

Probabilistic Tracker Management Policies for Low-Cost and Scalable Rowhammer Mitigation

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

Stephen W. Keckler
NVIDIA
skeckler@nvidia.com

Gururaj Saileshwar*
University of Toronto
gururaj@cs.toronto.edu

Abstract—This paper focuses on mitigating DRAM Rowhammer attacks. In recent years, solutions like TRR have been deployed in DDR4 DRAM to track aggressor rows and then issue a mitigative action by refreshing neighboring victim rows. Unfortunately, such in-DRAM solutions are resource-constrained (only able to provision few tens of counters to track aggressor rows) and are prone to thrashing based attacks, that have been used to fool them. Secure alternatives for in-DRAM trackers require tens of thousands of counters.

In this work, we demonstrate secure and scalable rowhammer mitigation using resource-constrained trackers. Our key idea is to manage such trackers with probabilistic management policies (PROTEAS). PROTEAS includes component policies like request-stream sampling and random evictions which enable thrash-resistance for resource-constrained trackers. We show that PROTEAS can secure small in-DRAM trackers (with 16 counters per DRAM bank) even when Rowhammer thresholds drop to 500 while incurring less than 3% slowdown. Moreover, we show that PROTEAS significantly outperforms a recent similar probabilistic proposal from Samsung (called DSAC) while achieving 11X - 19X the resilience against Rowhammer.

I. INTRODUCTION

DRAM scaling has led to higher DRAM capacities by packing DRAM cells more closely. This has increased inter-cell interference, leading to the problem of Rowhammer [21], whereby rapid activations of one DRAM row can cause charge leakage and bit-flips in neighboring rows. Such Rowhammer bit-flips are not just a reliability problem, but also a major security threat. Numerous studies have illustrated exploits using Rowhammer [1], [5], [8]–[10], [23], [37], [41]. Moreover, the vulnerability is worsening with each DRAM generation. The number of activations required to induce bit-flips, called the Rowhammer threshold (TRH), has dropped from 140K in DDR3 [21] to just 4.9K in LPDDR4 [18] over the last decade. Thus, there is a growing need for effective and scalable mitigations, as TRH is expected to drop further.

Mitigating Rowhammer effects within the DRAM module has been an ongoing challenge. Since the release of DDR4 in 2015, DRAM manufacturers have deployed an in-DRAM mitigation called Targeted Row Refresh (TRR). TRR and many subsequent solutions rely on a *tracking* mechanism to identify rapidly activated rows or aggressor rows and then issue a *mitigative action* by refreshing the neighboring victim rows [12]. The tracker typically consists of a group of counters within each DRAM bank that counts row activations and issues mitigations in the background of regular refresh commands when they are issued to DRAM by the memory controller.

Unfortunately, given the limited storage capabilities within the logic space of a DRAM module, TRR implementations often store less than 32 counters per DRAM bank [6], [13], [15]. Such limited capacity of in-DRAM trackers has made them significantly vulnerable to thrashing-based attacks such as TRRespass [8] and Blacksmith [15], which ensure aggressor rows are evicted from the tracker by activating a larger number of rows than the tracker capacity, as shown in Figure 1(b). Such thrashing-based attacks can continue to activate untracked rows far beyond TRH without a mitigation, thus inducing Rowhammer bit-flips and rendering resource-limited trackers such as TRR non-secure.

Emerging trackers attempt to avoid such thrashing-based attacks with deterministic tracking algorithms, which require a larger number of counters. Graphene [31] maintains activation counts using the Misra-Gries algorithm [30]; recent proposals [19], [27] store such counters within the DRAM logic area. However, the required counters per bank increases as the TRH decreases; as shown in Figure 1(a), Graphene and similar solutions require $\sim 5K$ counters per bank at TRH of 500 (680KB of SRAM per DRAM rank in DDR5 [32]); such high storage overheads make these solutions impractical for in-DRAM adoption. Alternative solutions maintain one counter per row, requiring 8K to 16K counters per bank. These counters are stored in memory [17], [32] and require additional DRAM accesses to fetch and update the counters, leading to high worst-case performance overheads of up to 70% [32]; storing them entirely within the DRAM array [3] requires complex redesigns of the DRAM MATs. Consequently, such deterministic trackers requiring thousands of counters per bank have been difficult to adopt in commodity DRAM.

While trackerless solutions such as PARA [21] exist which probabilistically issue mitigations to adjacent rows on activations, incurring no storage overhead, such solutions however cannot be implemented transparently within the DRAM. They are required to be implemented only within the memory controller, as mitigative refresh commands cannot be issued by the DRAM transparently after any given activation.

For an in-DRAM only solution, we seek a defense that (a) incurs low storage (b) has strong security (c) has low performance overheads (d) scales to low row hammer thresholds ($< 1K$), and (e) is compatible with current DRAM interfaces. Thus, we focus on low overhead mechanisms to secure in-DRAM trackers (like TRR) by making them thrash-resistant.

*Gururaj contributed to this work while he was affiliated with NVIDIA.

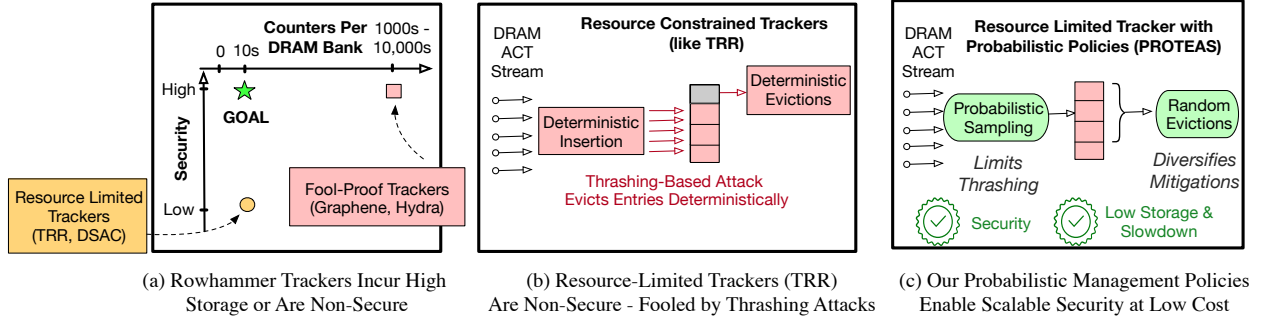


Fig. 1. (a) As Rowhammer thresholds drop below 1K, Rowhammer trackers require 1000s to 10,000s of counters to be secure, while resource-constrained trackers (like TRR or DSAC) with tens of counters, which are practical within the logic area in DRAM, provide little to no security. (b) This is because trackers with tens of counters are vulnerable to thrashing-based attacks, which can easily evict tracked entries to fool the tracker, exploiting its deterministic management. (c) PROTEAS provides strong security with a small 16-entry tracker, via thrash resistance through probabilistic sampling and random evictions.

This paper shows that in-DRAM trackers can be made thrash-resistant by enhancing the tracker management policies, which heavily influence the tracker’s security properties. Conventional tracker management policies typically consist of (a) a lookup policy (which DRAM activations should consult the tracker) (b) an update policy (how to update tracker state on hits) (c) an eviction policy (which entry to evict when it is capacity-limited), (d) an insertion policy (whether to insert entries on a miss), and finally (e) a mitigation policy (which entry to issue a mitigation for). For instance, the in-DRAM tracker in TRR [13] has a lookup policy where the activation stream is sampled at deterministic time instances, and mitigation/eviction policies, though not documented, that are also likely deterministic (as attack patterns aligned to mitigation instances have shown better success in deterministically evicting entries [6]). Such deterministic policies are the root cause of vulnerability to thrashing-based attacks.

Based on this observation, we propose *PRObabilistic TrackerEr mAnagement policieS* (PROTEAS) to enable thrash-resistant in-DRAM trackers. We observe that non-determinism can be best introduced by varying the insertions and evictions, rather than the mitigation policy. To that end, PROTEAS proposes (i) probabilistic sampling to minimize the number of insertions into the tracker and (ii) randomize replacement to ensure that a diversity of rows are retained in the tracker.

We systematically study two different sampling approaches to minimize thrashing. First, thrashing can be prevented by modifying the tracker insertion policy to perform *Probabilistic Miss Stream Sampling* (PMSS), where only a subset of the misses are inserted into the tracker while the others are bypassed (similar to the Bimodal Insertion Policy (BIP) [33] for caches). A similar scheme was proposed recently by Samsung’s probabilistic tracker, DSAC [14]¹. However, we show that sampling the miss stream is ineffective as the periodic insertions continue to thrash the small tracker. Alternatively, we can modify the tracker lookup policy to perform *Probabilistic Request Stream Sampling* (PRSS) where only a subset of the request stream looks up the tracker while the rest are bypassed. We observe that PRSS performs significantly better than PMSS (or DSAC) because sampling the request stream results in a working set smaller than the tracker capacity. This ensures that inserted rows get mitigated without the adversary being

able to thrash them, and thus provides much stronger protection against thrashing-based attacks. Consequently, PROTEAS adopts PRSS for its sampling-based thrash-resistance.

