2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

AutoRFM: Scaling Low-Cost In-DRAM Trackers to Ultra-Low Rowhammer Thresholds

Moinuddin Qureshi
Georgia Institute of Technology
moin@gatech.edu

***Abstract*—In-DRAM Rowhammer mitigation has the potential to solve the Rowhammer problem without relying on other parts of the system. In-DRAM mitigation requires space (to identify the aggressor rows) and time (to perform the victim refresh). To reduce the storage overheads of tracking, recent works have developed secure low-cost in-DRAM trackers that can probabilistically identify aggressor rows. To obtain the time required for mitigation, these trackers rely on the *Refresh Management (RFM)* command introduced in DDR5. As RFM stalls the bank for a latency of 200ns-400ns, frequent use of RFM can cause significant slowdowns. For example, scaling the recent MINT tracker to a threshold of 100 incurs 33% slowdown. The goal of this paper is to enable low-cost trackers to tolerate ultra-low thresholds (sub-100) while incurring negligible slowdown.**

This paper proposes *AutoRFM*, a transparent RFM mechanism that can provide mitigation time to the DRAM chips without stalling the bank. The key insight in AutoRFM is to leverage the subarray structure (e.g. each bank contains 256 subarrays) and perform mitigation on only one of the subarrays. Operations to all subarrays that are not under mitigation are serviced without any interruption. If activation occurs to the subarray under mitigation, the DRAM chip sends an ALERT signal informing the Memory Controller to retry after a predefined time. As AutoRFM works best if consecutive requests to the same bank do not get mapped to the same subarray, we use *Randomized Memory Mapping* to break the spatial correlation between memory accesses. Furthermore, we also develop a *Fractal Mitigation Algorithm* that can tolerate transitive attacks (such as Half-Double) without requiring recursive mitigations to the same subarray. Our design ensures that a declined request does not have to wait more than 200 ns before retrying, thus limiting the slowdown and avoiding any potential for denial of service. Our evaluations, with SPEC, GAP, and stream workloads, show that AutoRFM enables low-cost trackers to tolerate a threshold of as low as 74 while incurring an average slowdown of only 3.1%.

I. INTRODUCTION

Rowhammer is a data-integrity error that occurs when rapid activations of a DRAM row cause bit-flips in neighboring rows [21]. Rowhammer is a serious security threat, as it gives an attacker the ability to flip bits in protected data structures, such as page tables, which can result in privilege escalation [2], [3], [5]–[7], [46], [49] and breach confidentiality [25]. Rowhammer has been difficult to solve because the *Rowhammer threshold (T_{RH})*, which is the number of activations required to induce a bit-flip, has continued to decrease with successive generations of DRAM, reducing from 140K [21] to 4.8K [17] in the last decade, as shown in Figure 1(a). As Rowhammer thresholds continue to reduce, Rowhammer mitigations must be effective even at low thresholds.

Typical hardware-based mitigation for Rowhammer relies on a *tracking* mechanism to identify the aggressor rows and issue a refresh to the neighboring victim rows [9]. Such mitigation can be deployed at the memory controller (MC) or within the DRAM chip (in-DRAM). The in-DRAM approach is appealing, as it can solve the Rowhammer transparently within the DRAM chips without relying on other parts of the system. In this paper, we focus on in-DRAM mitigations.

In-DRAM Rowhammer mitigation requires both space and time, as shown in Figure 1(b). **First**, storage for tracking aggressor rows. The SRAM budget available for tracking the aggressor rows is quite small (a few bytes), so existing trackers (such as TRR) cannot track all aggressor rows and can be broken with stressful patterns [5], [12]. **Second**, time for doing victim refresh to perform the mitigation. Although initial in-DRAM trackers (such as TRR) exclusively relied on borrowing time during refresh (REF) to perform mitigation, such an approach is severely limits the amount of time that can be used for mitigation, thus limiting the tolerable threshold [29].

Although existing low-cost trackers from industry (such as TRR and DSAC [10]) have been broken, this does not mean that it is not possible to have secure low-cost in-DRAM trackers. In fact, developing secure, low-cost in-DRAM trackers is an area of active research. Examples of secure low-cost in-DRAM trackers include PARFM [18], PrIDE [11], and MINT [37]. These trackers select the activated row with a given probability. The selection probability determines the tolerated threshold. For example, for tolerating a threshold of of 1000, the selection probability is approximately 2.5%. Tolerating a lower threshold requires selection probability be increased proportionately.

Even if low-cost trackers can identify the aggressor rows accurately, the DRAM bank still needs time to perform mitigative refreshes. This is especially true at low thresholds when the tracker performs more frequent selection of aggressor rows. DDR5 specifications introduce a new command called *Refresh Management (RFM)* that can allow the MC to provision the DRAM chip with the time to perform the mitigation, as shown in Figure 1(c). The MC counts the activations for a given bank, and when a specified number of activations is reached, the MC inserts the RFM command, which stalls the bank for a specified time (200ns to 400ns). As RFM is a blocking command, frequent use of RFM can cause significant slowdowns, thus placing a practical limit on the thresholds tolerated by low-cost trackers.

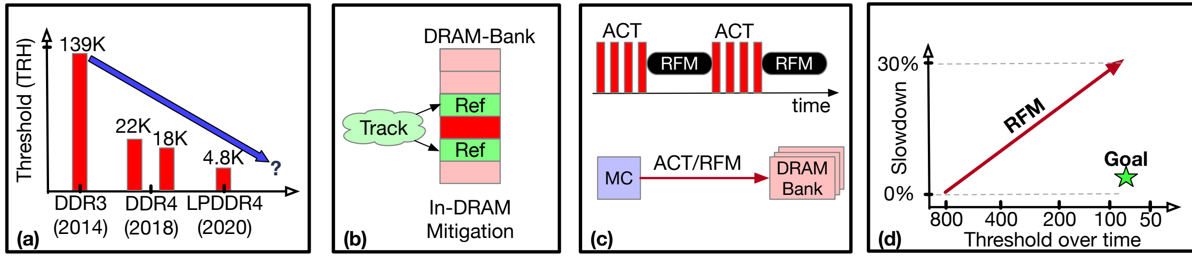

Fig. 1. (a) The trend in Rowhammer thresholds (TRH) (b) Overview of in-DRAM mitigation, which needs a tracker and time to do mitigation (c) Overview of RFM to provide time to DRAM chips to perform mitigation (d) The slowdown of RFM as the Rowhammer thresholds reduce over time.

The performance overheads of RFM are negligibly small for the current Rowhammer threshold (800 and higher). However, at lower thresholds, RFM needs to be invoked frequently. For example, for a threshold of 100, RFM must be invoked every 4 activations, which causes a slowdown of 33%, as shown in Figure 1(d). The goal of our paper is to enable low-cost and secure in-DRAM trackers to tolerate ultra-low thresholds (sub-100) while incurring negligible slowdowns.

Our solution is based on two observations. First, DRAM banks consist of a large number of independent modules called *subarrays* [22] [30] [33], each with its own *Row Buffer*. Typically, a bank contains 256 subarrays (512 rows each). While mitigation is performed on a subarray, operations can be performed on other subarrays within the bank. Second, recent DRAM specifications [14] allows the DRAM chip to use ALERT to indicate that it needs additional time for mitigation. We can similarly modify ALERT to indicate that an incoming request conflicted with a subarray performing a mitigation. Based on these observations, we propose *AutoRFM*, a transparent RFM mechanism that exploits a large number of subarrays to perform non-blocking RFM mitigations. AutoRFM enables low-cost trackers to scale to ultra-low Rowhammer thresholds (sub-100) while incurring negligible slowdown.

When the low-cost tracker identifies an aggressor row, the subarray corresponding to that row is marked for Rowhammer mitigation. We call such a subarray, *Subarray Under Mitigation (SAUM)*. The mitigation causes the SAUM to become unavailable for a defined time (with a blast radius of 2, we need to perform 4 victim refreshes, so about 200ns). While mitigation is performed on SAUM, all requests to the bank are processed in a normal manner if they map to a sub-array other than SAUM. If the bank receives an activation request (ACT) for a row that maps to SAUM, such a request cannot be serviced until the SAUM becomes free. Therefore, the DRAM chip inserts the ALERT signal to indicate that the ACT request has failed and that the memory controller (MC) can retry that ACT after the specified time (e.g., 200ns).

To ensure that the slowdown from AutoRFM remains low, we need to satisfy two conditions: (1) Consecutive requests should have only a negligibly small probability of mapping to the same subarray, so as to avoid the latency overheads of conflicting with SAUM. (2) Once the SAUM finishes mitigation, it must become free to perform the demand activations, so as to limit the slowdown and potential for denial of service.

To reduce the conflict with SAUM, we break the spatial correlation between memory accesses using the recently proposed *Randomized Memory Mapping*, called *Rubix* [42]. Rubix uses a low-latency block cipher to convert the line address into an encrypted line address and uses it to access memory. Rubix ensures that any activation has a negligible probability of conflicting with SAUM (1/256 with 256 subarrays). While Rubix sacrifices row-buffer hits, it improves the bank-level parallelism, so the impact on performance remains low (1.5%).

