2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems

Jeonghyun Woo
University of British Columbia
jhwoo36@ece.ubc.ca

Gururaj Saileshwar*
NVIDIA Research
gsaileshwar@nvidia.com

Prashant J. Nair
University of British Columbia
prashantnair@ece.ubc.ca

***Abstract*—As Dynamic Random Access Memories (DRAM) scale, they are becoming increasingly susceptible to Row Hammer. By rapidly activating rows of DRAM cells (aggressor rows), attackers can exploit inter-cell interference through Row Hammer to flip bits in neighboring rows (victim rows). A recent work, called *Randomized Row-Swap* (RRS), proposed proactively swapping aggressor rows with randomly selected rows before an aggressor row can cause Row Hammer.**

Our paper observes that *RRS* is neither secure nor scalable. We first propose the 'Juggernaut attack pattern' that breaks *RRS* in under *1 day*. Juggernaut exploits the fact that the mitigative action of RRS, a *swap* operation, can itself induce additional target row activations, defeating such a defense. Second, this paper proposes a new defense *Secure Row-Swap* mechanism that avoids the additional activations from *swap* (and *unswap*) operations and protects against Juggernaut. Furthermore, this paper extends *Secure Row-Swap* with attack detection to defend against even future attacks. While this provides better security, it also allows for securely reducing the frequency of swaps, thereby enabling *Scalable and Secure Row-Swap*. The *Scalable and Secure Row-Swap* mechanism provides years of Row Hammer protection with 3.3× lower storage overheads as compared to the RRS design. It incurs only a 0.7% slowdown as compared to a not-secure baseline for a Row Hammer threshold of 1200.

I. Introduction

Technology scaling has been a double-edged sword [38]. While it has enabled high-density Dynamic Random Access Memory (DRAM) chips, it has also uncovered security vulnerabilities. A key vulnerability called Row Hammer (RH) [20, 24, 30, 34] allows malicious processes to rapidly activate rows (aggressors) of DRAM cells and flip bits in their *immediate* neighboring (victim) rows [4, 11, 15, 17, 18, 27, 54].

There has been an arms race between RH attacks and defenses. To prevent RH, prior proposals tend to proactively refresh the contents of victim rows. This is called victim-focused mitigation (VFM) [15, 20, 24, 28, 44]. However, new attack patterns, such as the *half-double attack* from Google [16, 25], have shown that they could trigger RH even in distance-of-2 (or more) rows away from the aggressor row by exploiting the mitigative action of VFM. To overcome this, the state-of-the-art solution, *Randomized Row-Swap* (RRS) [51], uses an aggressor-focused mitigation mechanism. To this end, RRS swaps aggressor rows with random rows. Our paper finds that RRS is not secure. We show that, akin to the half-double attack, one can create a new access pattern by exploiting the mitigating action of RRS (the act of swapping rows) to break RRS. As a defense, our paper develops solutions that enables future-proof, secure, and scalable row swaps.

*This work was partially performed when Gururaj Saileshwar was affiliated with Georgia Institute of Technology.

Malicious processes must activate their aggressor rows above a certain threshold, to trigger RH. This threshold is called the RH threshold (T_{RH}). The RH threshold must be crossed on a single row within an epoch of a refresh window (typically 64ms) to cause bit-flips within victim rows. To prevent this, RRS proactively swaps aggressor rows with randomly chosen rows before they reach T_{RH}. The number of activations at which a row is swapped is denoted by T_S, and the ratio of T_{RH} to T_S (*i.e.*, $\frac{T_{RH}}{T_S}$) is called the 'swap rate'. The choice of swap rate has security and performance implications.

For security, the *swap rate* is chosen such that no row in memory can reach the T_{RH} number of activations within an epoch under years of attack. As shown in Figure 1, for a 32GB 16-bank DDR4-3200 system with a T_{RH} of 4800 and a swap rate of 6 (default in RRS), it would take more than 10^3 days (~3 years) for an untargeted attack to succeed (as studied in RRS). A higher swap rate is even better for security, as it increases the attack time by increasing the adversarial effort of finding the attacked rows repeatedly. So, our first goal is to investigate if a targeted attack pattern can break such defenses in under *1 day*. Our second goal is to develop a secure defense against not just the Juggernaut attack pattern but even future *unknown* attack patterns.

For performance, a lower *swap rate* is better as this reduces the memory bandwidth and latency overheads. At a T_{RH} of 4800 and a swap rate of 6, the system incurs an average slowdown of 0.3% due to swaps. But as T_{RH} drops in future generations (it has dropped 29× in 8 years [21, 24]), swaps will be needed after fewer activations, resulting in increased slowdowns and higher storage overheads to track more swapped rows. So, our third goal is to enable a low-cost swap mechanism that securely tolerates lower swap rates to minimize performance and storage overheads.

