ImPress: Securing DRAM Against Data-Disturbance Errors via Implicit Row-Press Mitigation

Anish Saxena
Georgia Tech
anish.saxena@cc.gatech.edu

Aamer Jaleel
NVIDIA
ajaleel@nvidia.com

Moinuddin Qureshi
Georgia Tech
moin@gatech.edu

***Abstract*—DRAM cells are susceptible to *Data-Disturbance Errors (DDE)*, which can be exploited by an attacker to compromise system security. *Rowhammer* is a well-known DDE vulnerability that occurs when a row is repeatedly activated. Rowhammer can be mitigated by tracking aggressor rows *inside DRAM (in-DRAM)* or at the *Memory Controller (MC)*. *Row-Press (RP)* is a new DDE vulnerability that occurs when a row is kept open for a long time. RP significantly reduces the number of activations required to induce an error, thus breaking existing RH solutions.**

Prior work on *Explicit* Row-Press mitigation, *ExPress*, requires the memory controller to limit the maximum row-open-time, and redesign existing Rowhammer solutions with reduced Rowhammer threshold. Unfortunately, ExPress incurs significant performance and storage overheads, and being a memory controller-based solution, it is incompatible with in-DRAM trackers.

In this paper, we propose *Implicit Row-Press mitigation (ImPress)*, which does not restrict row-open-time, is compatible with memory controller-based and in-DRAM solutions and does not reduce the tolerated Rowhammer threshold. ImPress treats a row open for a specified time as *equivalent* to an activation. We design ImPress by developing a *Unified Charge-Loss Model*, which combines the net effect of both Rowhammer and Row-Press for arbitrary patterns. We analyze both controller-based (Graphene and PARA) and in-DRAM trackers (Mithril and MINT). We show that ImPress makes Rowhammer solutions resilient to Row-Press transparently without affecting the Rowhammer threshold.

I. Introduction

Relentless scaling over the last four decades has increased the capacity of DRAM chips from a few megabits to several tens of gigabits. As DRAM cells get smaller, they become prone to inter-cell interference, where the activity in one cell can disturb the data in another cell, leading to *Data-Disturbance Errors (DDE)*. DDEs are not only a reliability concern, but also a serious security threat, as attackers can exploit DDEs to compromise system security [9, 40].

Rowhammer: The most well-known DDE vulnerability in DRAM is *Rowhammer (RH)* [22]. Rowhammer occurs when an *aggressor* row is activated a large number of times, which causes bit-flips in the neighboring *victim* rows. Several studies [2, 4, 6, 8, 9, 40, 43] have shown that Rowhammer can be exploited to compromise security. For example, an attacker can flip bits in page tables to escalate privilege [40], flip bits in instruction opcode to bypass authentication [35], or analyze flipped bits to infer the data of nearby pages [23].

The number of activations (ACTs) to the aggressor row required to induce a bit-flip is called the *Rowhammer Threshold (TRH)*. The latest publicly available characterization data reports a TRH of 4.8K [18]. Typical hardware-based defenses for Rowhammer rely on a tracking mechanism [17, 17, 19, 22, 28, 32, 34, 42] to identify aggressors and refresh the victim rows [10]. The tracking can be either at the *Memory-Controller (MC)* or within the DRAM *(in-DRAM)*. Solutions for mitigating RH are designed for a *specific* TRH, which assumes DRAM will not incur bit-flips if the activation count is below the specified TRH. These solutions can be broken if a vulnerability causes bit flips with fewer than TRH activations.

Row-Press: A recent paper [26] discloses a new DDE vulnerability, *Row-Press (RP)*, which occurs when a row is kept open for a long time. While the row is open, the cells of the neighboring rows *slowly* leak charge on the bit-lines. The cumulative charge loss increases with time. Therefore, a Row-Press pattern keeps the row open for as long as possible. The row may eventually close due to a row conflict or refresh operation. Such a Row-Press attack pattern is repeated until the charge on the neighboring cell is depleted enough to cause a flip. Figure 1 (a) compares the pattern of RH and RP.

Impact of Row-Press: The efficacy of Row-Press depends on how long the row is kept open. Each round incurs an activation of the given row. Luo et al. [26] provide a detailed characterization of Row-Press and show that the number of activation rounds required to succeed is 18x to 160x lower compared to the number of activations required by a standalone RH attack. If the row is kept open for 30ms[1], then a single round of Row-Press attack may be enough to flip a bit.

Figure 1 (b) captures the impact of Row-Press on TRH. RP reduces the activations required to cause a bit-flip to much lower than TRH [26]. Thus, RP breaks RH mitigation designed to tolerate a threshold of TRH, as such solutions inherently assume that no bit-flip occurs if the row gets fewer than TRH activations. Therefore, RP is a serious security vulnerability.

Explicit Row-Press Management: Luo et al. [26] also proposed a design to tolerate Row-Press attacks, which forces the Memory Controller (MC) to limit the amount of time a row can be kept open to *Maximum Row Open time (tMRO)*. For example, let TRH denote the threshold for the standalone RH attack. The number of activations, $T*$, required for Row-Press to flip bits is characterized, with the maximum aggressor open time (tON) being limited to tMRO. The proposal redesigns the RH mitigation to operate at a lower threshold, T*, instead of TRH. We term this design *Explicit Row-Press (ExPress) Mitigation*. Figure 1(c) provides an overview of ExPress.

[1]Millisecond-scale Row-Press is not possible as per DDR specifications, which require performing refresh within a few tens of microseconds.

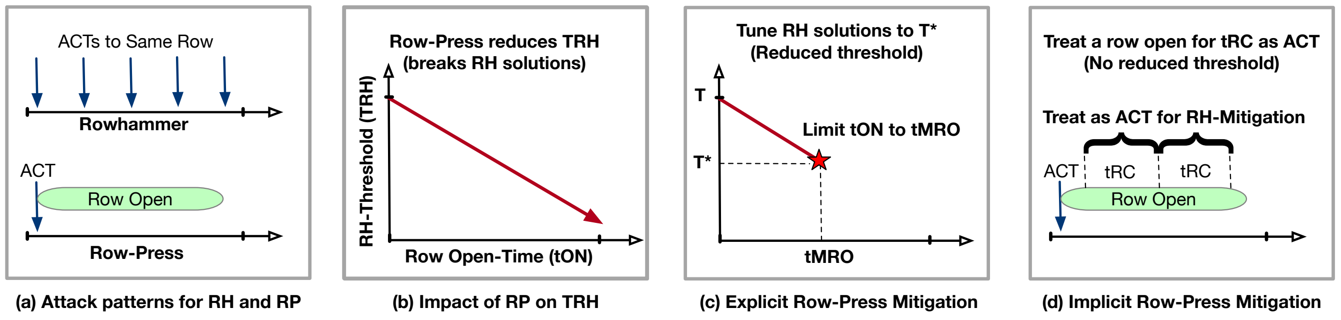
ACTs to Same Row
Rowhammer
ACT
Row Open
Row-Press
(a) Attack patterns for RH and RP
Row-Press reduces TRH (breaks RH solutions)
RH-Threshold (TRH)
Row Open-Time (tON)
(b) Impact of RP on TRH
Tune RH solutions to T* (Reduced threshold)
T
T*
Limit tON to tMRO
tMRO
(c) Explicit Row-Press Mitigation
Treat a row open for tRC as ACT (No reduced threshold)
Treat as ACT for RH-Mitigation
ACT
tRC
tRC
Row Open
(d) Implicit Row-Press Mitigation

Fig. 1. Towards practical Row-Press solution: (a) Attack pattern for Rowhammer and Row-Press (b) Impact of Row-Press on the Rowhammer Threshold (c) Explicit Row-Press (ExPress) [26] mitigation, which limits the aggressor row-open time (tON) to tMRO, reduces the tolerated RH threshold (d) Our proposal, Implicit Row-Press (ImPress) mitigation, treats a row-open for tRC as equivalent to an activation (ACT) for RH-mitigation, retains the same RH threshold.

Pitfalls of ExPress: The key shortcoming of ExPress is that it reduces the tolerated threshold from TRH to T*. Additionally, ExPress suffers from the following three problems:

(1) High performance overheads: Early row closure reduces the row buffer hits for workloads with good spatial locality. Furthermore, tuning the RH solution to a lower threshold (T*) increases the rate of mitigation and the associated penalty.

(2) High storage overheads: If the tracking mechanism is based on counters, the number of tracking entries increases due to the reduction in the threshold from TRH to T*.

(3) Incompatibility with in-DRAM Trackers: ExPress is a memory controller-based solution, as it must limit tON to tMRO. Therefore, it is incompatible with in-DRAM Rowhammer schemes that are unaware of the tMRO value unless the JEDEC specifications are revised to standardize tMRO.

Our Goal: The goal of our paper is a Row-Press solution which (i) does not place any limit on row-open time, (ii) does not affect the TRH tolerated by a Rowhammer solution, (iii) incurs low performance and storage overheads, and (iv) is applicable to both memory controller-based and in-DRAM RH solutions, without requiring changes to JEDEC specifications.

Our Insight: Secure RH-mitigation schemes are designed to handle the case when an activation occurs at every tRC (row-cycle time). If a row is open for a particular time period (say, tRC), then we treat it as *equivalent* to causing an activation, and the row participates in the RH mitigation. Doing so converts the Row-Press attack into an equivalent Rowhammer attack and lets the RH solution transparently handle RP without limiting the row open time.

Our Solution: We propose *Implicit Row-Press (ImPress) Mitigation*, as shown in Figure 1(d). To drive the design of ImPress, we first develop a *Unified Charge-Leakage Model* that combines the effect of both Rowhammer and Row-Press into a single metric. Both RP and RH *damage* the data in the cell by causing charge loss, albeit at a different rate. Our model normalizes the rate of damage caused by RP (per tRC) to the rate of damage caused by RH (per tRC). Our model estimates the combined damage caused by RH and RP for any pattern.

