For example, if the memory controller activates row Y in Bank 1 and the corresponding bit in the bit-vector is not set, DAPPER-H sets the bit and increments only the RGC in Table 2 (❶).

In contrast, if the bit is already set, both RGCs in Tables 1 and 2 are incremented, and the bit-vector entries for all other banks are cleared. For instance, if the memory controller activates row Z in Bank 0, where the bit for the bank is already set, DAPPER-H increments both RGCs in Tables 1 and 2 and clears the bits for other banks (❷). Additionally, since both RGCs reach N_M, the mitigation process is triggered, and DAPPER-H issues mitigative refreshes to row Z. This entire process, including address randomization and RGC table access, is completed within a single cycle (0.25 ns) [26], [51], [52], staying well within the *tRRD_S* latency of 2.5 ns.

3) Mitigative Refreshes and RGC Reset: Thanks to double-hashing, DAPPER-H avoids issuing mitigative refreshes to all

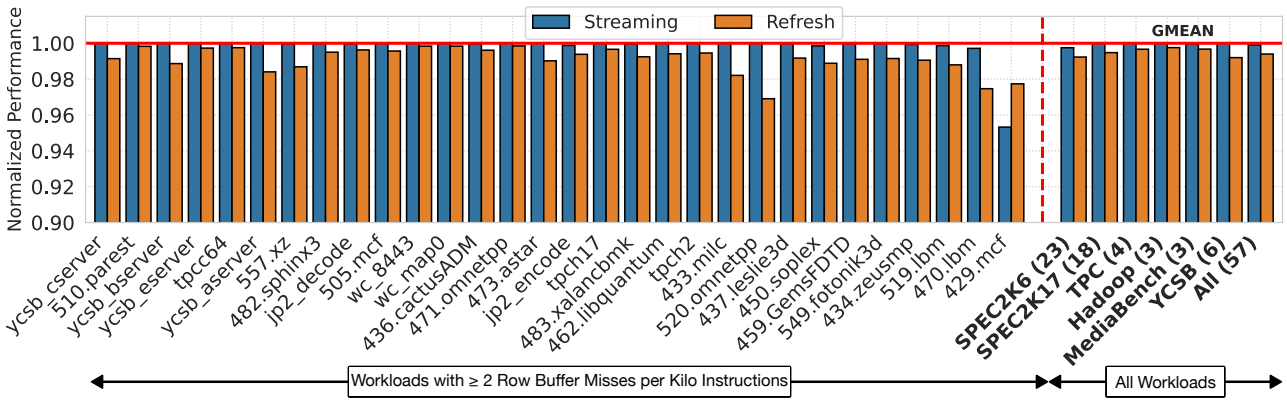


Fig. 10. Normalized performance of DAPPER-H under two *Mapping-Agnostic* attacks, streaming and refresh attacks, at an ultra-low RowHammer threshold (N_{RH}) of 500. The performance of three benign applications is normalized to a non-secure baseline system. Despite the active *Perf-Attacks*, DAPPER-H incurs less than 1% average slowdowns. The maximum slowdowns are 4.7% for the streaming attack and 2.3% for the refresh attack, highlighting the effectiveness of the proposed double-hashing, novel reset, and bit-vector techniques in mitigating *Perf-Attacks*.

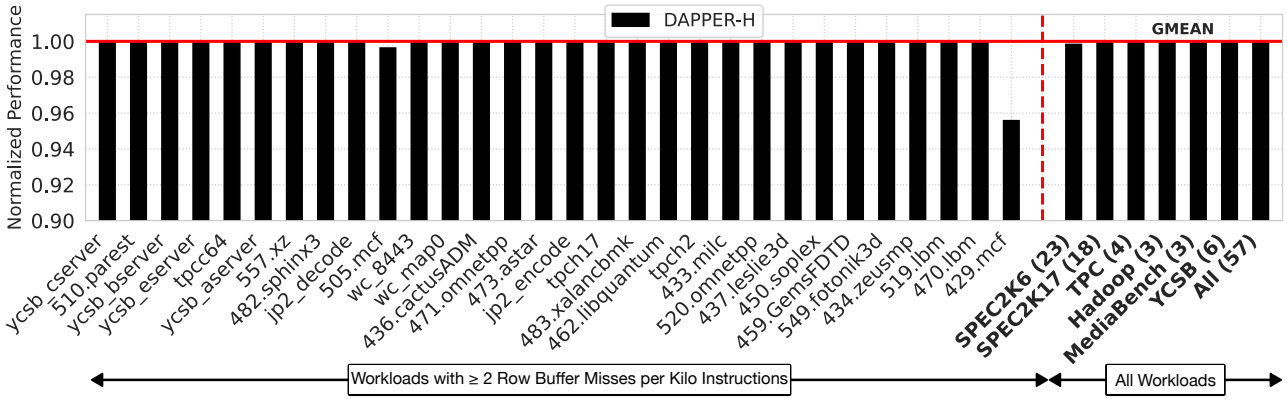


Fig. 11. Normalized performance of DAPPER-H on benign applications compared to an insecure baseline at an ultra-low RowHammer threshold (N_{RH}) of 500. DAPPER-H incurs only 0.1% slowdown on average and a maximum slowdown of 4.4%, as the double-hashing, novel reset, and bit-vector mechanisms effectively minimize unnecessary mitigative refreshes.

rows in an RGC. Instead, it refreshes only the rows shared between the RGCs of the two tables. This is achieved by decrypting the member rows of each RGC using multiple LLBC engines in parallel and identifying shared rows that require mitigative refreshes (e.g., Row A in ❸). However, this selective mitigation complicates the counter-reset process, as simply resetting all RGCs to zero is no longer sufficient. To address this, DAPPER-H tracks the counter value of each member row from the opposite RGC table during the decryption process using per-table reset counters. For example, as shown by ❸, DAPPER-H updates the reset counter value of RGC Table 2 from 20 to 100 because the counter value of member row X in Table 1 is larger than the current reset counter value. At the end of the mitigation process, DAPPER-H resets each RGC to its corresponding reset counter value and clears the bit-vector (as shown by ❹).