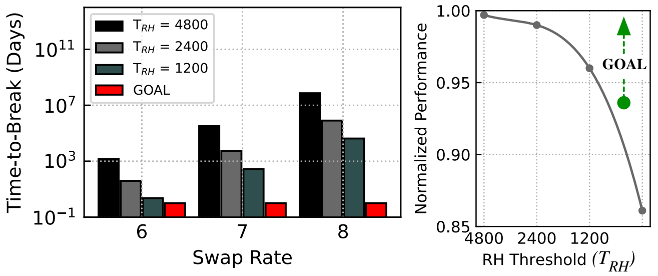

Fig. 1. (a) Time-to-break (in days) Randomized Row-Swap (RRS) with varying Swap Rate and Row Hammer Thresholds (T_{RH}). Our goal is to break RRS in under 1 day. (b) The normalized performance of RRS as values of T_{RH} vary. Our goal is to minimize the performance overheads of RRS at lower values of T_{RH} and enhance security; thereby making it scalable and secure.

Key Observation 1 – Security: The act of swapping rows, called the swap operation, itself incurs *additional* row activations to read and write the original row. These additional activations can be used to bias any target row towards higher row activation counts. In the case of RRS, which picks random rows for swaps, let us assume that the internal chip address of the aggressor row is Row$_{aggr}$ and that of the randomly chosen row is Row$_{rand}$. A swap requires separately activating both Row$_{aggr}$ and Row$_{rand}$ and copying each row to the other's locations. Thus, if we repeatedly cause Row$_{aggr}$ to be swapped to new locations, then we can increase Row$_{aggr}$'s activation count each time due to the mitigating action (*i.e.*, row swap) [1].

After a swap, if we continue to activate Row$_{aggr}$ for other T$_S$ activations, the memory controller must first *unswap* Row$_{aggr}$ before swapping it again with a newly chosen random location. The *unswap* operation itself also performs an additional row activation for Row$_{aggr}$. Thus, we can develop a targeted attack that uses a combination of *unswap-swap* operations on a single Row$_{aggr}$ to surpass the RH threshold (T$_{RH}$) within 64ms. For instance, even with 1 extra activation per *unswap-swap*, up to 1700 activations are possible for a row within 64ms, purely due to *unswap-swap* operations. This can significantly assist the demand activations, made to a row during an attack, to cross a T$_{RH}$ of 4800. Such an attack, called the *Juggernaut* attack, can break RRS in a significantly lower time (<1 day).

To defend against such attacks (and even future attacks), we propose *Secure Row-Swap (SRS)*. SRS avoids *unswap-swap* operations from biasing row activations and thereby protects against the Juggernaut attack. Moreover, we incorporate *attack detection* in SRS to detect future attack patterns. As any successful attack requires swapping a single row multiple times, we deploy swap counters for mitigated rows in SRS to flag potential attacks. Thus, SRS enables attack-detection capability for protection against even future attacks.

Key Observation 2 – Performance: At lower RH thresholds (T$_{RH}$ $\leq$ 4800), even benign workloads tend to have frequently activated rows [30], requiring frequent swaps that can cause a slowdown. While reducing the swap rate (*e.g.*, from 6 to 3) can reduce overheads and improve performance, this results in more frequent outlier rows that cross 3 swaps in an epoch (*e.g.*, once every few hours), causing potential security breaches. However, our swap-count-based attack detection mechanism can detect outlier rows, and then additional activations can be prevented by simply pinning these outliers in the Last Level Cache (LLC) for the rest of the refresh period (using <6% of the LLC). This enables extending SRS into a scalable design, called *Scale-SRS*, that can employ a swap rate of 3 at lower values of T$_{RH}$, reducing the performance and storage costs.

Contributions: This paper makes the following contributions.

1) We develop a new targeted attack pattern, *Juggernaut*, that breaks RRS by exploiting the mitigative action of row swaps (and unswap operations) in under 1 day.
2) We propose Secure Row-Swap (SRS), an RH mitigation that prevents *unswap-swap* operations and defends against the Juggernaut attack pattern. Moreover, SRS also includes attack detection to detect future attack patterns against row-swap-based RH defenses.
3) We propose Scale-SRS, a scalable solution that can securely reduce the swap rate by combining outlier-based attack detection and LLC-pinning of outlier rows as mitigation. This improves the performance, storage costs, and scalability of row-swap-based RH defenses at lower T$_{RH}$ values.

We show that Scale-SRS protects against the Juggernaut under reduced *swap rates*. Compared to a baseline system that does not protect against RH, Scale-SRS incurs an average slowdown of only 0.7%, even at the T$_{RH}$ of 1200. In a similar setup, we show that RRS can be broken in <1 day (regardless of the value of the swap rate), incurs a slowdown of >4%, and has 3.3× higher on-chip storage overheads.

[1]In spirit, this attack is inspired by the half-double attack against victim-focused mitigation. The half-double attack uses the mitigating act of refreshing neighboring rows to induce extra activations and trigger RH in farther rows.

II. Background and Motivation

A. Threat Model

We assume a target system in which an Operating System (OS) provides process isolation using virtual memory and page tables. The memory system is composed of DRAM modules that are vulnerable to Row Hammer (RH). The attacker(s) run a malicious program in the *user* privilege and activate DRAM rows rapidly. These rows, called the aggressor rows, can flip bits (by leaking charge) in their neighboring victim rows.

We assume that an attack succeeds if an aggressor row can trigger a bit-flip (*i.e.*, when it incurs more activations than the RH Threshold (T$_{RH}$) within a refresh interval of 64ms). Similar to prior work, to showcase the effectiveness of our technique, we use a T$_{RH}$ value of 4800 [51] (also lower T$_{RH}$ values to show scalability). It is the lowest demonstrated T$_{RH}$ value for any attack pattern, including Single-Sided [24], Double-Sided [54], or Half-Double [16] attack patterns.