Our first design, *Impress-N (Naive)*, operates on integer values of damage to demonstrate the impact of imprecise damage estimation. Impress-N divides the time interval between refresh into windows of tRC. If an activation occurs in the given tRC window, that row participates in RH-tracking. If a row is open for the full tRC window, that row is treated as equivalent to causing activation and participates in RH-tracking. Impress-N limits the impact of unmitigated RP to, at most, one tRC. Impress-N can underestimate Row-Press activity, reducing the threshold (T*) by 1.35x-2x, which is identical to ExPress at the corresponding tON. While ImPress-N has a similar impact on threshold, performance, and storage as ExPress, it is compatible with in-DRAM tracking as it does not limit row open time.

Our optimized design, *Impress-P (Precise)*, operates on precise values of damage, including non-integer values. ImPress-P dynamically tracks the row-open time (tON) and converts it into an *Equivalent Activation Count (EACT)*. We modify the RH trackers to operate on non-integer values. For example, a probabilistic solution that mitigates with probability p would now select the row with probability $p \times EACT$. A counter-based tracker would increment the counter by $EACT$ instead of 1. As Impress-P is precise, it maintains the same tolerated threshold (TRH) as a system that does not have RP mitigation.

We analyze ImPress with memory controller-based (Graphene and PARA) and in-DRAM (Mithril and MINT) trackers, and show that ImPress tolerates Row-Press for both categories. The storage required for ImPress-P is 1.25x, whereas it is 2x for both ExPress and ImPress-N.

This paper makes the following contributions:

1) We show that Row-Press can be transparently handled, without limiting tON, by treating a row open for tRC as equivalent to activation for RH schemes.
2) We develop a *Unified Charge-Leakage Model* to capture the net effect of RP and RH for any given pattern into a single number and use this metric to guide our design.
3) We propose *ImPress-N*, which treats a row open for the full-time window of tRC as equivalent to an activation. ImPress-N limits the impact of unmitigated Row-Press to tRC, reducing the Rowhammer threshold by 1.35x-2x.
4) We propose *ImPress-P*, which tracks tON and uses the damage due to RP and RH precisely into RH solutions. ImPress-P does *not* reduce the tolerated RH threshold and tolerates RP with negligible overhead.

II. Background and Motivation

A. Threat Model

We assume that the attacker can issue memory requests for arbitrary addresses. The attacker can choose the memory subsystem policies (e.g., open-page versus closed-page) best suited for the attack. The attacker knows the defense algorithm, including which row has been selected for mitigation. We declare an attack to be successful when it causes a bit-flip at any location in memory.

B. DRAM: Operation and Timings

DRAM chips are organized as banks, two-dimensional arrays of rows and columns. To access data from DRAM, the memory controller must first issue an activation (ACT) to open the row. The row can continue to be open until it is (a) proactively closed by the memory controller (e.g. closed-page policy) (b) closed due to a row conflict to service data from another row, or (c) closed to perform refresh.

DRAM has deterministic timings, specified as part of the JEDEC standards (see Table I). All data in DRAM is refreshed every tREFW. To reduce the latency impact of refresh, memory is divided into 8192 groups, and a refresh pulse is sent every tREFI interval to refresh one group. DDR5 specifications allow the postponement of up to 4 refreshes, so the time between refreshes can be up to 5 times tREFI.

TABLE I
DRAM Timings

Parameter	Description	Value
tACT	Time for performing ACT	12 ns
tPRE	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	Execution Time for REF Command	350 ns
tON	Time the current row is open (dynamic value)	–
tONMax	Max time a row can be kept open per DDR5	19.5 μs
tMRO	Max time a row can be kept open by the MC	–

DRAM systems are susceptible to *Data-Disturbance Errors (DDE)*, whereby the operations on one DRAM cell can corrupt the data stored in a nearby cell. An attacker could exploit DDE to compromise system security [2, 4, 6, 8, 9, 23, 40, 43]. In this paper, we focus on two specific modalities of DDE for DRAM: *Rowhammer (RH)* and *Row-Press (RP)*.

C. Rowhammer: Problem and Solutions

Rowhammer [22] occurs when a row (aggressor) is activated frequently, causing bit-flips in nearby rows (victim). The number of activations to an aggressor row to cause a bit-flip in a victim row is called the *Rowhammer threshold (TRH)*. Solutions to mitigate RH must ensure that the victim rows are refreshed before the aggressor row incurs TRH activations.

Typical solutions to mitigate RH rely on a *tracking mechanism* to identify the aggressor rows and perform a mitigative refresh on the victim rows. Identification of aggressive row could be done using *activation counters* [19, 28, 31, 32, 34, 37] or *probabilistically* [17, 17, 22, 42].

Tracking can be performed on the *Memory Controller (MC)* or transparently inside the DRAM chip *(in-DRAM)*. The advantage of in-DRAM tracking is the potential to solve the RH problem inside the DRAM without relying on other parts of the system. DDR5 supports in-DRAM tracking with *Refresh Management (RFM)*. Without loss of generality, we analyze the following four trackers in our study.

Graphene [32] (Counters, MC-Based): Graphene uses *Misra-Gries* algorithm to identify rows that reach TRH activations and issue a mitigation. The number of tracking entries (per bank) is inversely proportional to the threshold.

PARA [22] (Probabilistic, MC-Based): PARA selects each activation for mitigation with a probability p, which is determined based on a target failure rate.

Mithril [19] (Counters, in-DRAM): Mithril uses *Counter-based Summary* to identify heavily activated row. Mitigation is performed on receiving the RFM command (sent by MC every *RFMTH* activations) for the row with the highest count. The number of entries depends on RFMTH and TRH.

MINT [33] (Probabilistic, in-DRAM): MINT is a concurrent work that achieves secure mitigation with just a single entry per bank. At each RFM, MINT mitigates the identified aggressor row and randomly selects the activation slot that will be selected for mitigation in the upcoming RFMTH activations.

These trackers are designed to provide secure RH tolerance for a specific TRH. However, if the attacker can cause bit-flips in fewer than TRH activations, then the attacker can break all of these designs. A new vulnerability makes this possible.

D. Row-Press: Bit-Flips with Fewer Activations

Row-Press (RP) [26] is a new DRAM DDE vulnerability, which occurs when a row is kept open for a long time. When the row is open, the cells of the neighboring rows leak charge on the bit lines at a non-negligible rate. Over time, the total charge loss due to this leakage can become substantial. Figure 2 shows the access pattern for RP. Let tON be the time a row is kept open. With RP, we keep the aggressor open for a time that is much longer than tRAS. This pattern is repeated continuously until a bit-flip occurs.

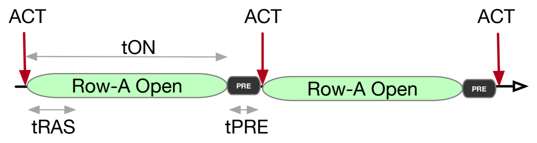

Fig. 2. Pattern for the Row-Press Attack (PRE denotes Precharge operation)

Luo et al. [26] characterized RP for DDR4 devices and showed that RP can reduce the number of activations required to induce a bit flip by 18x (on average, when the row is kept open for one tREFI, which is 7800ns in DDR4) to 156x (on average, when the row is kept open for 9 tREFI, which is 70 μs for DDR4) compared to standalone RH attacks. As RP can perform bit flips in much fewer than TRH activations, it can break RH mitigations designed for a threshold of TRH.

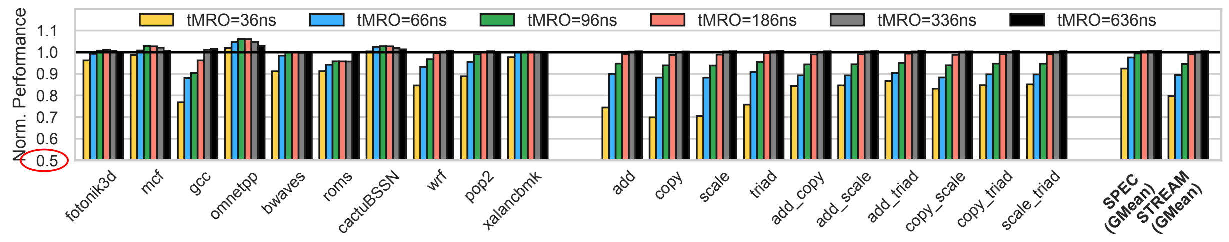

Fig. 3. Performance impact of limiting the time a row is open to a particular value, termed as tMRO (Maximum Row-Open Time). While SPEC workloads (low/medium spatial locality) are less sensitive to tMRO value, Stream workloads (high spatial locality) can suffer significant slowdown at low tMRO.

E. Tolerating RP by Limiting the Row-Open Time

Row-Press exploits the fact that an open row can continue to be open for a long time without any row conflicts. This time is constrained only by the time between refresh operations (tREFI, 3900ns for DDR5 and 7800ns for DDR4, although it can be extended with refresh postponement to 5 times tREFI in DDR5 and 9 times tREFI in DDR4).

Luo et al. [26] propose a solution to mitigate RP by using the Memory-Controller to limit the *Maximum Row-Open Time (tMRO)*. Figure 4 shows the number of activations required on the aggressor row with RP (i.e., change in TRH) as the tMRO is varied from 36ns (minimum value, tRAS) to 630ns. For example, if tON is limited to 186ns, the *effective* threshold (T*) reduces to 62%. The RH mitigation can be redesigned to tolerate this new threshold (T*). As this approach explicitly uses RP to change the threshold of existing algorithms, we term this solution *Explicit Row-Press (ExPress) Mitigation*.