Transitive attacks [48] use victim refreshes to inflict Rowhammer on distant rows [23]. So, mitigation must ensure that rows that undergo victim refresh can also trigger mitigation. Recent trackers, such as PrIDE [11] and MINT [37], enable rows that undergo victim refresh to also possibly trigger a mitigation. However, such recursive mitigations can potentially tie the subarray for several consecutive rounds. Ideally, we want the subarray to be busy only for a deterministic amount of time. To that end, we propose *Fractal Mitigation (FM)*, which is robust to transitive attacks, while incurring only a single round of mitigation. FM always refreshes the immediate neighbors on both sides and the distant neighbors with an exponentially reducing probability ($2^{(1-d)}$). We analyze the security of FM and provide a simple implementation. As FM avoids recursive mitigations, MINT + FM can operate at an even lower threshold than MINT (e.g., 74 instead of 96).

Our analysis with SPEC, GAP, and stream workloads shows that AutoRFM allows low-cost trackers to scale to a threshold of 74 while incurring an average of only 3.1% slowdowns.

Overall, our paper makes the following contributions:

1) We show that RFM latency is the main bottleneck for scaling low-cost in-DRAM trackers to ultra-low thresholds. We propose *AutoRFM*, which performs RFM transparently and uses ALERT for subarray conflicts.
2) We reduce the rate of subarray conflicts and ALERT in AutoRFM using randomized memory mapping.
3) We propose *Fractal Mitigation* that securely defends transitive attacks without requiring recursive mitigations.
4) We ensure that a subarray is busy for only 200ns after which it is available for service, thus limiting performance impact and denial-of-service concerns.

We also compare AutoRFM to PRAC [14]. PRAC requires significant area overheads and longer cycle times (tRC increases by 10%). AutoRFM is a low-cost means of tolerating low thresholds while avoiding the overheads of PRAC.

II. Background and Motivation

A. Threat Model

Our threat model assumes that an attacker can issue memory requests for arbitrary addresses and is free to choose the policy of the memory system that is best suited for the attack. We assume that the attacker knows the defense algorithm but does not have physical access to the system (e.g., the outcome of the random-number generator). Our defense aims to prevent Rowhammer from being attacked against all access patterns, including Half-Double [23]. We declare an attack to be successful when any row receives more than the threshold number of activations without any intervening mitigation. Row-Press [28] attack is out of our scope, as its effects can be mitigated using alternative techniques [39].

B. DRAM Architecture and Parameters

Figure 2 provides an overview of the DRAM architecture. DRAM cells store data as charge on the capacitor. DRAM chips are organized as two-dimensional 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*. Data can be accessed from the row buffer by providing the column address. The DRAM chip is divided into a number of banks (32 or 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* [22], [30], each with a dedicated row buffer and sensing circuit. For servicing a request, the bank routes the incoming access to the given subarray, and the routes the data from the subarray to an I/O buffer.

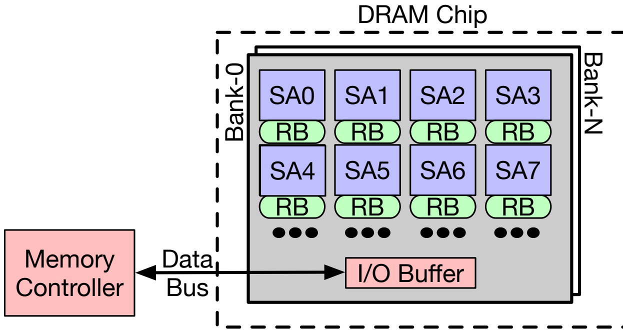

Fig. 2. DRAM Architecture and Operation

DRAM has deterministic timings, which are specified as part of the 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 first be precharged (PRE). To ensure data retention, the data in DRAM are refreshed every tREFW. To reduce the latency impact of refresh, memory is divided into 8192 groups, and one REF is issued every tREFI.

Within a tREFI of 3900ns, the tRFC time (410ns) is used for performing refresh; therefore, given a tRC of 48ns, we can perform a maximum of 73 activations within tREFI.

TABLE I
DRAM Timings (DDR5 specs).

Parameter	Description	Value
tRCD	Time for performing ACT	12 ns
tRP	Time to precharge an open row	12 ns
tRAS	Minimum time a row must be kept open	36 ns
tRC	Time between successive ACTs to a bank	48 ns
tREFW	Refresh Period	32 ms
tREFI	Time between successive REF Commands	3900 ns
tRFC	Time for REF Command	410 ns
tRFM	Time for RFM Command	205 ns

C. DRAM Rowhammer Attacks

Rowhammer [21] occurs when a row (aggressor) is frequently activated, causing bit-flips in nearby rows (victim). 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 for a single-sided pattern *(TRH-S)* or a double-sided pattern *(TRH-D)*. As shown in Table II, TRH has decreased significantly, from 139K (TRH-S) in 2014 [21] to 4.8K (TRH-D) in 2020 [17].

TABLE II
Rowhammer Threshold Over Time

DRAM Generation	TRH-S (Single-Sided)	TRH-D (Double-Sided)
DDR3-old	139K [21]	–
DDR3-new	–	22.4K [17]
DDR4	–	10K [17] - 17.5K [17]
LPDDR4	–	4.8K [17] - 9K [23]

Rowhammer is a serious security threat, as an attacker can use it to flip bits in the page table to perform privilege escalation [5], [6], [46], [54] or break confidentiality [25].

Solutions for mitigating Rowhammer typically rely on a mechanism to identify the aggressor rows and then mitigate by refreshing the victim rows. The identification of aggressor rows can be done either at the *Memory Controller (MC)* or within the DRAM chip *(in-DRAM)*. In-DRAM mitigation can solve Rowhammer within the DRAM chips without relying on other parts of the system, and enable DRAM manufacturers can tune their solution to the TRH observed in their chips. Therefore, in this paper, we consider in-DRAM mitigations.

D. Secure Low-Cost In-DRAM Trackers

As the thresholds reduce, an attacker can focus the attack on a larger number of aggressor rows. For current (future) thresholds, we can have hundreds (thousands) of aggressor rows per bank. Unfortunately, the SRAM budget available for tracking within the DRAM chip is quite small (a few tens of bytes), which is insufficient to track all the aggressor rows deterministically. Recent research has developed secure low-cost in-DRAM trackers using probabilistic selection.

PARFM [18]: PARFM buffers the row addresses that incur an activation. On mitigation, one of the buffered addresses is selected at random. The size of the buffer depends on the mitigation window, which determines the tolerated threshold.

PrIDE [11]: PrIDE selects each activation with a probability (p) and inserts it in a 4-entry FIFO. At mitigation, the oldest entry is chosen. The tolerated threshold depends on selection probability (p), loss probability of the buffer, and Tardiness (extra activations received between insertion and mitigation).

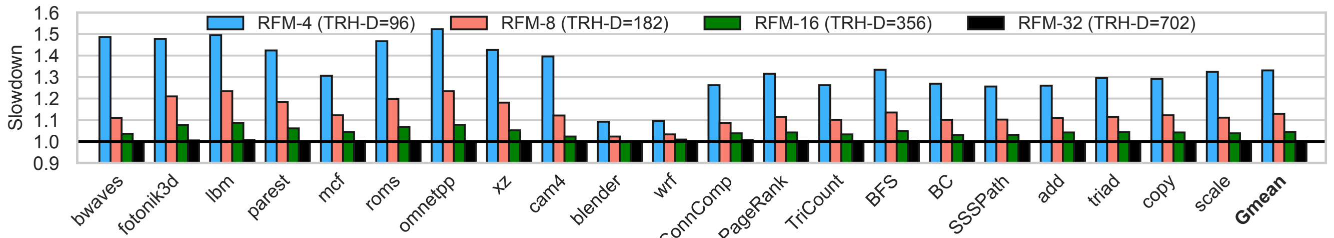

Fig. 3. Performance Impact of RFM. The average slowdown of RFM-4, RFM-8, RFM-16, and RFM-32 is 33%, 12.9%, 4.4%, and 0.2%. Thus, scaling low-cost trackers to tolerate ultra-low thresholds (sub-100) is not practical using RFM as it would cause unacceptable slowdowns (average 33%).

MINT [37]: MINT is a recent low-cost secure tracker. As shown in Figure 4, MINT operates over a window of N activations. At the start of the window, MINT randomly selects which of the N slots will be picked for mitigation. At the end of the window, the selected entry is mitigated, and the process repeats. MINT is a single-entry tracker and provides 25% lower threshold than PrIDE. MINT can tolerate *Transitive Attacks* [48] by selecting from N+1 positions (one position is reserved for the recently mitigated row, albeit victim refreshes for this row are performed at an increased distance).