C. Defending Mapping-Capturing Performance Attacks

Attackers can develop a *Mapping-Capturing* attack for DAPPER-H similar to those targeting DAPPER-S. They first choose a single target row and repeatedly activate it. However, due to the double-hashing design of DAPPER-H, they now should activate the target row only N_M - 2 times. Suppose an attacker were to activate the target row just below the Mitigation threshold (i.e., N_M - 1 times). In that case, their only chance of learning the mapping relies on randomly selecting a row shared by both row group counters (RGCs). This is highly improbable given the vast number of rows in the randomized address space (e.g., 2M rows in our default configuration).

A more effective strategy involves activating the target row N_M - 2 times and randomly accessing two additional rows. The attacker then activates the target row one final time to check if their attack was successful. If the two random accesses correctly target the required RGCs, mitigation will be triggered during the random accesses or the final check activation. This allows the attacker to determine the success of their attack. Given N row groups, the probability of success (P) per trial is described by Equation (6).

$$p = (1 - (1 - \frac{1}{N})^2) \times (1 - (1 - \frac{1}{N})^2) \quad (6)$$

Furthermore, the bit-vector implementation in DAPPER-H prevents attackers from targeting multiple banks with tRRD_S latency. Attempts to attack other banks merely set the bit vector and eliminate chances for additional guesses on the target row. This limits attackers to only 616K activations. Additionally, if the attacker fails to capture the mapping, the attacker must activate the target row $N_M - 2$ before retrying. Thus, every attack trial requires the full N_M limit, reducing the

total number of trials (T) to approximately 2.5K. The overall attack success probability is calculated using Equation (7).

$$P_S = 1 - (1 - p)^T \quad (7)$$

As a result, DAPPER-H effectively prevents *Mapping-Capturing* attacks with a 99.99% probability within tREFW.

Discussion: Although the mapping information can theoretically be compromised with a 0.01% probability within tREFW (32ms), attackers would only successfully launch a *Mapping-Capturing* attack during *one* tREFW in 10,000 tREFWs (on average). This is because DAPPER-H changes the seed randomly every tREFW. For Performance Attacks (*Perf-Attacks*) to significantly affect system performance, the attacker must breach a defense *frequently* across back-to-back tREFW intervals. In contrast, RH mitigations require the adversary to achieve a single successful attempt at flipping DRAM bits among millions of tries [57]. A single breach can be exploited to manipulate or extract data [31], [62], necessitating more complex solutions. Moreover, unlike the case of randomized caches, DAPPER-H does N-to-N encryptions, making it resistant to shortcut attacks [3], [47] on LLBC. Prior research also demonstrated this aspect [26].

D. Defending Mapping-Agnostic Performance Attacks

Figure 10 shows the performance of DAPPER-H under two *Mapping-Agnostic* attacks, streaming and refresh attacks, at an ultra-low N_{RH} of 500. The performance of three benign workloads is normalized to an insecure baseline system. DAPPER-H effectively mitigates both attacks, incurring less than 1% slowdowns on average, with maximum slowdowns of 4.7% for the streaming attack and 2.3% for the refresh attack. The double-hashing design of DAPPER-H efficiently counters the refresh attack by performing mitigative refreshes only on shared rows between the two RGCs. Most of the time, it refreshes a single shared row[5].

As described in Section V-E, the streaming attack that activates every DRAM row can rapidly increase all row group counters (RGCs) in DAPPER-S. This attack exploits bank-level parallelism, activating rows across different banks with tRRD_S latency to inflate RGCs. DAPPER-H effectively counters such attack through its bit-vector mechanism, as shown in Figure 10. DAPPER-H causes an average slowdown of only 0.2%, up to 4.7% in the worst case. Specifically, DAPPER-H sets the corresponding bit in the bit-vector but does not increment the RGC of Table 1 when rows from previously unaccessed banks are activated, preventing the overestimation of RGCs. For example, with a row group size of 256 and uniformly distributed addresses, each bank contains eight rows per group. With our default 2M randomized row space and up to 12 million activations per channel during tREFW, each row is activated approximately six times. Consequently, the theoretical maximum number of accesses per row group is about 48 (6 activations × 8 rows), far below the mitigation threshold (N_M) of 250. This ensures that DAPPER-H can effectively thwart the streaming attack. While ABACUS [45] also uses a per-bank bit-vector for performance reasons, it does not prevent targeted *Perf-Attacks*. Attackers can still overflow the spillover counter through repeated accesses to rows with different row IDs, leading to significant performance drops.

[5]The probability of shared rows between two RGCs is very low. With a default 2 million randomized address space and row group size of 256, there are 8K RGCs in each table, making shared rows unlikely. Our evaluation shows that DAPPER-H refreshes a single row 99.9% of the time under both benign applications and *Perf-Attacks*, ensuring negligible overhead.