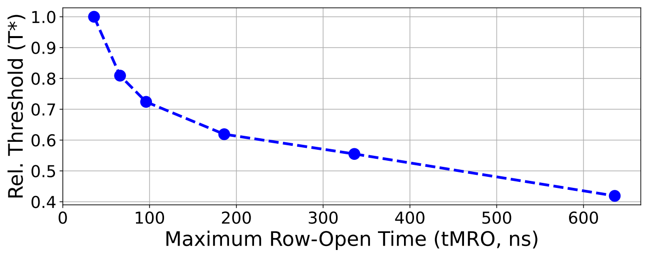

Fig. 4. Reduction in Tolerated TRH (T*) if the maximum tON is constrained to tMRO (Note: the data is obtained from Table-8 of [27])

Limiting the time a row can be open can reduce the row buffer hit rate due to premature closing of the row. The performance impact of early row closure depends on the type of workload. If the workload has poor row-buffer locality, early closure will not cause a slowdown (it may cause slight improvement due to removing precharge from the critical path). However, if the workload has a good spatial locality, early row closure can significantly impact performance. Figure 3 shows the normalized performance of our system for two classes of workloads, SPEC2017 and Stream, as the tMRO is varied from 36ns to 636ns. On average, for SPEC, low tMRO has a negligible performance impact, whereas, for Stream, low tMRO can cause a significant slowdown (e.g., on average, 10% for tMRO of 66ns). Thus, ExPress can cause substantial slowdowns for an entire category of applications.

ExPress causes additional slowdown as the design must cater to a lower threshold (T*), thus sending more mitigative refreshes. We analyze ExPress for our four trackers.

Graphene: Figure 5 shows the performance of Express as tMRO is varied. Stream workloads exhibit significant slowdown at low tMRO (due to reduced row-buffer hits). Furthermore, a higher tMRO increases the effective threshold (T*). So, Graphene must target an even lower threshold; hence, it would need more entries. At tMRO of 80ns, the storage of Graphene increases from 115KB to 160KB per channel.

PARA: Figure 5 also shows the relative performance of PARA when the tMRO is varied. The trend is similar to Graphene - negligible impact for SPEC workloads but significant slowdown for Stream workloads at low values of tMRO.

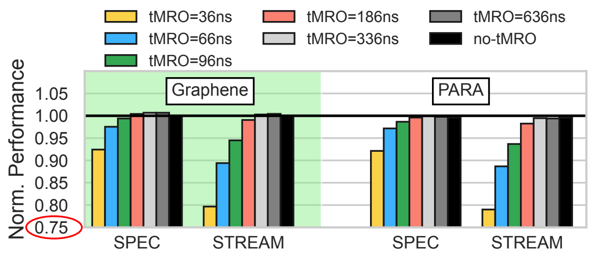

Fig. 5. Performance of Graphene and PARA as tMRO is varied. Stream has slowdown at low tMRO. [Note: All values are geometric means.]

MINT and Mithril: ExPress uses the MC to limit tON to tMRO. As the tMRO value is decided by the MC, this value is not visible to the trackers inside the DRAM chip, as current JEDEC standards do not allow such communication. Thus, ExPress is incompatible with in-DRAM trackers, so these trackers will continue to be vulnerable to RP. To make ExPress viable for in-DRAM tracking, JEDEC must include a new parameter (tMRO). Unfortunately, such changes are hard to incorporate in JEDEC, as all memory/processor vendors must adhere to new specifications. Furthermore, any such specification is likely to select the tMRO conservatively.

F. Goal of Our paper

An ideal solution should tolerate Row-Press transparently, without limiting tON (thus avoiding changes to JEDEC and letting systems choose what performs best for their workloads), should not lower the effective threshold, should have only a minor impact on performance and storage overheads, and should be compatible with both MC-based and in-DRAM designs. The goal of our paper is to develop such a solution.

III. Experimental Methodology

A. Performance Methodology

We use ChampSim [7], a cycle-level multi-core simulator, interfaced with DRAMSim3 [25], a detailed memory system simulator. We enhanced DRAMSim3 to support DDR5. Table II shows the configuration for our baseline system. We use a *Minimalist Open-Page (MOP)* memory mapping with 8 consecutive lines per row. For RFM, we assume a latency of 205ns (half of tRFC) and use a default RFMTH of 80.

TABLE II
Baseline System Configuration

Out-of-Order Cores	8 cores at 4GHz
Width, ROB size	6-wide, 352
Last Level Cache (Shared)	16MB, 16-Way, 64B lines, SRRIP
Memory size	64GB – DDR5
Channels	2 (32GB DIMM per channel)
Banks x Ranks x Sub-Channels	32×1×2
Memory-Mapping	Minimalist Open Page (8 lines)

We use two categories of workloads: First, the 10 SPEC2017 [1] (8-core rate mode) traces available from ChampSim to study the impact of tMRO on conventional workloads. Second, 4 streaming workloads [30] (8-core rate mode) and 6 mixed streaming workloads (two with 4 copies each), to study the impact of tMRO on high-locality workloads. For each workload, the trace represents the region of interest. We warm up for 50 million instructions and run each workload for 200 million instructions. We report performance as normalized weighted speedup.

B. Reliability Methodology for RH Trackers

We perform mitigation by refreshing the victim rows. To securely mitigate RH and RP, the parameters of the underlying RH mitigation scheme must be configured appropriately. We use a default TRH of 4K [18], and show sensitivity in Section VI-F. For probabilistic schemes, we use a target bank-failure rate of 0.1 FIT (1 failure per 10 billion hours, about 30x lower than the rate of naturally occurring errors [3]).

Based on our target failure rate, we configure PARA with p=1/184. For Graphene, the number of entries is inversely proportional to TRH. To tolerate a TRH of 4K, Graphene needs 448 entries per bank (115KB SRAM per channel).

Mithril performs mitigation transparently under the RFM command, issuing RFM every RFMTH activation per bank. For mitigation, Mithril selects the aggressor row with the highest counter value. For a given mitigation rate (1 per RFMTH), we determine the number of entries required to tolerate a given threshold using Theorem-1 of [19]. For example, for an RFMTH of 80, Mithril needs 383 entries per bank (86 KB SRAM per channel) to tolerate a TRH of 4K.

MINT requires a single entry per bank to keep track of the row to be mitigated at RFM. At each RFM, MINT mitigates the given aggressor row and then randomly selects which activation slot in the upcoming RFMTH (e.g. 80) activations will be chosen for mitigation at the next RFM. As MINT lacks configurability (for a fixed RFMTH), we report the threshold tolerated by MINT as the figure of merit.

IV. Unified Charge-Loss Model

To mitigate Row-Press transparently and at a low cost, we propose *Implicit Row-Press (ImPress) mitigation*. ImPress converts the time spent doing Row-Press to an equivalent activation count for Rowhammer. To design ImPress, we first develop a unified charge-loss model for RH and RP.

A. Relative Charge-Loss Model for Rowhammer

Consider a DRAM cell that is the target of a RH attack. After TRH activations to the aggressor row, the total charge loss suffered by the cell must be above some *critical* value to cause a bit flip. We need a model to quantify the *Total Charge Loss* incurred after K activations. We quantify charge-loss as a *relative* metric to keep our model simple. Let the *relative charge-loss per activation* (C_A) be 1 unit. The total charge loss (TCL_{RH}) after K activations is given by Equation 1.

$$TCL_{RH} = K \cdot C_A = K \cdot 1 = K \quad (1)$$

As a bit flip occurs after TRH activations, the total charge loss is TRH units, representing the value of the *critical* charge loss. Figure 6 shows the charge-loss model for RH. Note that the time is counted in terms of tRC. RH is a *perfect linear attack* – with unit damage in one unit of time.

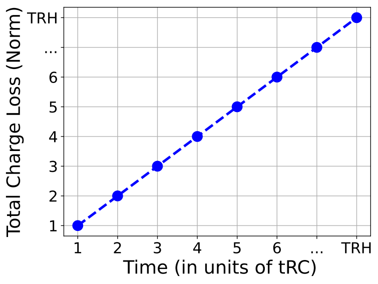

Fig. 6. Relative Charge-Loss Model for Rowhammer

B. Relative Charge-Loss Model for Row-Press

The charge loss for RP comes from two sources: (1) the activation and the time incurred in the first tRC, the impact of which is identical to an RH pattern, so this period incurs a charge-loss of 1 unit (2) the time-dependent charge loss that occurs because the row is kept open for an *additional time* $(tON - tRAS)$. As we normalize all times to tRC, we also normalize the *additional time* to tRC. The total charge loss (TCL_{RPA}) from an RP pattern that keeps a row open for tON time is given by Equation 2.

$$TCL_{RPA} = 1 + f(\frac{tON - tRAS}{tRC}) \quad (2)$$

The function f captures the *rate of charge leakage* per unit of time (in terms of tRC) for RP. This function can be estimated using the characterization data or picked conservatively to never be below the observed data.

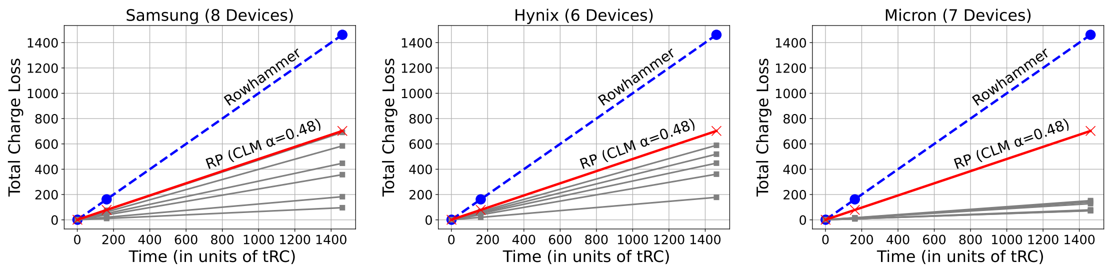

Fig. 7. Total Charge Loss (TCL) for long-duration RP attacks that last for 1 tREFI (162 tRC in DDR4) and 9 tREFI (1462 tRC in DDR4). For our CLM model for RP, we use alpha=0.48 as it covers all the devices across the three vendors (experimental data is reproduced from Appendix B of Luo et al. [27])

If we have the data for T* available, we can deduce the relative charge leakage incurred by a single round of an RP attack (for a given tON time) compared to a single round of an RH attack. For example, if the RP attack causes T* to be half of TRH, then each round of RP attack must leak 2x the charge as a single round of RH attack. We use this insight to estimate the charge leakage versus the attack time for an RP attack (note that the total time for an RP attack is tON+tPRE, as the attack eventually ends with a precharge). Figure 8 shows the total charge loss for the RP attack and compares it with the RH, as the attack time increases from 1 tRC to 8 tRC. RH is a linear attack (K units of charge-loss in K units of time). The *red dots* represent the charge-loss derived from the data of Luo et al. [26] (data is a reorganized version of Figure 4).

