2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

SALT: Track-and-Mitigate Subarrays, Not Rows, for Blast-Radius-Free Rowhammer Defense

Moinuddin K. Qureshi
Georgia Institute of Technology
moin@gatech.edu

***Abstract*—Typical in-DRAM Rowhammer mitigation operates by identifying aggressor rows and refreshing a limited number of victim rows on either side of the aggressor row. The number of victim rows is specified by the *Blast Radius*. JEDEC recently introduced state-of-the-art Rowhammer defense, which includes *Per-Row-Activation-Counting (PRAC)* to identify aggressor rows and *Alert-Back-Off (ABO)* to allow the DRAM chip to obtain time to refresh two victim rows on either side of the aggressor row. The implicit assumption in PRAC is that charge loss beyond the two victim rows is negligibly small and does not represent a threat to the security of PRAC. In this paper, we develop *Ripple Attack* that can amplify even a small amount of leakage in distant rows to cause charge loss equivalent to 3x-78x the activations tolerated by PRAC for the given threshold. The goal of our paper is to develop an in-DRAM mitigation that tolerates Rowhammer without relying on a pre-defined Blast Radius.**

We observe that as subarrays are spatially isolated from each other, activity in one subarray does not cause charge leakage in rows of another subarray. To develop Blast-Radius-Free Rowhammer mitigation, we propose *SALT (Subarray-Level Tracking and Mitigation)*. SALT tracks activation counts per subarray, and when the count exceeds a specified value, it triggers ABO to obtain time for refreshing a portion of the subarray. SALT bounds the maximum number of activations to the subarray before all rows are guaranteed to be refreshed, thus providing *Blast-Radius-Free* Rowhammer mitigation. To reduce the slowdown from ABO, SALT-C coordinates the demand refresh operations such that ABO is not required if the activations to the subarray are below what can be handled by the demand refresh, thus reducing ABO by 48x. SALT-C not only provides stronger security guarantees than PRAC due to Blast-Radius-Freedom, but also has 38x lower storage overhead than PRAC, and incurs lower slowdown (0.3% vs 1.7%) than PRAC.

I. Introduction

Rowhammer occurs when a row is frequently activated, leading to charge loss in neighboring rows. The number of activations required to cause a bitflip is called the *Rowhammer Threshold (TRH)*. Over the last decade, as the sizes of the DRAM cells have reduced, the TRH has reduced from 135K [20] to 4.8K [17]. Rowhammer provides an attacker with a powerful tool to flip bits in critical data structures, such as page tables, and cause privilege escalation. Thus, Rowhammer is not only a reliability concern, but a serious security threat [5], [6], [44], [52] [23]. Hardware mitigation of Rowhammer typically relies on a tracking mechanism to identify the *aggressor* rows and then refreshing a specific number of *victim* rows on either side of the aggressor row. As in-DRAM mitigations can transparently address Rowhammer within DRAM chips, we focus on in-DRAM mitigations.

Tracking Challenges: In-DRAM Rowhammer mitigation requires storage for tracking aggressor rows. For example, DDR4 devices contain *Targeted Row Refresh (TRR)* tracker with 1-28 entries [7]. As TRR cannot track all the aggressor rows, an attacker could craft a pattern to make the tracker forget some of the aggressor rows by attacking a large number of rows [5] or using decoy rows [12]. Thus, the system remains vulnerable to Rowhammer attacks even in the presence of TRR, as acknowledged by JEDEC [13], [14]. Optimal in-DRAM trackers, such as Mithril [19] and ProTRR [29], provide sufficient entries to track all aggressor rows for a given threshold; however, they require impractical storage overhead (for example, 4.35KB CAM per bank at TRH of 1K). To overcome the SRAM storage challenges of aggressor row tracking, JEDEC recently introduced *Per-Row-Activation-Counting (PRAC)* [1], which equips each DRAM row with an activation counter that is incremented for each activation. The DRAM timings are increased to accommodate the read-modify-write operation of the counter, thereby increasing the memory access latency. Although PRAC tracking incurs both storage and performance overheads, it represents a significant advance in Rowhammer defense, as it enables tracking of every row, thereby overcoming the shortcomings of TRR.

Mitigation Challenges: Typical in-DRAM solutions perform mitigation by refreshing a given number of victim rows, specified by the *Blast Radius*. The mitigation is typically done transparently during the refresh (REF) operation by borrowing time from the refresh. This type of transparent mitigation suffers from two problems. First, using the refresh time to perform Rowhammer mitigation reduces the time available for demand refreshes. As DRAM reliability degrades, it becomes difficult to sacrifice the available time to perform regular refreshes. Second, if mitigation can only be done periodically (every Nth REF), the attacker can use *Feinting Attack* [29] to cause a significant number of activations on a given row between mitigations. For example, if a PRAC system can mitigate one aggressor row every 4th or 8th REF, then the tolerated TRH (double-sided) would be 2.5K-4.6K. To allow DRAM chips to receive mitigation time as needed, JEDEC introduced *Alert-Back-Off (ABO)*. When the activation count of a row crosses an *Alert-Threshold (ATH)*, PRAC issues an ABO to refresh victim rows, as shown in Figure 1(a). Recent works [3], [37], [48] show that PRAC+ABO can tolerate low thresholds while incurring only negligible mitigations.

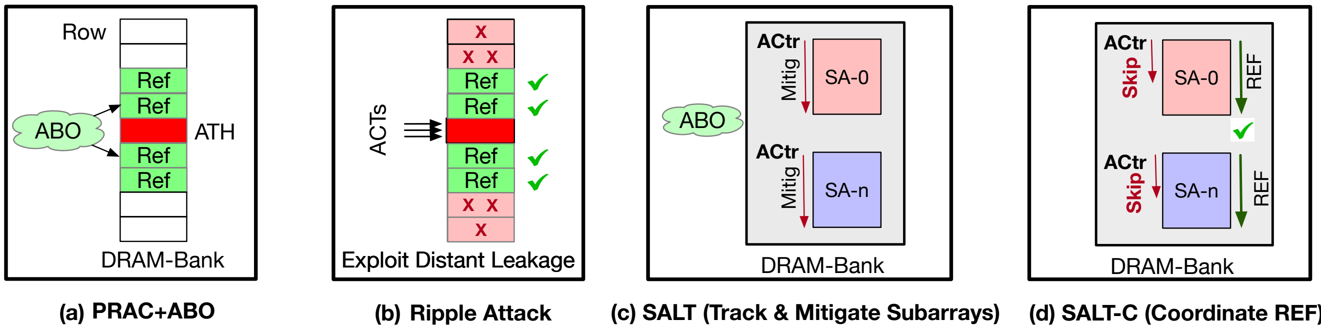

Fig. 1. (a) PRAC uses a per-row counter and obtains mitigation time using ABO to refresh two victim rows on either side. (b) Our Ripple Attack exploits non-zero leakage beyond two rows to cause 3x-78x charge loss on distant rows than permissible with PRAC. (c) SALT does subarray-level tracking and mitigation to provide Blast-Radius-Free Rowhammer mitigation. It provisions an Activation Counter (ACtr) per subarray and refreshes subarray gradually using ABO. (d) SALT-C coordinates demand refreshes such that mitigations are skipped if demand refreshes are sufficient to handle the activity per subarray.

The Vulnerability of Blast-Radius: The implicit assumption for PRAC-based mitigation is that there is a well-specified (and small) Blast-Radius, such that there is no charge loss for any row with a distance greater than the Blast-Radius. In general, charge loss is typically modeled as an exponential decay [2], [50] with respect to the distance (d) from the aggressor row. Thus, although the charge loss decreases for distant rows, it does not vanish. The conventional wisdom is that if the charge loss is small, it can be ignored. For example, if the charge loss on the distant row is only 1% (relative to the nearest victim row), then ignoring it will cause only a negligible impact on the tolerated threshold. We show that an attacker can cause charge loss on distant rows (even though mitigating activations to the nearby victim rows are accurately tracked by PRAC). We develop *Ripple Attacks*, which can cause significantly greater charge loss on distant rows than is permissible under the specified threshold in PRAC. Ripple Attack can cause bit-flips in rows well beyond the rows specified by the Blast-Radius, as shown in Figure 1(b).

The Charge-Loss Model: To analyze the impact of Ripple Attack, we develop a charge-loss model. We show that an attacker can continuously activate an aggressor row, triggering frequent mitigations. PRAC accurately counts activations, including those required to mitigate victim rows. However, if the mitigations on the victim rows are less than ATH, then the victim rows themselves do not trigger any mitigations. Thus, the activations on the aggressor row can continue to cause charge leakage to rows well beyond the Blast-Radius. Our analysis shows that Ripple Attacks can cause damage equivalent to activating an aggressor row for 3x-78x times the specific threshold without incurring any mitigations. Thus, Ripple Attacks can severely degrade the security of PRAC+ABO. The goal of our paper is to develop an in-DRAM mitigation without relying on a pre-defined Blast-Radius.

SALT: We observe that, as subarrays are spatially isolated from each other, activity in one subarray does not leak charge from rows of another subarray [26], [27]. To provide Blast-Radius-Free Rowhammer mitigation, we propose *SALT (Subarray-Level Tracking and Mitigation)*. Unlike PRAC, which tracks activations per row, SALT tracks activations per subarray. Unlike PRAC, which uses ABO to perform victim refresh for a specific aggressor row, SALT uses ABO to gradually refresh a fixed number of rows from a subarray, as shown in Figure 1(c). SALT bounds the total number of activations to the subarray before all rows are refreshed. Thus, SALT provides *Blast-Radius-Freedom*, in that the security property is independent of the distance between the victim and aggressor rows. We perform security analysis to determine the parameters of SALT for the desired TRH. For example, to tolerate a double-sided TRH of 1K, SALT issues an ABO every 26 activations. The slowdown of SALT ranges from 1.5% at TRH of 4K to 12.9% at TRH of 500.

Coordinate ACT-Based Refresh and Time-Based Refresh: To reduce the performance overheads of SALT, we observe that subarrays undergo time-based refresh (to ensure retention). By coordinating the time-based refresh with the ACT-based refresh, we avoid ABO if the time-based refresh is sufficient to handle the refresh of the subarray, as shown in Figure 1(d). So, ABO is needed only if the activity exceeds what can be handled by the demand refresh. This coordination reduces the ABO requirements by 48x. SALT with coordinated refresh *(SALT-C)* incurs an average slowdown of only 0.3% at TRH of 500. The storage overhead of SALT-C is quite small at 0.5KB SRAM per bank.

Contributions: Our paper makes the following contributions:

1) We introduce *Ripple Attack* on PRAC that can cause 3x-78x more damage to distant rows than is permissible under the given Rowhammer threshold with PRAC.
2) We propose *SALT* that performs tracking and mitigation per subarray and provides Blast-Radius-Free Rowhammer mitigation without requiring changes to the DRAM array.
3) We propose *SALT-C*, which coordinates demand refresh with act-based refresh, allowing it to skip ABO when demand refresh is sufficient (reduces ABO by 48x).