E. Performance of DAPPER-H on Benign Applications

Figure 11 shows the performance of DAPPER-H under benign applications at N_{RH} of 500, normalized to a non-secure baseline DDR5 system. Leveraging double-hashing, novel reset, and bit-vector techniques, DAPPER-H effectively minimizes mitigations, resulting in an average slowdown of only 0.1%. The highest performance overhead occurs in the most memory-intensive workload, *429.mcf*, which experiences a 4.4% slowdown due to frequent mitigations than others. Overall, DAPPER-H shows significantly low overheads for benign applications, even at an ultra-low N_{RH} of 500.

F. Sensitivity to Varying RowHammer Thresholds

Figure 12 illustrates DAPPER-H's performance sensitivity as N_{RH} varies from 125 to 4K. DAPPER-H incurs less than 1% overheads at $N_{RH} \geq 500$, even under active Performance Attacks (*Perf-Attacks*). Additionally, it minimizes its performance impact even at lower N_{RH} by avoiding unnecessary mitigative refreshes. For example, at N_{RH} of 125, DAPPER-H incurs only a 6% slowdown under the refresh attack, highlighting its resilience against *Perf-Attacks*.

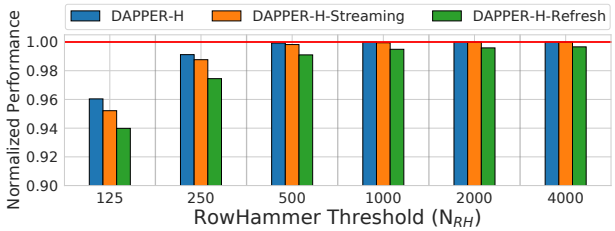


Fig. 12. Normalized performance of DAPPER-H under benign and *Perf-Attacks* as the RowHammer threshold (N_{RH}) varies. DAPPER-H incurs less than 1% slowdowns at $N_{RH} \geq 500$, and up to a 6% slowdown at N_{RH} of 125 under *Perf-Attacks*, demonstrating strong resilience against *Perf-Attacks*.

G. Impact of blast radius and Performance with DRFM

Figure 13 compares the performance of DAPPER-H with a blast radius (BR) of 1 (default) and BR of 2 under benign applications and *Perf-Attacks*. We use the refresh attack for *Perf-Attacks*, as they cause higher overheads than the streaming attack. Overall, DAPPER-H-BR2 exhibits slightly lower performance. Specifically, it incurs slowdowns of 1.8% or less at $N_{RH} \geq 500$, 4.4% at N_{RH} of 250, and 9.2% at N_{RH} of 125 under *Perf-Attacks*. Conversely, DAPPER-H incurs slowdowns of 0.9% or less, 2.5%, and 6% at the same thresholds. The increased overhead of BR of 2 stems from doubling the blocking duration for each bank during mitigative refreshes.

The JEDEC DDR5 specification supports the Directed Refresh Management (DRFM) command, allowing the memory

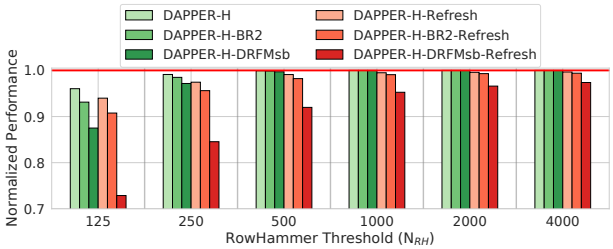


Fig. 13. Normalized performance of DAPPER-H with a blast radius (BR) of 1 (default) and BR of 2, as well as Same-Bank Directed Refresh Management ($DRFM_{sb}$), under benign applications and the refresh attack. Increasing the BR slightly raises slowdowns while using $DRFM_{sb}$ results in higher slowdowns, particularly at low N_{RH} due to blocking more banks. At N_{RH} of 500 and under the refresh attack, DAPPER-H and DAPPER-H-BR2 incur slowdowns of 1% and 2%, respectively, while DAPPER-H-$DRFM_{sb}$ experiences an 8% slowdown.

controller to issue mitigative refreshes for a specific target row, similar to VRR [38]. However, current DRFM commands support only All-Bank and Same-Bank granularity, blocking either all banks (32 banks) or the same bank across all bank groups (8 banks). This can result in higher overhead than VRR, which blocks only the accessed bank. Figure 13 shows the performance of DAPPER-H with Same-Bank DRFM ($DRFM_{sb}$). We use $DRFM_{sb}$ with a 240ns delay (supporting a BR of 2) as specified by the JEDEC DDR5 specification [38], assuming no rate limitations between DRFM commands[6], consistent with prior research [49], [60]. DAPPER-H-$DRFM_{sb}$ incurs higher overhead than DAPPER-H-BR2, as each mitigation penalizes more banks. In the presence of *Perf-Attacks*, DAPPER-H-$DRFM_{sb}$ causes slowdowns of 8% at N_{RH} of 500 and 27.1% at N_{RH} of 125, while DAPPER-H-BR2 incurs only 1.8% and 4.4% overheads, respectively, at the same N_{RH}. Our results highlight that supporting per-bank granularity RFM or DRFM commands can significantly reduce the performance overhead, particularly under *Perf-Attacks*. This observation aligns with findings from concurrent work [73].

H. Storage and Energy Overhead of DAPPER-H

DAPPER-H employs double-hashing, which requires dual row group counter (RGC) tables. With the default per-rank mapping, which contains 2M rows in our baseline system, and a Mitigative threshold (N_M) of 250, DAPPER-H requires 8K 1-byte RGC entries per table, resulting in a total of 32KB SRAM per 32GB DDR5 memory. Additionally, DAPPER-H utilizes a per-bank bit vector to mitigate the streaming attack, which requires 32KB of SRAM per rank or 64KB per 32GB DDR5 memory. Table III compares DAPPER-H's storage overhead to state-of-the-art scalable RH mitigations [4], [45], [50], [60] for 32GB DDR5 memory. The estimated die area overhead, calculated based on prior work [45], shows that DAPPER-H incurs a negligible $0.038mm^2$ overhead, similar to existing scalable RH mitigations.