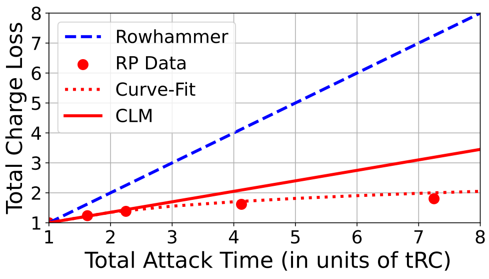

Fig. 8. Relative Charge-Loss Model for Row-Press

C. Conservative Linear Model (CLM)

We could do a curve-fit on experimental data (shown as the dotted red line in Figure 8). However, we have two key requirements: (1) the function must be simple for ease of implementation inside the DRAM chip, and (2) the function must not underestimate the TCL observed in the chips, as doing so may lead to reliability and security failures. Therefore, we develop a *Conservative Linear-Model (CLM)*, which provides a linear relationship[2], albeit a conservative one, where no observed data point exceeds the CLM line.

The general form of CLM is given by Equation 3.

$$TCL_{ON} = 1 + \alpha * (\frac{tON - tRAS}{tRC}) \tag{3}$$

[2]We assume a linear model because a DRAM cell can be modeled as an RC circuit and charge leakage can be approximated to be linear for short period of time (tRC of 48ns compared to retention of 32ms).

Where α *is the relative charge leakage per tRC for RP* ($\alpha \leq 1$, and $\alpha = 1$ gives RH). For the data from Luo et al. shown in Figure 8, $\alpha = 0.35$, leading to Equation 4.

$$TCL_{RPA} = 1 + 0.35 * (\frac{tON - tRAS}{tRC}) \tag{4}$$

An RP attack degenerates into an RH attack if tON= tRAS. Thus, Equation 3 represents both RH and RP for any pattern.

D. Row-Press at Large Time Scale

The data shown in Figure 8 is for a short-duration (sub-microsecond) RP attack. However, RP attacks can last up to one tREFI without refresh postponement and up to 9x of tREFI with refresh postponement (for DDR4). Appendix B of Luo et al. [27] also characterizes devices from the three memory vendors for long-duration RP attacks, specifically 1 tREFI (162 tRC in DDR4) and 9 tREFI (1462 tRC in DDR4). Figure 7 shows the total charge loss (TCL), as the time is normalized in terms of tRC. For comparison, the TCL of Rowhammer is also shown, if performed for an identical duration. We also show our CLM model for RP, and we set $\alpha = 0.48$, as it covers all the characterized devices. Thus, we can use Equations 3 to model short-duration and long-duration RP attacks.

E. Key Observations:

Our model enables us to estimate the combined effect of RH and RP patterns, where the RP length is limited by the DDR specifications. The takeaways from our model are:

1. Row-Press is a much slower attack than Rowhammer. Even with α of 0.48, RP causes less than half the *damage* (charge loss) per unit time as a standalone RH attack.

2. Any time spent in RP is the time that the attacker cannot perform RH. Therefore, doing RH alone is the fastest way to reach a critical charge loss, limited only by RH mitigation.

3. A secure RP solution must reduce the dependency on α as α may vary between chips, or select the value of α conservatively, so that it is guaranteed to work across all chips.

4. As the leakage of RH is due to activity (row activation) and RP is due to idling, it is unlikely α would exceed 1. So, using α of 1 avoids relying on per-device behavior.

The **key assumption** in our work is that the leakage rate (per unit time) for Row-Press is less than or equal to that of Rowhammer (i.e. $\alpha \leq 1$). This seems safe as Rowhammer is activity-based leakage, while Row-Press is idle leakage.

V. ImPress-N: The Naive Version

We propose two variants of ImPress. The first version is *ImPress-N*, the *naive* version, designed to handle only integer charge-loss values. ImPress-N aims to understand the impact of reduced precision on ImPress's effectiveness. ImPress-N divides the time into windows of tRC, and if a row is open for the entire window, it treats it as equivalent to activation for RH mitigation. Thus, ImPress-N limits the impact of any unmitigated Row-Press to at most one tRC window. In this section, we provide the design and analysis of ImPress-N and bound the worst-case effect of the unmitigated Row-Press.

A. *ImPress-N: Design and Operation*

The key insight in ImPress-N is that secure RH mitigations are designed to tolerate the worst-case RH pattern, which causes activation in each time window of tRC. With Row-Press, if a row is kept open for a long time, then by design, such a pattern will not cause as many activations as the worst-case. If we convert the RP activity into RH activity, we can use the existing RH framework to mitigate RP.

Figure 9 shows the overview and design of ImPress-N. ImPress-N divides time into windows of tRC. If a row activation occurs within the window, that row participates in the RH mitigation. This is the case for Row-A in the second window and Row-B in the fourth window. Furthermore, if a row is kept open for the entire tRC window, then it is treated as equivalent to causing a row activation for that open row. That open row again participates in RH mitigation. For example, Row-A, open for tRC during the third window, is treated as activating Row-A for RH mitigation.

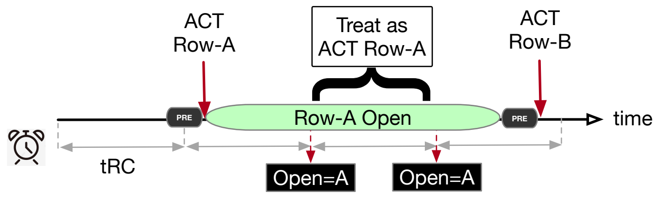

Fig. 9. Design and Operation of ImPress-N. A row open for tRC is treated as equivalent to causing an activation within that window.

To implement ImPress-N, the system requires two counters. First, a *Timer* register identifies each window's ending time. Second, an *Open-Row Address (ORA)* register that stores the row address of the open row. ORA is filled at the end of each window. Suppose that the address to store in ORA is the same as the address present in ORA. In that case, it indicates that the row was open for the entire window and participated in the RH tracking mechanism, similar to causing an activation.

ImPress-N is simple to incorporate into current RH mitigation solutions. It converts RP activity into a series of ACTs, which are already handled by RH mitigation, so the underlying tracker design does not need to be changed. The total storage for implementing ImPress-N is 1 byte for Timer and 3 bytes for ORA, for 4 bytes per bank (32 bytes per chip).

B. *Bounding the Impact of Unmitigated Row-Press*

ImPress-N converts an RP pattern that keeps a row open over multiple tRC windows into an equivalent number of ACTs (one per tRC). However, since it operates on integer values, it does not mitigate RP at a granularity lower than tRC. An attack can exploit this to reduce the threshold.

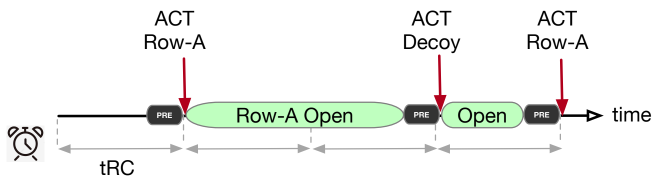

Fig. 10. The pattern for exploiting the unmitigated Row-Press of ImPress-N – an attacker can keep the row open for tRAS+tRC and evade RP mitigation.

Figure 10 shows the worst-case pattern for ImPress-N. The attacker is focused on causing undetected RP on Row-A. The pattern causes Row-A activation at a time within the precharge-time (PRE) of the ending of the current window. Row-A has not yet been opened and will not be stored in ORA. The pattern keeps Row-A open for a time equal to tRC+tRAS. As Row-A is open at the end of the current tRC window, the address of Row-A is stored in ORA. During the subsequent window, at a point slightly before the precharge time from the ending of the window, an ACT is sent for a decoy row, which causes precharge and closes Row-A. Thus at the end of the window, ORA gets an invalid row. The pattern is repeated.

For each round, the RH mitigation will see only a single ACT for Row-A and thus treat this as an RH attack, causing a charge loss of 1 per round for Row-A. As the tON time for Row-A is (tRC+tRAS), we can use Equation 3 to quantify the charge loss per round as $(1+\alpha)$. Thus, the *Effective Threshold* (T^*) with ImPress-N is given by Equation 5.

$$T^* = \frac{TRH}{(1+\alpha)} \tag{5}$$

The impact on the threshold depends on α. The value of alpha from experimental data (tON $\leq$ 2tRC) reported by Luo et al. is 0.35. So, T* is equal to TRH/1.35 or 0.74×TRH. If we want device independence, then α=1 and T* equals TRH/2.

C. *Protecting RH Trackers with ImPress-N*

Appendix A describes how ImPress-N can be applied to our four tracker designs: PARA, Graphene, Mithril, and MINT. For PARA and Graphene, ExPress and ImPress-N have similar performance overheads as they must be operated at a reduced threshold (e.g., 2x lower). Overall, ImPress-N can make Row-Press mitigation viable at small performance overheads for Mithril and MINT.

Key Takeaway: For MC-based trackers, ImPress-N has a similar impact as ExPress on threshold, performance, and storage. However, as ImPress-N does not limit tON, it can also be used with in-DRAM trackers, thus representing the first solution to protect such trackers from RP attacks.

VI. ImPress-P: The Precise Version

While ImPress-N is a simple design (no changes to the trackers, except for the number of entries), it can still incur performance overheads due to lowering the effective threshold resulting from unmitigated Row-Press that occurs at sub-tRC granularity. Furthermore, the impact of ImPress-N on the threshold depends on the value of α, and we want a solution that naturally offers protection of α=1 without any of the associated overheads. Our next design, *Impress-P (Precise)*, overcomes both shortcomings of ImPress-N. The key idea in ImPress-P is to measure the tON time of a row and use it to determine the *Equivalent Number of Activations (EACT)* between the time the row is opened and when it completes the precharge. ImPress-P ensures that there is no lowering of the threshold due to mitigating Row-Press. In this section, we provide the design and analysis of ImPress-P and study the impact of applying Impress-P to different trackers.