However, thrash resistance alone is not sufficient. If the sampling rate is not carefully selected, too low of a sampling rate can allow requests that escape sampling to be used for rowhammer, and too high of a sampling rate can lead to thrashing. Hence, we derive the optimal sampling rate where the tracker just begins to thrash, when the insertion probability equals the mitigation probability of the tracker (1 per tREFI). To prevent such thrashing from being exploitable, PROTEAS proposes a random replacement policy to ensure a diversity of rows are stored in the tracker, and thus the frequency-based mitigations are to diverse rows, as shown in Figure 1(c). The tracker sampling decisions are based on a pseudo-random number generator (PRNG) with a secret seed (which is periodically changed), so the attacker cannot align with and exploit sampling instances [15].

We evaluate PROTEAS on 500 uniform and non-uniform attack patterns based on TRRespass [8] and Blacksmith [15]. Across all patterns, we measure *maximum disturbance*, the maximum activations any row receives before a mitigation. With 1 mitigation per tREFI, PROTEAS limits the maximum disturbance to 2.1K, making it a suitable defense for current (LP)DDR4 DRAM with TRH of 4.9K to 9K [18], [22].

We also co-design PROTEAS with RFM (a feature in DDR5), which allows extra mitigation opportunities to the DRAM. We observe that a baseline deterministic tracker (like TRR), benefits minimally from additional RFM mitigations, as it is easily thrashed even in the shorter period between mitigations; it thus suffers a maximum disturbance of 73K to 64K activations with 1 to 8 mitigations per tREFI. Whereas, PROTEAS with RFM significantly benefits from additional mitigations: with 4 mitigations per tREFI, PROTEAS limits the maximum disturbance to 600, which reduces to 300 with 8 mitigations per tREFI. This makes PROTEAS a scalable defense even as TRH drops to 1K or 500 in the future.

¹ PROTEAS aligns closely with recent work by Samsung (i.e., DSAC [14]) and SK Hynix [20] who also employ non-determinism through PRNG based Rowhammer trackers. Samsung’s DSAC uses miss-sampling (like PMSS), making it less thrash resistant; in Section VI-B we find that it is insecure for Blacksmith-like attacks [15]. Hynix provides insufficient details for us to provide a reasonable security comparison.

Compared to recent probabilistic designs, our in-DRAM solution PROTEAS achieves a maximum disturbance that is 15% lower than PARA [21], a trackerless solution in the memory controller, and 19X lower than Samsung’s DSAC [14], a probabilistic in-DRAM tracker at equivalent mitigation costs.

Overall, this paper makes the following contributions:

- 1) We show that effective tracker management can enable in-DRAM trackers to be secure against row hammer attacks at Rowhammer thresholds of 1K or below.
- 2) We propose PROTEAS, a probabilistic tracker management policy, that uses request stream sampling and random replacement to prevent tracker thrashing.
- 3) We demonstrate how a recent probabilistic tracker, Samsung’s DSAC [14], is insecure and incurs a max disturbance beyond TRH, and 19X higher than PROTEAS.
- 4) We co-design PROTEAS with RFM on DDR5 systems to show PROTEAS can significantly lower the max disturbance below TRH of 1K and 500, unlike deterministic trackers (like TRR) in commodity DRAM which benefit minimally from extra RFM mitigations.

To our knowledge, this is the first work to systematically analyze the relationship between a tracker’s management policies and its susceptibility to rowhammer. Our Gem5 evaluations with SPEC CPU2017 workloads show that the average performance impact is less than 1% for a TRH of 1K and 3% for TRH of 500. Our in-DRAM implementation incurs negligible storage (less than 3KB per DRAM rank) that is similar to current TRR implementations.

II. BACKGROUND AND MOTIVATION

A. DRAM Architecture and Parameters.

To access data from DRAM, the memory controller first issues an activate (ACT) for a DRAM row to read it into a row-buffer, the 64B data is then transferred to the bus, and then the row may be closed and pre-charged (PRE). For reliable operation, the memory controller ensures a minimum time gap of t_{RC} between two successive ACTs to a bank. The charge of a DRAM row must also be refreshed periodically every t_{REFW} time period. For this, the memory controller issues a REF command every t_{REFI} period to refresh a subset of rows. There are 8192 REFs issued every t_{REFW} period to cover all DRAM rows. Table I provides these timing parameters.

TABLE I
DRAM PARAMETERS (FROM MICRON DDR4 DATASHEET [29])

| Parameter | Explanation | Value |
|---------------------|--|---------|
| t_{REFW} | Refresh Period | 64 ms |
| t_{REFI} | Time between successive REF Commands | 7800 ns |
| t_{RFC} | Execution Time for REF Command | 350 ns |
| t_{RC} | Time between successive ACTs to a bank | 45 ns |
| $ACTs-per-t_{REFI}$ | $(t_{REFI} - t_{RFC}) / t_{RC}$ | 165 |

B. DRAM Rowhammer Attacks

Kim et al. [21] discovered that rapid activations of a DRAM row (called Rowhammer) can cause charge leakage and bit flips in neighboring rows. The heavily activated row is called an “aggressor” row and the neighboring row with a bit-flip

is called the “victim” row. These bit flips can occur up to distance of 2 rows from the aggressor rows (also known as the “blast-radius”) [22]. The minimum number of activations to an aggressor row to cause a bit-flip in a victim row is called the “Rowhammer threshold” (TRH). With DRAM scaling, TRH has dropped significantly from 139K for DDR3 in 2014 [21] to 10K for DDR4 [18] and just 4.8K–9K for LPDDR4 in 2020 [18], [22]. (see Table II).

Rowhammer is not only a reliability issue but has also become a critical security threat. Numerous exploits have shown that Rowhammer bit-flips in page-tables or sensitive binaries can be used by attackers to escalate to kernel-level privileges [8], [9], [37], [46], or the data-dependent nature of the bit-flips can be used to leak confidential data [23]. Thus, it is imperative to reduce the risk of Rowhammer attacks.

TABLE II
ROWHAMMER THRESHOLD OVER TIME

| DRAM Generation | RH-Threshold |
|-----------------|------------------------|
| DDR3 | 22.4K [18] - 139K [21] |
| DDR4 | 10K [18] - 17.5K [18] |
| LPDDR4 | 4.8K [18] - 9K [22] |

C. Threat Model

Our threat model assumes an attacker with native execution capability, that can issue memory requests for arbitrary addresses. We assume the attacker knows the defense algorithm, but does not have physical access to the memory controller or DRAM to learn any secret information stored inside DRAM (e.g., seed used for random-number generator). Our defense aims to prevent all known forms of Rowhammer attacks, including TRRespass [8] and Blacksmith [15], that attempt to fool the tracker or Half-Double [22] which fools mitigative refresh. The recent RowPress [26] attack is out-of-scope, since its effects are orthogonal to Rowhammer and can be mitigated with a paging policy that limits the time a row is kept open.

D. Targeted Row Refresh (TRR) Mitigation in DDR4

DDR4 modules support in-DRAM mitigation against Rowhammer called Targeted Row Refresh (TRR) [8]. TRR maintains an aggressor row tracker within each DRAM bank. Each tracker entry holds the DRAM row number and a frequency counter. When the memory controller issues a REF command every t_{REFI} period, the DRAM issues a mitigation for the highest activated row in the tracker. The mitigation involves refreshing the neighboring victim rows (in a given blast radius) during t_{RFC} in the background of a REF command.

Recent Attacks. Recent studies [8], [13], [15] have shown that trackers typically contain less than 32 entries per bank due to DRAM storage constraints. The *TRRespass* [8] attack exploited this limited storage by uniformly hammering a large number of rows beyond the tracker capacity, to thrash the tracker, and evict resident entries. As a result, the hammered rows that are evicted escape a mitigation, and bit-flips can be induced by fooling the tracker. Recent work [15] also observed that some tracker implementations may perform deterministic temporal sampling of activations for insertion into the tracker. The *Blacksmith* attack [15] exploited this to

flip bits by aligning its hammering pattern to escape sampling and increasing the intensity of hammering for these rows.

Refresh Management (RFM). DDR5 includes a new feature, RFM, where memory controller can issue additional mitigations (RFM commands) to the DRAM when the activations per bank crosses a threshold. However, the actual row to be mitigated still depends on the in-DRAM tracker implementation. Consequently, if a vulnerable tracker like TRR [8], [15] can be fooled (by thrashing or escaping sampling) between mitigations, it can allow existing attacks to continue despite additional RFM mitigations, as we shown in Section VI-B.

Resource-constrained trackers (like TRR with <32 entries) are susceptible to attacks exploiting thrashing-based evictions or deterministic sampling to escape insertions.

where Hydra’s filtering may be ineffective, the worst-case slowdown can be as high as 70%.

Panopticon: Panopticon [3] and PRHT from Hynix [20] propose per-row counters stored within the DRAM array. This requires 512K counters per DRAM rank, as before, which may be updated without additional DRAM accesses. However, such solutions require a significant redesign of the DRAM MATs, and additional logic to store and update these counters in the background of a DRAM ACT, which can additionally impact DRAM timings. The required complex redesign of the DRAM arrays makes such solutions less desirable.

State-of-the-art trackers require large storage overhead (87K – 512K counters per DRAM rank) or require significant SRAM / DRAM changes making them undesirable.