SALT-C not only provides stronger security guarantees than PRAC (Blast-Radius-Freedom) but also has 38x lower area overheads and incurs lower slowdown (0.3% vs. 1.7%). We also show that, with minor modifications, SALT-C can also tolerate the newly discovered ColumnDisturb [53] error modality while incurring 0% performance overheads.

II. Background and Motivation

A. Threat Model

Our threat model assumes that an attacker can issue memory requests for arbitrary addresses. The attacker knows the defense algorithm and any relevant parameters. We declare an attack successful when a row (near or distant) incurs sufficient charge loss to cause a bitflip. We assume that activity in one subarray does not affect the data in another subarray due to physical isolation between subarrays [26], [27].

B. DRAM Architecture and Parameters.

DRAM chips are organized as arrays consisting of *rows* and *columns*. To access the data in DRAM, the row is accessed using the *word-line*, and the charge on the DRAM cells is sensed using a sense amplifier, and the data is stored in a *Row-Buffer (RB)*. The DRAM chip is divided into a number of banks (32 for DDR5), with only one row-buffer per bank architecturally visible to the Memory Controller. However, internally, each bank consists of smaller independent units called the *Subarray* [21], [30], each with a dedicated row buffer and sensing circuit.

DRAM has deterministic timing specifications as specified by JEDEC standards (see Table I). To access data from DRAM, the memory controller must first issue an activation (ACT) to open the row. To access data from another conflicting row, the opened row must be precharged (PRE) first. To ensure data retention, DRAM data is refreshed every tREFW. To reduce the latency impact of refresh, memory is divided into 8192 groups, and a REF command, issued every tREFI, refreshes one group. As our bank contains 128K rows, each REF must refresh 16 rows per bank.

TABLE I
DRAM Timings (DDR5-6000AN [15]).

Parameter	Description	Baseline
tRCD	Time for performing ACT	14 ns
tRP	Time to precharge an open row	14 ns
tRAS	Min. time a row must be kept open	32 ns
tRC	Time between two ACTs to bank	46 ns
tREFW	Refresh Period	32 ms
tREFI	Time between two REF Commands	3900 ns
tRFC	Execution Time for REF Command	410 ns

C. The Rowhammer Problem

Rowhammer [20] is a data disturbance error that occurs when an aggressor row is activated frequently, causing bit-flips in neighboring rows. The minimum number of activations to an aggressor row to cause a bit-flip in a victim row is called the *Rowhammer Threshold (TRH)*. TRH is reported either for a single-sided pattern *(TRHS)* or a double-sided pattern *(TRHD)*. TRH has dropped from 139K (TRHS) in 2014 [20] to 4.8K (TRHD) in 2020 [17]. As TRH could drop further as devices scale down, it is essential to develop solutions that can handle not only current thresholds (4.8K), but also future thresholds. Therefore, in this paper, we consider a TRHD range from 4K (akin to current thresholds) to 500 (about 10x lower than current thresholds).

Hardware solutions for mitigating Rowhammer typically rely on a tracking mechanism to identify the aggressor rows and then perform a mitigation by refreshing a given number of victim rows on either side of the aggressor row (controlled by a parameter called *Blast-Radius*). The identification of aggressor rows can be done at the *Memory Controller (MC)* or within the DRAM chip *(in-DRAM)*. The advantage of in-DRAM mitigation is that it can address Rowhammer transparently. Furthermore, DRAM manufacturers can tune their solution to the TRH relevant to their chips. Therefore, in this paper, we focus only on in-DRAM solutions. Two resources are required for in-DRAM mitigation: space (to track aggressor rows) and time (to perform mitigation).

D. In-DRAM Mitigation: Space Challenge

For guaranteed protection, the in-DRAM tracker must identify *all* aggressor rows and mitigate them before they receive TRH activations. Unfortunately, the space for tracking the aggressor rows is limited on DRAM chips. DDR4 designs incorporated a low-cost TRR tracker that could track 1-28 entries per bank. Unfortunately, such designs can be broken by patterns targeting a large number of aggressor rows [5] or decoy rows [12]. Table II shows the number of *CAM (content-addressable memory)* entries required to reliably identify aggressor rows for a given TRHD, in an idealized setting where mitigation can be performed as needed.

TABLE II
Number of Entries for tolerating a given TRHD (Graphene [34] with idealized setting where mitigation is done as needed).

TRHD	CAM Entries/Bank	Bytes-per-Bank
500	2486	8391
1000	1243	4351
2000	622	2253
4000	311	1165

Prior works [19] [29] bound the optimal number of entries needed to tolerate a given TRH under a given rate of mitigation. Unfortunately, such optimal trackers still require several hundred (thousands) of entries per bank to mitigate current (future) thresholds. The CAM overhead to implement such trackers makes them impractical for commercial adoption.

To address the space challenge of tracking, JEDEC introduced *Per-Row-Activation-Counter (PRAC)*, which modifies the DRAM array to provide each row with a counter. The counter is incremented for each activation. The DRAM timings are modified to support the counter update (e.g., tRC is increased from 46ns to 52ns). Although PRAC incurs storage and performance overheads due to the counter, it provides a reliable means of tracking all aggressor rows.

E. In-DRAM Mitigation: Time Challenge

Even with accurate activation-counting for each row, the tolerated threshold of PRAC is still dictated by the mitigation rate, provided mitigation can be performed only under REF. This is bounded by the *Feinting-Attack* [29], which spreads the activations over a given number of rows (R), and when one row is mitigated, it excludes that row and spreads the activations over the remaining rows. The TRHD tolerated by PRAC under a Feinting-Attack can be derived using an analytical model [37]. Let there be T activations per tREFI. Let there be one mitigation every M tREFIs. Then, the acts between mitigation (A) equals $M \cdot T$, and the total rounds of mitigation (R) within tREFW equals $8192/M$. The total activations (F) for the last mitigated row is given by Equation 1.

$$F = A \cdot [ln(R) + 0.577] \tag{1}$$

The TRHD tolerated by PRAC varies from 729 to 4567 as the mitigation rate is varied from 1 aggressor row per tREFI to 1 aggressor row per 8 tREFI. Thus, a scheme that relies solely on REF is challenging to scale to low thresholds (as mitigation is typically performed only once every 4-8 REF [7]).

To overcome the limitations of the Feinting-Attack, JEDEC introduced a new command, *Alert-Back-Off (ABO)*. When the activation counter of a row reaches an *Alert-Threshold (ATH)*, the DRAM chip asserts ABO to obtain time for Rowhammer mitigation. When ABO is asserted, the memory controller performs normal operations for 180ns, issues an *RFM (Refresh Management)* command to DRAM, and stalls for tRFM (350ns) [15] (the specifications allow 1-4 RFMs per ABO, for simplicity, we assume one RFM per ABO, so stall time of 350ns). ABO allows PRAC to overcome the Feinting-Attack. Recent works [3], [37], [48] show that PRAC+ABO can tolerate low thresholds without relying on any mitigation under REF. The conventional wisdom is that PRAC+ABO provides a strong defense against Rowhammer.

F. The Perils of Blast Radius

The time provisioned for ABO is sufficient to refresh two victim rows on either side of the aggressor and perform a counter-update for all five rows, including the aggressor (the counters of the victim rows are incremented, and the aggressor row is reset). The key assumption for PRAC+ABO is that there is no charge loss from activations beyond the Blast-Radius of 2. Unfortunately, charge loss typically follows an exponential decay with distance (d) from an aggressor row, so the charge loss for distant rows remains non-zero [2], [19], [50]. For example, Rowhammer characterization data [17], [20] show a few bit-flips even at a distance of 6 from an aggressor row. As thresholds reduce, the impact of charge loss at distant rows becomes more pronounced. An attacker can exploit such unmitigated leakage to cause failure at distant rows.

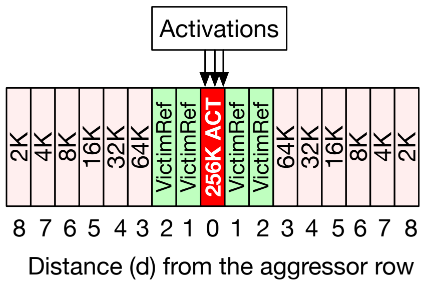

Fig. 2. Charge-Loss at a distance (d) from aggressor row receiving 256K activations (under exponential decay of 2x).

We illustrate this vulnerability with an example. Consider a system where the charge loss from an activation halves [50] with increasing distance (so 1x for the immediate neighbor, 0.5x for the next, 0.25x for the row after, and so on). If we induce 256K activations on an aggressor row, the charge loss incurred at a distance of d would be equivalent to the immediate neighbor of that row receiving 256K/(2^d) activations, as shown in Figure 2. Consider a PRAC design that mitigates when the activation count reaches 1K. Then, rows at d=1 and d=2 are protected as they receive a victim refresh every 1K activations (they undergo a total of 256 mitigations). However, distant rows leak far more than 1K ACT. For example, d = 3 suffers a leak equivalent to 64K unmitigated activations to its immediate neighbor, and even d = 8 leaks 2K units. Thus, this pattern can inflict 64x the damage allowed under PRAC.

G. Experimental Evidence of Bit-flips in Distant Rows

Prior studies show experimental evidence of bit-flips in distant rows. For example, Kim et al. [17] showed that LPDDR4-1y chips had (exponentially reducing number of) bit-flips up to distance six from the aggressor (Figure 3 is reproduced from [17]). Thus, PRAC (with a blast radius of 2) will still incur bitflips for such devices. The absence of bitflips at d>6 does not mean charge leakage at d>6 is zero; instead, it means that it is insufficient to cause bitflips at the current thresholds and may still cause bitflips at lower thresholds.

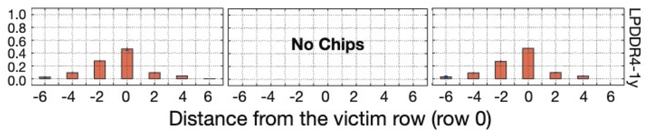

Fig. 3. Characterization from Kim et. al [17] showing bitflips up-to distance 6 from aggressor row. Y axis represents the fraction of bitflips observed and the three graphs are for three manufacturers. (Figure reproduced from [17])

Concurrent to our submission, a MICRO paper introduced *ColumnDisturb* [53], which experimentally demonstrates that for DDR4 chips, activity in one row can cause bit-flips in rows that are up to 446 rows away from the aggressor row (when aggressor and victim are in the same subarray).[1] Thus, solutions such as PRAC+ABO that perform victim refresh for a small number of rows adjacent to the aggressor row will still suffer bit-flips in the distant victim rows.

H. Goal of Our Paper

The security of PRAC+ABO relies on the fundamental assumption that charge leakage from an activation is zero for all rows beyond the blast radius (of 2). This assumption does not hold in practice, as leakage of distant rows is nonzero and decreases only gradually with distance. Ideally, we want a Rowhammer mitigation that does not rely on the knowledge of Blast-Radius. Our goal is to design *Blast-Radius-Free* Rowhammer mitigation (without requiring changes to the DRAM array) and at lower overhead than PRAC. We begin by developing a charge-loss model to analyze the distant leakage.