A. ImPress-P: Design and Operation

The key insight in ImPress-P is that secure RH mitigations are designed to tolerate the rate of damage that occurs under the RH pattern. So, we can treat every time unit in terms of tRC (integer or fractional) as equivalent to the number of ACTs (integer or fractional). This allows us to precisely convert any amount of RP activity into equivalent RH activity and use the existing RH framework to accurately mitigate RP without impacting the threshold.

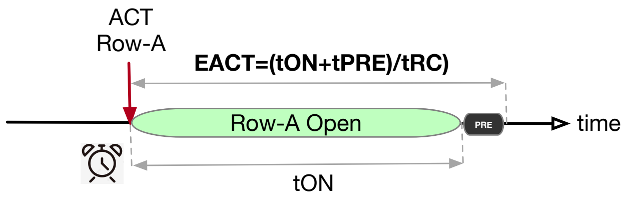

Fig. 11. Design and Operation of ImPress-P. ImPress-P measures the time the row is open and converts it into an equivalent number of ACT (EACT).

Figure 9 shows the design of ImPress-P. ImPress-P only requires a timer to measure when the row is open (tON). The timer starts when the row is opened and stops when the row is closed. The total duration for access must also include the time required for precharge, so the total time is equal to tON+tPRE. We divide the total time by tRC to get the *Equivalent Number of ACTs (EACTs)*. For example, if tON is equal to tRAS, this is the same as RH attack, and EACT is equal to 1. If tON is equal to tRAS+tRC, the access lasts for two tRC and we would get EACT=2. EACT is guaranteed to be at least 1, but it can be a fractional value (e.g., if tON = tRAS + tRC/2, EACT = 1.5). Thus, the RH-mitigation algorithms must be able to handle a non-integer number of ACT.

For counter-based algorithms, we modify the counters to support fractional values. Instead of increasing by 1, we increase the counter by EACT. We modify the selection probability from p to p*EACT for probabilistic solutions. Thus, ImPress-P applies to both types of trackers.

ImPress-P requires a single *Timer* (10 bits) per bank (32 per chip). All DRAM activity occurs and is measured at the granularity of the DRAM cycles. For our 2.66GHz DRAM, this means that tRC (48ns) equals 128 cycles; thus, the division by tRC can be implemented by shifting right by 7 bits.

B. Impact of Counter Precision on Effective Threshold

The fractional part of EACT is 7 bits (due to division by tRC). For counter-based tracking algorithms, this means that the counter must also be extended by 7 bits to incorporate the fractional values of EACT precisely. A design may modify the counter-based tracker with fewer bits to store the fractional value (to save on storage) at the expense of some tracking error, which reduces the effective threshold (T*).

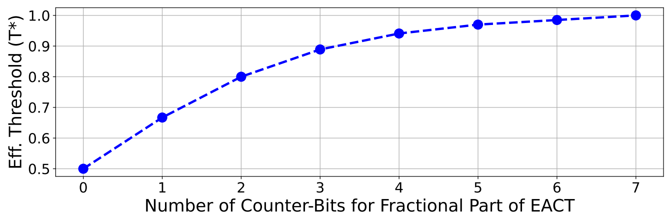

Fig. 12. Impact of number of counter-bits for storing the fractional part on the effective threshold of ImPress-P (value is normalized to TRH).

Figure 12 shows the effective threshold (T*) of ImPress-P as the number of counter-bits used to store the fractional part is varied from 0 to 7. With 7 bits, we track accurately, so T* equals TRH (no reduction in threshold). With fewer than 7 bits, say b bits, we get a precision equal to $\frac{1}{2^b}$, so the loss in accuracy is also equal to $\frac{1}{2^b}$. Thus, with 6 bits, the relative T* reduces to 0.985, 5 bits to 0.97, and 4 bits to 0.94. Finally, if we have 0 bits for the fractional part, ImPress-P degenerates to ImPress-N and has T* of 0.5 times TRH.

Our default implementation of ImPress-P uses 7-bits to store the fractional part. Thus, ImPress-P maintains the *same* TRH with Row-Press protection compared to a system without any Row-Press protection. Furthermore, ImPress-P avoids dependency on α, which is implicitly designed for α of 1. Thus, while implementing and comparing designs with ImPress-P, we will use α=1.

C. Protecting RH Trackers with ImPress-P

Unlike ExPress, ImPress-P does not place any limit on tON. Thus, it does not affect performance due to the early closure of an open row. Furthermore, as ImPress-P does not affect the threshold, it does not incur any additional mitigations due to activations compared to an idealized baseline that does not have Row-Press. However, ImPress-P can still incur additional mitigations due to a row being kept open for a long time.

We analyze ImPress-P, ImPress-N, and ExPress for our trackers. We implement ExPress with tMRO of tRAS+tRC. As ExPress is incompatible with in-DRAM tracker designs (Mithril and MINT), we compare ImPress-P with only ImPress-N for these two designs. We describe the changes required in the tracking algorithms to support ImPress-P.

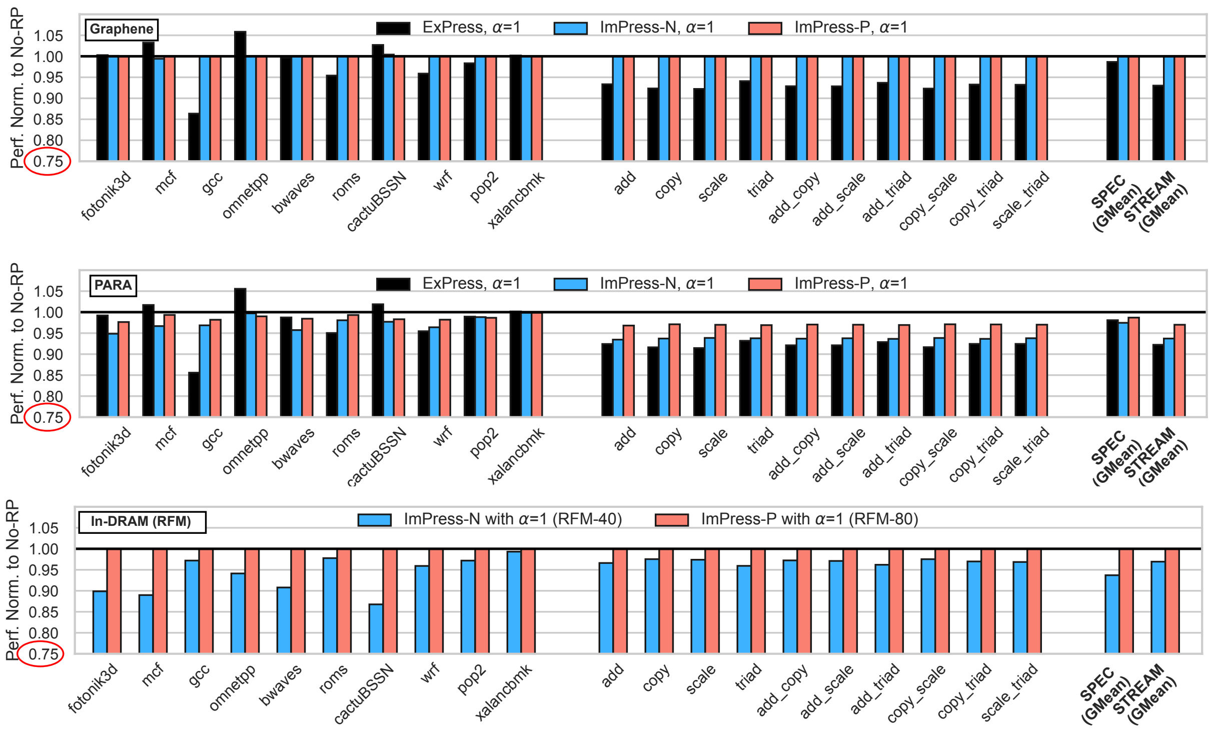

Fig. 13. Performance of (a, top) Graphene and (b, mid) PARA for ExPress, ImPress-N, and ImPress-P (c, bottom) Performance of in-DRAM (MINT) for ImPress-N and ImPress-P (ExPress is not shown as it is not applicable to in-DRAM trackers). Note: All performance is normalized to No-RP.

Impact on Graphene: For TRH of 4K, Graphene requires 448 entries per bank. ExPress and ImPress-N (α of 1) double it to 896 per bank. With ImPress-P, the number of entries remains unchanged at 448. However, each entry now requires 7-bits of extra storage to store fractional value of EACT, hence ImPress-P incurs 25% storage overhead (each entry is 28-bits). Thus, the total storage required for ImPress-P is only 1.25x of No-RP, whereas it was 2x for both ImPress-N and ExPress.

Figure 13 shows the performance of Graphene with ExPress, ImPress-N and ImPress-P, normalized to a baseline that does not suffer from RowPress. As ImPress-P does not affect the threshold or restrict tON, it incurs a negligible overhead.

Impact on PARA: Conventionally, PARA uses a constant probability p for all activations. For TRH of 4K, p=1/184, and ImPress-N and ExPress p=1/92. ImPress-P changes PARA to use a variable value for p for each activation, depending on the tON time. For each activation, PARA uses $\hat{p} = p * EACT$.

Figure 13 shows the performance of PARA with ExPress, ImPress-N, and ImPress-P, normalized to a baseline without Row-Press. ImPress-P has significantly reduced performance overheads (especially for Stream) compared to ExPress.

Impact on Mithril: For TRH of 4K and a default RFMTH of 80, Mithril requires 383 entries. This increases to 1545 entries (4x) with ExPress and ImPress-N (α=1). With ImPress-P, the number of tracking entries remains unchanged at 383. However, each entry must now be provisioned with 7 more bits to track the fractional values, resulting in 25% storage overheads, much less than the 4x overhead required for ExPress and ImPress-N. The performance overheads of Mithril, due to RFM commands, remain the same as the No-RP baseline.