E. Storage Overheads of State-of-the-Art Trackers

State-of-the-art trackers typically store a large number of entries to deterministically prevent thrashing-based attacks. Such trackers may be stored in SRAM, DRAM or in both SRAM and DRAM. At one end of the spectrum, Graphene [31] and Mithril [19] perform approximate counting and require fewer counters, while at the other end of the spectrum, solutions utilize one counter per row [3], [17], [32] to perform exact tracking. Other trackers [24], [38], [44] lie between these.

Graphene: Graphene [31] uses activation counters based on Misra-Gries summaries, a solution to the frequent element problem, to identify aggressor rows from a stream of row activations in the memory controller. When any counter in the tracker crosses a predefined threshold ($TRH/2$), the associated row is mitigated. The counts in this tracker are always guaranteed to be greater than or equal to the actual row activation counts, so long as the number of tracker entries is greater than $ACTs\text{-per-tREFW} / (TRH/2)$. For TRH of 500, this requires 5440 counters per DRAM bank (87K counters per rank) consuming 240 KB SRAM per DRAM rank; additionally, this must be organized as a 5400-entry CAM per bank, which may be beyond practical capabilities [32].

Mithril: Mithril [19] stores a similar Misra-Gries tracker within DRAM, and co-designs it with RFM to allow a smaller tracker size when the mitigations are issued at a higher frequency (with lower RFM_{TH}). Unfortunately, Mithril also faces a similar problem that at low thresholds of 500, to keep performance overheads low, it requires a tracker of few thousand counters per DRAM bank, which is impractical given that deployed in-DRAM defenses only have tens of counters.

Hydra: Hydra [32] and CRA [17] store one counter per DRAM row, in a reserved portion of the DRAM. A 4GB DRAM Rank (with 8KB rows) requires 512K counters per rank. Unfortunately, Hydra faces performance overheads due to additional DRAM accesses to fetch and update the counters. Although these solutions deploy complex caching and filtering mechanisms on-chip within the memory controller to avoid extra DRAM accesses, recent work [32] shows that the average slowdown for CRA is 25%, and for pathological workloads

F. Challenge with Existing Probabilistic Mitigations

PARA [21] or PRA [17] are trackerless probabilistic solutions that issue a neighboring row activation from the memory controller with a probability p on each DRAM activation. Such solutions are trackerless, so they incurs no storage overheads. Unfortunately, this cannot be implemented transparently within the DRAM as only the memory controller can issue additional activation commands at arbitrary time instances to specific rows; thus, such solutions must be implemented within the memory controller. This also means that the memory controller requires the knowledge of adjacent rows, which is not currently exposed by DRAM to the memory controller. Prior works [39], [45] have attempted to minimize mitigations in PARA, but at the cost of lowering the security.

Recent proposals like PRoHIT [39] and Samsung’s DSAC [14] employ stochastic replacement in resource constrained in-DRAM trackers, to filter out decoy rows in TR-Respass attacks. DSAC achieves this by modifying the tracker insertion policy to progressively reduce the probability of eviction from the tracker on misses as the minimum count in the tracker increases. PRoHIT uses two tracker tables (hot table and cold table) and probabilistically promotes entries from the cold to hot table to limit thrashing. While such designs provide security against TRRespass attack, they are vulnerable to newer attacks: e.g., DSAC’s consistent reduction in insertion probability allows rows that escape sampling to still be hammered (like in the Blacksmith attack [15]). In Section VI-B, we show that both PRoHIT and DSAC are insecure against Blacksmith-like attack patterns, which were not evaluated in the original submissions.

G. Goal: Low Cost, Scalable, and Secure Trackers

Ideally, we desire a secure Rowhammer mitigation solution that incurs low performance and storage overheads and is compatible with existing DRAM protocols to be easily adoptable by current and future DRAM. Furthermore, we desire a scalable solution as Rowhammer thresholds drop to 500 (by 10x compared to 2020 levels). To that end, we study storage-limited trackers (like TRR) and redesign them to be secure by systematically enabling them to be thrash-resistant.

TABLE III
ATTACK PATTERN OF ACTIVATIONS UNDER STUDY

| Type | Pattern | Parameter Sweep |
|-------------|--|--|
| Uniform | $(r_1, \dots, r_j)^N$ | $j = 2, 4, 8, 16, 20, 32, 40, 80, 120, 140$ |
| Non-Uniform | $[(r_1, \dots, r_j)^X, (d_1, \dots, d_k)]^N$ | $j = 2, 4, 8, 16, 20, 32, 40, 80, 120, 140$ $X = 2, 3, 4, 5$ $k = 5, 10, 20, 32, 40, 80$ |

III. METHODOLOGY

To analyze the thrash-resistance properties of trackers, we use an empirical methodology based on access patterns derived from recent row hammer attacks, like TRRespass [8], Blacksmith [15], and SMASH [6]. We develop a trace-driven Rowhammer simulator which models our trackers and uses DRAM activation traces of attacks as input.

Attack Patterns: We use *uniform* [8] and *non-uniform* [15] attack patterns (see Table III). A uniform attack pattern resembles a TRRespass attack [8], consisting of a cyclical reference to a set of target rows r with length j (larger than the tracker capacity). A non-uniform pattern is like Blacksmith [15], where activations occur to a set of target rows r with length j with a higher intensity (X), compared to a second set of rows d with length k , with an intensity of 1. The difference in intensity (X) and sequence lengths (j, k) achieves the effect of varying intensity, phase, and frequency like in Blacksmith [15]. We evaluate 10 uniform activation patterns for different values of j (see Table III). For non-uniform activation patterns, we evaluate ten different values for j , four different values for X , and six different values for k for a total of 240 non-uniform patterns. Recent attacks [6] show that aligning activation patterns to tREFI can further increase the success of row hammer attacks. Thus, we evaluate aligned and unaligned versions of our 250 attack patterns for a total of 500 patterns. These patterns cover footprints of 2 – 220 unique addresses.

Simulator: We use a trace-based simulator modeling an aggressor-row tracker assuming DRAM parameters listed in Table I. We assume a baseline 16-entry fully-associative tracker. We run our 500 attack patterns through the simulated tracker. For our baseline tracker, on hits, the frequency counter of the associated entry is incremented. On every miss, a new entry is inserted into the tracker with the frequency counter set to zero. In the event the tracker is full, the Least Frequently Used (LFU) entry in the tracker is evicted. At tREFI intervals, the Most Frequently Used (MFU) entry is selected for mitigation after which the entry is invalidated. The attack patterns are continually repeated for a value N that covers one refresh interval. Like prior work [14], we report “maximum disturbance” which is the maximum activations received by any row in the attack pattern before being refreshed.

IV. PROBABILISTIC TRACKER MANAGEMENT

We first formalize the management policies of a typical aggressor-row tracker. We then systematically study probabilistic tracker management policies for thrash protection.

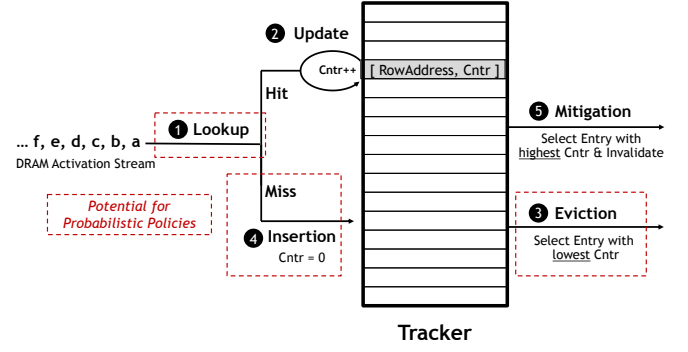


Fig. 2. Overview of Tracker Management Policies and Potential for Probabilistic Policies for Thrash Resistance

A. Formalizing Tracker Management Policies

Structure. Figure 2 shows a representative aggressor-row tracker typical in DRAM defenses like TRR. We study small trackers (e.g. 16 entry) which can be fully-associative in-DRAM tables. Each entry stores the DRAM row address and a frequency counter, which counts the hits received by the entry while resident in the tracker. Since the entries with the highest counts are selected for mitigation, a fool-proof tracker management policy that retains frequently activated rows is critical for effective mitigation.

Policies. As shown in Figure 2, tracker management is composed of many different policies: ① Lookup Policy, ② Update Policy, ③ Eviction Policy, ④ Insertion Policy, and ⑤ Mitigation Policy. In state-of-the-art trackers (like TRR [13] or Graphene [31]), the Lookup Policy typically consults the tracker on *all* (or a deterministic subset of) DRAM activations [15]. On a hit, the Update Policy increments the associated counter. On a miss, the Insertion Policy typically inserts *all* missing entries into the tracker [31] with a frequency count of zero. If the tracker is full², the Eviction Policy typically employs a Least Frequently-Used (LFU) policy where the entry with the lowest count is evicted. Finally, when issuing a mitigation, the Mitigation Policy selects the entry with highest count for mitigation [12], [31], after which the entry is invalidated to enable other frequently accessed rows to receive mitigations. Without loss of generality, we assume mitigations are issued at tREFI intervals and refresh neighboring rows within a blast-radius of 2 (to protect against Half-Double [22]).