[1]ColumnDisturb also causes data disturbance in nearby subarrays. We extend our solution SALT with Column-Disturb-Protection in Appendix A.

III. Ripples Don't Die, They Just Fade Away

To exploit leakage in distant rows, we develop *Ripple Attack*. To understand the efficacy of Ripple-Attacks, we develop a charge-loss model that relates leakage to distance (d) from the aggressor row. We then analyze the vulnerability of PRAC under Ripple Attacks. We also differentiate Ripple-Attack from Half-Double [22] and discuss the pitfall of previous approaches to handle the decaying effect beyond Blast-Radius.

A. Relative Charge-Loss for Immediate Neighbor

Consider a DRAM cell that is the target of a single-sided Rowhammer attack. After TRHS activations to the aggressor row, the total charge loss suffered by the cell must cross a *critical* value to cause a bit flip. We need a model to quantify the *Total Charge Loss (TCL)* incurred after K activations. We quantify charge-loss with a *relative* and abstract metric to keep our model simple. Let the *relative charge loss per activation* (C_A) be 1 unit. The total charge loss ($TCL_{d=1}$) after K activations is given by Equation 2.

$$TCL_{d=1} = K \cdot C_A = K \cdot 1 = K \tag{2}$$

As a bit-flip occurs after TRHS activations, the total charge loss is $TRHS$ units, representing the *critical* charge loss for the single-sided pattern. Similarly, for a double-sided attack that sandwiched one victim row and performs TRHD activations on two aggressors, the TCL will be 2·TRHD units.

B. Relative Charge-Loss for Distant Rows

The charge loss for distant rows depends on the distance (d) from the aggressor row. In prior studies [2], [50], this has been modeled as an exponential decay,[2] where for each unit increase in the distance, the charge loss gets reduced by an *Attenuation Factor (E)* [2]. The typical value of E used in previous studies ranges from 2x [50] to 10x [2]). The TCL on a row at a distance of d from an aggressor row subjected to K activations is given by Equation 3.

$$TCL_d = K \cdot E^{1-d} \tag{3}$$

We analyze three values of E: 10x (less leaky), 5x (medium leaky), and 2x (more leaky). Table III captures the maximum unmitigated charge loss at a distance of up to 6 for a hypothetical system that does not perform Rowhammer mitigation.

TABLE III
Leakage rates as a function of distance (d) from the aggressor row.

distance (*d*)	1	2	3	4	5	6
E=2 (high leak)	100%	50%	**25%**	12.5%	6.3%	3.1%
E=5 (med leak)	100%	20%	**4%**	0.8%	0.2%	0.0%
E=10 (low leak)	100%	10%	**1%**	0.1%	0.0%	0.0%

[2]A prior study [19] also captured leakage-decay as a geometric function of distance ($1/d^2$). Analysis with geometric decay results in longer-distance attacks as a geometric series dissipates less quickly than an exponential.

Consider Ripple-Attack, which continuously activates a single row. For our system, an attacker can cause 625K activations per tREFW (32ms). Now consider that PRAC triggers an ABO at ATH of 1K. This pattern would trigger 625 ABO (the count for the two neighbors on each side is 625, which is not enough to trigger an ABO on its own). Table IV shows the Total-Charge-Loss for distant rows (d>2) under Ripple Attack. To simplify our analysis, we do not include the leakage caused by victim refreshes (the impact is relatively small for PRAC).

TABLE IV
Maximum unmitigated damage from Ripple-Attack (* denotes damage is limited due to victim refresh by PRAC+ABO)

d	1	2	3	4	5	6
E=2 (high leak)	1K*	500*	**156K**	78K	39K	19.5K
E=5 (med leak)	1K*	200*	**25K**	5K	1K	187
E=10 (low leak)	1K*	100*	**6250**	625	62	0

A secure PRAC+ABO system configured for TRHD=1K must limit the damage to any row to at most 2000 units. Under Ripple-Attack, the damage to the row at d=3 ranges from 6.25 K to 156 K, approximately 3.25x-78x that permissible under PRAC. Note that a damage of 156K is equivalent to having 156K unmitigated activations on an immediate neighbor.

C. Difference from Half-Double

The main vulnerability of Half-Double [22] stems from the unaccounted activations of victim refreshes to immediate neighbors (d=1). For TRR, these *silent* activations can be enough to cause bit-flips to d=2 neighbors. PRAC counts mitigating activations, so it does not have silent activations. For Ripple-Attack, the d=1 and d=2 rows would have activation counts of 625, not enough to trigger mitigation at ATH=1K. Also, Half-Double requires a few activations on the near-neighbors, whereas Ripple-Attack can be done using one row.

D. Pitfalls of Prior Approach for Handling Blast-Radius

Prior studies [2], [19], [50] try to account for the decaying effect of Blast-Radius by reducing the effective threshold for the aggressor row. For example, if E=10, this modification to PRAC would reduce the threshold for triggering an ABO by a factor of 1.11 (1+0.1+0.01+...). Thus, instead of issuing an ABO at 1K, we would issue an ABO at 900 (1K/1.11). However, such a modification does not protect PRAC against Ripple-Attack, as the d=3 row still suffers 6.25K damage (the same as before). We note that even if the ATH is revised from 1K to 500, one could perform a double-sided Ripple-Attack (312.5K activations at d=0 and d=6) and ensure that the victim row (d=3) suffers the same damage as the single-sided attack.

E. Need for Blast-Radius Freedom

The vulnerability of PRAC to Ripple-Attacks stems from nonzero leakage beyond the Blast-Radius. The lack of characterization data for DDR5 devices (due to on-die ECC) also makes it harder to estimate the decay function for current/future devices. So, we would either need to indiscriminately perform victim-refresh for a large number of rows, or develop a mitigation that does not rely on the Blast-Radius.

IV. SALT

To develop a Blast-Radius-Free Rowhammer mitigation, we propose *SALT (Subarray-Level Tracking and Mitigation)*. SALT mitigates Rowhammer by ensuring that all rows within a subarray get refreshed within a defined number of activations to the subarray, thus providing a much stronger Rowhammer protection, independent of the Blast-Radius. In this section, we discuss the design of SALT, provide security analysis to derive parameters, and conduct a performance evaluation.

A. Design and Overview

Figure 4 shows the overview of SALT for a single subarray. To count activations per subarray, SALT equips the subarray with an *Activation Counter (ACtr)*. When ACtr reaches an *Alert-Threshold (ATH)*, SALT asserts an ABO to refresh a specific number of rows within the subarray. We term the number of rows refreshed within an ABO a *Bundle*.

We assume that a refresh of one row incurs the same latency as tRC; therefore, we assume that an ABO (of 350ns) can refresh **7 rows per bank** (comparatively, a regular REF refreshes 16 rows per bank within 410ns of tRFC).

As SALT refreshes the rows of a subarray sequentially, we must track the bundle to refresh next. In our design, each subarray contains 512 rows and 74 bundles (the last bundle has a single row). SALT assigns a 7-bit *Bundle-Pointer (BPtr)* to each subarray that indicates the following bundle to refresh. On receiving an ABO (triggered by the given bank or a neighboring bank), SALT refreshes one bundle (pointed by BPtr) and increments BPtr.

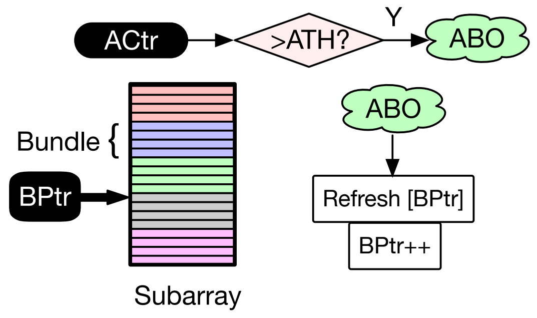

Fig. 4. Overview of SALT (for a single subarray). SALT provisions a subarray with an Activation Counter (ACtr) and a Bundle-Pointer (BPtr). Upon receiving ABO, one bundle (as indicated by BPtr) is refreshed.

B. Gradual Mitigation By Spreading ABO

One way to design SALT is to continuously trigger ABO until all bundles in the subarray have been refreshed. Unfortunately, such a *Bulk* design can incur high performance overheads. Therefore, we opt for a *Gradual* design that distributes the 74 ABO required to refresh the subarray. To achieve this, we introduce a new parameter *Activations-Per-Mitigation (APM)*. On each ABO, the subarray undergoing refresh reduces the ACtr by APM. Therefore, for the gradual design, each subarray would get one ABO per APM activation. To better utilize ABO across all banks, we set APM to be below the *Alert Threshold (ATH)* (as ABO is channel-wide).

C. Which Subarray to Mitigate Per Bank?

When a bank receives an ABO (for example, triggered by another bank), it must choose one subarray to refresh one bundle. A bank may contain a large number of subarrays (e.g., 256), and ideally, we want to select the subarray with the highest ACtr (as it has the most pending mitigation). To avoid having to read 256 counters to decide the subarray with the maximum value ACtr, each bank performs a greedy tracking with a single register per bank (called the *Currently Selected Subarray or CSS*), as shown in Figure 5. CSS contains a valid (Vld) bit, SubarrayID, and ACtr. Whenever an ACT occurs, we read the ACtr of the accessed subarray. If the ACtr of the accessed subarray is greater than the ACtr of the CSS, we overwrite the CSS with the ID and ACtr of the accessed subarray. Thus, CSS always contains the highest ACtr subarray accessed since the last mitigation. On ABO, we mitigate the subarray pointed by CSS (if valid) and decrement ACtr of CSS by APM. If ACtr is at or below zero, the CSS is invalidated.[3] This greedy selection requires only two extra bytes per bank.

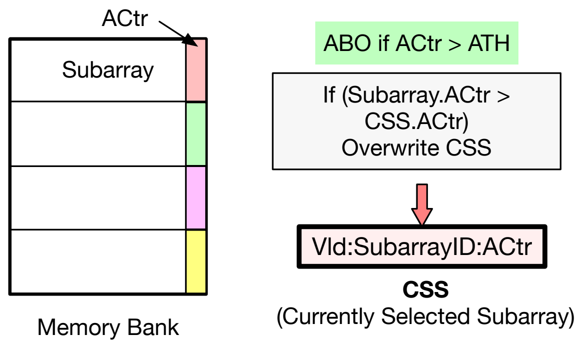
ACtr
Subarray
Memory Bank
ABO if ACtr > ATH
If (Subarray.ACtr > CSS.ACtr) Overwrite CSS
Vld:SubarrayID:ACtr
CSS
(Currently Selected Subarray)

Fig. 5. Tracking subarray with highest ACtr (since last mitigation) per bank

D. Security Model and Determining Parameters of SALT

SALT requires two parameters: APM and ATH. We perform security analysis to determine the values of these two parameters. As SALT is Blast-Radius-Free, to reason about security, we determine the maximum number of activations *(MaxACT)* to the subarray before all rows are refreshed. MaxACT consists of three components: (1) ATH to trigger the first ABO, (2) an additional 73 ABO to refresh all the bundles in the subarray, and (3) Additional activations under Feinting Attack over multiple subarrays. We analyze all three components.