B. Memory Organization and Timing Parameters

A DRAM-based memory system consists of independent channels that are managed by individual memory controllers. Each channel consists of ranks which are composed of several banks that operate in parallel over a common memory bus. Each bank contains rows of DRAM cells that are accessed via a *row-buffer*. The memory controller issues an activate (*ACT*) command to bring data into the row-buffer. To access another row, the memory controller must replace the existing data in the row buffer by issuing the precharge (*PRE*) command and subsequently issuing another ACT command.

Each *ACT* command leaks a small fraction of the charge within the DRAM cells of neighboring rows. DRAM cells also leak charge naturally and employ refresh operations (typically at 64ms intervals) to maintain data integrity. The time between consecutive ACT commands into the same bank is determined by the parameter t_{RC} (Row Cycle Time). t_{RC} is approximately 45ns for DDR4 systems. Thus, if we discount the time spent on refresh, a bank can experience up to 1.36 million activations (ACT_{max}) in the 64ms refresh window.

C. Row Hammer (RH) Thresholds Over Time

The attacker(s) can use RH to flip bits in victim rows by activating an aggressor row above the RH Threshold (T_{RH}). To make matters worse, the value of T_{RH} has reduced dramatically due to technology scaling. Table I shows the demonstrated values of T_{RH} across different DRAM generations. The table uses *old* and *new* to distinguish different versions of the same standard that span multiple technology nodes. The value of T_{RH} has reduced by nearly 29× in the last 8 years – specifically from 139K [24] to 4.8K [21].

TABLE I
ROW HAMMER THRESHOLD – FROM 2014 TO 2021

DRAM Generation	RH-Threshold
DDR3 (old)	139K [24]
DDR3 (new)	22.4K [21]
DDR4 (old)	17.5K [21]
DDR4 (new)	10K [21]
LPDDR4 (old)	16.8K [21]
LPDDR4 (new)	4.8K [21] - 9K [16]

In practice, attackers have used RH to flip bits in the page table and cause privilege escalation [11, 15, 17, 54]. Attackers have also used RH to read confidential data [27].

D. Tracking Rows

A key area of research has focused on developing efficient designs to track aggressor rows [28, 44, 56]. Tracking aggressor rows helps issue timely mitigation. The row trackers could be placed within DRAM chips or memory controllers [15, 23, 45]. As the tracking mechanism is orthogonal to our mitigation mechanism, it is not our main focus. We evaluate our design with the state-of-the-art trackers, Hydra [45] and the Misra-Gries tracker (used in RRS [51] and Graphene [44]), although our mitigation is compatible with any aggressor tracker.

E. Victim-Focused Mitigation

The victim-focused mitigation (VFM) refreshes the victim rows before the aggressor row receives more than T_{RH} activations [20, 23, 24, 28, 43, 56]. The number of victim rows near an aggressor row is determined by the *blast radius* [29]. If the blast radius is n (where $n > 0$), we would need to refresh *n* rows on both sides of an aggressor row.

VFM tends to have two key concerns. First, VFM mechanisms implemented in the memory controller need to know the internal chip mappings of DRAM rows, specifically the set of neighboring rows for any row. Unfortunately, this proprietary internal row mapping information is not exposed to the memory controller [23]. Alternatively, VFM methods can be implemented inside DRAM chips, but this requires an additional interface to coordinate with the memory controller. Second, as shown by the recent half-double attack [16, 25], refreshing *n* victim rows can itself cause RH on the $n{+}1^{th}$ victim row. To overcome this, recent proposals suggest using aggressor-focused mitigation. These proposals either blacklist the aggressor rows or break their spatial correlation with victim rows by displacing the aggressor rows [51, 52, 59].

F. Aggressor-Focused Mitigation: Randomized Row-Swap

Randomized Row-Swap (RRS) [51] is the state-of-the-art aggressor-focused mitigation mechanism. RRS uses the memory controller to swap aggressor rows with randomly selected rows. The activation threshold for initiating a swap is typically much lower than T_{RH} and is denoted by T_S. The fraction, $\frac{T_{RH}}{T_S}$, is called the *swap rate*. Typically, the *swap rate* is chosen such that RRS can tolerate several years of attacks. For instance, for a DRAM bank with 128K rows with a T_{RH} of 4800, RRS, with a *swap rate* of 6, can tolerate more than 3 years of attacks by an adversary continuously hammering randomly selected rows. This is because it is challenging for the attacker to guess the location of the aggressor rows as they are constantly shuffled. Since swaps impact both the security and performance of RRS, we dive deeper into its design.

Swaps and Unswaps in RRS: RRS swaps a candidate row each time it crosses T_S activations with a randomly chosen row (*swap-partner*) within the bank. If the row needs to be swapped again in the same refresh window (due to T_S more activations), the row and its current swap-partner need to be *unswapped* before they may be swapped with new partners.