B. Probabilistic Tracker Management Policies (PROTEAS)

Unfortunately, *deterministic* policies in state-of-the-art trackers are susceptible to *thrashing* attack patterns. Such attack patterns are designed with a *working set* larger than the tracker capacity to cause thrashing, or are cleverly designed to avoid insertions into the tracker exploiting the deterministic policies. To address this, we enable thrash resistance with PROBABILISTIC Tracker Management policies (PROTEAS)³.

²The insertion policy prioritizes inserting into invalid entries first.

³Named after the Greek God Proteus who could change his form at will. PROTEAS probabilistically changes insertion patterns seen by the tracker.

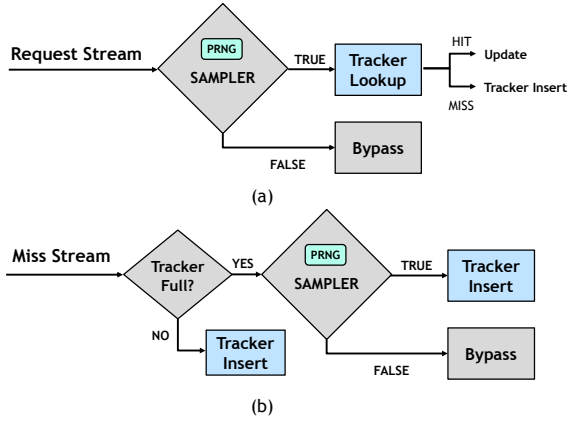


Fig. 3. Probabilistic Sampling of (a) Request Stream and (b) Miss Stream

PROTEAS consists of two key components to introduce non-determinism and diversity in rows resident within the tracker: probabilistic sampling of insertions to avoid thrashing and a random-replacement based Eviction Policy. Unlike conventional tracker policies, both of these mechanisms enable diversity in mitigations. We systematically explain the design of these two components and their benefits. We retain the baseline Update and Mitigation Policy in Figure 2 (i.e., frequency counter update on hits and MFU-based mitigations) since we still desire that the most heavily activated rows are tracked and mitigated accurately.

C. Probabilistic Sampling to Limit Tracker Thrashing

Our goal is to prevent thrashing based attacks from being able to fool the tracker by deterministically evicting tracked entries. Probabilistic sampling of insertions aims to reduce the number of insertions to limit thrashing capability of the attack. At the same time, the probabilistic sampling decisions are based on a pseudo-random-number generator (PRNG), whose seed is a secret stored within the DRAM and not known to the attacker, so the attacker may not surgically avoid sampling into the tracker. Moreover, the seed can be changed periodically, to prevent an attacker from reverse-engineering the sequence. Intuitively, sampling to limit insertions can be implemented either at the Miss Stream or Request Stream. We analyze both.

Probabilistic Miss Stream Sampling (PMSS): Typical Insertion Policies insert *all* missing entries into the tracker. When the tracker is full, the Eviction Policy is used to select a suitable victim. Consequently, when the access pattern has a footprint that is larger than the tracker size, the tracker starts thrashing. Past studies have shown that thrashing can be avoided by inserting only a subset of the miss stream and bypassing the rest [33]. Figure 3(b) illustrates the implementation of PMSS. On a Miss, if there is available capacity in the tracker, the missing entry is installed in the empty locations. If the tracker is full, PMSS uses a PRNG with a sampling probability p to insert a subset of misses into the tracker. This preserves a portion of the working set in the tracker, and intermittently installs new entries, thus allowing new untracked rows to be tracked. In fact, such a technique is deployed in

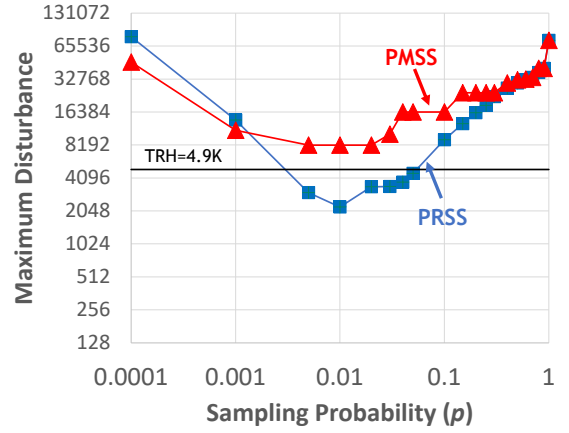


Fig. 4. Sensitivity of PMSS and PRSS to Sampling Probability (p). PRSS achieves a significantly lower minima for the maximum disturbance (2.2K) compared to PMSS (5.1K), due to its more effective thrash reduction.

thrash-resistant cache replacement policies like BIP [33] and in DSAC’s stochastic replacement for aggressor-row tracking.

Probabilistic Request Stream Sampling (PRSS): An alternative approach to limit thrashing is to probabilistically sample the request stream and only use a subset of the activations to consult the tracker. As shown in Figure 3(a), PRSS uses a sampling probability p , based on a PRNG, to select a subset of the requests to lookup the tracker. The key idea is that frequently accessed rows have a higher chance of being sampled. Only the sampled requests update on hits and or can insert into the tracker on misses, while the non-sampled requests bypass the tracker.

For both PMSS and PRSS, the sampling probability p must be selected appropriately. If p is high, the tracker can get thrashed; if p is very low, the non-sampled activations can induce sufficient hammering while escaping mitigations.

Results. Figure 4 shows the *maximum disturbance* (i.e., the maximum number of activations any DRAM row receives before a mitigation) as the sampling probability p varies for PRSS and PMSS (using the baseline LFU replacement), across the 500 attack patterns described in Section III. The baseline policy of consulting the tracker on all requests/misses ($p = 1$ or 100%) incurs a maximum disturbance exceeding 64K, indicating that the tracker can be easily thrashed. As p reduces, PRSS and PMSS both reduce tracker thrashing because the maximum disturbance decreases until $p = 0.01$ (1%). However, as p reduces below 1%, maximum disturbance starts increasing again. This is because with such low sampling rates, the tracker is severely underutilized, enabling non-sampled rows to induce hammering. This suggests that the sampling rate must ensure at least one insertion per tREFI period. With a maximum of 165 activations per tREFI (see Table I), ensuring at least one entry in the tracker is populated requires a sampling probability of at least $1/165$ (i.e., 0.6%).

The lowest maximum disturbance value for PRSS is 2.2K while that of PMSS is 8.1K (both at $p = 0.01$ (or 1%)). PRSS significantly outperforms PMSS because of its underlying implementation. PMSS by design ensures that the tracker

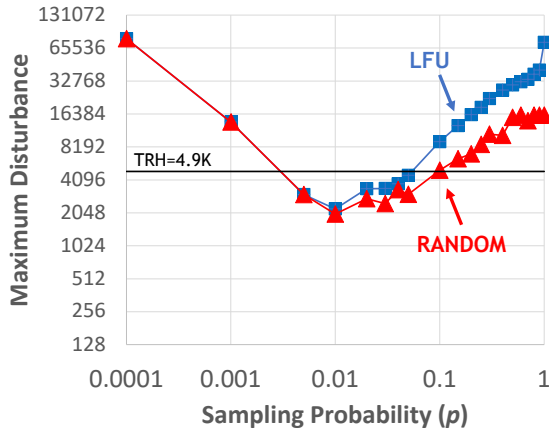


Fig. 5. Sensitivity of PRSS to Replacement Policy. PRSS with Random Replacement achieves a 10% lower minima for maximum disturbance (2K) compared to LFU Replacement (2.2K).

stays fully occupied because invalid entries are prioritized for insertion (see Figure 3(b)). Thus, in the steady state, whenever PMSS performs an insertion, a tracker entry must be evicted. As such, PMSS still thrashes the tracker and the portion of access pattern that are not resident escape mitigation similar to the baseline. On the other hand, with PRSS, the tracker does not thrash and the occupancy is often below the maximum capacity (on average 80% in our studies) and it performs best when the sampling rate equals the mitigation rate. Given the low chance of eviction, the sampled activations have a higher chance of receiving mitigations, and thus PRSS achieves a much lower maximum disturbance than PMSS. Thus for PROTEAS, we choose PRSS with $p = 0.01$ (1%) as our default design. As our tracker just begins to have infrequent evictions at this probability, we now study replacement policy.

D. Random Replacement Policy for Diversity in Tracking

On tracker misses, state-of-the-art trackers [14], [31] employ least frequency used (LFU) replacement to select an eviction candidate. The goal is to retain heavily accessed rows resident in the tracker so they have the opportunity to be selected for mitigation. However, the disadvantage of such deterministic replacement is that it can be exploited by attack patterns to dislodge target entries deterministically. In doing so, the target entries may be used for continued hammering while escaping mitigations (since they are not resident in the tracker). To address this vulnerability, we propose *random replacement* on evictions. This ensures that evictions are unpredictable to the adversary, thus thwarting attacks that may attempt to dislodge a specific entry. Furthermore, random replacement ensures a diversity of rows are retained in the tracker allowing diversity in mitigations. Note that we only modify the Eviction Policy to use random replacement. We retain and continue to update the frequency counters used by the MFU-based Mitigation Policy, which selects the entry with the highest counter for mitigation.