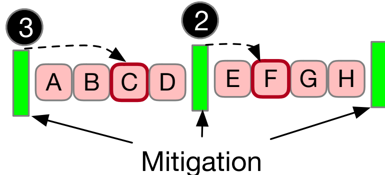

Fig. 4. Overview of MINT. At each mitigation one of the activation slots from the upcoming window is marked for mitigation at the end of the window.

As MINT has low storage overheads (single-entry) and it can operate at lower thresholds than prior low-cost trackers, we select MINT as the representative low-cost tracker in our work. We also note that, unlike PrIDE, MINT is guaranteed to select exactly one row (no more, no less) in a given window; therefore, the amount of mitigation time for a given window remains constant. The threshold tolerated by MINT is purely determined by the window size (see Appendix-A). Table III shows the threshold (TRH-D) tolerated by MINT as the window size (W) is varied. To tolerate lower thresholds, MINT requires shorter window (frequent mitigation).

TABLE III
THRESHOLD TOLERATED BY MINT

Window Size (W)	TRH-D (Double-Sided)
4	96
8	182
16	356
32	702

E. Refresh Management (RFM): Finding Time

As DRAM is a synchronous device, it needs to be provisioned with time to perform Rowhammer mitigation (victim refresh of neighboring rows). At high thresholds (e.g. several thousand), DRAM can exclusively rely on borrowing time during Refresh to perform mitigation. However, at a low threshold, we require frequent mitigations, so relying exclusively on refresh for mitigation becomes impractical.

DDR5 introduced a new command, *Refresh Management (RFM)*, which allows the Memory Controller (MC) to provide the DRAM chips the time to perform mitigation (the responsibility of tracking the aggressor rows still remains within the DRAM chip). The MC keeps track of the number of activations for each bank in a counter (RAA). When the RAA of a bank reaches a specified value, the MC inserts an RFM command to the DRAM chip. The RFM command incurs a latency (tRFM) of either tRFC/2 (205ns) or tRFC (410ns). The bank remains inaccessible during RFM, so no demand operations can be performed on the bank. RFM reduces the RAA counter by a value of *RFMTH*. Furthermore, a refresh operation also reduces RAA by 50% or 100% of RFMTH. We refer RFM with RFMTH value of N as *RFM-N*.

F. Performance Impact of RFM: Key Limiter of Threshold

As RFM is a blocking operation, frequent use of RFM causes significant slowdown, as shown in Figure 3. For these studies, we vary the RFMTH (window of MINT) from 4 to 32. We assume a tRFM of 205ns, with REF reducing RAA by RFMTH. We do not place any per-tREFI limit on RFM.

At RFMTH of 32, there is negligible performance loss (0.2%) with RFM, as banks typically do not receive 32 activations per tREFI, and hence, the RAA counter is reset to zero at almost every REF. At RFMTH of 16, the performance overhead is still relatively small (4.4% on average), thus RFM is a practical means to tolerate a TRH-D of as low as 350.

However, at lower thresholds, the overheads increase significantly. At RFMTH of 8, which is equivalent to TRH-D of 180, we observe a slowdown of 12.9%. Furthermore, to scale MINT to an ultralow threshold (sub-100), we would need an RFMTH of 4, which would incur unacceptable performance overheads (33%). The latency overhead due to RFM is thus a key limiter to scaling low-cost trackers to ultra-low thresholds.

G. Goal: Scaling Low-cost Trackers to Ultra-Low Threshold

As Rowhammer thresholds reduce, ideally, we want to use secure low-cost trackers to tolerate ultra-low thresholds (sub-100). RFM is not scalable due to high latency overheads. An alternative RFM mechanism that enables the DRAM chips to perform mitigation without blocking the bank can enable these low-cost trackers to tolerate ultra-low thresholds. The goal of our paper is to develop such a mechanism. We describe our experimental methodology before describing our solution.

III. Experimental Methodology

We use *memsim* [1], a cycle-level multi-core simulator with a detailed memory model. Table IV shows our configuration. We use DDR5 memory timings (see Table I). We use eight out-of-order cores coupled with a DRAM channel that has a total of 64 banks. As we are interested in server systems, we use the memory mapping of AMD Zen [13] (this mapping exploits bank-level parallelism by keeping two lines of a 4KB page in the same bank and distributing the page across 32 banks). For this mapping, closed-page policy performs better than an open-page policy (our design permits row-buffer hits if a later request gets serviced within tRAS).

TABLE IV
Baseline System Configuration

Out-of-Order Cores	8 core, 4GHz, 4-wide, 256 entry ROB
Last Level Cache (Shared)	8MB, 16-Way, 64B lines
Memory specs	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	AMD Zen Mapping [13]

We used 11 SPEC-2017 benchmarks that have at least one ACT per 1K instruction (ACT-PKI), 6 benchmarks from the Graph-Analytics Platform (GAP) [40], and 4 benchmarks from Stream [31]. We use a representative slice of one billion instructions. We run the workloads in 8-core rate mode until each core completes 1 billion instructions. We measure performance using weighted speedup. Table V shows the characteristics of the workload, including ACT-PKI and ACT-per-tREFI (per bank), which highligts the need for RFM (for example, at RFMTH=32, RFM is rarely needed as RAA gets reset at REF).

TABLE V
Workload Characteristics
(ACT-per-tREFI is averaged per bank).

Suite	Workloads	ACT-PKI	ACT-per-tREFI
SPEC2K17	bwaves	35.7	27.7
	fotonik3d	26.7	33.0
	lbm	25.5	34.4
	parest	20.0	28.4
	mcf	22.0	31.4
	roms	13.4	26.7
	omnetpp	9.5	29.0
	xz	5.9	25.0
	cam4	4.2	18.2
	blender	1.4	9.7
	wrf	1.0	6.6
GAP	ConnComp	80.7	35.0
	PageRank	40.9	31.5
	TriCount	35.2	26.1
	BFS	31.1	30.4
	BC	16.0	26.3
	SSSPath	9.0	23.9
Stream	add	12.1	29.2
	triad	10.3	28.6
	copy	9.3	27.8
	scale	7.6	27.1

IV. AutoRFM: Enabling Non-Blocking RFM

For low-cost in-DRAM trackers, scaling to low thresholds becomes impractical, as the latency overhead of RFM causes a significant slowdown. RFM, as currently defined, is a blocking operation, which means a bank becomes unavailable for demand operations during RFM. To enable low-cost in-DRAM trackers to scale to low thresholds, we propose *AutoRFM*, a transparent RFM design that leverages the subarray structures (already present in DRAM banks) to enable a non-blocking RFM operation. AutoRFM relies on ALERTs to indicate a conflict with a subarray undergoing a mitigation.

A. AutoRFM: Overview and Operation

Figure 5 shows an overview of AutoRFM. For simplicity, we show only one bank. The bank is equipped with a low-cost tracker (e.g., MINT). With AutoRFM, the memory controller (MC) no longer sends *explicit* RFM to the DRAM. Instead, the DRAM module performs the mitigation transparently. For example, if MINT selects a row for mitigation and that row is assigned to subarray *SA2*, the DRAM bank would perform a mitigation on SA2. This subarray is called *Subarray Under Mitigation (SAUM)*. Our design ensures that mitigation is started only on a precharge operation to the bank; therefore, the only request that can conflict with the SAUM is a subsequent activation request to a row that maps to the SAUM.

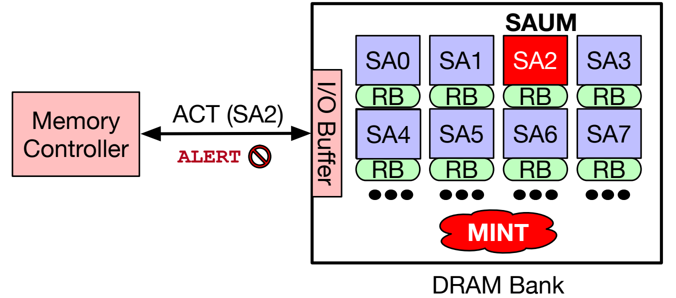

Fig. 5. Overview of AutoRFM (SA=Subarray, RB=Row-Buffer, Figure not to scale). AutoRFM uses subarrays to enable non-blocking RFM.

If a request from the MC maps to a subarray other than SAUM, the DRAM chip can service it normally, without any interruption. If the MC sends an ACT request to a row that maps to the SAUM, the DRAM chip cannot service the request, so it asserts an ALERT signal. If the MC receives an ALERT during an ACT operation, it marks the ACT request as failed.[1] This failed request can be re-tried only after a specified time (t_M). Without loss of generality, our design refreshes a total of four victim rows, so t_M is approximately equal to 200 ns (four times tRC). Our design ensures that SAUM becomes free to service demand request after t_M time period (by avoiding recursive mitigations). Thus, an ACT request that has failed once is guaranteed to succeed upon retrying after t_M. Thus, AutoRFM incurs a deterministic latency overhead.

[1] As the channel contains multiple chips, an ACT can get declined by only a subset of chips, and other chips complete the ACT (ALERT is ORed, so ALERT is asserted). To ensure correctness, the MC sends a precharge for any failed ACT to ensure all chips have the conflicted row in a closed state.