Single-Subarray Attack: We first consider a pattern that causes activations only on a single subarray. Assuming ACtr is zero at the start, the subarray will induce the first ABO after ATH+1 activations. If the subarray has B bundles, then this pattern will cause (B-1) more ABO, once every APM activation. At this point, all rows in the subarray would be refreshed. So, the maximum activations incurred by the subarray ($MaxACT_s$) is given by Equation 4.

$$MaxACT_s = ATH + 1 + (B-1) \cdot APM \tag{4}$$

[3]Optionally, we can overwrite invalid CSS with the ID of the subarray with the highest count. Such a design incurs similar slowdown for typical workloads, but can help with skewed patterns that access only a few banks.

Multi-Subarray Attack: The attacker can increase MaxACT by attacking multiple subarrays. For example, consider that the attacker keeps all 256 subarrays in the bank at $MaxACT_s$ minus one activations. The subsequent activation of any subarray would trigger ABO. The attacker would use the four activations under ABO (three during the normal period of ABO and one after ABO ends) to cause a Feinting Attack over the 256 subarrays. We determine the maximum activations in the last subarray using Equation 1 (A=4 and R=256), yielding 24 additional activations. So, the MaxACT for SALT is given by Equation 5.

$$MaxACT = ATH + (B-1) \cdot APM + 25 \qquad (5)$$

For a double-sided Rowhammer pattern, TRHD=MaxACT/2 (acts spread over two rows). Table V shows the parameters of SALT for different TRHD values. We use a default ATH of twice that of APM. At a TRHD of 1K, SALT requires an APM of 26, which incurs an ABO every 26 activations on the bank.

TABLE V
PARAMETERS OF SALT

MaxACT	Target TRHD	APM	ATH
1000	500	13	26
2000	1000	26	52
4000	2000	53	106
8000	4000	106	212

E. Performance Methodology

For performance evaluation, we use memsim [37], a cycle-level multi-core simulator with a detailed memory model. Table VI shows our configuration. We used a *Minimalist Open-Page Mapping* [16] that places four consecutive lines of page in the same row. We assume rows are striped across subarrays (consecutive rows go to consecutive subarrays). We use a policy that closes the page if there are no pending requests to the opened row, as such a policy outperformed open-page.

TABLE VI
BASELINE SYSTEM CONFIGURATION

Out-of-Order Cores	8 core, 4GHz, 4-wide, 256 entry ROB
Last Level Cache (Shared)	16MB, 16-Way, 64B lines
Memory specifications	32 GB, DDR5
Banks x Sub-channel x Rank	32×2×1
Rows	128K rows per bank, 4KB rows
Subarrays	256 per bank (512 rows per subarray)
Memory-Mapping Policy	Minimalist Open-Page Mapping [16]
Memory Page-Closure Policy	Closed Page (if no pending requests)

We use 13 benchmarks from SPEC-2017 with at least one L3-Miss per 1K instructions (L3-MPKI), six from GAP [39], four from STREAM [31], and two data-analytics benchmarks (KMeans [25] and MassTree [28]). We run the workloads in 8-core rate-mode, until each core completes 1 billion instructions (representative slice). We measure performance using weighted speedup. Table VII shows workload characteristics, including L3-MPKI and ACT-per-tREFI (per bank). With tFAW constraints (10.6ns for 4 ACTs), on average, banks can have at most 41 ACTs per tREFI, so our workloads are memory-intensive (reaching 67% of the theoretical maximum).

TABLE VII
WORKLOAD CHARACTERISTICS

Suite	Benchmark	L3-MPKI	ACT-per-tREFI (per Bank)
SPEC2K17	bwaves	42.7	17.7
	fotonik3d	28.3	26.0
	lbm	26.7	27.2
	parest	23.2	15.6
	mcf	23.1	18.1
	roms	11.6	16.6
	omnetpp	9.3	23.4
	xz	5.1	23.6
	cam4	5.0	13.9
	cactuBSSN	3.5	16.3
	wrf	1.2	4.6
	xalancbmk	1.1	4.7
	blender	1.0	5.2
GAP	ConnComp	86.2	27.6
	PageRank	46.4	21.5
	TriCount	52.2	11.2
	BFS	37.8	19.0
	BC	20.7	14.2
	SSSPPath	10.3	12.5
STREAM	add	15.6	14.2
	triad	13.4	14.0
	copy	12.5	13.0
	scale	10.4	12.7
ANALYTICS	kmeans	8.9	18.6
	masstree	4.8	15.8

F. Impact on Performance and ALERT

We configure SALT to tolerate TRHD of 4000 to 500. Figure 6 (a) shows the slowdown from SALT for the four thresholds. The average slowdown of SALT increases from 1.5% (at TRHD of 4K) to 12.9% (at TRHD of 500). Thus, SALT can provide strong security guarantees (Blast-Radius-Freedom) while incurring only a slight slowdown at current thresholds (4K). Omnetpp incurs a maximum slowdown of 29% at TRHD of 500. Thus, even for our worst-case workload, the evaluated slowdown is only about half of the maximum slowdown estimated under performance attacks. In the next section, we significantly reduce the slowdown of SALT.

The slowdown of SALT is due to ALERTs. Figure 6 (b) shows the ALERTs-per-tREFI from SALT as the TRHD varies from 500 to 4K. At the current TRHD of 4K, the ALERT rate is low, approximately once every 7 tREFI, so the slowdown is small. However, at a TRHD of 500, the ALERT rate increases to 1.2 per tREFI, resulting in higher slowdowns.

G. Storage Overhead

SALT requires an ACtr and BPtr with each subarray. Table VIII shows the storage overhead of SALT (per subarray and per bank) as the target TRHD varies from 500 to 4K. The SRAM overhead ranges from 400-512 bytes per bank, which is much smaller than optimal trackers (1.2KB-8KB, see Table II) and without incurring the complexity of CAM lookups.

TABLE VIII
STORAGE OVERHEAD FOR DIFFERENT TRHD VALUES

TRHD	ACtr-Bits	BPtr-Bits	Bits/Subarray	Bytes/Bank
500	6	7	13	416
1K	7	7	14	448
2K	8	7	15	480
4K	9	7	16	512

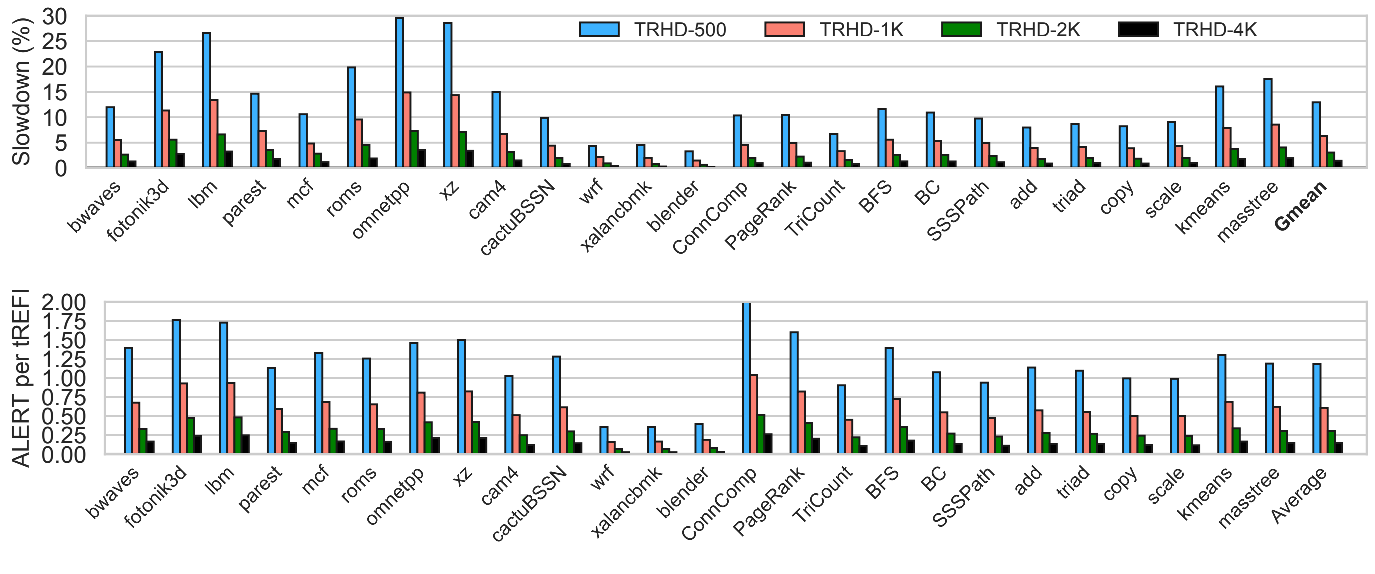

Fig. 6. (a) Slowdown from SALT when configured for TRHD=500 to 4K. The average (Geometric mean) slowdown ranges from 12.9% to 6.3% to 3.0% to 1.5%. (b) ALERTs-per-tREFI (per sub-channel). On average, ALERT varies from 1.2 per tREFI to 0.60 to 0.30 to 0.15 (about one per 7 tREFI).

V. SALT-C: Coordinating Refresh

To reduce the overhead of SALT, we exploit the observation that subarrays periodically undergo a time-based refresh (all rows are refreshed within 32ms) to ensure data retention. We leverage the time-based refresh (performed by REF) to avoid ABO. For example, if the activity on a subarray can be handled by time-based refresh, then we can skip the activity-based refresh (via ABO), thus **eliminating** the time overheads of Rowhammer mitigation. We call such a coordinated design of SALT with time-based refresh as *SALT-C*.

A. Synergistic Refresh Schedule

DRAM chips perform refreshes gradually, with the refresh task spread over 8192 refresh intervals (denoted by tREFI). Upon receiving a REF signal, each DRAM bank must refresh 1/8192 of the rows in the bank. As our bank has 128K rows, each REF must refresh 16 rows per bank. JEDEC specifications do not dictate the sequence in which a refresh is performed within a bank, provided that all rows are refreshed within 32 ms. For example, the 16 rows to be refreshed can come from 2-16 different subarrays. At the next REF, the bank may either continue refreshing other rows from the same set of subarrays as in the last REF or proceed in a round-robin fashion to other subarrays.

To be synergistic with SALT, a subarray must not receive all the REF in short succession (e.g. 1 millisecond) and then no REF for a long time (e.g. 31 milliseconds). Therefore, we pick a refresh schedule that does round-robin across subarrays at each REF. The bank in our design contains 256 subarrays. At each REF, we refresh **one row each from 16 consecutive subarrays** (this also balances refresh power over subarrays). At the next REF, we refresh one row each from the next 16 subarrays, and so on. Thus, each subarray would refresh one row every 16 tREFI.