Impact on MINT: MINT contains three registers: SAN (Selected Activation Number), CAN (Current Activation Number), and SAR (Selected Address Register). Both SAN and SAR remain unchanged. We modify CAN to have 7 more bits corresponding to the fractional value of EACT. For each activation, we increase CAN by the value of EACT. Thus, each activation gets a selection probability in proportion to the EACT. If CAN crosses SAN, the row address is stored in SAR. At RFM, the row address in SAR (if valid) is mitigated, and a new value for SAN is selected. ImPress-P increases the storage overhead of MINT from 4 bytes to 5 bytes. With ImPress-N, the threshold increases from 1.6K to 3.1K, whereas with ImPress-P, it remains unchanged at 1.6K. Figure 13(c) shows the performance of No-RP, ImPress-P, and ImPress-N. ImPress-P has an identical performance to No-RP.

D. Summary of Comparisons

Table III compares ExPress, ImPress-N, and ImPress-P. The shortcomings are highlighted in **bold**. ImPress-P requires minor changes (to include EACT) and provides near-ideal performance. Therefore, we will assume that by default, ImPress is implemented only as ImPress-P (ImPress-N was an intermediate step to emphasize the importance of precision).

TABLE III
COMPARISONS OF EXPRESS, IMPRESS-N, AND IMPRESS-P

Property	ExPress	ImPress-N	ImPress-P
Puts Limit on tON	**Yes**	No	No
Affects Threshold (T*)	**Yes (up to 2x)**	**Yes (up to 2x)**	No (1x)
Performance Overheads	**High**	Medium	Low
More Tracking Entries	**Yes (up to 2x)**	**Yes (up to 2x)**	No (1x)
Wider Tracking Entries	No	No	**Yes (Minor)**
In-DRAM Trackers	**Incompatible**	Compatible	Compatible
Device Dependency	**Yes (alpha)**	**Yes (alpha)**	No

E. Activation and Energy Overheads

Tolerating Row-Press may cause extra activations due to row closure or mitigations. Figure 14 shows the average mitigations relative to an unprotected baseline. Graphene without RP protection (No-RP) causes less than 1% extra activations. With ExPress, mitigative activations remain low, however, early row closure increases activations by 56% (for both Graphene and PARA). Graphene with ImPress-P does not incur additional activation overhead. For PARA, demand activations with ImPress-P increase negligibly by 2% on average. However, the mitigative activations increase by 12%. ImPress-P has significantly lower activation overhead than ExPress, reducing it from 56% to 1% for Graphene, and 61% to 14% for PARA.

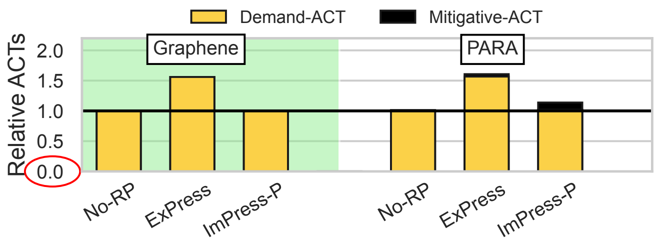

Fig. 14. Relative activation overhead of Graphene and PARA (No-RP, Express, and Impress-P), broken down into demand activations and mitigative activations (all normalized to activations in the unprotected baseline).

Energy Overheads: On average, activations account for 11% of the baseline DRAM energy. ExPress increases DRAM energy by 6% for Graphene (7% for PARA), while for Impress-P, the increase in energy is 1% for Graphene (2% for PARA).

F. Scalability to Lower Rowhammer Threshold

Figure 15 shows the performance of Graphene and PARA normalized to an unprotected baseline as TRH varies from 1K to 4K. At TRH of 1K. Graphene incurs no slowdown for No-RP and ImPress-P, while ExPress has 4.4% slowdown. PARA incurs a 1.5% slowdown for No-RP, and ExPress increases the slowdown to 8.9%. ImPress-P reduces it to 7.7%. The storage overheads of Graphene and performance overheads of PARA make them impractical for low TRH. To tolerate low TRH, JEDEC [15] announced *Per-Row Activation Counting (PRAC)* where the DRAM array stores a counter for each row (8KB). ImPress can be used with PRAC by dedicating 7 bits of the counter to store the fractional EACT. Alternatively, the PRAC counter can be incremented by *round-up* of EACT, which would ensure security, although with slightly more mitigations.

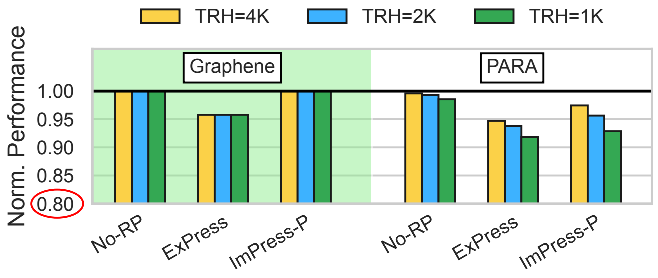

Fig. 15. Performance of Row-Press mitigation with Graphene and PARA, normalized to an unprotected baseline. [Note: All values are Geo-Mean.]

VII. Related Work

To the best of our knowledge, ImPress-P represents the first tracker implementation that can securely tolerate both Rowhammer and Row-Press. ImPress exploits the observation that the row-open time can be converted into equivalent activity for Rowhammer. Two prior works have made similar observations. For example, ProTRR [28] suggests "increasing the counter for victims of the (aggressor) row that remains active". However, ProTRR does not provide any methodology to convert the row-open time into equivalent RH (notably, ProTRR appeared one year before Row-Press was publicly known and characterized, so the lack of such details is understandable). Furthermore, ProTRR operates with integer-valued counters, and ImPress-N shows that the integer-valued design would have a significantly higher threshold than ImPress-P.

Although DSAC [11] uses *time-weighted* counting, it suffers from three problems: (1) the weight is a logarithmic function of time. For example, for tON=256 tRC, the weight will be approximately 8, whereas, the Row-Press characterization [26] shows that the weight should be about 0.48*256 = 122 (15x higher). Thus, DSAC significantly underestimates the RP damage, (2) Row-Press is ignored for the row getting installed in the tracker, as it always uses a weight of 1, (3) DSAC uses integer counter values and would suffer from the same problem as ImPress-N, even if the weights were accurate. We note that DSAC can be broken with Blacksmith [12], so assessing the security of DSAC against Row-Press is impractical.

Several studies [17] [22] [48] [42] [17] [41] [24], [32] [34] have investigated efficient trackers to identify aggressor rows. Our design can work with these trackers. We do not consider In-DRAM designs of TRR [6], DSAC [11], and PAT [21] as these can be broken with simple patterns [6] [14]. Our work applies to secure in-DRAM trackers like Mithril [19], PrIDE [13], MINT [19], ProTRR [28], and PRHT [21].

Prior works have looked at alternative mitigation techniques, such as rate-limiting [46] or Dynamic row-migration [36] [39] [45] [44]. Prior studies [4, 5, 16, 20, 38] have also proposed using ECC and detection codes to tolerate Rowhammer. All these works can reduce, but not eliminate, DDE errors. REGA [29] and HiRA [47] modify the DRAM module to support multiple concurrent mitigative activations.

VIII. Conclusion

The scaling of DRAM to single-digit nanometers results in new modalities of *Data-Disturbance Errors (DDE)*. While Rowhammer is well known, a new pattern, Row-Press, was recently discovered, which causes charge leakage by keeping the row open for a long time. RP reduces the number of activations required to induce a bit-flip. Prior work proposed to mitigate RP by limiting the maximum time a row can be kept open, however, that proposal incurs high overheads and is incompatible with in-DRAM tracking. We propose *Implicit Row-Press (ImPress) mitigation*, which converts RP activity into an equivalent amount of RH activity, and uses the RH framework to mitigate RP. Our solution does not restrict tON, incurs low overhead, and applies to all trackers.

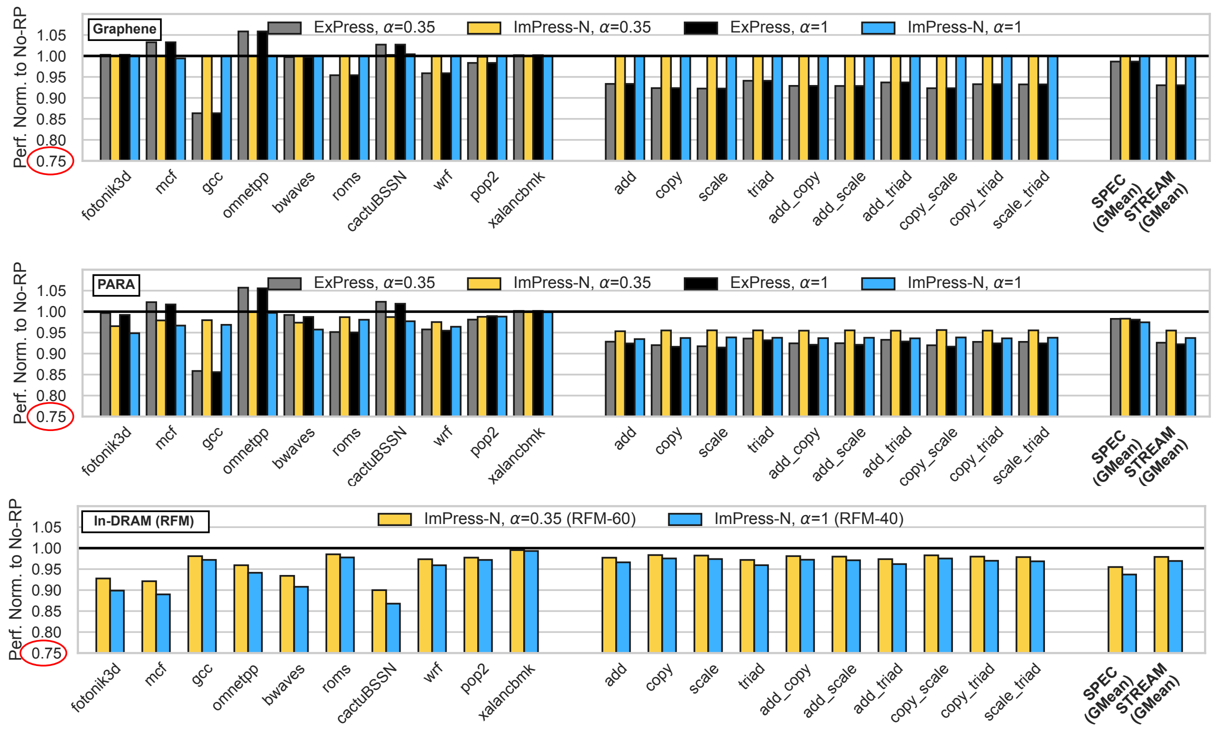

Fig. 16. Performance of (a, top) Graphene and (b, mid) PARA for ExPress and ImPress-N (both designs for α of 0.35 and 1) (c, bottom) Performance of in-DRAM (MINT) for ImPress-N (ExPress is not shown as it is not applicable to in-DRAM trackers). Note: All performance is normalized to No-RP.