Results. Figure 5 shows the maximum disturbance for PRSS with LFU and Random replacement. We observe that PRSS with Random replacement is almost always better than LFU replacement. In fact, it achieves a 10% lower minima for

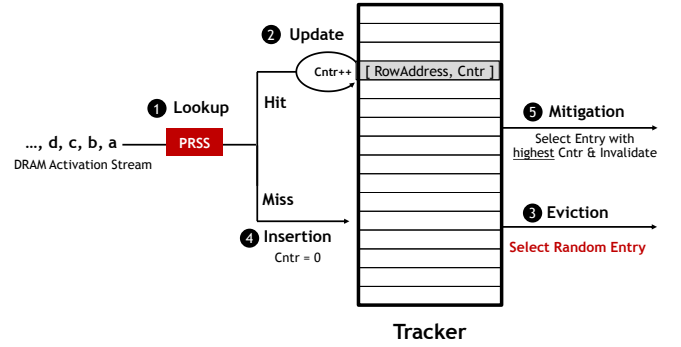


Fig. 6. Overview of Probabilistic Tracker Management Policies (PROTEAS)

the maximum disturbance (2K) compared to LFU based replacement (2.2K). This is because random replacement further avoids thrashing from being easily exploitable by introducing non-determinism into the tracker replacement behavior.

E. Putting it all together: PROTEAS

Based on Figure 5, PROTEAS uses PRSS with random replacement as the default design choice. Figure 6 shows the complete design of PROTEAS, with the PRSS and random replacement enhancements. As illustrated in the figure, PROTEAS requires minimal changes to the baseline tracker design. We introduce only two pseudo-random number generators (PRNG): one for PRSS and one for random replacement. Section VI-G discusses the design overhead of these modifications.

F. Sizing the Probabilistic Tracker

Figure 7 illustrates the sensitivity of PROTEAS to tracker size as it varies from 2 to 128 counters. Across tracker sizes, the lowest value for the maximum disturbance is achieved close to $p = 0.01$ (1%), at which point the tracker insertion rate (approximately 1.6 per tREFI) is close to the mitigation rate (1 per tREFI). As tracker size increases from 2 to 128, the maximum disturbance decreases. For a 16-entry tracker, the maximum disturbance (at $p = 0.01$) is 2K. As the tracker size increases to 32, 64, and 128, the maximum disturbance decreases to 1834, 1538, and 1431 respectively. This is because the extra tracker capacity makes the attacker take longer to thrash the tracker. Thus, the chance that a resident entry receives a mitigation before it is evicted increases thereby reducing the maximum disturbance for an attack pattern. On the other hand, as tracker size decreases from 16 to 4 and 2, the maximum disturbance increases as high as 2.5K. Out of an abundance of caution, we choose the default tracker size of 16 for PROTEAS (at $p = 0.01$), where the maximum disturbance of 2K is less than half the current TRH of 4.9K.

V. COMBINING PROTEAS WITH RFM FOR SCALABILITY

Thus far, we studied PROTEAS with one mitigation per tREFI, which is issued at the time of the regular refresh during tRFC. Such a design is appropriate for LPDDR4 and DDR4 (and HBM2) memories, where the opportunity for issuing

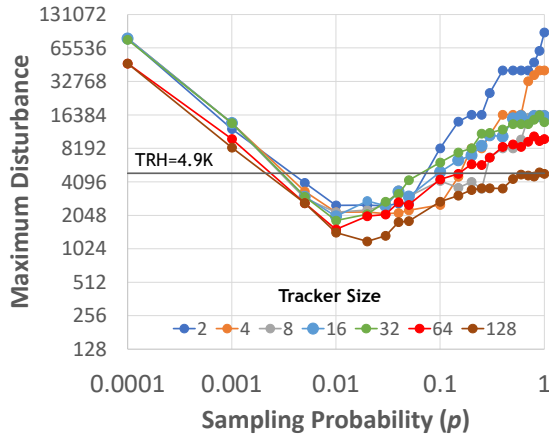


Fig. 7. Sensitivity of PROTEAS to Tracker Size (varies from 2 to 64 counters)

mitigations is only on DRAM refresh commands. Given the limited tRFC, it is not practical to issue mitigations for more than one aggressor row per tREFI which involves refreshing victim rows within a blast radius of 2 or 4.

In DDR5 and HBM3 standards, the memory controller can issue Refresh Management (RFM) commands [19] to the DRAM for additional mitigative refreshes per tREFI. We now show how PROTEAS can leverage these additional mitigations per tREFI to further limit the maximum disturbance.

Figure 8 shows our design of PROTEAS with RFMs for DDR5 and LPDDR5 (and HBM3). The memory controller maintains a Rolling Accumulation of ACTs (RAA) counter [19] per bank. When any RAA counter crosses a threshold, RFM_{TH} , the memory controller resets the RAA counter and issues an RFM command for the corresponding DRAM bank. To ensure k mitigations per tREFI under continuous activations to a DRAM bank, we set the RFM_{TH} to $ACTs-per-tREFI/k$, i.e., once every $165/k$ activations. On each RFM-based mitigation, as illustrated in Figure 6, the Mitigation Policy (MFU-based) selects an entry from the tracker and correspondingly issues refreshes to victim rows within the blast-radius (radius of 2 by default).

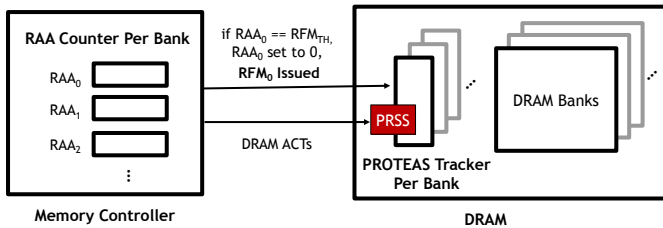


Fig. 8. Design of PROTEAS with RFM

Figure 9 shows how maximum disturbance varies with sampling probability (p), as the number of mitigations per tREFI is increased (RFM_{TH} is lowered). As the number of mitigations per tREFI is increased from 1 to 2, 4, and 8, the maximum disturbance decreases from 2K to 1K, 533, and 290. Additionally, the sampling probability (p) at which the minima for the maximum disturbance increases as the

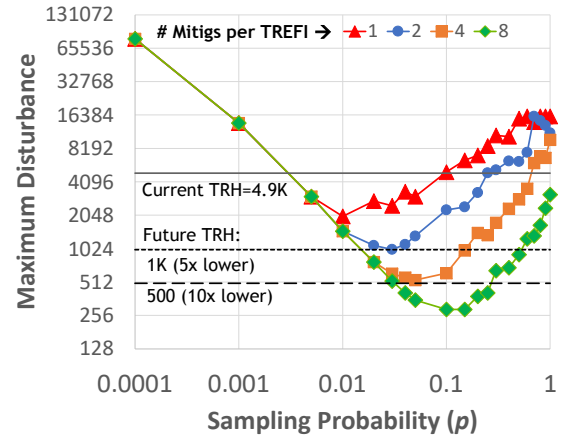


Fig. 9. Sensitivity of PROTEAS to Number of Mitigations per tREFI. With 4 and 8 mitigations per tREFI, the maximum disturbance decreases below future Rowhammer thresholds of 1K and 500.

frequency of mitigation increases, with it being at $p = 0.01$ for the baseline and increasing to 0.03, 0.05, and 0.10 for 2, 4, and 8 mitigations per tREFI. This shows that the sampling rate must be proportional to the mitigation rate. If the sampling rate is lower than the mitigation rate, then the proposed policy would be unable to issue mitigations since the tracker can be empty. Overall, we observe that PROTEAS combined with RFM is suitable for TRH of 1K and 500. Thus, PROTEAS is a simple, practical, and scalable solution as Rowhammer thresholds drop to 500 (by 10x compared to 2020 levels).

VI. RESULTS

A. Evaluation Methodology

We now evaluate PROTEAS for Rowhammer protection efficacy, performance overheads, and storage overheads.

Rowhammer mitigation efficacy. We use our Rowhammer simulator to compare PROTEAS with prior probabilistic mitigations such as DSAC [14] and PARA [21]. We collect the maximum disturbance (number of activations before a refresh) across all 500 attack patterns (Section III). The reported disturbance is averaged over 100 different seeds provided to the random number generator. We assume a 16-entry tracker (like TRR) with a baseline policy of deterministic lookup and insertion (100% sampling). We configure DSAC to target a TRH of 500. We configure the baseline, PARA, and DSAC with similar mitigation frequency as PROTEAS (1, 2, 4, or 8 mitigations per tREFI). For PARA, we achieve a rate of 1, 2, 4, and 8 mitigation per tREFI with mitigation probabilities of 0.6%, 1.2%, 2.5%, and 5%.