Figure 7 shows the refresh schedule that we use for SALT-C. The first refresh (Ref-0) refreshes one row each in Subarrays SA-0 to SA-15. The second refresh (Ref-1) refreshes one row each in Subarrays SA-16 to SA-31, and so on. In 16 refreshes, one row is refreshed in all 256 subarrays (SA-0 to SA-255). The process repeats for the next set of refreshes.

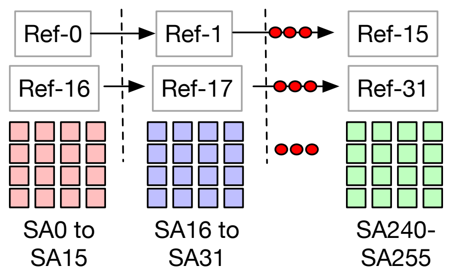

Fig. 7. Refresh Schedule for SALT-C: At each REF, one row is refreshed from 16 consecutive subarrays. Refresh goes across all the subarrays in a round-robin manner. Thus, all subarrays refresh one row every 16 tREFI.

B. Overview of SALT-C

We observe that, for a subarray, the time-based-REF performs a different amount of mitigation work (1 row refreshed in a subarray) compared to the activity-based-refresh issued by an ABO (7 rows refreshed in a subarray). SALT-C modifies the design to handle both types of mitigation in a unified manner.

Figure 9 shows an overview of SALT-C. Instead of BPtr (Bundle Pointer), SALT-C contains *RPtr (Row Pointer)*, which points to a row in a subarray. On receiving an ABO, SALT-C refreshes seven rows of a subarray beginning with the row pointed by RPtr. SALT-C reduces ACtr by APM and increments RPtr by seven (wrapping around if needed). On receiving a REF, SALT-C refreshes the row pointed by RPtr, increments RPtr, and reduces ACtr by an amount called *APR*

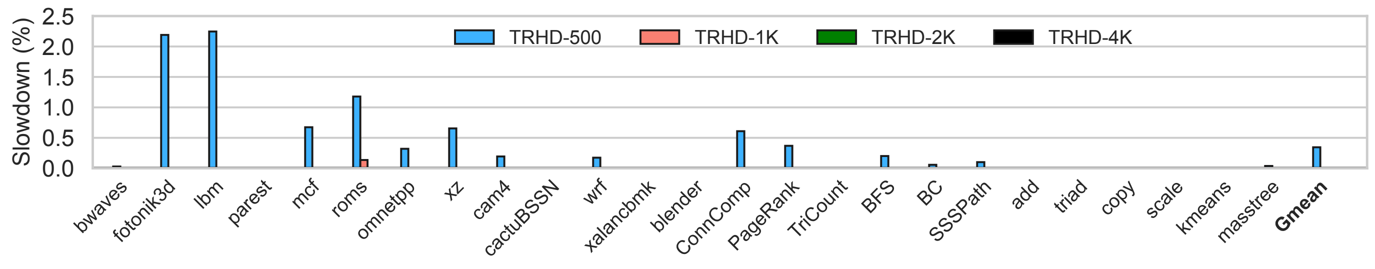

Fig. 8. Slowdown from SALT-C for TRHD=500 to 4K. The average (Geometric mean) slowdown at TRHD=500 is 0.3%, and 0% for TRHD of $\geq$ 1K.

(Activations Per Refresh). As REF does one-seventh of the refreshes compared to ABO, we set APR as APM/7 to ensure a proportional decrease in ACtr (every 7 REF to subarray refreshes 7 rows and reduces ACtr by APM).[4]

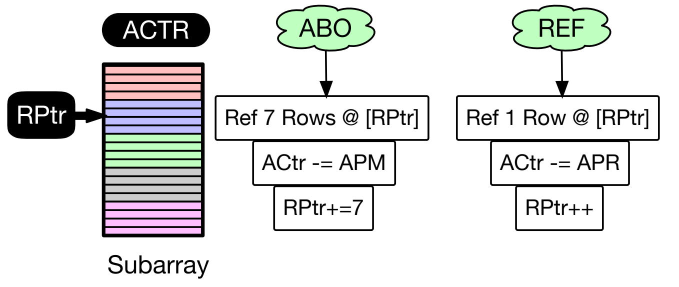

Fig. 9. Overview of SALT-C (one subarray). On ABO, SALT-C refreshes 7 rows, increments RPtr by 7, and reduces ACtr by APM. On REF, SALT-C refreshes 1 row, increments RPtr by 1, and reduces ACtr by APR.

We observe that, since each subarray can refresh 7 rows every 112 tREFI, SALT-C can handle an APM number of activations within this time period without triggering an ABO. For example, consider an application that performs 16 activations per tREFI per bank (average for our workloads). Then, over 112 tREFI, the workload would perform 16*112 = 1792 activations per bank, or, equivalently, 7 activations per subarray (much less than our APM of 13). Thus, we expect SALT-C to skip most ABOs, as our designs use an APM of 13 or higher. However, activations are not uniformly distributed over time or across subarrays, so some subarrays may still receive more activations than REF can handle, and we expect SALT-C to trigger an ABO occasionally.

C. Impact on Performance and ALERT

Figure 8 shows the slowdown of SALT-C for TRHD of 500 to 4K (APM set similar to SALT and varies from 13 to 106). On average, at TRHD=500, SALT-C incurs 0.25% slowdown. At TRHD of 1K and higher, SALT-C experiences 0% slowdown. Table IX compares the average slowdown and ALERT-per-tREFI for SALT and SALT-C. At TRHD=500, SALT-C reduces the ALERTs-per-tREFI from 1.2 to 0.025 (48x reduction). At TRHD of 1K and higher, SALT-C eliminates all ABOs, hence SALT-C has 0% slowdown.

TABLE IX
SLOWDOWN AND ALERT RATE VS TRHD

TRHD	Slowdown		ALERT-per-tREFI	
	SALT	SALT-C	SALT	SALT-C
500	12.8%	0.3%	1.2	0.025 (48x lower)
1000	6.3%	0%	0.6	0 (eliminated)
2000	3.0%	0%	0.3	0 (eliminated)
4000	1.5%	0%	0.15	0 (eliminated)

D. Impact on Power Overheads

SALT-C incurs two types of power overhead: (a) the extra refreshes under ABO and (b) the power consumed in the SRAM structures. To compute DRAM power, we use the DRAM-power model provided by Micron [33]. We configure it with parameters based on DDR5. Since ABO refreshes 7 rows rather than 16 per REF, we assume that ABO consumes half the power of each REF. Figure 10 shows the DRAM power for baseline, SALT, and SALT-C for TRHD=500. We assume the baseline performs no mitigations.

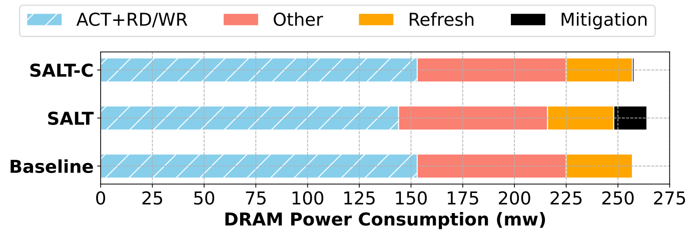

Fig. 10. DRAM power (per chip) for baseline, SALT, and SALT-C at TRHD=500. The baseline consumes 257mW. SALT incurs additional activations under ABO, which consume 16 mW. SALT-C reduces it to 0.3mW.

Our baseline consumes 257mW per chip. SALT incurs a slowdown, so the power consumed (without mitigations) reduces to 248 mW; however, it also incurs an additional 16 mW for mitigation, increasing the total power to 264 mW (2.7% increase). With SALT-C, the mitigations are reduced by a factor of 48, so the mitigation power is reduced to 0.3 mW, and the power consumed per chip is 257.3 mW (0.1% increase). The SRAM structures of SALT-C incur approximately 0.7 milliwatts per chip, accounting for 0.25% of the DRAM-chip power consumption. Thus, the power consumption of SALT-C is 0.35% of the baseline DRAM-chip power consumption.

[4]Our design avoids the complexity of fractional counting by doing appropriate integer decrements. For example, for APM of 13, we reduce ACtr by 1 on every seventh refresh and by 2 otherwise, thereby achieving a steady-state APR of 13/7. This is implemented using a 3-bit counter (BCtr) per bank, which is incremented every 16 REF operations for that bank. BCtr counts from 0 to 6 (incrementing BCtr from 6 takes it to 0). SALT-C uses APR of 1 if BCtr equals 0 and APR of 2 otherwise.

E. Impact on Storage Overheads

As SALT-C requires a pointer to a row rather than a bundle, it has slightly higher storage overhead than SALT. At TRHD=500, we would need ACtr of 6-bits and RPtr of 9-bits, for a total of 15-bits per subarray, or **480 bytes per bank**, much lower than prior counter-based trackers (8KB CAM).

F. Estimating Worst-Case Slowdowns

Figure 6 shows that SALT causes a maximum slowdown of up to 29% (for Omnetpp at THRD=500). While this is the worst-case slowdown observed experimentally for our workloads, the slowdown experienced by other workloads outside our evaluations could be even higher. In this section, we seek to understand the maximum slowdown that SALT can incur across all workloads (even outside our evaluations).

To analyze the worst-case slowdown of SALT, we make two observations. First, the slowdown of SALT occurs due to the stalls from ABO, so the maximum slowdown from SALT would occur for an application that performs activations at the maximum possible rate, thus triggering maximum ABOs. Second, for such a worst-case application, the loss of performance can be estimated by the reduction in the rate of activations (due to stall from ABO), as the key performance metric is the number of activations performed per unit time.

In the steady state, SALT incurs an ABO every APM activation, and an ABO stalls the bank for 350ns. Thus, the proportion of time the memory is stalled due to ABO would be 350ns/(46ns*APM), and this fraction represents the relative increase in the time required for our worst-case application to perform a given number of activations. Table X shows the worst-case slowdown for SALT as TRHD varies from 500 to 4K. The 7%-59% slowdown for SALT under stressful patterns is relatively small compared to other memory performance attacks, such as row-buffer conflicts (2x-3x slowdown), SALT does not introduce a new or worse way of inducing denial-of-service attacks than what is already possible in the baseline system. We note that, because execution of typical benign workloads does not cause continuous activations, they incur much less slowdown than the theoretical maximum.

TABLE X
WORST-CASE SLOWDOWNS WITH SALT

Target TRHD	APM	Up-Time (T_u)	Stall-Time (T_s)	Theoretical Max. Slowdown
500	13	598 ns	350 ns	59%
1K	26	1196 ns	350 ns	29%
2K	53	2438 ns	350 ns	14%
4K	106	4876 ns	350 ns	7%

VI. ALTERNATIVE DESIGNS FOR SALT

We observe that SALT-C requires virtually zero refresh overhead for performing mitigations. In this section, we leverage the low overhead of SALT-C to reduce per-bank storage overhead. We also develop a transparent design for SALT/SALT-C that operates without ABO and is particularly well-suited for near-term implementations.