APPENDIX-A: PERFORMANCE IMPACT OF IMPRESS-N

Unlike ExPress [26], ImPress-N does not place any limit on tON. Therefore, it does not suffer from reduced row-buffer hits due to premature closing of an open row due to tMRO. However, ImPress-N still incurs performance overhead from the extra mitigations due to the reduction in threshold (T*) and from considering rows opened for tRC as an ACT.

We analyze ImPress-N and ExPress for our four trackers. To ensure that both schemes target the same T*, we evaluate ExPress with tMRO set to (tRAS+tRC).

Impact on Graphene: For TRH of 4K, Graphene uses an internal threshold of 1333 (mitigation is sent when counters reach the internal threshold), requiring 448 entries per bank (a total of 115KB SRAM per channel). To make Graphene Row-Press tolerant with ExPress or ImPress-N, the number of entries must be increased directly to $(1+\alpha)$. Thus, for α of 0.35, Graphene requires 605 entries per bank (a total of 155KB SRAM per channel), and α of 1, Graphene requires 896 entries per bank (a total of 230KB SRAM per channel). Thus, ExPress and ImPress-N require a total storage overhead of 1.35x-2x compared to the No-RP design.

Figure 16 shows the performance of Graphene with ExPress and ImPress-N, normalized to No-RP. As graphene efficiently sends mitigative refreshes, the slowdown comes mainly from reducing row-buffer hits. For Stream workloads, ExPress incurs an average slowdown of 7.5%, whereas ImPress-N incurs a negligible slowdown. For SPEC, both ExPress and ImPress-N have similar performance.

Impact on PARA: For TRH of 4K, PARA requires p to be 1/184. At α of 0.35, p increases by 1.35x to 1/136, for both ExPress and ImPress-N. At α of 1, p increases to 1/92 for both ExPress and ImPress-N. Figure 16 shows the performance of PARA with ExPress and ImPress-N, normalized to No-RP. On Stream workloads, ExPress incurs an average slowdown of 8% (at α of 0.35) and 8.4% (for α of 1), whereas ImPress-N incurs an average slowdown of 4.7% (at α of 0.35) and 6.7% (for α of 1). Overall, ImPress-N performs better than ExPress.

ExPress is incompatible with in-DRAM trackers, so we evaluate Mithril and MINT only with ImPress-N.

Impact on Mithril: We assume a default RFM Threshold (RFMTH) of 80. For such RFMTH to handle a TRH of 4K, Mithril requires 383 entries. To account for the unmitigated RP of ImPress-N, Mithril would need to target a revised threshold (T*) of either 2963 (α=0.35) or 2000 (α=1). Thus, the number of entries increases from 383 to 615 (α=0.35) or 1545 (α=1).

We assume a system that already performs RFM at a RFMTH of 80 (to tolerate Rowhammer). Therefore, Mithril and MINT do not incur additional performance overheads.

Impact on MINT: We use RFMTH of 80 for MINT. Therefore, for No-RP, MINT can tolerate a TRH of 1.6K. Due to the unmitigated Row-Press of ImPress-N, the tolerated threshold increases to 2.1K (α=0.35) and 3.1K (α=1). Alternatively, we could reduce RFMTH to 60 (α=0.35) or 40 (α=1) to retain the same tolerated TRH (of 1.6K). Figure 16 shows the slowdown of RFM-60 and RFM-40 compared to RFM-80. The average slowdown is small and ranges from 3% to 5%.

Appendix-B: Performance Impact of Attacks

We analyze the performance implications of ImPress-P under attacks that combine both Rowhammer and Row-Press. We note that such patterns affect the performance only for memory-controller-based mitigations. The performance of in-DRAM Rowhammer mitigations remains independent of the access patterns as mitigations are performed under REF.

A. *Parameterized Attack Patterns for RH and RP*

The RH and RP patterns can be parameterized, as shown in Figure 17. In this pattern, an activation keeps the row open for tRAS. Then, the row is kept open for an additional K times tRC time, where K is the Row-Press parameter. Finally, the row is closed, incurring the tPRE time. Thus, the total time for one pattern loop is $(K+1)*tRC$. The pattern degenerates to Rowhammer for $K=0$. The pattern keeps the row open for a full tREFI for $K=72$. The pattern is repeated continuously, and we want to find the relative time taken to perform a large number of attack iterations (N).

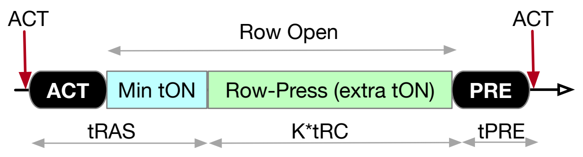

Fig. 17. Attack Loop for combined Rowhammer and Row-Press pattern.

B. *Analyzing the Performance Impact on Graphene*

For the Rowhammer threshold of T, Graphene performs mitigation when the counter reaches $T/2$ activations. For each mitigation, we need 4 activations, assuming a Blast Radius of 2. Thus, Rowhammer performs four additional activations every $T/2$ demand activation for a throughput loss of $8/T$. The slowdown is 0.2%, 0.4%, and 0.8% for the thresholds of 4K, 2K, and 1K, respectively, as shown in Figure 18.

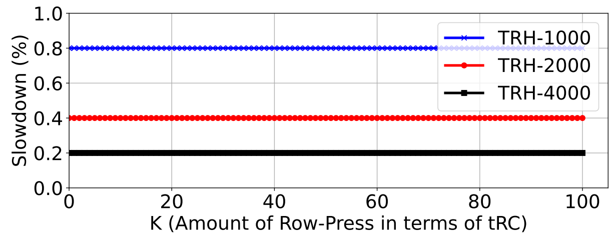

Fig. 18. Slowdown of ImPress-P with Graphene for the attack pattern.

To analyze Row-Press, we vary the parameter "K". The total time for N iterations in the unprotected baseline is N * (K + 1) * tRC. In each loop iteration, the counter of Graphene increases by (K+1). When it reaches T/2, Graphene issues a mitigation (4 activations). Thus, the slowdown will be four activations per (T/2)/(K+1) iteration of the loop. To simplify our analysis and without loss of generality, consider the case where the attack loop is repeated N = (T/2)/(K+1) times.

$$t_{mitigation} = 4 \cdot tRC \tag{6}$$

$$t_{one-iter} = (K+1) \cdot tRC \tag{7}$$

$$t_N = (K+1) \cdot tRC \cdot \frac{(T/2)}{(K+1)} = (T/2) \cdot tRC \tag{8}$$

$$Slowdown = t_{mitigation}/t_N = \frac{4 \cdot tRC}{(T/2) \cdot tRC} = 8/T \tag{9}$$

Thus, the slowdown of Graphene remains at 8/T, independent of "K" (the effect of Row-Press). This is expected as ImPress-P converts Row-Press into an equivalent amount of Rowhammer; therefore, the slowdown per unit attack time remains the same, regardless of whether the pattern is Rowhammer or Row-Press. Figure 18 shows the slowdown of Graphene as the amount of Row-press is varied (for T of 4K, 2K and 1K), and the slowdown does not depend on Row-Press.

C. *Analyzing the Performance Impact on PARA*

Consider the case of Rowhammer ($K = 0$). For each activation, PARA issues a mitigation with probability p. Each mitigation performs four activations (two victims on each side of the aggressor row). Thus, the overhead of PARA is $4p$ per activation. For thresholds of 4K, 2K, and 1K, the value of p is equal to 1/84, 1/42, and 1/21. At p=1/84, the mitigation overhead of PARA is 4.76%, as shown in Figure 19.

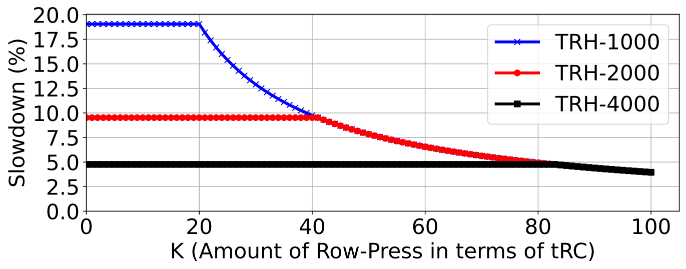

Fig. 19. Slowdown of ImPress-P with PARA for the attack pattern.