1. Security Implication: Each mitigative action of *swap* on an aggressor row itself causes one additional *latent activation* at the original physical location of the aggressor row, which may be exploited by a new attack. To see why this occurs, consider the five steps in a swap operation, as shown in Figure 2:

1) First, the aggressor row, Row_{aggr} (activated by the attacker), is read out to the memory controller, as shown by ❶.
2) Then, the randomly chosen row, denoted as Row_{rand}, is activated, and this closes row Row_{aggr}, as shown by ❷.
3) The original data of Row_{rand} is read out, as shown by ❸.
4) The data of Row_{aggr} is then written into the physical location of Row_{rand}, and that row is closed, as per ❹.
5) Finally, the original location of Row_{aggr} is activated, and the data contents of Row_{rand} are written into this location. This causes a latent activation, as shown by ❺.

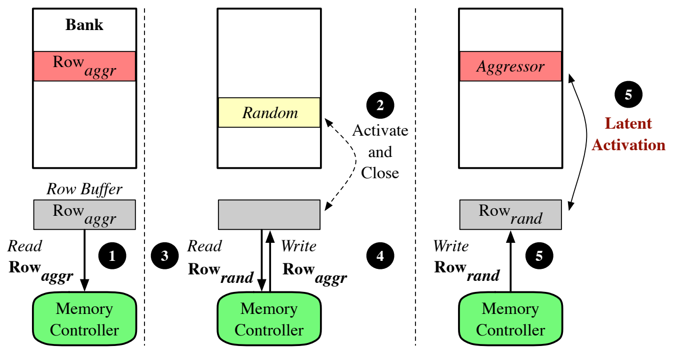

Fig. 2. The latent activation on the aggressor row caused by a swap operation. This is primarily due to the fact that it takes five steps to activate two different rows (Row_{aggr} and Row_{rand}) and thereby exchange their data contents.

Thereafter, if any one of the pairs of swapped rows continues to receive T_S activations, RRS would first unswap both these rows and then swap the aggressor row again to a new location.

All subsequent swaps for the aggressor row, within the refresh window, would be accompanied by an unswap, and together the unswap-swap operations cause up to *two* latent activations at the aggressor's original physical row. As shown in Figure 3, the first latent activation comes during the *unswap*, which copies back the swapped aggressor row to its original location (as shown in ❶). Then the *swap* of the aggressor row with the new location (Row$_{next\text{-}rand}$) also causes an additional activation to the aggressor's original location (as shown in ❷). Both steps incur extra activations because the row movements happen within the same bank, share a single row buffer, and require row-close and row-activate after each movement.

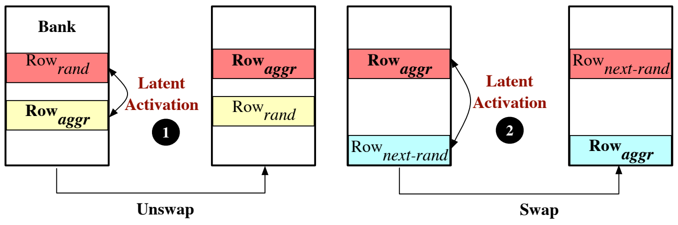

Fig. 3. Latent activations on the aggressor row caused by an unswap followed by a swap operation. These operations result in two additional activations.

Notably, if an attacker continuously activates the physical address of Row$_{aggr}$, its latent activations increases. In such a scenario, RRS issues mitigations that first cause one swap and then 'N' *unswap-swap* operations. Thus, the physical location originally storing Row$_{aggr}$ would have incurred up to $2N + 1$ latent activations. This may be exploited by a new targeted attack to increase the activations for a location by exploiting latent activations from the mitigative operations.

2. Performance Implications from Swaps and Unswaps: Unswap operations coupled with swaps are essential to ensure low-performance costs. This is because if an aggressor row is continuously swapped without first unswapping to its original location, it creates a chain of swapped rows that can introduce a large latency spike to unravel towards the end of a refresh interval. Figure 4 shows that if RRS does not employ immediate unswaps, it can cause an additional 3% - 7% slowdown on average compared to a design with immediate unswaps.

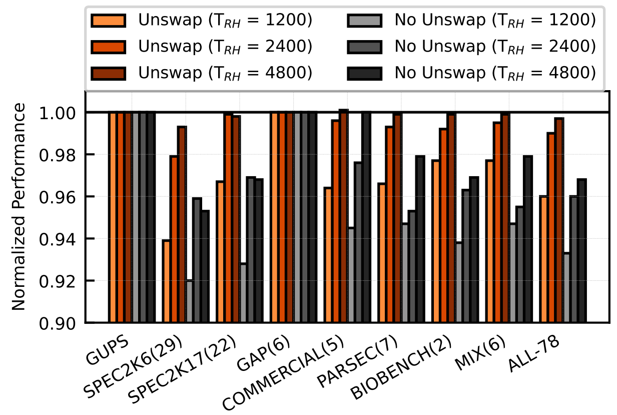

Fig. 4. The normalized performance of RRS, with and without immediate unswap operations, with respect to a baseline that does not mitigate against Row Hammer (RH). On average, not employing immediate unswap operations causes an additional slowdown of 3% to 7% at any given T_{RH}.

Consider a scenario in which Row$_A$ is swapped with Row$_B$. If Row$_A$ is continuously activated, it would need to be swapped again. Without the unswap, the new location containing Row$_A$ is now directly swapped with Row$_C$, and Row$_A$ is now in place of Row$_C$, while Row$_C$ is in place of Row$_B$, and so on. At the end of the refresh interval, all the swapped rows (Row$_A$, Row$_B$, Row$_C$, . . .) need to be placed back into their original locations. In practice, even one aggressor row can displace 1000s of random rows as it is swapped. Placing these random rows back together at the end of an epoch can cause a system to freeze up under hammering access patterns. Thus, designing a practical row swap mitigation without unswaps is non-trivial.