A. Ganged SALT-C: Reducing Storage Overhead

As SALT-C avoids performance overheads at TRHD of 1K and higher, we explore a SALT-C design that can help reduce the storage overheads. Similar to SALT, the default design for SALT-C provisions an ACtr and RPtr for each subarray. To reduce storage overhead, we increase the granularity of tracking and mitigation from a single subarray to multiple subarrays, and we refer to this unit as a *gang*. Figure 11 shows an overview of SALT-C with a gang containing four subarrays. The RPtr size is increased by 2 bits to track 4x as many rows in the gang. Since there are 64 gangs (from 256 subarrays) and a REF refreshes one row from 16 gangs, each gang receives a row refresh every 4th REF.

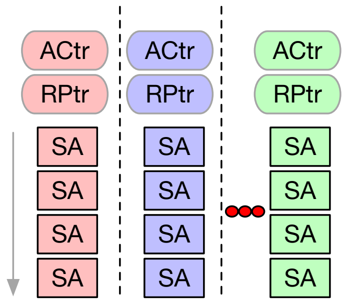

Fig. 11. Ganged SALT-C reduces storage overheads of ACtr and RPtr by tracking gangs containing multiple subarrays (shown four subarrays per gang).

For SALT-C with gangs, we continue to use the same APM/ATH parameters as TRHD=500; however, as this design tracks and mitigates a larger area, the tolerated TRHD increases in proportion to the number of subarrays in the gang. Table XI shows the gang size, APM/ATH parameters, and storage overheads per bank as TRHD varies from 500 to 4K.

TABLE XI
STORAGE AND PERFORMANCE OF GANG SALT-C

Target TRHD	Gang Size	APM/ ATH	Storage (Per Bank)	Average Slowdown
500	1 subarray	13/26	480B	0.3%
1000	2 subarray	13/26	256B	0.3%
2000	4 subarray	13/26	136B	0.3%
4000	8 subarray	13/26	**72B**	0.3%

SALT-C can tolerate current thresholds (4K) at a storage overhead of only 72 bytes of SRAM per bank, which is lower than even some of the TRR designs (e.g., a 28-entry TRR requires approximately 90 bytes of SRAM per bank). Thus, SALT-C is a practical design that can be incorporated in current systems to provide Rowhammer protection with guarantees stronger than even PRAC (Blast-Radius Freedom).

SALT-C with gangs continues to incur negligible performance overhead. The average slowdown across different gang sizes remains 0.3%. This occurs because the ACtr decrements with time-based REF, which is sufficient to handle the activations per gang. Even though the activity per gang increases in proportion to the number of subarrays (e.g., 4x more ACtr increments with a gang containing four subarrays per gang), the REF-based decrement also occurs more frequently (e.g., 4x more frequent ACtr decrement with four subarrays per gang).

B. *Designing SALT without ABO*

Our default design of SALT and SALT-C assumes that the mitigation time (the refresh of a bundle in one subarray) is obtained via the ABO command. ABO is a new command, which was recently released as part of the PRAC+ABO specifications [15]. As ABO is optional, in the near term, likely, the SOC vendors may not implement the ABO support on their memory controllers (in fact, even though RFM was introduced five years ago, a recent study [11] found that Intel and AMD still do not support RFM, even if required by the DRAM chips). Ideally, in the near term, DRAM vendors would like an in-DRAM mitigation that operates transparently, without relying on any ABO/RFM support from SOC vendors. In this section, we enable such a transparent design of SALT.

The transparent design performs mitigations under REF. During REF, in addition to performing the time-based refresh (16 rows, one in each of the 16 subarrays), this design can also refresh 7 additional rows from one of the subarrays (similar to refreshes under ABO). As mitigation is performed only under REF, on-demand mitigation is not viable (no ATH needed).

Transparent SALT: The transparent design of SALT performs the subarray-level mitigation under REF only if the subarray has an ACtr counter more than APM. Otherwise, mitigation is skipped to reduce the power overhead associated with mitigation. Transparent SALT can still incur significant refresh overheads for mitigation, as subarray-level mitigation is invoked under some REF operations.

Transparent SALT-C: For transparent design SALT-C, we reduce the ACtr of subarrays that undergo refresh due to the time-based REF (same as our default design) and employ the subarray level mitigation (7 rows refreshed under REF) only if the ACtr exceeds APM. As the time-based REF reduces the ACtr of subarrays, the need for issuing subarray-level mitigation (7 rows refreshed under REF) is avoided. Thus, SALT-C eliminates almost all refresh overhead, while still ensuring that all rows in the subarray are refreshed within the specified number of activations.

Parameters and Results: As 75 activations are possible (per bank) between REF, the minimum APM for such a transparent design is 75, and it can handle TRHD of 3K or higher (sufficient for near-term designs). Table XII shows the APM for SALT/SALT-C for TRHD of 3K/4K/5K, and the associated refresh overheads. SALT needs 1.52-0.9 rows per REF, and given that time-based refresh does 16 row refreshes, the increase in refresh power is 9.5% to 5.7%. SALT-C eliminates the refresh overheads. The slowdown of the transparent design (both SALT/SALT-C) is zero, as it performs mitigations under REF, thereby avoiding the time overheads.

TABLE XII
APM and Refresh Overhead of Transparent SALT/SALT-C

Target TRHD	APM	SALT (Row Refs. per tREFI)	SALT-C (Row Refs. per tREFI)
3000	75	1.52	0
4000	100	1.13	0
5000	125	0.91	0

VII. Related Work

A. *Silver Bullet: Region-Based Counting and Refresh*

The work most closely related to ours is *Silver Bullet* [4], [51]. Silver Bullet divides the memory bank into regions and performs activity-based refresh of the entire region. As regions are formed in a subarray-agnostic manner, to ensure security, an additional BR rows (where BR is the defined Blast-Radius) are also refreshed in the two adjacent regions. Figure 12(a) shows an overview of Silver Bullet containing 3 regions (R1-R3). To mitigate Region R2, rows C-G must be refreshed before Region R2 reaches the threshold number of activations.

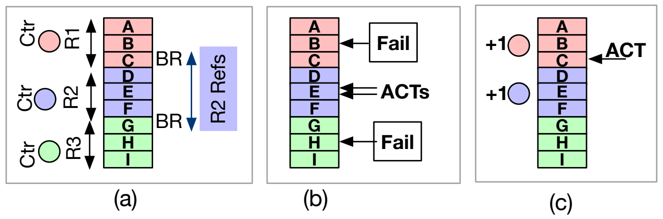

Fig. 12. (a) Overview of Silver Bullet (b) Ripple-Attack on Silver Bullet (c) Silver Bullet can double count ACTs, so it needs higher mitigation.

Silver Bullet suffers from three shortcomings.

1. Blast-Radius Dependent Security (Less Secure): As Silver Bullet relies on a defined Blast-Radius, it does not provide Blast-Radius-Freedom and is vulnerable to Ripple-Attacks. For example, Figure 12(b) shows Ripple-Attack on Row-E with 625K ACTs, which causes Row-C and Row-H to have leakage equivalent to 156K-6250 ACTs.

2. Need 2x Higher Mitigation Rate: To ensure security, an activation at the border row (e.g. Row-C in Figure 12(c)) must increment the counters of both adjacent regions (R1 and R2). Silver Bullet must be designed to tolerate a pattern that doubles the increments with each activation. Thus, Silver Bullet requires 2x as many mitigations as SALT/SALT-C.

3. No Refresh Coordination: Silver Bullet does not leverage time-based-REF to avoid the activity-based-REF, so it incurs significant overheads at low TRHD. SALT-C leverages time-based refresh to reduce mitigation overhead by 48x.

Figure 13 shows the slowdown of Silver Bullet, SALT, and SALT-C for THRD of 500 to 4K. All designs use a region size of 512 rows. Silver Bullet suffers 30% to 3% slowdown, much higher than SALT. The refresh coordination in SALT-C enables negligible/zero overhead. Thus, our two key insights of subarray-granularity regions and refresh coordination achieve both stronger security and much lower overheads.

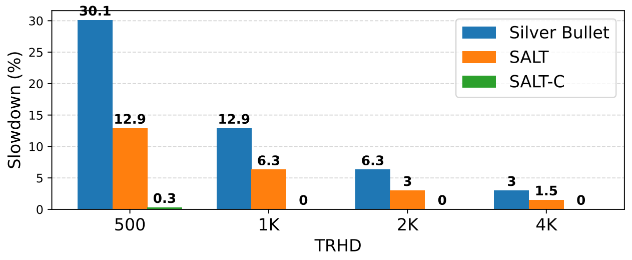

Fig. 13. Slowdown of Silver Bullet, SALT, and SALT-C.

B. PRAC+ABO: State-of-the-Art

JEDEC recently introduced specifications for *Per-Row-Activation-Counter (PRAC)* and *Alert-Back-Off (ABO)*. Recent works, such as Chronus [3], MOAT [37] and QPRAC [48], use PRAC to securely tolerate Rowhammer for the specified Blast Radius. However, PRAC suffers from three shortcomings. First, it performs a Blast-Radius-based mitigation and is vulnerable to Ripple-Attacks (3.5x-78x more charge loss than permissible for the TRH). Second, performance overhead, as the increase in memory timings (e.g. tRC from 46ns to 52ns) causes slowdown due to higher latency for memory requests. Third, it incurs significant storage overhead due to the per-row counter. SALT-C improves PRAC in all three respects.

Security: SALT-C has stronger security than PRAC as it provides *Blast-Radius-Free* Rowhammer mitigation. Note that PRAC+ABO cannot tolerate ColumnDisturb [53], whereas SALT-C (with minor modifications) can tolerate ColumnDisturb with 0% performance overheads (see Appendix-A).

Area Overheads: SALT-C requires a much lower area overhead than PRAC. PRAC requires one counter per row (512 counters per subarray, so 768 bytes DRAM per subarray). SALT-C requires only 2 bytes of SRAM per subarray. Typical DRAM density is about 10x higher than SRAM [9], so SALT-C incurs **38x** lower area overhead than PRAC.

Performance: SALT-C incurs a lower slowdown than PRAC. The slowdown of PRAC is highly sensitivity to the page policy [18] (open-page versus closed-page). Figure 14 shows the performance overhead of PRAC and SALT-C for both open-page and closed-page systems, and policies are normalized to the respective systems. The overheads of PRAC are constant across TRHD of 125 to 4K, as the slowdown is mainly due to the longer timing (tRC) and not mitigation. PRAC suffers 1.7% slowdown for closed-page systems and 5.1% for open-page systems. Until TRHD of 500, SALT-C incurs negligible overheads (0.1% for open-page and 0.3% for closed page). At TRHD=250, the overheads increase to 2.8% and 4.5%.