Similar to RFM, a key parameter for AutoRFM is the number of activations to the bank that causes one mitigation. We call this parameter *AutoRFM Threshold (AutoRFMTH)*. Each bank has a counter that counts the number of activations, and when it reaches AutoRFMTH, the bank selects the aggressor row, triggers the mitigation, and resets this per-bank counter.

B. Low-Cost Tracker and Mitigation Schedule

AutoRFMTH determines the window over which MINT operates. For example, if AutoRFMTH equals 4, then the window for MINT equals 4 activations (e.g. A, B, C, D) from which one is randomly selected as an aggressor row. In fact, MINT pre-decides, at the start of the window, which slot in the upcoming window will get selected for mitigation. For example, if MINT had pre-selected the third slot to be selected for mitigation, then Row-C would get marked as an aggressor, as shown in Figure 6. In the second window, MINT randomly selects the second slot (Row-F), and so on.

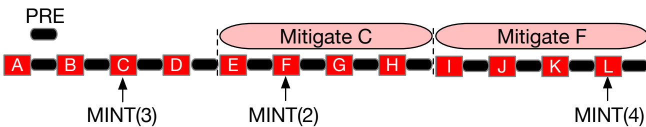

Fig. 6. Timeline of activations and mitigation for AutoRFM Threshold of 4. MINT identifies an aggressor over 4 activations, and mitigation is performed on the aggressor at the 4th precharge (mitigation latency is four tRC).

Each ACT operation on a bank is eventually followed by a precharge (PRE) operation. To simplify our design, we initiate mitigation only when the last PRE operation occurs in the window, as it marks the time when the memory controller infers that no row is open in the bank. Therefore, the MC would not send any read, write, or PRE operations to the bank. However, the MC can send an ACT to open a new row.

Without loss of generality, we assume that mitigation is performed by refreshing two rows on either side of the aggressor row (so, for mitigating Row C, we would refresh C+2, C+1, C-1, and C-2). As the mitigation keeps the subarray busy for a period of four tRC, the minimum value of AutoRFMTH would be 4 activations. We note that even lower value of AutoRFMTH can be supported if we reduce the number of rows that receive victim refresh from 4 to 2. However, to keep our design simple, we do not consider these options.

If the AutoRFMTH is larger than 4, then there would be times when no subarray is under mitigation, and therefore no request would have any subarray conflict. Thus, longer AutoRFMTH would result in a lower slowdown. We are specifically interested in AutoRFMTH between 4 and 16, as RFM becomes a low-overhead option beyond RFMTH of 16.

We note that at any time at most one subarray (SAUM) in a bank is undergoing a mitigation. The SAUM receives no demand activations while undergoing mitigation, so no row of SAUM is selected by MINT for mitigation in that window (we discuss how to handle transitive mitigation of SAUM in the next section). Therefore, the SAUM is guaranteed to become free to service demand request during the next window.

C. Changes to the Memory Controller

AutoRFM requires only minor support from the memory controller (MC). First, it needs the MC to have the ability to respond to the ALERT signal issued by the DRAM chip (in response to a conflicted ACT). This means that the ACT must be considered as failed, so the MC must continue to infer that there is no open row in the corresponding bank. Second, the MC needs the ability to retry a failed ACT request. We propose a simple change to the MC to facilitate both, as shown in Figure 7. For each bank, the MC keeps a busy bit and a timestamp when the bank will become free. When an ACT fails, the corresponding bank is marked as busy, and the timestamp is set to current time plus t_M. When the current time exceeds the timestamp of the busy bank, the busy bit is reset. A bank with a busy bit is prevented from sending any demand requests to the DRAM chip.

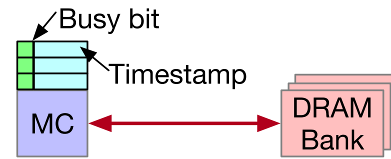

Fig. 7. Changes to the MC. MC keeps a busy-bit and timestamp (when the busy bank becomes free) on a per-bank basis. (Figure not to scale)

We note that while a bank encounters a failed ACT, it is theoretically possible for the bank to service demand operations from subarrays other than the SAUM. This would require a busy-bit and timestamp for each entry in the request queue of the memory controller. Our simple design avoids the complexity of maintaining per-entry metadata and timestamps. If conflicts are infrequent (as we have a large number of subarrays per bank), the simple design provides similar performance compared to a more complex design.

D. Changes to the DRAM Chip and Interface

AutoRFM also requires minor changes to the DRAM chip and interface. We note that the subarrays leveraged for AutoRFM are already present in the current DRAM chips, so no change is required to incorporate or use the subarrays.

The DRAM chip needs two changes. First, it needs the ability to internally perform mitigation on the SAUM for the row specified by MINT. Performing a mitigation requires doing a total of four victim refreshes (two rows on each side of the aggressor row). We note that the DRAM chips already have circuitry to perform victim refreshes (to support in-DRAM trackers and to support RFM commands). Second, the DRAM chip needs to store the address of the SAUM, and compare the subarray of the incoming ACT request to the SAUM. If the results match, then the DRAM chips skips doing the ACT operation and informs the MC of the failed ACT.

AutoRFM also requires support from the DRAM interface to allow the DRAM chips to inform the MC that an ACT has failed. To facilitate this, we reuse the ALERT signal (similar to recent DDR5 specifications [14] that reuse ALERT to convey that DRAM needs more time to perform mitigation). An ALERT in response to ACT indicates the ACT has failed.

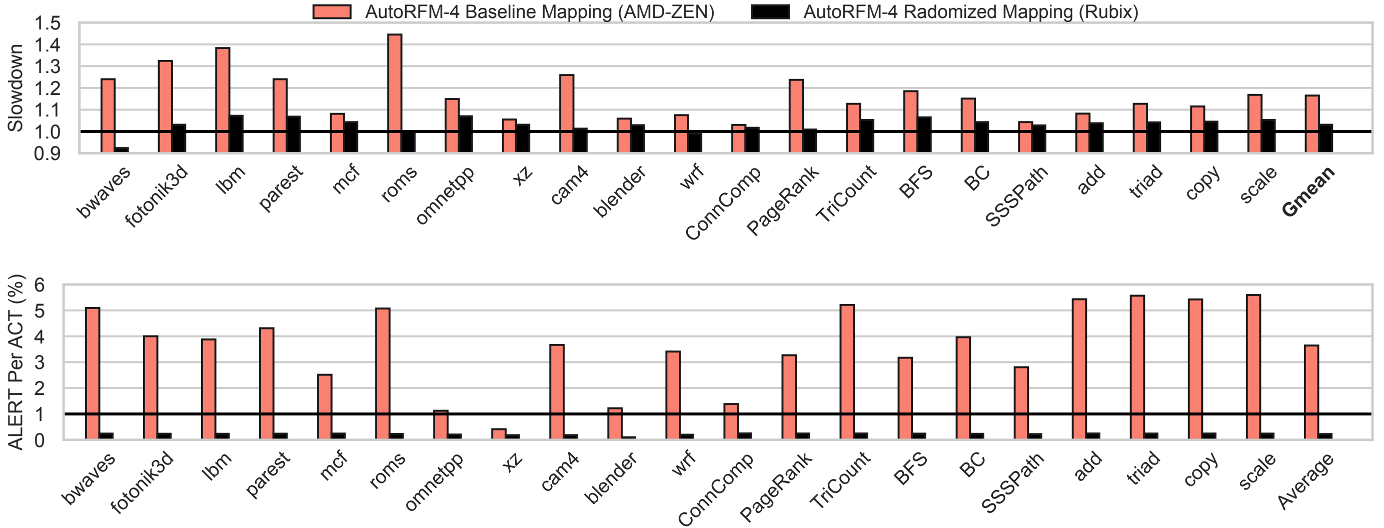

Fig. 8. Impact of memory mapping policy on AutoRFM-4 (a) Slowdown and (b) Likelihood of ALERT per ACT (due to conflict with SAUM), with the baseline mapping (AMD-ZEN) and Randomized Mapping (Rubix). Baseline mapping has a high rate of ALERT (3.7%, on average) and has a significant slowdown (16.5% on average). Randomized mapping reduces the ALERT probability to 0.2% and has a significantly lower slowdown (3.1% on average).

E. Impact of Memory Mapping

With 256 subarrays in a bank, we would expect the likelihood of an ACT to conflict with SAUM to be small. However, the likelihood of a conflict is strongly dependent on the memory mapping, which decides not only the line-to-bank mapping but also the set of lines that are co-resident in a given row. If a mapping places an entire 4KB page in a row, then the likelihood of consecutive requests mapping to the same row (and hence the same sub-array) is significant, and therefore the likelihood of conflict also becomes significant.

Our baseline mapping (AMD-Zen) places two lines from a 4KB page within the same row and spreads the 4KB page over 32 banks to boost the bank-level parallelism. Therefore, it is possible for two requests from the same page, occurring in a short time of each other, to cause two ACTs and the second ACT request to get directed to the SAUM (caused by mitigation for the first request). Therefore, the second request will have a conflict, trigger ALERT, and cause slowdown.