In the next section, we demonstrate how the latent activations of unswap-swaps can be exploited to break the defense and how a secure defense might be designed without unswap-swaps.

III. Juggernaut Attack Pattern

A. Intuition and Overview

The default attack studied in RRS employs a *random-guess* strategy, where the attacker continuously picks random aggressor rows to activate and makes T_S activations on it before it gets swapped. Eventually, the attacker hopes to repeatedly activate a single chip address by repeatedly guessing which row currently maps to it. For an RH threshold (T_{RH}) of 4800 and T_S of 800, the attacker would need to correctly guess the mapping $\frac{4800}{800} = 6$ times – essentially the *swap rate*. This attack pattern exploits the *birthday paradox* and takes years to break RRS. Rather than using *only* the birthday paradox attack pattern, we develop a more effective attack pattern, *Juggernaut*, that uses both latent activations and random guesses.

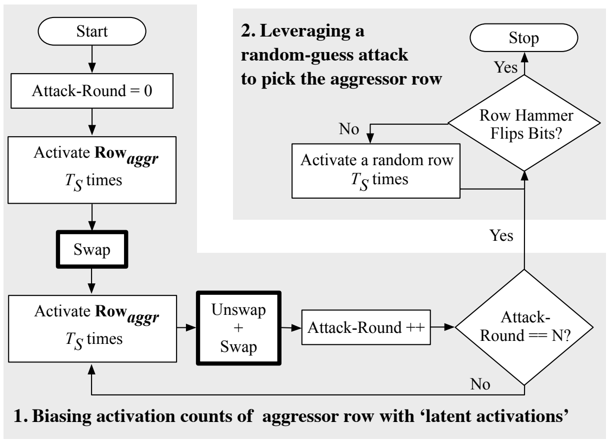
Start
Attack-Round = 0
Activate Row$_{aggr}$ T_S times
Swap
Activate Row$_{aggr}$ T_S times
Unswap + Swap
Attack-Round ++
Attack-Round == N?
No
Yes
2. Leveraging a random-guess attack to pick the aggressor row
Stop
Yes
Row Hammer Flips Bits?
No
Activate a random row T_S times
1. Biasing activation counts of aggressor row with 'latent activations'

Fig. 5. The high-level flow of the Juggernaut attack pattern. It consists of two parts. The first part biases an aggressor row with latent activations. The second part employs a random-guess attack.

Figure 5 shows the high-level flow of Juggernaut (with latent activations and random guesses). Juggernaut uses latent activations to bias activations to a single chip address and thus reduces the adversarial effort for random guesses, as follows:

1) First, we use latent activations to bias any one aggressor row towards a higher activation count. For instance, for a T_{RH} of 4800, if the aggressor row incurs 800 unswap-swaps (N), then its original chip location would have incurred 1601

$(2N+1)$ latent activations[2] (as described in Section II-F). Additionally, it would have incurred T$_S$ (800) activations before its initial swap, and in total, 2401 activations.

2) Subsequently, a *random-guess* attack only needs to land T$_S$ (800) activations 3 times on the aggressor row for it to cross T$_{RH}$ (4800) activations. As 3 is much lower than the *swap rate* (6), it enables us to break RRS quickly.

B. Analytical Model of Juggernaut Attack Pattern

We model our Juggernaut attack pattern statistically to better understand its impact. Table II shows the parameters used in its analysis. We also assume a memory controller with a closed-page policy similar to prior work [32].

TABLE II
KEY PARAMETERS USED IN THE ANALYTICAL MODEL

Parameter	Definition
N	Number of rounds of repeated unswap-swaps
L	Latent activations per round (up to 2)
G	Number of Random Guess
R	Number of Rows per Bank
t_{RC}	Row Cycle Time
t_{reswap}	Unswap-swap Latency = Reswap latency in RRS
t_{swap}	Swap Latency

Goal: For a successful RH attack, any aggressor row (Row$_{aggr}$) should incur $\geq$ T$_{RH}$ activations (ACTs).

1. Biasing an Aggressor Row with Latent Activations

We consider N attack rounds. Each round increases the latent activations of Row$_{aggr}$ by L – as shown in footnote 2, L is 1.5. Furthermore, if the attack is timed precisely, an attacker can target a row $2 \times T_S - 1$ times before encountering an initial mitigative action (*i.e.*, swap operation) that causes one latent activation. This exploits the fact that the refresh operations may not be synchronized with the reset of trackers [43, 45]. Equation 1 shows the number of activations in the aggressor row (ACT$_{aggr}$) after $2 \times T_S$ initial activations, composed of $2 \times T_S - 1$ direct activations and one latent activation, and N rounds of latent activations (L).

$$ACT_{aggr} = 2 \times T_S + (L \times N) \tag{1}$$

After N rounds, the additional activations required for Row$_{aggr}$ to cause a bit flip are denoted as ACT$_{left}$ and can be represented using Equation 2.