TABLE IV
BASELINE SYSTEM CONFIGURATION

| | |
|--|---|
| Out-of-Order Cores | 4 core, 3GHz, 8-wide fetch, 192 entry ROB |
| Last Level Cache (Shared) | 4MB, 16-Way, 64B lines |
| Memory size, bus speed | 16 GB, DDR4, 1.2 GHz (2400 MT/s) |
| t_{RCD} - t_{CL} - t_{RP} - t_{RC} | 14.2-14.2-14.2-45 ns |
| Banks x Ranks x Channels | 16x1x1 |
| Rows | 128K rows, 8KB row buffer |

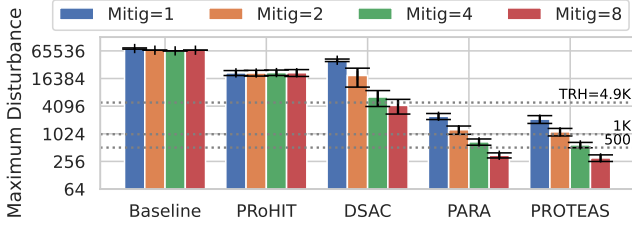


Fig. 10. Maximum Disturbance of PROTEAS and other probabilistic mitigations. We report the means across 100 runs with different seeds and plot error-bars for 95% confidence intervals.

Performance evaluations. We also model PROTEAS in Gem5 [25], a cycle-accurate simulator in the Syscall Emulation (SE) mode. We model a 4-core out-of-order CPU with DDR4 2400MT/s memory, with timings based on Micron datasheets [29]. Table IV shows the configuration. To model the effects of our mitigative action, for 1 mitigation per tREFI configuration, we assume it is issued during tRFC, similar to TRR. For additional mitigations per tREFI, we model the overhead similar to RFM, where the bank is busy for a period of $2 \times \text{blast-radius} \times t_{RC}$. We evaluate our design with 17 SPEC2017 [40] *rate* workloads and 17 mixed workloads. We fast-forward the workloads by 25 billion instructions to reach regions of interest and simulate for 250 million instructions.

B. Comparing Maximum Disturbance of PROTEAS

Figure 10 compares the Max-Disturbance for PROTEAS with prior probabilistic defenses, PRoHIT [39], DSAC [14], and PARA [21]. We compare against a baseline deterministic tracker which samples all lookups and uses LFU for evictions.

At one mitigation per tREFI, PROTEAS with 1% sampling achieves a max-disturbance of 2.1K, which is 35X lower than baseline which has a max-disturbance of 74K. As mitigations per tREFI increase to 2, 4, and 8, the baseline deterministic tracker continues to see a high max-disturbance (65K-67K), whereas PROTEAS has max-disturbance that reduces to 1128, 585, and 305 at sampling rates of 3%, 5%, and 10%, respectively. PROTEAS achieves 60X to 222X decrease over the baseline because the deterministic tracker gets easily thrashed despite the additional mitigations per tREFI, and the resident entries are easily evicted between an insertion and potential mitigation, despite the shorter time window with higher mitigation frequency. However with PROTEAS, the tracker can get thrashed over a much longer period, and hence frequent mitigations limits the time available to an attacker, thus reducing the maximum disturbance linearly.

In comparison, the prior probabilistic tracker PRoHIT [39], achieves a max-disturbance of 21k with one mitigation per tREFI (10x higher than PROTEAS). This is because Blacksmith patterns deterministically evict entries from PRoHIT’s cold table before they are promoted to the hot table, thus avoiding mitigations for hammered rows. Similarly, Samsung’s DSAC [14], achieves a max-disturbance of 41K with one mitigation per tREFI (19x higher than PROTEAS), which decreases to 4K at 8 mitigations per tREFI (14x higher than

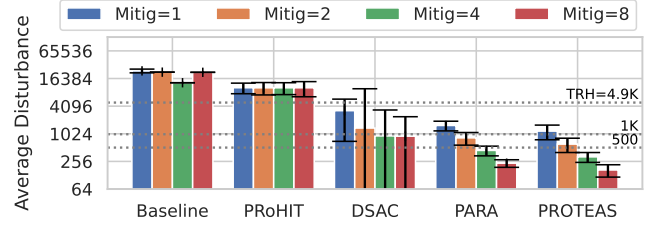


Fig. 11. Average Disturbance of PROTEAS and other probabilistic mitigations. We report the means across 100 runs with different seeds and plot error-bars for 95% confidence intervals.

PROTEAS). This is because DSAC adopts stochastic evictions similar to PMSS in Figure 4, which reduces thrashing to a less extent compared to PRSS in PROTEAS. Moreover, the sampling probability in DSAC varies inversely to the minimum count in the tracker. We observe the p in DSAC dynamically varies between 1 and 0.05 leaving it more vulnerable to thrashing than our PRSS which always operates at the optimal sampling probability. The highest maximum disturbance for DSAC is achieved for non-uniform attack patterns (where decoy rows are accessed immediately after few iterations of target rows similar to Blacksmith [15]) which effectively thrash this tracker and severely degrade its security – these patterns were not evaluated in DSAC.

The maximum disturbance with PROTEAS is generally lower than PARA at similar mitigation costs. PARA with one mitigation issued per tREFI achieves a max-disturbance of 2.4K (15% higher than PROTEAS), and with 8 mitigations per tREFI, this reduces to 350 (14% higher than PROTEAS). This is because, unlike PARA where the mitigations are fully probabilistic, PROTEAS samples probabilistically into the tracker and then intelligently selects entries for mitigation based on frequency. This allows PROTEAS to achieve slightly better resilience at equivalent mitigation costs. Moreover, unlike PARA which is required to be implemented in the memory-controller, that requires knowledge of neighboring rows in DRAM, PROTEAS is an in-DRAM solution that can be managed directly by memory vendors.

Overall, Figure 10 shows that with 1 mitigation per tREFI, PROTEAS reduces max disturbance by 35X versus baseline, by 10X compared to PRoHIT, by 19X compared to DSAC, and by 15% compared to PARA. With 8 mitigations/tREFI, these reductions with PROTEAS are 222X vs baseline, 72x vs PRoHIT, 14X vs DSAC, and 14% vs PARA.

C. Comparing Average Disturbance of PROTEAS

We also compare the average disturbance for PROTEAS to that of PRoHIT [39], DSAC [14], and PARA [21]. Again, we compare these against a baseline deterministic tracker which samples all lookups and uses LFU for evictions. Figure 11 illustrates the average disturbance received across the different schemes. The figure shows that with 1 mitigation per tREFI, PROTEAS reduces average disturbance by 20X versus baseline, by 9X compared to PRoHIT, by 3X compared to DSAC, and by 30% compared to PARA. With 8 mitigations/tREFI,

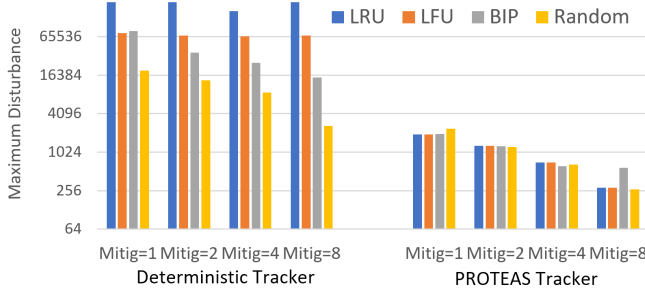


Fig. 12. Sensitivity to Eviction Policy for Baseline and PROTEAS Tracker

these reductions with PROTEAS are 135X vs baseline, 62x vs PRoHIT, 6X vs DSAC, and 40% vs PARA. Thus, PROTEAS reduces both the maximum disturbance and the average disturbance for the hammered rows compared to all prior probabilistic defenses.

D. Sensitivity of PROTEAS to Tracker Eviction Policy

To better understand the impact of replacement algorithms on trackers, we study sensitivity to tracker eviction policy in both deterministic and probabilistic trackers. We study LRU, LFU, BIP [33], and random replacement as the eviction policy but maintain a LFU-based mitigation policy. Figure 12 illustrates the maximum disturbance for the different eviction policies on the baseline deterministic tracker (no sampling based insertion) and a PROTEAS tracker. For a deterministic tracker, LRU has the highest maximum disturbance (226K) followed by LFU (70K), which remains high even if mitigation frequency is increased because they are easily thrashed. BIP and Random replacement minimize thrashing and reduce maximum disturbance to 15K and 2K with 8 mitigations per tREFI. BIP is thrash-resistant in that it retains a fixed portion of the access stream in the tracker. Random replacement on the other hand retains different portions of the access stream allowing them to receive mitigations over time [33].

For PROTEAS, there is limited benefit from intelligent replacement policies since the maximum disturbance across all policies are similar. This is because intelligent replacement policies like LRU/LFU/BIP lose temporal locality information in the request stream and thus behave like random.

E. Performance Overheads of PROTEAS

We implement PROTEAS in Gem5 and measure its performance overheads⁴ at different Rowhammer thresholds, using 1, 2, 4, and 8 mitigations per tREFI which correspond to TRH values of 5K, 2K, 1K, and 500 respectively. As shown in Figure 13, across 17 SPEC-CPU 2017 workloads and 17 mixed workloads (random combinations of SPEC workloads), PROTEAS incurs negligible slowdown at TRH of 5K and 2K. At these operating points, PROTEAS requires RFM_{TH} of

⁴For space reasons we only report performance of PROTEAS. PROTEAS and PARA incur similar slowdown. PROTEAS requires 15% fewer mitigation refreshes to achieve similar Max-Disturbance vs PARA, so it achieves slightly better performance (but both are within 1% of each other). Unlike PROTEAS, PARA is a memory controller solution that cannot be deployed in-DRAM.

166 and 83, which results in 1 mitigation every 166 or 83 activations in the worst case. As activations are only a subset of memory accesses, the resulting overheads are insignificant.