Figure 8(a) shows the slowdown with AutoRFM-4 (AutoRFMTH is set to 4), with the baseline memory mapping. We observe that AutoRFM-4 incurs significant slowdowns: more than 40% for `roms`, and more than 30% for `fotonik3d` and `lbm`. On average, with the baseline memory mapping, AutoRFM-4 incurs a slowdown of 16.5% on all 21 workloads.

The key reason for the high slowdown of AutoRFM-4 is the high rate of conflicts under the baseline mapping. Figure 8(b) shows the ALERT per ACT (as an ACT can be declined only once, this metric indicates the probability of ALERT) under the baseline memory mapping. Ideally, with 256 subarrays, we would expect a conflict probability of less than 1%. However, with the baseline mapping, we observe a significantly higher rate of conflicts: 7 workloads have more than 5% probability of conflict. On average, across the 21 workloads, the probability that an ACT can get an ALERT is 3.7%.

F. Reducing Conflicts with Randomized Memory Mapping

The slowdown of AutoRFM can be reduced with a memory mapping that decreases the likelihood of conflicts with SAUM. Our key insight is to use randomization to break the spatial correlation between the line-to-subarray mapping. With randomization, the likelihood that a given line maps to a particular subarray within the bank will be 1/256 regardless of the spatial location within the access stream. To design a practical *Randomized Memory Mapping*, we leverage the recent proposal called *Rubix* [42].

Rubix randomizes the line-to-row mapping using the low-latency K-cipher [24] (latency of 3 cycles). The memory controller converts a given *line-address* into an *encrypted-line-address* and uses the *encrypted-line-address* to access memory. With address encryption, we break any spatial correlation between the given line-address and banks, subarrays, and rows.

While Randomized Memory Mapping sacrifices row-buffer locality, it increases the bank-level parallelism as the access stream gets spread (almost uniformly) across all the banks in the system. The increased bank-level parallelism can offset the performance lost due to reduced row-buffer hits.

Figure 8(a) shows the slowdown with AutoRFM-4, with randomized mapping. The average slowdown gets reduced to 3.1%. This occurs mainly due to the significant reduction in the probability of conflict for ACT. The slight speedup for `bwaves` is because of the higher bank-level parallelism due to randomized mapping. Figure 8(b) shows the ALERT per ACTs. On average, our mapping reduces the ALERT per ACT to 0.22% (16x reduction compared to AMD-Zen mapping).[2] We note that out of the 3.1% slowdown, 1.5% slowdown is due to randomized mapping and 1.6% is due to conflicts.

[2]We note that this is lower than 1/256 (0.4%), as our workloads do not use all the activation slots. If a bank has only 50% of the activation slots used, then for 50% activations, there is no SAUM. So, the conflicts become 0.2%.

V. Fractal Mitigation for Transitive Attacks

Thus far, we have assumed that the mitigative action (of refreshing 2 victim rows on either side of an aggressor row) is sufficient to ensure security. Unfortunately, at low thresholds, the mitigative action of victim refresh can itself be used to cause Rowhammer attacks. In this section, we first describe the impact of such *Transitive Attacks* and the current means of mitigating them via *Recursive Mitigation*. We then describe the pitfalls of recursive mitigation and propose *Fractal Mitigation* to overcome those shortcomings.

A. Transitive Attacks and Impact

Victim refresh replenishes the charge on the neighboring victim rows. At low thresholds, the act of victim refresh can itself be used to perform a Rowhammer attack on a distant row (for example, Half-Double [23]). Such indirect attacks are called *Transitive Attacks* [48]. Figure 9 (a) shows an example of a transitive attack. Row-E is subjected to a lot of activations, which causes victim-refresh on Row-C, Row-D, Row-F, and Row-G. The activations caused by these victim refreshes can be enough to cause errors in Row B and Row H. This becomes a severe problem at low Rowhammer thresholds (sub-1000) as it requires frequent victim-refresh operations.

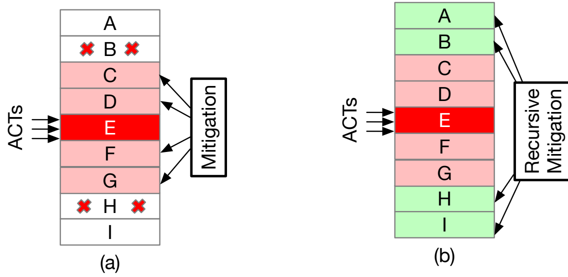

Fig. 9. (a) Transitive Attack (b) Defense with Recursive Mitigation.

B. Recursive Mitigation Defense and Pitfalls

To handle transitive attacks, MINT uses recursive mitigation, where the victim-rows themselves have the possibility of triggering a subsequent mitigation. Figure 9(b) shows an example of transitive mitigation where victim-refreshes of C, D, F, and G trigger (with some probability) a *level-2* mitigation, which performs victim refreshes for A, B, H and I. The concept is applied recursively, where level-2 can trigger a level-3 mitigation (with some probability), and so on.

For a window of N activations, MINT probabilistically selects from N+1 slots, with 1 slot reserved for recursive mitigation (by increasing the mitigation level). Thus, with N=4, MINT selects each demand activation with only 20% (and not 25%) probability. The lower probability of selection increases the threshold tolerated by MINT.

Another important impact of recursive mitigation is that it can keep the same subarray busy for several consecutive rounds (e.g. 10 rounds every second). Thus, the time SAUM is busy can become non-deterministic (200ns to 2000ns), which can cause repeated failures for a given ACT. Ideally, we want AutoRFM to have constant latency overheads to avoid slowdowns and alleviate any denial-of-service concerns.

C. Our Proposal: Fractal Mitigation

To avoid recursive mitigation, we propose *Fractal Mitigation (FM)*. The key insight with FM is to use the mitigative refreshes probabilistically. Two of the four victim refreshes always refresh the immediate neighbors, which we refer to as the distance 1 (d=1) neighbor. The remaining two victim refreshes are directed to distant neighbors, where each distance "d" neighbor has a probability $2^{(1-d)}$ of getting refreshed. So distance 2 neighbors are refreshed with probability 1/2, d=3 with 1/4, d=4 with 1/8, etc., as shown in Figure 10(a).

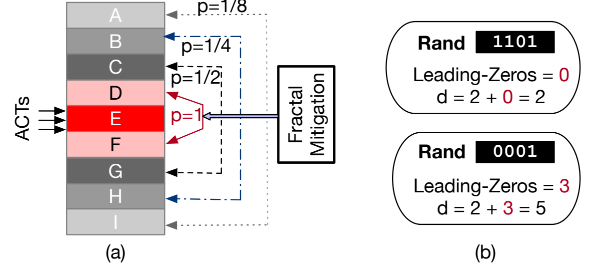

Fig. 10. Fractal Mitigation (a) Overview, d=1 is always refreshed, and the other pair is selected probabilistically (b) Implementation using a Random Number (Rand), the distance (d) to refresh equals 2 + leading-zeros in Rand.

To implement FM in a simple manner, we leverage the insight that the number of leading zeros in a random number (Rand) follows an exponentially reducing distribution. So, no leading zeros occur 50% of the time, one leading zero occurs 25% of the time, and so on. We use a 16-bit random number, as the likelihood of d=18 getting refreshed is negligibly small (even if the same row is activated continuously, d=18 will receive less than 1 victim refresh per 32ms). With FM, we can handle transitive attacks while ensuring that there are only 4 victim refreshes per mitigation without requiring subsequent mitigative actions (specifically for transitive attacks).[3]

D. Security of Fractal Mitigation

An attacker could use the mitigative refreshes generated by FM to cause unmitigated activations on distant neighbor rows. In Appendix B, we analyze the security of FM under such attacks. Our model shows that such FM attacks become viable only at thresholds of TRH-D below 53. Given that AutoRFM achieves a minimum TRH-D of 74, direct attacks to the row are the most potent way to cause Rowhammer, even with FM.

E. Results: Impact on Threshold and Performance

Fractal Mitigation offers two key advantages. First, it avoids recursive mitigations on the same subarray, thus allowing AutoRFM to have deterministic latency. Any ACT that gets declined will find the SAUM free for service after the mitigation time (200ns). Second, it allows MINT to choose from N slots instead of (N+1); therefore, it has a higher probability of selection and can tolerate a lower Rowhammer threshold.

[3] We observe that, with FM, the d=2 neighbors get refreshed with 50% probability, which is different from the baseline mitigation that always refreshes both d=1 and d=2. Recent characterization [26] shows that the neighbor at d = 2 suffers less than 10% charge loss compared to the neighbor d = 1, so refreshing both of them with the same probability is wasteful. Instead, FM uses the 2 victim refreshes judiciously over more distant neighbors.