$$ACT_{left} = T_{RH} - ACT_{aggr} \tag{2}$$

2. Employing the Random-Guess Attack

To further activate Row$_{aggr}$, as the attacker does not know its original location, they can repeatedly choose a random row (Row$_{rand}$) and activate it T$_S$ times. Some of these choices could land on the original location of Row$_{aggr}$. The number of *swaps* (k) needed for this attack is denoted with Equation 3.

$$k = \lceil \frac{ACT_{left}}{T_S} \rceil \tag{3}$$

[2]Although a naive unswap-swap operation causes two latent activations, it is possible to optimize the unswap-swap using swap buffers in RRS (to be described in Section IV). In this case, depending on which row is selected first, a row gets either one or two additional latent activations. Thus, in this paper, we take an average of 1.5 latent activations per attack round.

t_{RC} (45ns) is the minimum delay between activations. Let us assume a 64ms refresh interval (epoch). A DRAM bank performs 8192 refresh operations during an epoch, and each operation takes t_{RFC} (350ns). Thus, only the remaining time the attacker can use (t_{actual}) is described by Equation 4.

$$t_{actual} = 64ms - t_{RFC} \times 8192 \tag{4}$$

In addition, the attacker has N attack rounds (t_{aggr}) to bias the target aggressor row towards a higher activation count. As each attack round incurs T$_S$ activations to force an unswap-swap operation, with each unswap-swap operation incurring t_{reswap} (5.4μs) latency[3], t_{aggr} can be expressed by Equation 5.

$$t_{aggr} = ((T_S - 1) \times t_{RC} + t_{reswap}) \times N \tag{5}$$

The time the attacker spends to cause an initial swap should also be considered. As the attacker could generate $2 \times T_S - 1$ activations until to cause an initial swap with t_{swap} (2.7μs) latency, the total time left (t_{left}) for employing the Random-Guess attack is denoted by Equation 6.

$$t_{left} = t_{actual} - t_{aggr} - (t_{RC} \times (2 \times T_S - 1) + t_{swap}) \tag{6}$$

The total number of possible random guesses (G) within a refresh interval (epoch) is calculated using Equation 7. Each randomly chosen row (Row$_{rand}$) is activated T$_S$ times. These rows only incur the initial swap (t_{swap}) latency. This is because most of these rows are picked only once.

$$G = \frac{t_{left}}{t_{RC} \times (T_S - 1) + t_{swap}} \tag{7}$$

Assuming a bank with R (128K) rows, a row has a probability of $p = \frac{1}{R}$ of being selected. Thus, the probability (p_{k,T_S}) of a row having been selected k times within G random guesses is described by Equation 8.

$$p_{k,T_S} = {}_G C_k \times p^k \times (1-p)^{(G-k)} \tag{8}$$

Since we only have a single target row, the expected number of iterations (AT_{iter}) and the time (AT_{time}) for a successful attack are represented by using Equation 9 and Equation 10.

$$AT_{iter} = \frac{1}{p_{k,T_S}} \tag{9}$$

$$AT_{time} = 64ms \times AT_{iter} \tag{10}$$

C. Juggernaut: Determining the Attack Rounds

Figure 6 shows the time-to-break RRS with Juggernaut for different RH thresholds (T$_{RH}$) and varying rounds of attack. We also perform event-driven Monte Carlo simulations to validate our analytical model [40, 47]. As shown in Figure 6, the results with 100,000 iterations of our Monte Carlo simulations closely match the values from our analytical model.

For a T$_{RH}$ of 4800, even after using a T$_S$ of 800 (swap rate of 6), Juggernaut takes only *4 hours* to break RRS. In contrast, the naive attack pattern using only the birthday-paradox attack (used in RRS) takes >3 years to cause RH with T$_S$ of 800.

[3]Note that, the Row Indirection Table (RIT) in RRS [51] evicts entries of the previous epoch *before* the swap or unswap-swap operations (to be described in Section IV). To enable this, the attacker can fill RIT *after* the first refresh interval (epoch).

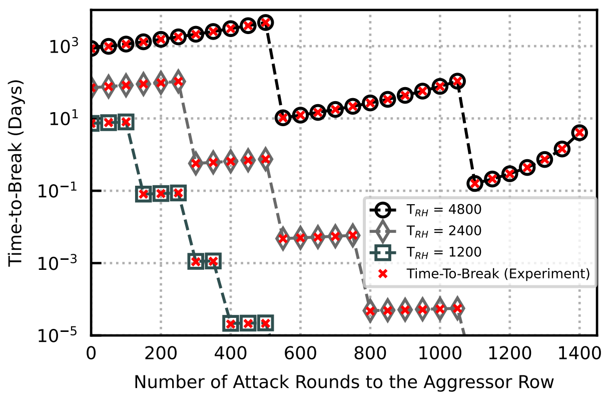

Fig. 6. Time-to-break RRS [51] with Juggernaut with varying attack rounds - both analytical and experimental results are shown. This analysis uses a swap rate of 6 for RRS. Juggernaut can break RRS in under 4 hours.