Table IV presents the energy overhead of DAPPER-H for N_{RH} values from 125 to 4K, measured using DRAMPower [8]. At N_{RH} of 500, DAPPER-H incurs only a 0.1% overhead

[6]The JEDEC specification limits the DRFM command rate to one DRFM command per 2 tREFI.

TABLE III
STORAGE OVERHEAD PER 32GB DDR5 MEMORY

	SRAM (KB)	CAM (KB)	Die Area Overhead (mm2)
Hydra [50]	56.5	-	0.044
CoMeT [4]	112	23	0.139
START [60]	4	-	0.003
ABACUS [45]	19.3	7.5	0.038
DAPPER-H	96	-	0.075

for benign applications and a 1.1% overhead under the refresh attack. Even at N_{RH} of 125, the energy overhead under the refresh attack remains limited to 7.5%. The overhead primarily arises from mitigation operations (i.e., mitigative refreshes). DAPPER-H effectively reduces unnecessary mitigations. The double-hashing mechanism ensures the mitigative refreshes target a single shared row 99.9% of the time, minimizing overhead under the refresh attack. Additionally, the bit-vector mechanism prevents the streaming attack and eliminates redundant mitigations, further optimizing energy efficiency.

TABLE IV
ENERGY OVERHEAD OF DAPPER-H

RowHammer Threshold	Benign	Streaming Attack	Refresh Attack
125	4.5%	7.0%	7.5%
250	0.9%	1.3%	3.2%
500	**0.1%**	**0.2%**	**1.1%**
1000	0%	0.1%	0.6%
2000	0%	0%	0.5%
4000	0%	0%	0.4%

I. Comparison to BlockHammer

BlockHammer [75] is a throttling-based RH mitigation that uses a Counting Bloom Filter to track aggressor rows and blacklist them. Figure 14 compares the performance of BlockHammer and DAPPER-H under benign applications as N_{RH} varies from 125 to 4K. BlockHammer incurs a 7.5% slowdown at N_{RH} of 1K and suffers significant overhead at ultra-low N_{RH} ($N_{RH} \leq 500$). Specifically, it shows slowdowns of 25% at N_{RH} of 500 and 66% at N_{RH} of 125. These overheads stem from its inability to accurately distinguish malicious threads, leading to unnecessary throttling of memory requests. This vulnerability can be exploited to launch active *Perf-Attacks* by hammering rows that share Bloom Filter entries with target rows, effectively blocking the memory requests for target rows and causing severe performance degradation, as highlighted in prior work [25], [57], [59]. In contrast,

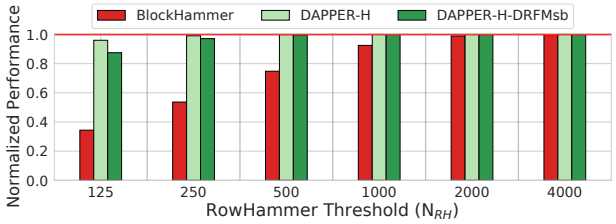


Fig. 14. Performance comparison of DAPPER-H and BlockHammer [75] under benign applications as the RowHammer threshold (N_{RH}) varies. BlockHammer significantly drops performance at ultra-low N_{RH} ($N_{RH} \leq 500$) due to unnecessary throttling. At N_{RH} of 250, for example, BlockHammer incurs considerable 46.4% slowdown, while DAPPER-H and DAPPER-H-$DRFM_{sb}$ incurs only 0.9% and 2.8% slowdowns, respectively.

DAPPER-H and DAPPER-H with DRFM demonstrate superior scalability. At N_{RH} of 500, they incur less than 0.3% slowdowns. Even at N_{RH} of 125, DAPPER-H incurs only a 4% overhead, while DAPPER-H-DRFM$_{sb}$ experiences a 13% slowdown, significantly outperforming BlockHammer.

J. Comparison to Probabilistic Mitigations

Figure 15 compares the performance of DAPPER-H under benign applications to two state-of-the-art probabilistic mitigations, PrIDE [19] and PARA [28], as N_{RH} varies from 125 to 4K. All evaluated methods are assumed to support per-bank granularity mitigations by default, with additional comparisons to the existing DRFM$_{sb}$ and RFM$_{sb}$ commands.

Both PARA and PrIDE experience higher slowdowns at ultra-low N_{RH} compared to DAPPER-H. At N_{RH} of 125, PARA and PrIDE incur overheads of 8.5% and 16.7%, respectively, whereas DAPPER-H causes only a 4% slowdown. When paired with Same-Bank RFM (RFM$_{sb}$) or DRFM (DRFM$_{sb}$) commands, the slowdowns are even more significant for probabilistic mitigations. At N_{RH} of 500, PARA-DRFM$_{sb}$ and PrIDE-RFM$_{sb}$ incur slowdowns of 18.4% and 11.5%, respectively, while DAPPER-H-DRFM$_{sb}$ incurs only 0.3% overhead. This is due to the stateless nature of probabilistic mitigations, which require more frequent mitigations. In contrast, DAPPER-H minimizes unnecessary mitigations through accurate tracking enabled by double-hashing and bit-vector mechanisms. Additionally, PARA is more affected by Same-Bank mitigations than PrIDE due to the longer delay of the DRFM$_{sb}$ command (240ns) compared to RFM$_{sb}$ (190ns), resulting in greater bandwidth reduction.