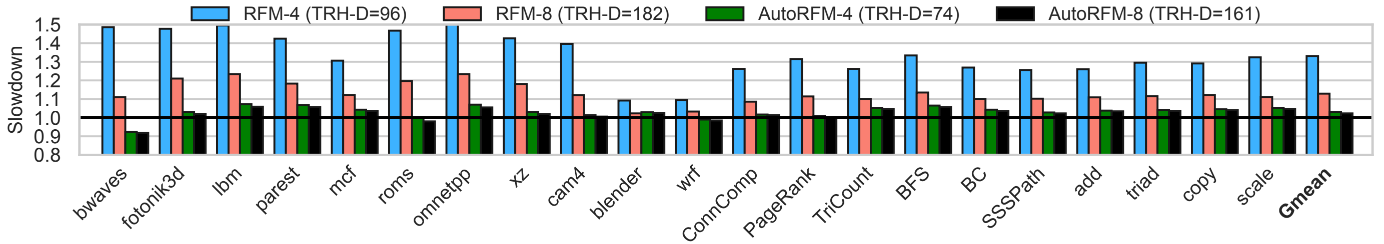

Fig. 11. Performance Impact of RFM and AutoRFM. The average slowdown of RFM-4 and RFM-8 is 33% and 12.9%, respectively. Whereas, with AutoRFM, it reduces to 3.1% and 2.3% respectively. We note that AutoRFM uses both Randomized Mapping and Fractal Mitigation.

Table VI shows the performance and the tolerated threshold (TRH-D) with recursive mitigation (RM) and fractal mitigation (FM) as the AutoRFM threshold is varied. FM achieves a lower threshold at the same performance overhead as RM, or a lower slowdown for a given TRH-D. With AutoRFMTH of 4 (the minimum possible value for our design), FM achieves a TRH-D of 74 instead of 96 with RM. Thus, our design can tolerate thresholds as low as 74 while having a low slowdown.

TABLE VI
SLOWDOWN AND TOLERATED THRESHOLD (TRH-D) FOR RECURSIVE-MITIGATION AND FRACTAL MITIGATION.

AutoRFMTH	Slowdown	Recursive Mit. TRH-D	Fractal Mit. TRH-D
4	3.1%	96	**74**
5	2.8%	117	96
6	2.7%	139	117
8	2.3%	182	161

VI. RESULTS AND ANALYSIS

A. Comparison with RFM

Both RFM and AutoRFM provide the DRAM chips with the time to perform mitigation. RFM does so in a blocking manner, as the bank performing RFM is prohibited from performing any other operations for a given time period (tRFM). With AutoRFM, the mitigations become non-blocking, as only the subarray under mitigation (SAUM) is kept busy, and all other subarrrays can still service memory requests. Figure 11 shows the slowdown with RFM and AutoRFM as the RFMTH and AutoRFMTH changes from 4 to 8.

RFM incurs significant slowdowns for RFM-4. Several workloads incur almost 50% slowdown. On average, RFM-4 incurs a 33% slowdown, which makes RFM unappealing for use in the sub-100 threshold regime. However, AutoRFM-4 has an average slowdown of only 3%. As thresholds increase, the gap between RFM and AutoRFM reduces. For example, the slowdown of RFM-8 is 13%, while for AutoRFM-8 it is 2.3%. We note that the speedup of `bwaves` is because AutoRFM uses Randomized Memory mapping, which increases bank-level parallelism compared to the AMD-Zen mapping.

We also note that for the same value of RFMTH or AutoRFMTH, the AutoRFMTH scheme has a lower tolerable threshold. This is because AutoRFM uses Fractal Mitigation. We note that our idea of Fractal Mitigation can also be applied to designs that use RFM to get reduced thresholds.

B. Power and Energy Overheads

AutoRFM incurs power overhead for two reasons. First, additional activations due to the randomized mapping (18% higher than baseline). Second, the mitigative refreshes for rowhammer mitigation. To compute DRAM power, we use the publicly available DRAM-power model provided by Micron [32]. We configure it to have the parameters based on the DDR5. Figure 12 shows the DRAM power for baseline, Rubix, AutoRFM-8, and AutoRFM-4. We break the power into four components: (a) Activations and Read/Write operations, (b) Other, indicating standby and termination, (c) Refresh, and (d) Mitig, indicating Rowhammer Mitigation. We assume the baseline and Rubix (standalone) do not do any mitigations.

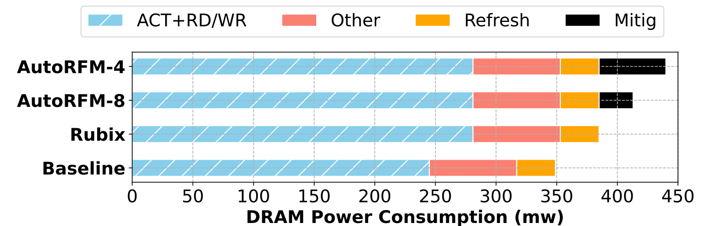

Fig. 12. DRAM power for baseline, Rubix, AutoRFM-8, and AutoRFM-4. Rubix incurs extra actiations, and AutoRFM incurs mitigations. On average, AutoRFM-8 and AutoRFM-4 increase average power by 65mW and 92mW.

Standalone Rubix incurs a power overhead of 36 mW (due to extra activations). Additionally, the mitigations required for AutoRFM-8 and AutoRFM-4 incur an overhead of 28mW and 55mW, respectively. Overall, AutoRFM incurs an additional power overhead of 65mW-92mW. If DRAM contributes 10% to the total system power, AutoRFM would increase the overall system power by 1.25%-2.5%. We also note that if the system is idle, AutoRFM does not incur any power overheads, so AutoRFM also has a nice property of energy proportionality.

C. Storage Overheads

AutoRFM incurs small storage overhead, both at the memory controller and within the DRAM chips. At the memory controller, we need a busy bit and a 15-bit timestamp (a total of 2 bytes) for each bank. With 64 banks, the memory controller needs **128 bytes of SRAM**. Each DRAM bank requires storage to identify SAUM (1 valid bit + 8 bits) and the MINT tracker (4 bytes), so a total of **5 bytes per bank**. The DRAM chip also needs a PRNG for random numbers.

VII. Related Work

A. *PRAC and ABO*

The inability of the DRAM industry to solve Rowhammer securely and efficiently has led to revised DDR5 specifications [14], which includes two optional features: *Per-Row Activation-Counting (PRAC)* and *Alert Back-Off (ABO)*. PRAC solves the space issue with Rowhammer tracking by redesigning the DRAM array to keep a per-row activation counter, which gets incremented on each activation to the row (the DRAM timings are increased to do the read-modify-write of the counter). ABO solves the time issue with Rowhammer tracking by allowing the DRAM chip to assert the ALERT signal, if it requires time to perform mitigation. PRAC+ABO represents not only the biggest changes to DRAM arrays and interface but also a principled means to tolerate Rowhammer.

Unfortunately, PRAC incurs both significant area overheads and performance overheads. The recent Hynix design [20] reports that the per-row counters incur significant area overheads. As PRAC increases the DRAM timings (**tRP increases by almost 150%**), it also incurs significant performance overheads, even at thresholds where an alternative low-cost tracking would incur negligible overheads.

ABO specifications permit a few activations before stalling for ALERT (in order to limit denial-of-service attacks); however, an attacker can use these inter-ALERT activations to cause significantly more activations on an attack row compared to the ALERT thresholds. Recent works [34], [36] show that such attacks can cause 20-30 additional activations, so PRAC+ABO is viable only for thresholds of greater than 50.

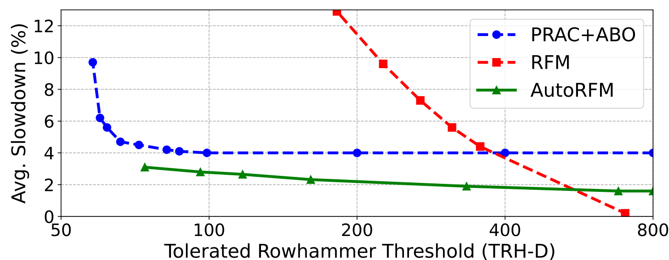

Fig. 13. Average slowdown of PRAC, RFM, and AutoRFM with varying thresholds. PRAC incurs a slowdown of at least 4% regardless of threshold due to increased tRC. RFM incurs significant slowdowns for thresholds below 300. AutoRFM scales to sub-100 thresholds with slowdown lower than PRAC.

Figure 13 shows the average slowdown for PRAC+ABO (implemented with MOAT [36]), RFM, and AutoRFM, as the tolerated threshold is varied. For PRAC+ABO, even at high thresholds, there is a 4% slowdown, due to longer DRAM timings to update counters. Therefore, PRAC not only incurs significant area overheads but also has non-negligible slowdowns, even for the current thresholds. RFM incurs negligible slowdown at thresholds of 700 and higher. However, RFM overheads increase significantly as the thresholds reduce, incurring 13% at a threshold of 180. The slowdown of AutoRFM is 2% at near-term thresholds (200-800) and increases to 3.1% at ultra-low thresholds (74). Thus, AutoRFM offers a low overhead option of scaling to low Rowhammer thresholds.