It is noteworthy to observe periodic 'steep cliffs' in the time-to-break. This is because, as shown in Equation 3, the value of k (*new swap rate*) is an integer. Thus, gradually varying the attack rounds can change the value of k from one integer value to another – which is manifested as a cliff in the time-to-break. Figure 7 shows how the number of guesses required to break RRS (k) varies with attack rounds. As we increase the attack rounds, the attacker only needs fewer guesses. At a T_{RH} of 4800, if the attacker uses ≤ 500 attack rounds, they would need to land at the original location of the aggressor row at least 4 times. In contrast, if we increase the attack rounds (say ≥ 1100), the attacker needs to guess the original location only twice. Also, within the same required number of correct random guesses (k), we see that the time-to-break increases as the attack rounds increase, as shown in Figure 6. This is because a larger number of attack rounds decreases the number of guesses (G) in Equation 7.

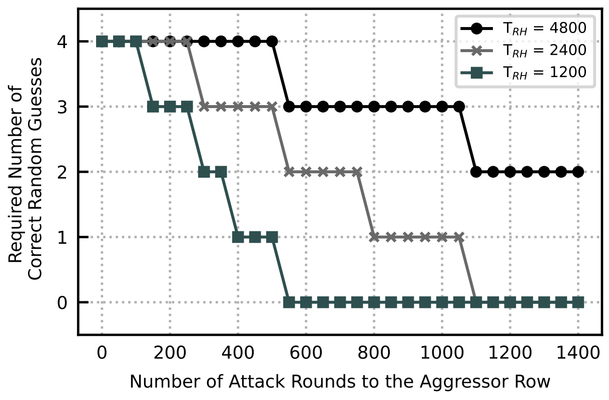

Fig. 7. The number of correct guesses required as the attack rounds vary. As the attack rounds increase, the attacker needs fewer guesses.

Hence, we pick the number of attack rounds (N) such that it minimizes the value of k, while also maximizing the number of guesses (G). For instance, at a T_{RH} of 4800, selecting N as 1100 shows the best attack performance – breaking RRS in under 4 hours. It is noteworthy to mention that, as shown in Figure 7, Juggernaut can break RRS in just 1 refresh period (64ms) using only the latent activations (unswap-swaps) at lower T_{RH} values (*e.g.*, 2400 and 1200). To make matters worse, the T_{RH} value is highly likely to drop further due to the DRAM technology scaling – T_{RH} has already dropped by 29× from 2014 to 2022. Thus, it is vital to develop a low-cost protection technique not only against the Juggernaut attack but also other unknown attack patterns.

We also analyze a multiple-bank attack, where the attacker targets multiple banks instead of a single bank. However, such an approach considerably reduces the attack effectiveness. This is because it significantly decreases the number of possible activations in one refresh interval due to bank-to-bank activation delays and row migration latencies [51]. For instance, at a T_{RH} of 4800 with a swap rate of 6, targeting all (16) banks in a channel increases the attack time from 4 hours to 9.9 years. Thus, we only focus on a single bank attack.

IV. Mitigating Juggernaut with Secure Row-Swap

A. Overview and Intuition

Secure Row-Swap (SRS) leverages the observation that latent row activations are due to the subsequent unswap and swap (*unswap-swap*) operations. As latent activations are key to the success of Juggernaut, SRS prevents latent activations by avoiding *unswap-swap* operations.

SRS observes that *unswap-swap* operations create pairs of tuples of row mappings. This implies that if Row_A maps to Row_X, then Row_X also maps to Row_A. The pairs of tuples of mappings enable RRS to immediately unswap these rows.

Unlike RRS, SRS manages row mappings such that it can only employ the swap operation. For instance, in SRS, if Row_A is repeatedly activated T_S times, it will perform a swap operation by choosing a random row (say Row_Z), thereby destroying the original tuple pair. SRS is designed to lazily unswap rows (across epochs) into their original locations by using a small per-bank place-back buffer. The lazy unswap operations help mitigate performance overheads.

B. Row Indirection Table

The Row Indirection Table (RIT) tracks row remappings in RRS. SRS also uses a modified RIT. RIT is constructed as a Collision Avoidance Table (CAT) [50]. The total number of entries in RIT ($\text{RIT}_{entries}$) depends on T_S and the maximum number of activations (ACT_{max}) in a refresh interval (epoch). Additionally, the CAT structure is over-provisioned to prevent collision-based attacks [50, 51]. Furthermore, RRS stores RIT entries as tuples to enable efficient *unswap-swap* operations.

For instance, if Row_A and Row_B are swapped, the RIT will have the tuples $< A, B >$ and $< B, A >$. If either Row_A or Row_B gets additional T_S activations, both rows are unswapped and swapped. Assuming Row_A is swapped with Row_C and Row_B is swapped with Row_D, then the RIT will now have the tuples $< A, C >$, $< C, A >$, $< B, D >$, and $< D, B >$.

A lock bit is set for both tuple entries when they are brought into RIT. The lock bits are reset at the end of the epoch. RIT randomly evicts tuples from the previous epoch to insert new tuples. RIT uses lock bits to identify if the tuples are indeed from the previous epoch.