To analyze Row-Press, we vary the parameter "K". The time for each iteration is (K+1) * tRC. With ImPress-P, the mitigation probability of PARA for each loop iteration would increase proportionately to (K+1), so it becomes $(K+1)*p$. The probability can reach a maximum value of 1, so the mitigation probability for each loop is effectively $MIN(1, p*(K+1))$. Thus, the mitigation overhead of PARA becomes $4*MIN(1, p*(K+1))*tRC$ per $(K+1)*tRC$ time period, as shown in Equation 10.

$$Slowdown = \frac{4 * MIN(1, p \cdot (K+1))}{(K+1)} \tag{10}$$

Figure 19 shows the slowdown of PARA for thresholds of 4K, 2K, and 1K, as the amount of Row-Press (K) ranges from 0 to 100. We note that Rowhammer is still the most potent attack. The slowdown of Row-Press remains similar to Rowhammer until a critical point, after which the slowdown starts to reduce. This is because the loop becomes large, and PARA's probability, p, saturates at 1. Note that PARA has high mitigation overhead for both Rowhammer and Row-Press.

APPENDIX-C: ARTIFACT

A. Abstract

This artifact presents the code, traces, and methodology to reproduce the evaluation results for ImPress. Our evaluations use ChampSim, a cycle-level multi-core simulator, interfaced with DRAMSim3, a detailed memory system simulator. We provide the complete code base and all traces used in our experiments. The code base includes documentation and scripts to compile ChampSim and DRAMSim3, download traces, launch experiments, parse results, and plot graphs. Most of the simulator code is in C++, scripts for launching experiments are in Bash, meta-scripts for creating job files and collecting stats are in Perl, and plotting scripts are in Python. This artifact enables the recreation of motivation Figures 3 and 5 as well as the key result Figure 13. This artifact and simulation infrastructure have been adapted from the START artifact [37].

B. Artifact check-list (meta-information)

- **Algorithm:** RowPress mitigations – ImPress-P, ImPress-D, and ExPress – and Rowhammer mitigations – PARA, Misra-Gries, and RFM.
- **Program:** ChampSim multi-core simulator interfaced with DRAMSim3 memory-system simulator and execution traces from SPEC2017 [1] and STREAM [30] workloads.
- **Compilation:** Tested with cmake v3.23.1 and gcc v10.3.0.
- **Binary:** ChampSim simulator binary and DRAMSim3 simulator as a dynamically loaded library.
- **Data set:** Dynamic execution traces from 10 SPEC2017 and 4 STREAM workloads.
- **Run-time environment:** All experiments were run on RHEL Server 7.9 running Linux kernel v3.10.0 on x86_64 processors.
- **Hardware:** Requires many-core server with at least 4GB memory per core. We used a scale-out HPC cluster with hundreds of cores and TBs of memory.
- **Run-time state:** 4GB of memory per core is required to store the dynamic execution state of simulations.
- **Execution:** One processor core is required per workload simulation experiment. All workloads and configurations run independently and can be fully parallelized. The artifact includes 35 configurations with 20 workloads each for 700 experiments.
- **Metrics:** Graphs use normalized weighted speedup as the performance metric.
- **Output:** Recreating motivation Figures 3 and 5 and key result Figure 13.
- **Experiments:** Instructions to set-up and run experiments, parse results, and plot graphs are available in the README file.
- **How much disk space is required (approximately)?:** 4.7GB for the traces and less than 100MB for the simulator and scripts.
- **How much time is needed to prepare workflow (approximately)?:** Downloading traces might take 30 minutes to an hour (depending on network bandwidth). Compiling the simulator binaries takes less than a minute per configuration, and there are 35 configurations (so about 30 minutes).
- **How much time is needed to complete experiments (approximately)?:** Each experiment runs for about 6 hours on average, so recreating all 700 experiments requires 4.2K core hours (approximately three days on one 64-core server). Note that some experiments can take up to 12 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0.
- **Data licenses (if publicly available)?:** MIT License.
- **Workflow framework used?:** We extend run-scripts of START [37], which is a recent Rowhammer tracker.
- **Archived (provide DOI)?:** https://zenodo.org/doi/10.5281/zenodo.13743004.

C. Description

1) How to access: The ChampSim simulator code and instructions on how to evaluate the artifact are available on GitHub at https://github.com/Anish-Saxena/impress_micro2024. The traces can be downloaded from Dropbox at https://www.dropbox.com/scl/fi/qeh3rztdh4md76lhsm0u4/traces.tar.gz?rlkey=xq1yu8zithl497dnef1jpp0gv&st=r1iam07b&dl=0.

2) Hardware dependencies: The artifact requires many core server(s) to run all configurations and workloads. The 700 workload simulations are stemming from 35 configurations with 20 workloads. As all workloads can run in parallel, it would take about three days of runtime on one 64-core server. At least 4GB of memory per core is required.

3) Software dependencies: Compilation requires gcc/ g++, cmake, and make. Launch scripts use Bash. Job creation scripts require Perl, although we supply default job files (for slurm cluster manager) that can be easily adapted to the experimental system. Trace download is streamlined using Dropbox, although they can also be downloaded using wget. The plotting scripts use Python (specifically, the matplotlib library) and Jupyter Notebook.

4) Data sets: SPEC2017 and STREAM workload dynamic execution traces.

D. Installation

Please clone the GitHub repository and follow the step-by-step instructions available in the README file.

E. Experiment workflow

The workflow setup includes downloading the 14 execution traces, cloning simulator repositories, compiling simulator binaries, and making changes to run scripts (either using helper scripts or manually) as required. Once set up, experiments are launched in parallel (depending on compute resources). Finally, the simulation results are parsed, and graphs are plotted to recreate relevant figures.

F. Evaluation and expected results

The artifact provides scripts to parse the simulation results to derive the normalized weighted speedup. The relevant commands are provided in the README. The Python scripts in the Jupyter Notebook plot the relevant graphs. This artifact enables the recreation of motivation Figures 3 and 5 as well as the key result Figure 13.

G. Experiment customization

Running all of the configurations discussed in the paper requires significant computing resources (about 4,200 core hours). If compute is constrained, the experiments can be sped up by reducing the simulated instructions or by running a subset of workloads. This requires changing the run scripts and job-creation scripts.

H. Notes

Please contact the authors in case of any questions or issues.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

Acknowledgment

We thank Salman Qazi and the anonymous reviewers of MICRO-2024 for their comments and suggestions.

References

[1] “SPEC CPU2017 Benchmark Suite.” [Online]. Available: http://www.spec.org/cpu2017/

[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*, 2016.

[3] M. V. Beigi, Y. Cao, S. Gurumurthi, C. Recchia, A. Walton, and V. Sridharan, “A systematic study of ddr4 dram faults in the field,” in *HPCA*, 2023.

[4] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, “Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks,” in *IEEE SP*, 2019.

[5] 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 *HPCA*, 2022.

[6] 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 *IEEE SP*, 2020.

[7] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, “The championship simulator: Architectural simulation for education and competition,” *arXiv:2210.14324*.

[8] 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 *IEEE SP*, 2018.

[9] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer.js: A remote software-induced fault attack in javascript,” in *DIMVA*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., 2016.

[10] 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*, 2021.

[11] 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:2302.03591*, 2023.

[12] A. Jaleel, S. W. Keckler, and G. Saileshwar, “Probabilistic tracker management policies for low-cost and scalable rowhammer mitigation,” *arXiv:2404.16256*, 2024.

[13] 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.

[14] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, “BLACKSMITH: Rowhammering in the Frequency Domain,” in *IEEE SP*, 2022.

[15] JEDEC, “Jesd79-5c ddr5 sdram standard,” Apr 2024.

[16] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, M. Lipp, and D. Gruss, “CSI: rowhammer-cryptographic security and integrity against rowhammer,” in *IEEE SP*, 2022.

[17] D.-H. Kim, P. J. Nair, and M. K. Qureshi, “Architectural support for mitigating row hammering in dram memories,” *IEEE CAL*, 2014.

[18] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, “Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques,” in *ISCA*, 2020.

[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 *HPCA*, 2022.

[20] 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 *MICRO*, 2023.

[21] W. Kim *et al.*, “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 *ISSCC*, 2023.

[22] 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.

[23] A. Kwong *et al.*, “Rambleed: Reading bits in memory without accessing them,” in *IEEE SP*, 2020.

[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. Li, Z. Yang, D. Reddy, A. Srivastava, and B. L. Jacob, “DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator,” *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 110–113, 2020.

[26] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, “Rowpress: Amplifying read disturbance in modern dram chips,” in *ISCA*, 2023.

[27] 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,” 2023.

[28] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, “ProTRR: Principled yet optimal in-dram target row refresh,” in *IEEE SP*, 2022.

[29] 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.

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

[31] A. Olgun *et al.*, “ABACuS: all-bank activation counters for scalable and low overhead rowhammer mitigation,” *USENIX Security*, 2024.

[32] 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.

[33] M. Qureshi, S. Qazi, and A. Jaleel, “MINT: Securely mitigating rowhammer with a minimalist in-dram tracker,” in *MICRO*, 2024.

[34] 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 *ISCA*, 2022.

[35] K. Razavi *et al.*, “Flip feng shui: Hammering a needle in the software stack,” in *USENIX Security*, 2016.

[36] 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 *ASPLOS*, 2022.

[37] A. Saxena and M. Qureshi, “START: Scalable tracking for any rowhammer threshold,” in *HPCA*, 2024.

[38] 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 *DSN*, 2023.

[39] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, “AQUA: Scalable rowhammer mitigation by quarantining aggressor rows at runtime,” in *MICRO*, 2022.

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

[41] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, “Mitigating wordline crosstalk using adaptive trees of counters,” in *ISCA*, 2018.

[42] M. Son, H. Park, J. Ahn, and S. Yoo, “Making dram stronger against row hammering,” in *DAC*, 2017.

[43] 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 *ACM CCS*, 2016.

[44] 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 *HPCA*. IEEE, 2023.

[45] J. Woo, G. Saileshwar, and P. J. Nair, “Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems,” in *HPCA*, 2023.

[46] A. G. Yağlikçi *et al.*, “Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows,” in *HPCA*, 2021.

[47] A. G. Yağlikçi *et al.*, “HiRA: hidden row activation for reducing refresh latency of off-the-shelf dram chips,” in *MICRO*, 2022.

[48] J. M. You and J.-S. Yang, “Mrloc: Mitigating row-hammering based on memory locality,” in *DAC*, 2019.