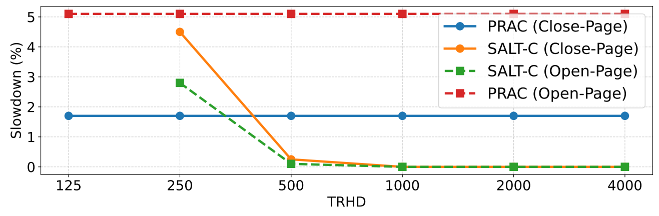

Fig. 14. Slowdown of PRAC and SALT-C for Closed-Page and Open-Page.

AT TRHD below 165, SALT-C requires an ALERT faster than once per 4 ACT, whereas JEDEC specifications permit a minimum of 4 ACTs between ALERTs. So, SALT-C is not viable below a TRHD of 165. Nonetheless, SALT-C provides a practical solution with low storage, low performance overhead, and stronger security for the foreseeable future, when TRHD is likely to be at or above 500. In practice, it is important to have a substrate that can reliably handle current and emerging failure modes rather than one that scales to theoretically low thresholds but causes failure with emerging modalities [53].

C. Changing Subarray with Refresh Generating Activations

REGA [30] provides Blast-Radius-Free Rowhammer mitigation by modifying the DRAM array to generate one (or more) mitigative refreshes on each activation. It does so by modifying the DRAM circuitry to decouple the row buffer after activation, allowing the subarray to service the refresh of another row. Our goal is to mitigate Rowhammer without requiring changes to the DRAM array, making our solution more practical for commodity devices. We also observe that activation is a high-power operation, and JEDEC specifies a rate limit on the number of activations permitted per channel within a given time period (as defined by the tFAW parameter, which specifies the minimum time for four activations). As REGA increases the work done per activation (due to mitigative refreshes), it may exacerbate the tFAW constraints and thus incur performance overheads. SALT does not affect the work done per activation; therefore, it does not affect the tFAW constraints. Finally, SALT-C can tolerate TRHD of $\geq$ 1K without requiring any mitigative refreshes (time-based refresh is sufficient, per Table IX). In fact, our insight of refresh coordination can also help REGA avoid the power overheads of mitigative refreshes.

D. Victim Refresh Using SRAM Trackers

Several studies have proposed storage-efficient trackers for identifying aggressor rows, and mitigating using refresh of victim rows for a defined Blast-Radius. Table XIII compares storage overhead (per bank) of SALT-C with prior schemes for TRHD of 500. Prior schemes incur 17x-116x storage overhead and still suffer from Blast-Radius dependence, making them vulnerable to distant leakage via Ripple-Attacks.

TABLE XIII
SRAM-TRACKER BASED ROWHAMMER MITIGATIONS

Design	SRAM Overhead	Blast Radius?	Secure Against Distant Leakage?
SALT-C	0.5KB (1x)	No	Yes
Graphene [34]	8.4KB (17x)	Yes	No
CBT [45]	37KB (74x)	Yes	No
TWiCE [24]	58KB (116x)	Yes	No

E. Probabilistic In-DRAM Solutions

The security of probabilistic in-DRAM trackers, such as PARFM [19], PrIDE [10], and MINT [38], is dependent on reliable on-chip pseudo-random-number-generator (PRNG). Prior work [35] showed that LFSR-based PRNG (used in DSAC [8]) can be easily broken. Similarly, a recent attack, Phoenix [32], exploits the fact that certain positions are unlikely to get sampled and are prime targets for Rowhammer attacks. Most likely, this non-sampling of certain positions arises from imprecise randomization by the PRNG. Furthermore, when probabilistic solutions fail, reproducing the failure is difficult due to non-determinism; therefore, deterministic solutions are preferred by industry. SALT avoids reliance on PRNG. Probabilistic solutions also require frequent mitigations, leading to greater slowdowns (due to the frequent use of RFM commands, which stall the processor).

MIRZA [46] is our prior work (developed one year earlier), which reduces the mitigation overheads of MINT by performing region-based filtering. MIRZA invokes MINT-based mitigation only if the activation count of the region is more than a *filtering threshold (FTH).* (e.g. about 1500 for TRHD of 1K). Compared to MIRZA, SALT provides the following advantages: (1) it removes the reliance on Blast-Radius, so it provides stronger security, (2) it provides a deterministic solution that avoids reliance on PRNG, and (3) it can easily be extended to handle the new ColumnDisturb failures. Furthermore, SALT/SALT-C suffers much lower slowdown under performance attacks compared to MIRZA (e.g., at TRHD=2K, 60% slowdown for MIRZA versus 15% for SALT/SALT-C),

F. Limitations of Fractal Mitigation

A recent work [36] proposed *Fractal Mitigation (FM)*, which guarantees refresh for only d=1 victim rows, and other victim rows are refreshed with probability $p = 1/2^{(d-1)}$, which helps with mitigating distant rows. FM was designed to handle only indirect attacks (due to mitigative activations) and not due to leakage beyond Blast Radius. We observe that under an exponential leakage model, FM can still cause significant unmitigated leakage for rows d>1. For example, for a mean-time-to-failure of 10K years, the damage on victim rows (d>1) is approximately 40K for ATH=1K. Thus, even if PRAC is implemented with FM, our Ripple-Attack can cause about 20x the damage permitted under PRAC.

G. Mitigation via Dynamic Row Migration

Thus far, we have focused on victim-refresh. Blast-Radius-Freedom can also be achieved through alternative mitigating actions, such as Row-Migration (e.g., RRS [40], AQUA [43], SRS [49]). However, such designs are incompatible with in-DRAM mitigations, as they require additional time to migrate a row from one location within the DRAM bank to another (during migration, the DRAM channel remains busy for several microseconds). Furthermore, these schemes require high-cost SRAM trackers (e.g., Graphene) to track hot rows. These solutions also cause significant slowdowns at TRHD below 1K [42]. Our solution avoids significant SRAM overhead for tracking, avoids slowdowns, and is in-DRAM compatible.

SHADOW [47] proposes in-DRAM row migration where rows are randomized within their respective subarrays. To aid row-swap, each subarray is provisioned with an extra row. As migrations relocate rows, SHADOW maintains the per-subarray physical-to-device mapping in a table resident in the spare row. Each activation must first retrieve the mapping, and only then can it perform the activation at the remapped location – this indirection increases the overall activation latency by 30%. Compared to SHADOW, SALT avoids the extra space for a spare row (40x lower area overhead), does not increase DRAM timings (thereby avoiding timing-related slowdowns), and does not rely on randomization for security. SALT-C uses refresh coordination to avoid slowdowns up to a TRHD of 500, whereas SHADOW relies on periodic RFMs, which cause significant slowdowns at low TRHD.

H. Mitigation via Rate Control of Aggressor Rows

Another way to obtain Blast-Radius freedom is via Blockhammer [50], which limits the number of activations to a given row during the refresh window (32ms) to less than TRHD. While Blockhammer can tolerate Ripple-Attack, it suffers from two key drawbacks. First, Blockhammer is incompatible with in-DRAM settings. DRAM chips are required to deterministically service a given request (within the specified time); therefore, it is not possible for them to arbitrarily delay requests to conform to rate limits. Second, Blockhammer suffers from an unacceptably large worst-case slowdown. For example, consider an application that continuously activates two rows within a bank, the baseline can service one request every tRC (46ns), however, at TRHD of 1K, Blockhammer can service only 500 requests in 32ms resulting in 1400x slowdown (1 request per 64 microseconds). Table XIV shows the theoretical worst-case slowdown for Blockhammer and SALT as TRHD is varied. The potential for 100x-1400x slowdowns with Blockhammer makes it impractical for adoption due to concerns of denial-of-service attacks, whereas the worst-case slowdowns with SALT are pretty small (within 2x).

TABLE XIV
WORST-CASE SLOWDOWN FOR BLOCKHAMMER AND SALT

TRHD	4K	2K	1K	500
Blockhammer	174x	384x	696x	1391x
SALT	1.08x	1.15x	1.3x	1.6x

VIII. CONCLUSION

Conventional in-DRAM solutions for mitigating Rowhammer, including PRAC, rely on refreshing a fixed number of victim rows (as indicated by the Blast-Radius) on either side of an aggressor. In this paper, we exploit the fact that charge leakage decays exponentially with distance, so rows beyond the blast radius still exhibit nonzero (albeit small) leakage. Our Ripple-Attack shows that even small leakage can be amplified to cause a 3x-78x greater charge loss than what is permissible under PRAC. We develop *SALT (Subarray-Level Tracking and Mitigation)* that provides Blast-Radius-Free Rowhammer mitigation, with lower slowdown (0.3% vs. 1.7%) and area overhead (38x) than PRAC.

ACKNOWLEDGEMENTS

Thanks to Reviewer-C for the pointer to ColumnDisturb, which was still not published at the time of the review. Discussing ColumnDistrub helped strengthen our paper.

This work is the culmination of five years of Rowhammer research in our group. Rowhammer has been a great source of fun and a productive (and hopefully impactful) research topic. The author would like to thank Salman Qazi, Aamer Jaleel, Stefan Saroiu, Kaveh Razavi, Michele Marazzi, Prashant Nair, Gururaj Saileshwar, and Giray Yağlikçi for all the intellectually stimulating conversations over the years, which have helped shape much of the Rowhammer work from our group.

Appendix-A: Tolerating ColumnDisturb with SALT

In this section, we focus on *ColumnDisturb* [53], which is a newly discovered data-disturbance error that shows that activations in a row can cause bit-flips at rows hundreds or rows from an aggressor row. ColumnDisturb would break existing row-granularity defenses (such as PRAC+ABO) that rely on Blast Radius, thereby further motivating our central argument that tracking and mitigation must be performed at subarray granularity and without relying on Blast Radius.

A. ColumnDisturb: Mechanism and Impact

Concurrent to our submission, a MICRO paper presented a new data disturbance error, called *ColumnDisturb* [53]. The key idea exploited in this pattern is that different rows of a subarray share the column line. Furthermore, the column line is shared between adjacent subarrays in an open-bit-line architecture that uses a single sense amplifier to serve both subarrays (thereby breaking physical isolation).

The Pattern: The pattern for ColumnDisturb is opening a row and keeping it on for tON time-period, closing it, and repeating this pattern continuously for t_{CD} milliseconds.

The Errors: The authors characterized DDR4 and HBM2 chips and showed that, even with t_{CD} less than the refresh time window (64ms for DDR4), existing chips can still suffer a large number of bit flips. Notably: (1) the bitflips occurred at distant rows (about 374-446 rows from the aggressor row, when aggressor and victim were in the same subarray), and (2) sometimes the bitflips occurred in adjacent subarrays, so an activation of a row in one subarray can cause vulnerability for up to 3 subarrays (current + two immediate neighbors).

The Impact: As the cells affected by ColumnDisturb are hundreds of rows away, any mechanism (such as PRAC+ABO) that counts per-row activations and mitigates only a few rows on either side of an aggressor will not be able to tolerate ColumnDisturb. This error modality again underscores that, for robust defense, it is important to think in terms of subarrays (rather than rows) for both tracking and mitigation.