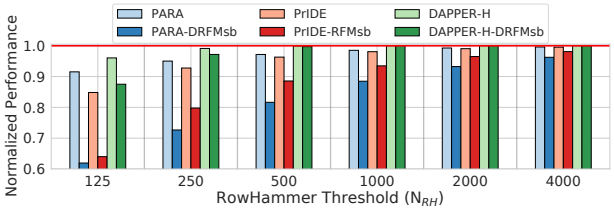


Fig. 15. Normalized performance of DAPPER-H, PARA [28], and PrIDE [19] on benign applications as the RowHammer threshold (N_{RH}) varies. At N_{RH} of 500, PARA and PrIDE incur slowdowns of 3% and 7%, while PARA-DRFM$_{sb}$ and PrIDE-RFM$_{sb}$ cause significant slowdowns of 18% and 12%. Conversely, DAPPER-H and DAPPER-H-DRFM$_{sb}$ incur less than 0.3% slowdowns.

Figure 16 compares the performance of DAPPER-H, PrIDE, and PARA under *Perf-Attacks*. At N_{RH} of 125, DAPPER-H incurs only a 6% performance drop, while PARA and PrIDE suffer significant slowdowns of 14.6% and 22.8%, respectively, due to frequent mitigations and reduced DRAM bandwidth. These attacks force PARA and PrIDE to perform mitigations every 2 or 4 activations, limiting bandwidth for co-running benign applications. DAPPER-H minimizes the performance impact of Same-Bank mitigations more effectively than probabilistic solutions. At N_{RH} of 1K, DAPPER-H-DRFM$_{sb}$ shows a 4.8% performance drop, while PARA and PrIDE experience slowdowns of 23% and 16%, respectively. These findings underscore the potential benefit of incorporating per-bank RFM or DRFM commands in future DRAM generations to mitigate *Perf-Attacks*, as discussed in Section VI-G.

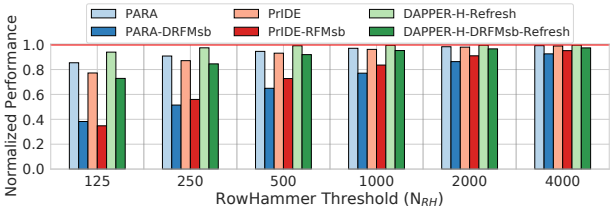


Fig. 16. Normalized performance of DAPPER-H, PARA [28], and PrIDE [19] under *Perf-Attacks*. At the RowHammer threshold (N_{RH}) of 125, DAPPER-H incurs only 6% overhead, while PARA and PrIDE cause significant slowdowns of 15% and 23%, respectively. DAPPER-H minimizes the performance impact of RFM$_{sb}$ and DRFM$_{sb}$ commands more effectively than PARA and PrIDE. At N_{RH} of 1K, DAPPER-H-DRFM$_{sb}$ incurs a 5% slowdown, while PARA and PrIDE experience considerable slowdowns of 23% and 16%, respectively.

K. Comparison to Per Row Activation Counting (PRAC)

The recent JEDEC DDR5 specification [38] introduces Per Row Activation Counting (PRAC) for precise in-DRAM aggressor tracking, utilizing per-row activation counters and an Alert Back-Off (ABO) protocol. While PRAC can support sub-100 N_{RH} [48], [73], it requires read-modify-write operations for every activation to update counters, leading to significant overhead even at relatively high N_{RH} ($N_{RH} \geq 1K$) [6].

Figure 17 compares the performance of DAPPER-H and PRAC under benign applications and *Perf-Attacks*. We implement PRAC based on the state-of-the-art secure QPRAC design [73]. PRAC incurs an average of 7% and up to 20% overhead on benign applications, even at N_{RH} of 4K, mainly due to frequent counter updates. Its overhead remains relatively constant across evaluated N_{RH} values because per-row tracking minimizes mitigations even at low N_{RH}. Furthermore, PRAC is less impacted by *Perf-Attacks* at all evaluated N_{RH} values, as its precise aggressor tracking effectively reduces unnecessary mitigative refreshes during *Perf-Attacks*. These findings align with recent work [6].

In contrast, DAPPER-H incurs less than 4% slowdowns across all evaluated N_{RH} values on benign applications, with only 1% overhead at $N_{RH} \geq 500$ and 6% at N_{RH} of 125 under *Perf-Attacks*. While DAPPER-H-DRFM$_{sb}$ outperforms PRAC at $N_{RH} \geq 250$ for benign applications and $N_{RH} \geq 1K$ under *Perf-Attacks*, its overhead increases significantly at $N_{RH} \leq 250$ due to more frequent mitigations and greater bandwidth loss caused by DRFM$_{sb}$ commands.

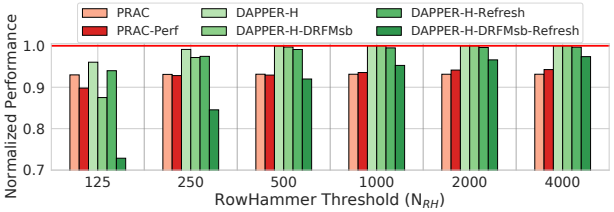


Fig. 17. Normalized performance of DAPPER-H and PRAC under benign applications and *Perf-Attacks*. While PRAC is less affected by *Perf-Attacks* due to precise activation counting, it incurs high overhead on benign applications, causing 7% slowdown even at N_{RH} of 4K. In contrast, DAPPER-H incurs less than 4% slowdowns for all N_{RH} under benign applications and only 6% even at N_{RH} of 125 under *Perf-Attacks*. DAPPER-H-DRFM$_{sb}$ outperforms PRAC at $N_{RH} \geq 1K$ but causes similar or higher overhead at lower N_{RH} due to more frequent mitigations and limited bandwidth due to DRFM$_{sb}$.