B. *Self Managed DRAM (SMD)*

As DRAM is a deterministic device, a scheduled operation (read, write, ACT) must finish within the predefined time. This requirement means that if DRAM chips need to perform internal maintenance operations (such as refresh or scrub or Rowhammer mitigation), such operations must not interfere with the commands from the memory controller. Thus, maintenance operations require support from JEDEC specifications, which hinders the adoption of new maintenance operations. A recent work, *Self-Managed DRAM (SMD)* [8], eases the adoption of maintenance operations by splitting the memory into regions and performing maintenance on at-most one region. If an incoming ACT conflicts with the region under maintenance, it is declined using a `ACT_NACK` signal. SMD analyzed three maintenance operations: Refresh, Rowhammer mitigation, and Scrubbing. While AutoRFM framework is similar to SMD, AutoRFM is designed explicitly for Rowhammer mitigation at ultra-low (sub-100) thresholds and solves two critical bottlenecks necessary to enable such a framework to tolerate ultra-low Rowhammer thresholds efficiently.

1. Breaking Spatial Correlation: At ultra-low Rowhammer threshold (sub 100), mitigation is frequent. For example, on average, every 4th activation to a bank triggers a mitigation. So, accesses that occur in a short succession of each other must not get mapped to the same row/subarray. Otherwise, they will get declined due to a conflict with mitigation and suffer slowdown. SMD evaluated PARA with p=1/5 (per Section 9.8 of [8]) and reports 11.3% slowdown, indicating that SMD incurs high overheads for tolerating ultra-low Rowhammer threshold. This is similar to our analysis (see Figure 8), which shows that with conventional mapping, AutoRFM causes a slowdown of 16.5% at p=1/4 (MINT window of 4). This slowdown occurs mainly because accesses are spatially correlated and go to similar rows/subarray. AutoRFM breaks the spatial correlation in accesses via Rubix-based randomized mapping, reducing the slowdown to 3.1%. Our work demonstrates that randomized mapping is vital for mitigating low thresholds.

2. Tolerating Transitive-Attacks: Transitive attacks are a significant vulnerability at ultra-low threshold. For example, at TRH=100 and p=1/4 (row selected for mitigation with 25% probability), inflicting just 400 activations on a row triggers 100 mitigations, which are enough to cause transitive failures. SMD does not discuss how to tolerate transitive attacks. While SMD analyzes the impact of the different blast radius, a higher blast radius is insufficient to tolerate transitive attacks, as the row immediately after the blast radius experiences bit-flips.

We note that, treating rows undergoing victim refreshes as also eligible for PARA is not a viable way to tolerate transitive attacks at ultra-low thresholds. For example, if PARA had p=1/4 and blast-radius of 2, then, after mitigation, PARA will select one victim row for transitive mitigation, which in turn will trigger one more row for transitive mitigation, and so on. Thus, the number of transitive victim refreshes keeps growing.

Our proposed *Fractal Mitigation* protects AutoRFM from transitive attack and incurs deterministic latency overheads.

C. Using RFM for Efficient Mitigation

RFM is a primary means of providing the in-DRAM tracker to perform frequent mitigations. Prior research has used RFM to reduce the storage overhead of tracking and scaling probabilistic trackers to lower thresholds. For example, both Mithril [18] and ProTRR [29] use RFM to tolerate low thresholds or to reduce the number of counters in the tracking structures or both. PARFM [18], PrIDE [11], and MINT [37] use RFM to provide more frequent mitigations (compared to REF) and enable the tracker to tolerate low thresholds. For example, PrIDE uses RFM-16 to obtain a TRH-D of 400.

D. Rowhammer Tracking and Mitigation

Several proposals reduce the SRAM overhead of aggressor-row tracking. Example of such works includes CAT [47], TWiCE [27], and Graphene [35]. START [43] uses the last-level cache for tracking but can cause significant LLC capacity loss. Examples of low-cost in-DRAM trackers that were broken include TRR and DSAC. As our objective is secure mitigation, we do not consider such trackers in our study. CRA [16] and Hydra [38] keep the counters in DRAM and use caches or filters to reduce the DRAM lookups for counters, however, they can still cause significant slowdowns.

Our design uses victim refresh for mitigation. RRS [41], AQUA [45], SRS [51], SHADOW [50] perform mitigation with row migration, whereas, Blockhammer [52] uses rate limits. REGA [30] changes the DRAM module to provide a mitigating refresh on each demand activation. However, it does not scale to sub-100 thresholds. HiRA [53] changes the interface to allow multiple activations per bank.

E. ECC to Tolerate Rowhammer

Recent work has used ECC to tolerate Rowhammer failures. For example, SafeGuard [4], CSI-RH [15], PT-Guard [44], and Cube [19] modify the ECC codes to correct Rowhammer failures. Unfortunately, with such solutions, uncorrectable failures can still occur, leading to data loss.

VIII. Conclusion

Although low-cost in-DRAM trackers can probabilistically identify aggressor rows, they require frequent mitigation. *Refresh Management (RFM)* can provide DRAM chips with the time required for mitigation; however, it incurs significant overhead at low thresholds (33% slowdown for a threshold of 100). In this paper, we propose *AutoRFM*, a transparent RFM design that uses the intra-bank subarrays to perform mitigation without requiring the bank to be stalled. We show that AutoRFM, when combined with *Randomized Mapping* and *Fractal Mitigation*, can tolerate thresholds as low as 74 while incurring a slowdown of only 3.1%. We also show that AutoRFM provides a more cost-effective alternative for scaling to low thresholds than the recent PRAC+ABO proposal.

Acknowledgments

We thank Salman Qazi, Kuljit Bains, Sudhanva Gurumurthi, Yoongu Kim, Kaveh Razavi and Michele Marazzi for feedback. This research was supported by NSF grant 2333049.

Appendix-A: Analytical Model for MINT+RFM

Consider MINT with a window of W activations. For MINT, the best attack pattern contains unique W rows activated continuously in a circular fashion. For W=4, the access pattern can be represented as $(ABCD)^K$ for K repeats.

Consider Row-A. The probability that this row gets selected for mitigation in one iteration is p=1/W. Let the single-sided Rowhammer threshold (TRH-S) be T, then the probability that the row is not selected in T iterations is given by Equation 1.

$$P_T = (1 - 1/W)^T \tag{1}$$

Given the selection probability is 1/W, the average number of iterations for receiving a mitigation for Row A is W. In every mitigation, we start a new *epoch* for Row A, and the probability of failure for that round is P_T. Let t_M be the mitigation time (RFM latency, 200 ns in our case), then the total epoch time (t_E) is given by Equation 2.

$$t_E = W^2 \cdot tRC + t_M \tag{2}$$

Each epoch has failure probability of P_T. Equation 3 shows the failure rate (FRate) per unit time for attacking one row.

$$FRate = \frac{P_T}{t_E} = \frac{(1 - 1/W)^T}{W^2 \cdot tRC + t_M} \tag{3}$$

If we attack all W rows in the window, the overall failure rate ($FRate_W$) across all rows is given by Equation 4.

$$FRate_W = W \cdot \frac{(1 - 1/W)^T}{W^2 \cdot tRC + t_M} \tag{4}$$

Mean Time to Failure (MTTF) is the inverse of the failure rate. Equation 5 shows the MTTF.

$$MTTF = \frac{W \cdot tRC + t_M/W}{(1 - 1/W)^T} \tag{5}$$

Rearranging Equation 5, we determine the threshold (T) and TRH-D as shown in Equation 6 and Equation 7.

$$T = ln(\frac{W \cdot tRC + t_M/W}{MTTF})/ln(1 - 1/W) \tag{6}$$

$$TRH_D = T/2 = ln(\frac{W \cdot tRC + t_M/W}{MTTF})/2 \cdot ln(1 - 1/W) \tag{7}$$

Figure 14 shows the TRH-D tolerated by MINT in varying window sizes (W) for Recursive and Fractal Mitigation.

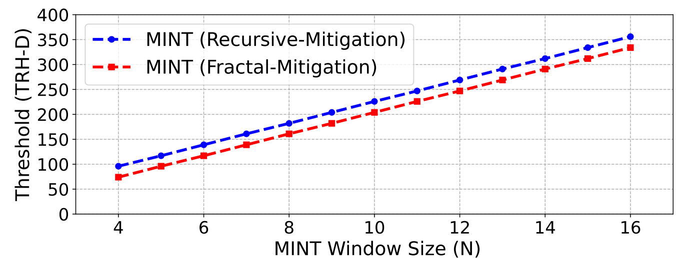

Fig. 14. Threshold tolerated by MINT versus window size

Appendix-B: Security of Fractal Mitigation

An adversary can do continuous activations on an aggressor row and try to use the victim refreshes from Fractal Mitigation (FM) to attack. In this section, we model such an attack.