The ColumnDisturb paper mentions two mitigations to limit t_{CD} to 8ms. First, reduce the DRAM refresh time from 32ms to 8ms, which incurs significant energy and performance overheads. Second, a scheme called *PRVR (Proactively Refreshing ColumnDisturb Victim Rows)*, which refreshes all rows in three subarrays once before the aggressor row is hammered or pressed enough times to induce a bit-flip. Unfortunately, the paper does not provide any design details for PRVR, such as the tracking mechanism, the threshold number of activations to target 8ms, and the rate of mitigation across the three subarrays to ensure that mitigations are completed within 8ms. The paper observes that PRVR still incurs approximately 30% of the performance overhead and 25% of the energy overhead of an 8ms refresh, indicating that PRVR's overheads remain high. We show our insight of refresh coordination can help SALT-C tolerate ColumnDisturb with virtually zero overhead (by leveraging the time-based refresh for mitigations).

B. Extending SALT and SALT-C to Handle ColumnDisturb

To handle ColumnDisturb, we introduce SALT with *CDP ColumnDisturb-Protection (CDP)*. SALT with CDP requires two changes. First, on an activation to a subarray, CDP increases the ACtr of three subarrays: the given subarray and the two neighboring subarrays. Second, CDP converts long row-open time into multiple activations [41]. For example, each period of 500ns the row is open is treated as one activation. Thus, if a row is kept open for a long time, it will cause multiple increments for ACtr, and the attacker is forced to use a tON of 500ns or less. Therefore, if an attacker targets a t_{CD} of 8 ms, the attacker must inflict 16K activations on the aggressor row. To ensure protection, CDP must keep MaxACT below this threshold. We note that with CDP, each activation increments 3 subarrays, so APM must be reduced by 3x (to provision for 3x higher mitigation activity) compared to the SALT design that targets a MaxACT of 16K (TRHD of 8K, so APM of 212). Therefore, to tolerate an 8ms ColumnDisturb, CDP uses APM of 70, ATH of 140.

C. Impact on Performance

To highlight the importance of refresh coordination for reducing performance overhead, we evaluate two variants of CDP: one built on SALT (no refresh coordination) and one built on SALT-C (with refresh coordination). Figure 15 compares the performance overhead of these two designs along with two prior solutions: (1) Increasing refresh to 8ms, and (2) PRVR* design that is estimated to incur 30% overhead of 8ms refresh (as the design details of PRVR are not available, we cannot do a direct comparison with PRVR).

With 8ms refresh, the slowdown for ColumnDisturb protection is significant (22%). PRVR* is estimated to incur 30% overhead of the 8ms refresh scheme, so it still incurs high overheads (6.7%). SALT (CDP) has comparatively lower overheads (2.3%). SALT-C (CDP) leverages time-based-ref for mitigation and eliminates all performance overheads. Thus, SALT-C (CDP) is a robust and low-overhead solution for mitigating ColumnDisturb. Thus, our critical insights of doing tracking and mitigation at subarray granularity, and leveraging time-based-refresh to avoid activity-based-refresh are both powerful even for this new modality of data-disturbance error, which appeared several months after our paper was submitted.

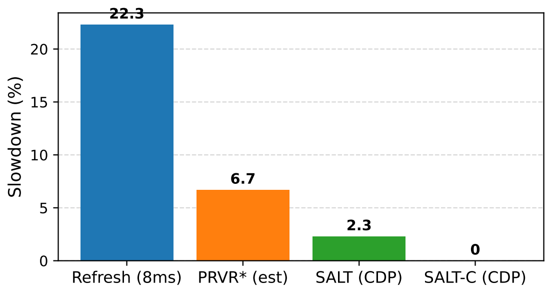

Fig. 15. Slowdown of different ColumnDisturb protections. Refresh coordination helps SALT-C (CDP) tolerate ColumnDisturb with 0% slowdown.

Artifact Appendix

This artifact presents the code for SALT, our Rowhammer mitigation based on Subarray-Level Tracking and Mitigation. We provide C++ code to determine the performance overhead (and ALERTs-per-tREFI) for different SALT configurations. We provide scripts and code for the following analysis and to recreate the data in the following key studies:

1) Fig-6a: The slowdown due to SALT for different TRHD.
2) Fig-6b: The ALERTs-per-tREFI for different TRHD.
3) Fig-8: The slowdown due to SALT-C for different TRHD.

The artifact is at: https://zenodo.org/records/17869821

Artifact check-list (meta-information)

- **Algorithm**: SALT/SALT-C mitigation.
- **Compilation**: Tested with g++ (Apple clang-1500.3.9.4)
- **Run-time environment**: Tested on Mac OS Sonoma 14.5 and Linux.
- **Hardware**: The artifact can be run on a Linux machine (64 core) or a laptop (e.g. Macbook).
- **Execution**: Simulator-based performance Models.
- **Metrics**: Slowdowns for the tolerated thresholds
- **Output**: Recreating performance results: Fig-6ab, Fig-8.
- **Experiments**: Instructions to run the analysis and plot graphs are available in the README file.
- **How much disk space is required (approximately)?**: 5 GB.
- **How much time is needed to prepare workflow (approximately)?**: 10 minutes
- **How much time is needed to complete experiments (approximately)?**: 3 hours (on Linux server) or 15 hours (Laptop)
- **Publicly available?**: Yes.
- **Workflow framework used?**: Simulation Models.
- **Archived (DOI)?**: Yes, DOI:10.5281/zenodo.17860231.

Description

How to access: The code is available at the following link: https://zenodo.org/records/17869821

Hardware dependencies: Evaluations can be run on most generic Linux machines or laptops.

Software dependencies: g++ is used for compilation, and Python3 with matplotlib and numpy to plot graphs.

Experiment workflow

See README for detailed instructions. Run the **./runall.sh** script to run experiments and **./graph.sh** to display figures.

Evaluation and expected results

The artifact provides the scripts to run both the performance evaluations and display the graphs. The relevant commands are provided in the artifact README. The artifact recreates the key results – Figure 6 and Figure 8.

Experiment customization

Scripts are self-contained and require no customization.

Methodology

Submission, reviewing, and badging methodology:

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

References

[1] "JEDEC Updates JESD79-5C DDR5 SDRAM Standard: Elevating Performance and Security for Next-Gen Technologies," https://www.jedec.org/news/pressreleases/jedec-updates-jesd79-5c-ddr5-sdram-standard-elevating-performance-and-security.

[2] (2021) On Setting Thresholds for Counter-based Rowhammer Defenses. https://stefan.t8k2.com/rh/thresholds/index.html.

[3] O. Canpolat, A. G. Yağlıkçı, G. F. Oliveira, A. Olgun, N. Bostancı, İ. E. Yüksel, H. Luo, O. Ergin, and O. Mutlu, "Chronus: Understanding and securing the cutting-edge industry solutions to dram read disturbance," *HPCA*, 2025.

[4] F. Devaux and R. Ayrignac, "Method and circuit for protecting a dram memory device from the row hammer effect," Patent US 10,885,966 B1, 2021, filed 2020-08-04; Priority FR2006541 (2020-06-23).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[10] A. Jaleel, G. Saileshwar, S. W. Keckler, and M. Qureshi, "Pride: Achieving secure rowhammer mitigation with low-cost in-dram trackers," in *ISCA*. IEEE, 2024.

[11] P. Jattke, M. Marazzi, F. Solt, M. Wipfli, S. Gloor, and K. Razavi, "Mcsee: evaluating advanced rowhammer attacks and defenses via automated dram traffic analysis," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[12] 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.

[13] JEDEC, "Near-term dram level rowhammer mitigation (jep300-1)," 2021.

[14] JEDEC, "System level rowhammer mitigation (jep301-1)," 2021.

[15] JEDEC, "JESD79-5C: DDR5 SDRAM Specifications," 2024.

[16] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 24–35.

[17] 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.

[18] J. Kim, S. Baek, M. Wi, H. Nam, M. J. Kim, S. Lee, K. Sohn, and J. H. Ahn, "Per-row activation counting on real hardware: Demystifying performance overheads," *IEEE Computer Architecture Letters*, 2025.

[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] 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.

[21] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[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 Symposium*, 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] S. P. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–136, 1982.

[26] K. Loughlin, J. Rosenblum, S. Saroiu, A. Wolman, D. Skarlatos, and B. Kasikci, "Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, 2023.

[27] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci, "Stop! hammer time: rethinking our approach to rowhammer mitigations," ser. HotOS '21, 2021.

[28] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," 2012, pp. 183–196.

[29] 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.

[30] 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.

[31] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE (TCCA) Newsletter*, 1995.

[32] D. Meyer, P. Jattke, M. Marazzi, S. Qazi, D. Moghimi, and K. Razavi, "Phoenix: Rowhammer attacks on ddr5 with self-correcting synchronization," in *IEEE Symposium on Security and Privacy (SP)*, USA, 2026.

[33] Micron Technology Inc., "System Power Calculators," https://www.micron.com/support/tools-and-utilities/power-calc.

[34] 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.

[35] S. Qazi and D. Moghimi, "Soothsayer: Bypassing dsac mitigation by predicting counter replacement," *DRAMSec4*, 2024.

[36] M. Qureshi, " AutoRFM: Scaling Low-Cost In-DRAM Trackers to Ultra-Low Rowhammer Thresholds ," in *HPCA*, 2025.

[37] M. Qureshi and S. Qazi, "MOAT: Securely mitigating rowhammer with per-row activation counters," *ASPLOS-2025*, 2025.

[38] M. Qureshi, S. Qazi, and A. Jaleel, "Mint: Securely mitigating rowhammer with a minimalist in-dram tracker," in *MICRO*. IEEE, 2024.

[39] K. A. S. Beamer and D. Patterson, "The gap benchmark suite," in *arXiv preprint arXiv:1508.03619*, 2015.

[40] 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*, 2022, pp. 1056–1069.

[41] 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)*. IEEE, 2024, pp. 935–948.

[42] A. Saxena, S. Mathur, and M. Qureshi, "Rubix: Reducing the overhead of secure rowhammer mitigations via randomized line-to-row mapping," ser. ASPLOS '24, 2024.

[43] 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.

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

[45] 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.

[46] H. Taneja, A. Hajiabadi, M. Marazzi, K. Razavi, and M. Qureshi, "Mirza: Efficiently mitigating rowhammer with randomization and alert," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2026.

[47] 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.

[48] J. Woo, C. S. Lin, P. J. Nair, A. Jaleel, and G. Saileshwar, "Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues," *HPCA*, 2025.

[49] 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.

[50] A. G. Yağlıkçi, 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.

[51] A. G. Yağlıkçı, J. S. Kim, F. Devaux, and O. Mutlu, "Security analysis of the silver bullet technique for rowhammer prevention," 2021. [Online]. Available: https://arxiv.org/abs/2106.07084

[52] 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.

[53] İsmail Emir Yüksel, A. Olgun, F. N. Bostancı, H. Luo, A. G. Yağlıkçı, and O. Mutlu, "Columndisturb: Understanding column-based read disturbance in real dram chips and implications for future systems," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025.