VII. Related Work

A. RowHammer-Based Performance Attacks

To the best of our knowledge, our paper is one of the first to thoroughly analyze the vulnerability of existing RowHammer (RH) mitigations to Performance Attacks (*Perf-Attacks*) and propose a *Perf-Attack*-resilient solution. BreakHammer [7] is a concurrent work that evaluates the impact of *Perf-Attacks* on RH mitigations and introduces throttling-based defenses. BreakHammer identifies potentially malicious threads by tracking the number of triggered RH mitigations per hardware thread and throttles their memory requests. However, it depends on prior RH solutions for mitigation and requires microarchitectural modifications, such as changes to caches, to propagate thread information to the throttling logic in the memory controller. In contrast, DAPPER provides standalone *Perf-Attack* resilience with minimal overhead without requiring additional microarchitectural changes. Furthermore, DAPPER can be combined with BreakHammer to enhance protection against *Perf-Attacks*. Another concurrent work, RogueRFM [68], also highlights the vulnerability of the current All-Bank Refresh Management (RFM_{ab}) command to *Perf-Attacks*. RogueRFM proposes limiting the number of RFM_{ab} commands to an average of one per tREFI interval. However, this approach compromises the security of existing RFM-based solutions [19], [25], [49], which require at least two RFMs to scale to $N_{RH} \leq 1.5K$.

Several prior studies have acknowledged the concern of RH mitigations against *Perf-Attacks*. For example, ABACUS [45] and CoMeT [4] recognize that their defenses are vulnerable to adversarial patterns similar to those discussed in Section III-B. However, CoMeT does not offer a solution, and ABACUS suggests increasing the number of Misra-Gries (MG) tracker entries to match the number of rows in the bank. This would require 200KB or 400KB CAM per channel, which is impractical for implementation in the memory controller. In contrast, DAPPER mitigates *Perf-Attacks* with just 96KB SRAM per 32GB of memory. Additionally, Panopticon [2], UPRAC [6], MOAT [48], and QPRAC [73] highlight potential issues with Per Row Activation Counting (PRAC) against *Perf-Attacks*. MOAT and UPRAC do not provide solutions, while Panopticon proposes randomly initializing per-row activation counters after each mitigation as a solution without any analysis. QPRAC suggests adding Same-Bank or Per-Bank RFM commands for the Alert Back-Off (ABO) protocol based on analytical models. In contrast, our paper provides both through security analysis and evaluation results.

B. Randomization in RowHammer Mitigations

Recent RH solutions, such as RRS [57] or SRS [74], mitigate RH by swapping aggressor rows with randomly selected rows [57], [72], [74] or migrating aggressor rows to a quarantine region [61]. However, these row randomization techniques remain vulnerable to *Perf-Attacks*, as adversaries can exploit the high latency of swap or migration operations by triggering frequent mitigations. Rubix [59] randomizes physical-to-DRAM address mappings to distribute activations across DRAM rows uniformly. However, it relies on existing RH solutions for mitigation and reduces row-buffer hit rates, potentially incurring high overhead in many workloads. In contrast, DAPPER randomizes *only* counter addresses, not impacting normal DRAM operations such as row-buffer hit rates while providing scalable and standalone security.

C. Host-Side RowHammer Mitigations

Host-side or memory controller-based RH mitigations address RH either probabilistically [22], [24], [28], [65], [78] or by tracking potential aggressor rows and refreshing their victims [4], [45], [46], [50], [60], [64] or throttling access to them [75]. However, at low RH thresholds ($N_{RH} \leq 1K$), probabilistic solutions incur significant overhead, particularly with the Directed Refresh Management (DRFM) commands. The inefficiency of current DRFM/RFM granularities–blocking access to all banks or the same bank across bank groups–leads to substantial DRAM bandwidth reduction for co-running applications. Similarly, tracking-based solutions either require impractical storage overhead [46], [64] or remain vulnerable to *Perf-Attacks* [4], [45], [50], [60]. DAPPER is the only known secure and scalable solution resilient to *Perf-Attacks* at these ultra-low RH thresholds, even with current DRFM commands.

D. In-DRAM RowHammer Mitigations

In-DRAM mitigations address RH by refreshing potential aggressor rows during the specific refresh operations (i.e., every N^{th} refresh) [17], [20], [21] or relying on RFMs. These solutions either work probabilistically [18], [19], [27], [49] or track potential aggressor rows using counters [25], [32], [37]. However, they face challenges at ultra-low RH thresholds, including high overhead or impractical storage requirements for tracking structures. Furthermore, as demonstrated in Section VI-J, RFM-based solutions can experience substantial slowdowns under *Perf-Attacks* with current Same-Bank RFM command (RFM_{sb}). In contrast, DAPPER requires no DRAM modifications and is more scalable with the existing Same-Bank DRFM ($DRFM_{sb}$) command.

E. Memory Performance Degradation Attacks

Prior research has proposed both memory Performance Attacks (*Perf-Attacks*) [40], [41] and defenses [41], [67]. These works address *Perf-Attacks* through memory request scheduling algorithms, which is orthogonal to our focus on RH-based *Perf-Attacks*. However, both types of attacks can coexist, potentially exacerbating performance degradation. Thus, addressing both attacks is critical.