At TRH of 1K and 500, PROTEAS incurs slowdowns of 0.3% and 2.9%. The comparatively higher slowdown is because PROTEAS requires more frequent mitigations at these thresholds (i.e., one mitigation per 42 and 21 activations). With a default blast-radius of 2, each mitigation incurs 4 activations. Thus, PROTEAS at lower thresholds can cause 10% and 20% extra DRAM activations, which leads to higher slowdowns. The highest slowdowns are for workloads like lbm (14.9%) and blender (13.4%) at TRH of 500, which have the highest DRAM activation rates of more than 20 per thousand instructions (PKI). Workloads such as cactuBSSN and those after it in the sorted list in Figure 13 have a DRAM activation rate of less than 1 PKI and thus incur negligible slowdown even at lower thresholds.

F. Sensitivity of Performance to Blast Radius

Figure 14 shows the performance-sensitivity of PROTEAS to the blast-radius of the mitigation. Our default defense refreshes victim rows within a blast radius of 2 from the aggressor (refreshes to 2 rows above and 2 rows below) to ensure protection against Half-Double [22] attacks which flip bits in distance-2 victims. At TRH of 5K and 2K, we observe negligible slowdown even for blast radius up to 4. At TRH of 1K, as blast-radius of mitigation increases from 1 to 2 to 4, PROTEAS slowdown increases from 0.3%, 0.7%, and 1.5%, whereas at TRH of 500, the slowdown increases from 1% to 3% to 6%. As mitigation blast radius increases from 1 to 4, the number of neighbor rows refreshed per mitigation increases from 2 to 8. This increases the slowdown at lower thresholds.

G. Storage Overheads of PROTEAS

PROTEAS requires 16 counters (each requiring 5-bytes: 21 bit counter and 17-bit rowID) per DRAM-bank (Table V). In DDR4, across 16 banks, this incurs a storage overhead of 1.3 KB per rank, and in DDR5 (32 banks) requires 2.6 KB of counters per rank in-DRAM. In the memory controller, RFM requires one 8-bit counter per DRAM bank, incurring 32B of storage within the memory controller for 32 banks per rank in DDR5. These overheads are similar to TRR and RFM.

TABLE V
STORAGE OVERHEADS OF PROTEAS

| Structure | Per-Bank | Per-Rank | |
|--------------------|--------------|----------|-------|
| | | DDR4 | DDR5 |
| Tracker in DRAM | 16 x 40-bits | 1.3KB | 2.6KB |
| RFM Counters in MC | 1 x 8-bits | - | 32B |

PROTEAS also requires extra logic for two PRNGs per DRAM bank (for PRSS and random replacement), requiring a few thousand gates [2] in-DRAM per bank. We believe this is practical especially since recent work by Hynix [20] showcased a DRAM chip fabricated with a PRNG per bank.

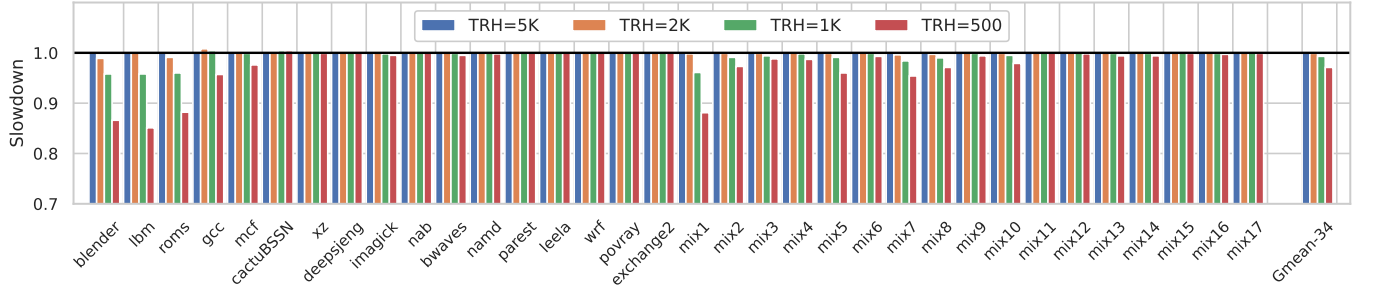


Fig. 13. Performance Overhead of PROTEAS (workloads sorted in descending order of rate of DRAM ACTs). PROTEAS incurs negligible slowdown for TRH of 5K and 2K (1 and 2 mitigations per tREFI), and 0.3% and 2.9% slowdown at TRH of 1K and 500 (4 and 8 mitigations per tREFI respectively).

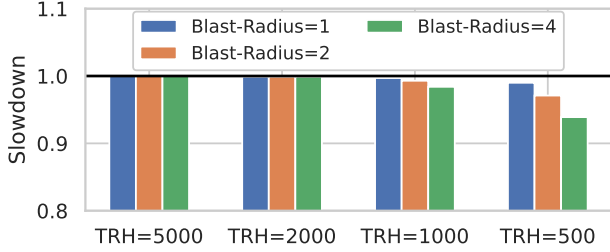


Fig. 14. Performance of PROTEAS as Blast Radius (BR). PROTEAS incurs negligible slowdown at TRH of 5K and 2K. At TRH of 1K, as Blast-Radius increases from 1 to 2 to 4, PROTEAS slowdown increases from 0.3%, 0.7% and 1.5%, while at TRH=500, the slowdown increases from 1% to 3% to 6%.

H. Energy Overheads of PROTEAS

PROTEAS relies on mitigative refreshes to victim rows to prevent Rowhammer, which can lead to an increase in DRAM energy consumption. On one hand, the additional mitigations require extra activation and precharge operations internally within the DRAM, which increase dynamic energy consumption. On the other hand, extra mitigations may also cause the overall execution time to increase, which causes an increase in static DRAM energy (due to normal refreshes and leakage power). Table VI shows the average energy overheads for PROTEAS across 34 workloads reported from Gem5, compared to a baseline without extra mitigations. On average, PROTEAS incurs negligible energy overheads at TRH of 5K, whereas at TRH of 1K, this overhead goes up to 0.5%, and 2.4% at TRH of 500. While the dynamic energy increases by 4.2% at TRH of 500, the static energy, which makes up the majority of the DRAM energy for several workloads, increases by only 2.3%, making the overall increase in energy of 2.4%.

TABLE VI
DRAM ENERGY OVERHEADS OF PROTEAS (BLAST-RADIUS = 2)

| PROTEAS Config | Static Energy | Dynamic Energy | Total Energy |
|----------------|---------------|----------------|--------------|
| TRH = 5K | 0.0% | 0.0% | 0.0% |
| TRH = 1K | 0.6% | 0.5% | 0.5% |
| TRH = 500 | 2.3% | 4.2% | 2.4% |

VII. RELATED WORK

A. Deterministic Aggressor-Row Trackers

Prior Rowhammer detection mechanisms use resource intensive trackers to identify frequently accessed DRAM rows

deterministically. These structures can either be stored on-chip in SRAM or off-chip within the DRAM itself. Table VII compares the storage overheads of these trackers with PROTEAS.

At one end of the spectrum, Graphene [31] uses the Misra-Gries algorithm for tracking, which provides a lower bound on the number of counters required for guaranteed detection of rows that may be activated beyond the Rowhammer threshold. Mithril [19] and PROTRR [27] leverage a similar algorithm and store such a tracker within the logic portion of the DRAM. However, at TRH of 500, and with the doubling of number of banks in DDR5, such structures require a SRAM storage of 640KB per rank, which is too high to be stored in SRAM on-chip [31] or in the logic portion of the DRAM [19].

At the other end of the spectrum, Hydra [32] (and CRA [17]) stores one counter per row in DRAM, with additional filters or caches in on-chip SRAM to limit extra accesses to DRAM. While such trackers require reserving 2.3MB of storage in DRAM, which is not significant, the SRAM-based performance optimizations are access-pattern dependent, which can lead to a worst-case slowdown of up to 70% with pathological workloads. Storing such counters in the DRAM array [3], [20] to avoid these performance issues requires a significant redesign of the DRAM arrays, which may affect DRAM access times, and is less desirable.

Other prior trackers typically incur different storage and performance overheads. TWiCe [24] maintains a table of counters to track activations per row, and prunes the entries that are unlikely to reach the Rowhammer threshold to save space. CAT [4] maintains a dynamic tree of counters which allocates more counters to hot rows, to enable a storage-efficient tracker. The storage overheads of such solutions scale as the number of attacked rows increase. At low thresholds of 500, such solutions incur higher storage overheads than Hydra or CRA, making them impractical.

D-CBF [44] uses a counting bloom-filter to identify potential aggressor rows that cross a certain threshold of activations. Unfortunately, as this is a blocklisting-based tracker, rows once inserted into the filter will continue to be flagged as aggressors until the end of the refresh period (64ms) when the tracker can be reset, making it incur high mitigation costs. Additionally, this incurs a high SRAM storage overhead of 1.5MB per rank.

In contrast, PROTEAS incurs negligible storage costs of less than 3KB SRAM stored within the DRAM logic area

(equivalent to currently deployed TRR) and has low mitigation costs, while providing significantly better security than TRR by enabling effective thrash protection.