Let there be N episodes of FM due to mitigation of an aggressor row. Consider a row R that is a distant neighbor of the aggressor row. This row has two neighbors, $R-$ and $R+$, as shown in Figure 15. $R-$, R, and, $R+$ receive mitigative refreshes due to to FM with probabilities p, $p/2$, and $p/4$, respectively. To cause bit flips in R, we want to maximize the activations in R+ and R- (total *Damage*) while ensuring a high probability that R receives no activations (escape).

Fig. 15. Attack R by maximizing ACTs on R+ and R- while no ACT on R.

The amount of *Damage* on R is given by Equation 8.

$$Damage_N = p \cdot N + 0.25 \cdot p \cdot N = 1.25 \cdot p \cdot N \quad (8)$$

The escape probability (P_{escape}) is given by Equation 9.

$$P_{escape} = (1 - p/2)^N \approx e^{-pN/2} \approx e^{-Damage/2.5} \quad (9)$$

For our target-MTTF of 10K years, the escape probability is 10^{-18}, so the maximum Damage is given by Equation 10. Thus, FM is safe for systems that have a TRH-D $\geq$ 53.

$$e^{-Damage/2.5} = 10^{-18} \Rightarrow Damage = 104 \Rightarrow TRH_D = 52 \quad (10)$$

Mixed Attacks: An attacker could try to combine the damage on an attack row R using both *direct* activations on neighbors and the *indirect* damage using FM. However, this results in a less effective attack than simply using direct activations. We note that the escape probability for damage due to direct activations on neighbors with MINT is given by $(1-1/W)^{Damage}$, where W is the MINT window. Figure 16 shows the escape probability for both FM and MINT-4 as the damage count is varied. The TRH-D with FM is 52, and with MINT is 74. If the attacker combines the two patterns, say 40 activations from FM (point X) and 80 activations from MINT (point Y), to cause 120 total activations, the total probability of escape will be $10^{-7} \times 10^{-10}$, so 10^{-17}, which is 100x lower than the 10^{-15} with MINT alone (point Z). Thus, even in the presence of FM, direct attacks on neighboring rows are still the fastest and most viable way to cause Rowhammer. Thus, such attacks do not impact the threshold for MINT designs with TRH-D $\geq$ 53 (so, both MINT-4 and MINT-8 are secure).

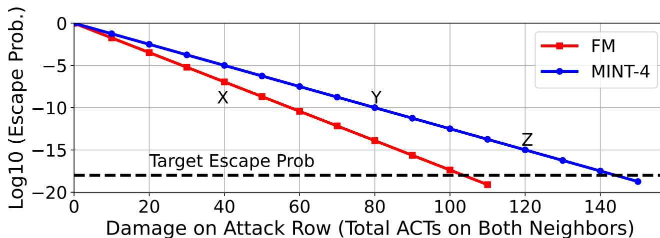

Fig. 16. Escape probability as a function of Damage (for MINT and FM)

Appendix-C: Impact of Rubix on RFM

As Rubix-based memory address-space randomization is highly effective at reducing the slowdown for AutoRFM, it would be reasonable to think that it can also help reduce the overheads of RFM. Figure 17 shows the average slowdown from RFM on a system with AMD-Zen mapping and with Rubix mapping, both normalized to the respective baseline system without RFM. On average, RFM incurs higher overheads on the Rubix system (e.g. 35.1% vs. 33.1% for RFM-4).

This may seem counter-intuitive, but it is expected behavior. Rubix spreads the activations over a larger number of rows (so it reduces the variance of ACT per row), but overall it increases activations to the bank (so it increases the mean ACTs/row). RFM is issued by the bank by counting the number of ACTs to the bank, so, with more activations per bank due to Rubix, we get more RFMs and a higher slowdown for Rubix+RFM.

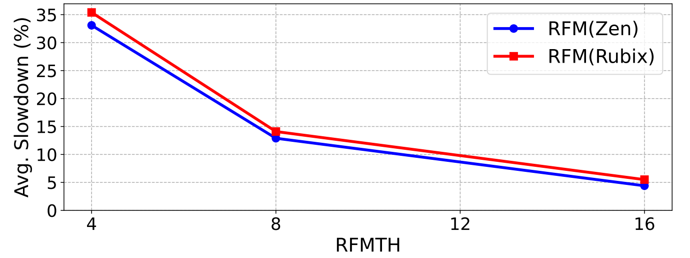

Fig. 17. Impact of RFM on Rubix and Zen. Rubix has higher slowdowns.

Appendix-D: AutoRFM with Other Trackers

Similar to RFM, AutoRFM is a generalized solution that can work with any DRAM tracker (randomized or deterministic). We briefly discuss AutoRFM with PrIDE [11] and Mithril [18].

PrIDE: We sample with probability 1/4 (1/8) for AutoRFM-4 (8) and insert the entry in a FIFO buffer. Once every 4 (8) ACTs to the bank, we mitigate one entry from the FIFO buffer.

Mithril: As Mithril is counter-based, the tracking remains unchanged. For AutoRFM-4 (8), once every 4(8) activations to the bank, we mitigate the row with the highest count.

Similar to RFM, the slowdown of AutoRFM is not dependent on the in-DRAM tracker and is dedicated only by AutoRFMTH. Figure 18 shows the TRHD tolerated by PriDE, MINT, and Mithril. With AutoRFMTH-4 all three trackers can tolerate sub-125 TRHD. Mithril needs $>$ 30K entries/bank. MINT has a lower threshold and storage than PrIDE.

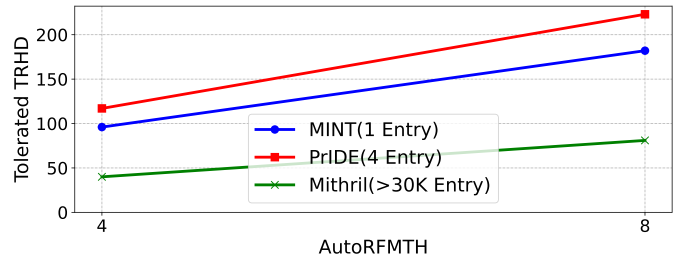

Fig. 18. The TRHD tolerated by PrIDE, MINT, and Mithril using AutoRFM.

References

[1] "Memsim: Memory System Simulator," 2024. [Online]. Available: https://github.com/mqureshi4/memsim

[2] 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," *ASPLOS*, vol. 51, no. 4, pp. 743–755, 2016.

[3] 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.

[4] 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.

[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] 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.

[8] H. Hassan, A. Olgun, A. G. Yaglikci, H. Luo, and O. Mutlu, "Self-Managing DRAM: A Low-Cost Framework for Enabling Autonomous and Efficient DRAM Maintenance Operations," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[9] 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.

[10] 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.

[11] A. Jaleel, G. Saileshwar, S. Keckler, and M. Qureshi, "PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers," in *Annual International Symposium on Computer Architecture*, 2024.

[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] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms," in *USENIX Security*, Aug. 2024, https://comsec.ethz.ch/wp-content/files/zenhammer_sec24.pdf.

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

[15] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, 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.

[16] 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.

[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] 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.

[19] M. J. Kim, M. Wi, J. Park, S. Ko, J. Choi, H. Nam, N. S. Kim, J. H. Ahn, and E. Lee, "How to kill the second bird with one ecc: The pursuit of row hammer resilient dram," in *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[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] 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.

[23] 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.

[24] M. Kounavis, S. Deutsch, S. Ghosh, and D. Durham, "K-cipher: A low latency, bit length parameterizable cipher," in *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020, pp. 1–7.

[25] 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.

[26] Z. Lang, P. Jattke, M. Marazzi, and K. Razavi, "Blaster: Characterizing the blast radius of rowhammer," in *Worshop on DRAM Security (DRAMSec)*, 2023.

[27] 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.

[28] 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 *ISCA*, 2023.

[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 Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

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

[33] H. Nam, S. Baek, M. Wi, M. J. Kim, J. Park, C. Song, N. S. Kim, and J. H. Ahn, "Dramscope: Uncovering dram microarchitecture and characteristics by issuing memory commands," in *2024 IEEE/ACM International Symposium on Computer Architecture (ISCA)*. [Online]. Available: https://arxiv.org/abs/2405.02499

[34] G. F. d. O. J. A. O. O. E. O. M. Oğuzhan Canpolat, Giray Yaglikci, "Understanding the security benefits and overheads of emerging industry solutions to dram read disturbance," in *Worshop on DRAM Security (DRAMSec)*, 2024.

[35] 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.

[36] M. Qureshi and S. Qazi, "MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters," in *ASPLOS*, 2025.

[37] M. Qureshi, S. Qazi, and A. Jaleel, "MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker," in *2024 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024. [Online]. Available: https://arxiv.org/abs/2407.16038

[38] 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.

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

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

[41] 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.

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

[44] 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.

[45] 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.

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

[47] 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.

[48] L. C. Stefan Saroiu, Alec Wolman, "The price of secrecy: How hiding internal dram topologies hurts rowhammer defenses," in *Proceedings of International Reliability Physics Symposium (IRPS)*, 2022.

[49] 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

[50] 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.

[51] 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.

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

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

[54] 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.