F. Randomization in Memory Systems

Randomization is widely used to enhance the reliability and security of memory systems. For instance, cache randomization techniques [33], [42], [70], [71] like CEASER [51], [52] and MIRAGE [56] randomize line-to-set mappings to defend against conflict-based cache attacks. Similarly, other work [53], [63] leverages randomization in Non-Volatile Memory mappings for wear-leveling.

VIII. Conclusion

RowHammer (RH) vulnerabilities pose a severe threat to modern memory systems. While shared counters and tracking structures offer cost-effective mitigation, they are vulnerable to Performance Attacks (*Perf-Attacks*) that exploit these structures, reducing DRAM bandwidth and causing performance drops. We address these challenges with secure hashing mechanisms. Our paper introduces DAPPER, a novel low-cost tracker resilient to *Perf-Attacks*, even at low RH thresholds. We first present DAPPER-S, a tracker template utilizing secure hashing, and further enhance it with DAPPER-H, which integrates double-hashing, advanced reset, and bit-vector strategies. Our security analysis shows that DAPPER-H effectively mitigates *Perf-Attack* patterns. Experimental results on 57 workloads from six benchmark suites show that DAPPER-H incurs just a 0.9% under active *Perf-Attacks*, requiring only 96KB of SRAM per 32GB of DRAM.

IX. Acknowledgments

This project is a part of the Systems and Architecture Laboratory (*STAR Lab*) at the University of British Columbia (UBC). We thank the Advanced Research Computing (ARC) Center team at UBC [1]. We also thank the anonymous reviewers (MICRO 2024 and HPCA 2025) for their invaluable feedback. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) [funding reference number RGPIN-2019-05059] and a gift from Meta Inc. The views and conclusions contained herein are those of the authors. They should not be interpreted as representing the official policies or endorsements of NSERC, the Canadian Government, Meta Inc., or UBC.

References

[1] "UBC Advanced Research Computing, "UBC ARC Sockeye." UBC Advanced Research Computing, 2019, doi: 10.14288/SOCKEYE."

[2] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.

[3] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 9–12, 2020.

[4] F. N. Bostanci, I. E. Yüksel, A. Olgun, K. Kanellopoulos, Y. C. Tuğrul, A. G. Yağliçi, M. Sadrosadati, and O. Mutlu, "Comet: Count-min-sketch-based row tracking to mitigate rowhammer at low cost," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 593–612.

[5] F. Canale, T. Güneysu, G. Leander, J. P. Thoma, Y. Todo, and R. Ueno, "SCARF – a Low-Latency block cipher for secure Cache-Randomization," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1937–1954. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/canale

[6] O. Canpolat, A. G. Yağlıkçı, G. F. Oliveira, A. Olgun, O. Ergin, and O. Mutlu, "Understanding the security benefits and overheads of emerging industry solutions to dram read disturbance," in *Workshop on DRAM Security (DRAMSec)*, 2024.

[7] O. Canpolat, A. G. Yağlıkçı, A. Olgun, I. E. Yuksel, Y. C. Tuğrul, K. Kanellopoulos, O. Ergin, and O. Mutlu, "Breakhammer: Enhancing rowhammer mitigations by carefully throttling suspect threads," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 915–934.

[8] K. Chandrasekar, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "Towards variation-aware system-level power estimation of drams: an empirical approach," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488762

[9] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 55–71.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[11] S. P. E. Corporation, "Spec cpu2006 benchmark suite," 2006. [Online]. Available: http://www.spec.org/cpu2006/

[12] A. Fakhrzadehgan, Y. Patt, P. Nair, and M. Qureshi, "Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2022.

[13] A. Foundation, "Apache hadoop." [Online]. Available: http://hadoop.apache.org/

[14] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "Mediabench ii video: Expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, no. 4, pp. 301–318, 2009, media and Stream Processing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014193310900026X

[15] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[16] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 300–321. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15

[17] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[18] S. Hong, D. Kim, J. Lee, R. Oh, C. Yoo, S. Hwang, and J. Lee, "Dsac: Low-cost rowhammer mitigation using in-dram stochastic and approximate counting algorithm," 2023. [Online]. Available: https://arxiv.org/abs/2302.03591

[19] A. Jaleel, G. Saileshwar, S. W. Keckler, and M. Qureshi, "Pride: Achieving secure rowhammer mitigation with low-cost in-dram trackers," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1157–1172.

[20] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 716–734.

[21] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer attacks on AMD zen-based platforms," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1615–1633. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/jattke

[22] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in dram memories," *IEEE CAL*, vol. 14, no. 1, pp. 9–12, 2014.

[23] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 638–651.

[24] K. Kim, J. Woo, J. Kim, and K.-S. Chung, "Hammerfilter: Robust protection and low hardware overhead method for rowhammer," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 212–219.

[25] M. J. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. J. Ham, J. W. Lee, and J. H. Ahn, "Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1156–1169.

[26] M. J. Kim, M. Wi, J. Park, S. Ko, J. Choi, H. Nam, N. S. Kim, J. H. Ahn, and E. Lee, "How to kill the second bird with one ecc: The pursuit of row hammer resilient dram," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 986–1001. [Online]. Available: https://doi.org/10.1145/3613424.3623777

[27] W. Kim, C. Jung, S. Yoo, D. Hong, J. Hwang, J. Yoon, O. Jung, J. Choi, S. Hyun, M. Kang, S. Lee, D. Kim, S. Ku, D. Choi, N. Joo, S. Yoon, J. Noh, B. Go, C. Kim, S. Hwang, M. Hwang, S.-M. Yi, H. Kim, S. Heo, Y. Jang, K. Jang, S. Chu, Y. Oh, K. Kim, J. Kim, S. Kim, J. Hwang, S. Park, J. Lee, I. Jeong, J. Cho, and J. Kim, "A 1.1v 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)*, 2023, pp. 1–3.