TABLE VII
STORAGE OVERHEADS OF TRACKERS PER 16GB DRAM RANK AT TRH OF 500 (ALL STORAGE IS IN SRAM UNLESS SPECIFIED OTHERWISE)

| Name | DDR-4 (16 banks/rank) | DDR-5 (32 banks/rank) |
|----------------|------------------------------|------------------------------|
| Hydra | 26 KB (SRAM) 2.3MB (DRAM) | 52 KB (SRAM) 2.3MB (DRAM) |
| TWiCe | 2.3 MB | 4.6 MB |
| CAT | 1.5 MB | 3.0 MB |
| D-CBF | 768 KB | 1.5 MB |
| Graphene | 340 KB | 640 KB |
| PROTEAS | 1.3 KB (in DRAM) | 2.6 KB (in DRAM) |

B. Alternative Rowhammer Mitigation Mechanisms

PROTEAS uses a mitigative action of refreshing the neighboring victim rows [13], [14], [19], [31], [32]. Alternatively, REGA [28] proposed a mitigation of refreshing additional rows in parallel to DRAM activations, at no slowdown, by modifying the DRAM sense amplifiers. This avoids slowdown but has the cost of significant additional refreshes and increased DRAM energy consumption. PROTEAS can be combined with REGA, to invoke such refreshes selectively only for the neighborhood of aggressor rows, thus significantly lowering the DRAM energy overheads, while enjoying the performance benefits of simultaneous refresh and activation.

Prior works have also proposed alternative mitigation that penalizes the aggressors. For instance, recent schemes like RRS [34] and SRS [43] move aggressor rows within memory by periodically swapping them with another random row in memory. AQUA [36] similarly moves an aggressor row to a quarantine region once it has been activated beyond a certain threshold. As such mitigative actions are invoked from the memory-controller, they cannot be directly applied with PROTEAS, which is an in-DRAM defense. On other hand, in-DRAM mitigations like SHADOW [42] or CROW [11] which similarly relocate aggressor or victim rows in-DRAM within the sub-array at low cost can be combined with PROTEAS.

Alternatively, memory-controller-based solutions like Blockhammer [44] limit the access rate to potential aggressor rows. Such mechanisms incur significant worst-case performance overheads and as such cannot be combined with PROTEAS given that it is implemented within DRAM.

C. Cryptographic Detection of Rowhammer Bit-Flips

Recent works propose storing cryptographic signatures like Message Authentication Codes (MACs) for data in DRAM, and using them to verify that data is free from tampering on DRAM reads. SafeGuard [7] and CSI-RH [16] store MACs for each 64 byte data in DRAM, whereas PT-Guard [35] only stores MACs for OS page-table data. Since these solutions detect a Rowhammer attack when bits flip, uncorrectable multi-bit flips can lead to data loss. On the other hand, because tracker-based solutions like PROTEAS adopt an orthogonal approach of tracking aggressors and issuing mitigations to prevent a Rowhammer bit-flip, such data loss is prevented.

VIII. CONCLUSION

In current DRAM modules, DRAM vendors have deployed Target Row Refresh (TRR) which uses a small in-DRAM tracking structure with tens of counters to detect row hammer attacks. Unfortunately, thrashing-based attack patterns have rendered TRR vulnerable. This paper proposes *PROBABILISTIC TRACKER MANAGEMENT POLICIES* (PROTEAS) to make in-DRAM trackers thrash resistant. PROTEAS employs Probabilistic Request Stream Sampling (PRSS) and random replacement to prevent thrashing and introduce diversity in entries held within the tracker. PROTEAS significantly reduces the maximum disturbance (i.e., number of activations before a mitigation) by 35X compared to a deterministic tracker baseline at current TRH of 4.9K, and by 220X at thresholds of 500 when co-designed with RFM. PROTEAS requires only 2.6KB storage overhead on DDR5 systems while incurring only 0.3% and 3% slowdown for TRH of 1K and 500, respectively.

APPENDIX A

ANALYTICALLY DERIVING OPTIMAL SAMPLING RATES

All entries inserted into the tracker exit the tracker either when they are evicted on a capacity/conflict miss OR when the entry is mitigated. PROTEAS desires to operate at a sampling probability where thrashing is minimized such that entries inserted exit from the tracker predominantly on mitigations. At this point, the tracker insertion rate should be at least equal to the tracker mitigation rates, as inserting at a lower rate than the mitigation rate means that the tracker is not fully utilized. The insertion rate should be no more than the mitigation rate, as that would overflow the tracker and cause thrashing.

For sequence of A activations to a tracker with sampling rate S and a given *MissRate*, the total insertions (I_A) is:

$$I_A = A \times \text{MissRate} \times S \quad (1)$$

The total mitigations (M_A) for A activations assuming mitigation rate M (number of mitigations per activation) is:

$$M_A = A \times M \quad (2)$$

To avoid thrashing, PROTEAS desires I_A equal M_A . By equating (1) and (2), the sampling rate S is:

$$S = M / \text{MissRate} \quad (3)$$

Without loss of generality, assuming a *MissRate* = 0.5; the optimal analytical sampling rate of PROTEAS is $= M \times 2$. Table VIII shows that the empirical sampling rate closely matches the analytical sampling rate.

TABLE VIII
SAMPLING RATES ANALYTICAL VS EMPIRICAL

| Mitigation Rate | Sampling Rate (analytical) | Sampling Rate (empirical) |
|---------------------------|----------------------------|---------------------------|
| 1 Mitigation per 166 ACTs | 1.2 | 1 |
| 2 Mitigation per 166 ACTs | 2.4 | 3 |
| 4 Mitigation per 166 ACTs | 4.8 | 5 |
| 8 Mitigation per 166 ACTs | 9.6 | 10 |

REFERENCES

- [1] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.
- [2] M. Bakiri, "Hardware implementation of a pseudo random number generator based on chaotic iteration," Ph.D. dissertation, Bourgogne Franche-Comté, 2018.
- [3] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.
- [4] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 117–130.
- [5] 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)*. IEEE, 2019, pp. 55–71.
- [6] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript," in *USENIX Security 21*, 2021.
- [7] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. 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.
- [8] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.
- [9] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoecl, 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.
- [10] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 300–321.
- [11] H. Hassan, M. Patel, J. S. Kim, A. G. Yaglikci, N. Vijaykumar, N. M. Ghiasi, S. Ghose, and O. Mutlu, "Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 129–142.
- [12] 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.
- [13] 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.
- [14] 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," *arXiv preprint arXiv:2302.03591*, 2023.
- [15] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Rowhammering in the Frequency Domain," in *43rd IEEE Symposium on Security and Privacy'22 (Oakland)*, 2022, https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.
- [16] J. Juffinger, L. Lamster, A. Kogler, M. Eichseder, M. Lipp, and D. Gruss, "Csi: Rowhammer-cryptographic security and integrity against rowhammer," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 236–252.
- [17] 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.
- [18] 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 *ISCA*. IEEE, 2020, pp. 638–651.
- [19] 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)*. IEEE, 2022, pp. 1156–1169.
- [20] 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.
- [21] 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," *ISCA*, 2014.
- [22] 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," in *USENIX Security*, 2022.
- [23] 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)*. IEEE, 2020, pp. 695–711.
- [24] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: preventing row-hammering by exploiting time window counters," in *ISCA*, 2019.
- [25] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Niekoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and E. F. Zulfian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.
- [26] 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>
- [27] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 735–753.
- [28] M. Marazzi, F. Solt, P. Jattke, K. Takashi, and K. Razavi, "REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023.
- [29] "DDR4 SDRAM Datasheet (MT40A2G4)," Micron Technology Inc., 2015. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf
- [30] J. Misra and D. Gries, "Finding Repeated Elements," *Science of Computer Programming* 2, pp. 143–152, 1982.
- [31] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *MICRO*. IEEE, 2020, pp. 1–13.
- [32] 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*, 2022, pp. 699–710.
- [33] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 381–391. [Online]. Available: <https://doi.org/10.1145/1250662.1250709>
- [34] 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 Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 1056–1069.

- [35] A. Saxena, G. Saileshwar, J. Juffinger, A. Kogler, D. Gruss, and M. Qureshi, “Pt-guard: Integrity-protected page tables to defend against breakthrough rowhammer attacks,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.
- [36] 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)*. IEEE, 2022, pp. 108–123.
- [37] M. Seaborn and T. Dullien, “Exploiting the DRAM rowhammer bug to gain kernel privileges,” *Black Hat*, vol. 15, p. 71, 2015.
- [38] 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.
- [39] M. Son, H. Park, J. Ahn, and S. Yoo, “Making dram stronger against row hammering,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.
- [40] “SPEC CPU2017 Benchmark Suite,” Standard Performance Evaluation Corporation. [Online]. Available: <http://www.spec.org/cpu2017/>
- [41] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic rowhammer attacks on mobile platforms,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16, New York, NY, USA, 2016, p. 1675–1689. [Online]. Available: <https://doi.org/10.1145/2976749.2978406>
- [42] 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.
- [43] 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)*. IEEE, 2023, pp. 374–389.
- [44] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, “Block-hammer: 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.
- [45] J. M. You and J.-S. Yang, “Mrloc: Mitigating row-hammering based on memory locality,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.
- [46] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, “Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 28–41.