[28] 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," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[29] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[30] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-double: Hammering from the next row over," Aug. 2022, 31st USENIX Security Symposium : USENIX Security '22, USENIX '22 ; Conference date: 10-08-2022 Through 12-08-2022.

[31] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 695–711.

[32] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "Twice: preventing row-hammering by exploiting time window counters," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 385–396.

[33] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[34] K. Loughlin, S. Saroiu, A. Wolman, Y. A. Manerkar, and B. Kasikci, "Moesi-prime: Preventing coherence-induced hammering in commodity workloads," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 670–684.

[35] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "Rowpress: Amplifying read disturbance in modern dram chips," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589063

[36] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu, "Ramulator 2.0: A modern, modular, and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 112–116, 2024.

[37] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 735–753.

[38] JEDEC. JESD79-5C. https://www.jedec.org/document_search?search_api_views_fulltext=jesd79-5c.

[39] Micron. DDR5 SDRAM Datasheet. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf.

[40] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in Multi-Core systems," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi

[41] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 146–160.

[42] P. J. Nair, B. Asgari, and M. K. Qureshi, "Sudoku: Tolerating high-rate of transient failures for enabling scalable sttram," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 388–400.

[43] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 72–83.

[44] P. J. Nair, V. Sridharan, and M. K. Qureshi, "Xed: Exposing on-die error detection information for strong memory reliability," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 341–353.

[45] A. Olgun, Y. C. Tugrul, N. Bostanci, I. E. Yuksel, H. Luo, S. Rhyner, A. G. Yaglikci, G. F. Oliveira, and O. Mutlu, "ABACuS: All-Bank activation counters for scalable and low overhead RowHammer mitigation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1579–1596. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/olgun

[46] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. Ho Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1–13.

[47] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 987–1002.

[48] M. Qureshi and S. Qazi, "Moat: Securely mitigating rowhammer with per-row activation counters," 2025.

[49] M. Qureshi, S. Qazi, and A. Jaleel, "Mint: Securely mitigating rowhammer with a minimalist in-dram tracker," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 899–914.

[50] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 699–710.

[51] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[52] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 360–371. [Online]. Available: https://doi.org/10.1145/3307650.3322246

[53] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 14–23. [Online]. Available: https://doi.org/10.1145/1669112.1669117

[54] M. K. Qureshi, D.-H. Kim, S. Khan, P. J. Nair, and O. Mutlu, "Avatar: A variable-retention-time (vrt) aware refresh for dram systems," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 427–437.

[55] SAFARI Research Group, "ABACuS — GitHub Repository," 2023. [Online]. Available: https://github.com/CMU-SAFARI/ABACuS

[56] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating Conflict-Based cache attacks with a practical Fully-Associative design," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1379–1396. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[57] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1056–1069. [Online]. Available: https://doi.org/10.1145/3503222.3507716

[58] A. Saxena, A. Jaleel, and M. Qureshi, "Impress: Securing dram against data-disturbance errors via implicit row-press mitigation," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 935–948.

[59] A. Saxena, S. Mathur, and M. Qureshi, "Rubix: Reducing the overhead of secure rowhammer mitigations via randomized line-to-row mapping," in *Proceedings of the 29th ACM International Conference*

on Architectural Support for Programming Languages and Operating Systems, Volume 2, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1014–1028. [Online]. Available: https://doi.org/10.1145/3620665.3640404

[60] A. Saxena and M. Qureshi, "Start: Scalable tracking for any rowhammer threshold," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 578–592.

[61] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, "Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 108–123.

[62] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[63] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 383–394. [Online]. Available: https://doi.org/10.1145/1815961.1816014

[64] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 612–623.

[65] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[66] Standard Performance Evaluation Corporation, "SPEC CPU2017 Benchmark Suite," 2017. [Online]. Available: http://www.spec.org/cpu2017/

[67] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 8–15.

[68] H. Taneja and M. Qureshi, "Roguerfm: Attacking refresh management for covert-channel and denial-of-service," 2025. [Online]. Available: https://arxiv.org/abs/2501.06646

[69] Transaction Processing Performance Council, "TPC Benchmarks." [Online]. Available: http://tpc.org/

[70] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 494–505, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273440.1250723

[71] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 675–692. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[72] M. Wi, J. Park, S. Ko, M. J. Kim, N. S. Kim, E. Lee, and J. H. Ahn, "Shadow: Preventing row hammer in dram with intra-subarray row shuffling," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 333–346.

[73] J. Woo, S. Lin, P. J. Nair, A. Jaleel, and G. Saileshwar, "Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues," in *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[74] J. Woo, G. Saileshwar, and P. J. Nair, "Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 374–389.

[75] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows," in *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 345–358.

[76] F. Yao, A. S. Rakin, and D. Fan, "DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1463–1480. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yao

[77] A. G. Yağlikçi, A. Olgun, M. Patel, H. Luo, H. Hassan, L. Orosa, O. Ergin, and O. Mutlu, "Hira: Hidden row activation for reducing refresh latency of off-the-shelf dram chips," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 815–834.

[78] J. M. You and J.-S. Yang, "Mrloc: Mitigating row-hammering based on memory locality," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.