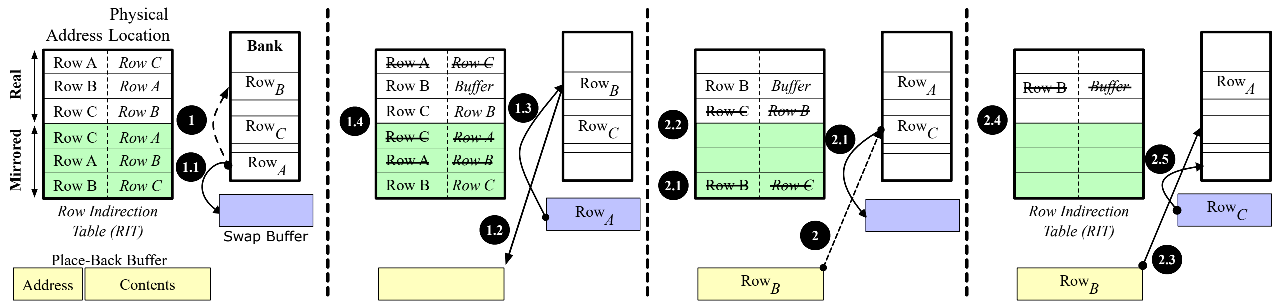

Fig. 8. An overview of the *place-back* operation and *place-back* buffer for enabling Secure Row-Swap (SRS). SRS does not require tuples of row addresses in RIT. The *place-back* buffer helps lazily store the rows that are displaced from the original location.

C. SRS: Swap-Only Row Indirection

SRS splits the RIT into two equal parts, namely, the real part and the mirrored part. Cumulatively, they have the same size as the RIT from RRS and retain the properties of CAT. The original mappings are stored in the real part, and the reverse mappings are stored in the mirrored part of the RIT.

1. Initial Swap: Let us assume that Row$_A$ swaps with Row$_B$. The original RIT now contains the tuples $< A, B >$ and $< B, A >$. The mirrored RIT contains the tuples $< B, A >$ (for $< A, B >$) and $< A, B >$ (for $< B, A >$).

2. Subsequent Swaps: Thereafter, if Row$_A$ receives T$_S$ activations again, then Row$_A$ is simply *swapped* again – *without unswapping*. Let us assume Row$_A$ now swaps with Row$_C$. The $< A, B >$ entry in the original RIT is now updated to $< A, C >$. Additionally, as Row$_C$ is now placed in the original location of Row$_B$, a new $< C, B >$ is also added. However, the original RIT still maintains the valid entry $< B, A >$.

The mirrored RIT is also updated with the reverse mappings of the entries in real RIT. Therefore, the mirrored RIT now contains $< C, A >$, $< A, B >$, and $< B, C >$. Figure 9 shows these row mappings. A key difference between SRS and RRS is that the RIT tuples in SRS do not have fixed pairs. *As there are no unswap operations, there is no latent row activation on the original location of the swapped rows.*

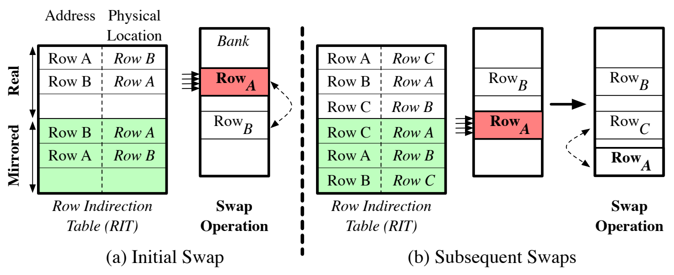

Fig. 9. The RIT (real and mirrored) provides indirections to the rows involved in the *swap* operations in SRS. The tuples in SRS do not have fixed pairs.

D. SRS: Lazy Evictions and the Place-Back Buffer

SRS employs lazy evictions of RIT entries from the previous epoch. These lazy evictions occur periodically in the current epoch. This design serves two purposes. First, the lazy evictions create space in the RIT for new entries for the next epoch. Second, due to their lazy nature, these evictions mitigate latency spikes as they are spread across the entire epoch.

SRS uses a per-bank 'place-back' buffer that holds the contents of the rows that are being evicted. Consider a scenario where RIT is performing lazy evictions for the entries of the previous epoch. If the RIT has 1700 valid entries from the previous epoch, each valid entry will be lazily evicted periodically at the rate of $\frac{Epoch_{Time}}{1700}$ (*i.e.*, $\frac{64ms}{1700}$). Note that, similar to RRS, the RIT is designed as a CAT. Thus, it can never be fully occupied and is resilient to conflict-based attacks.

As shown in Figure 8, let us assume that the RIT contains mappings for Row$_A$, Row$_B$, and Row$_C$. If Row$_A$ is lazily evicted from the RIT, as shown by 1, it will be first moved into the swap-buffers (already present in the original design of RRS [51]), as shown by 1.1. Then, Row$_B$ is copied into the *place-back* buffer. This is shown by 1.2. Row$_A$ then moved to its original location, as shown by 1.3. As the last step for the first place-back operation, the RIT invalidates the entries for Row$_A$ and updates the physical location of Row$_B$ in the *real part* as the place-back buffer This is shown by 1.4.

The next place-back operation moves Row$_B$ into its original location, as shown by 2. Similar to the first place-back operation, as shown by 2.1, it first moves the row (Row$_C$) in its original location into the swap buffer. The RIT invalidates the entries for Row$_C$, as shown by 2.2. Now, Row$_B$ is moved into its original location, as shown by 2.3. The RIT invalidates the entry for Row$_B$, as shown by 2.4. Finally, Row$_C$ is migrated to its original location, and the lazy eviction process is completed. This is shown by 2.5.

E. Security Analysis

We quantitatively analyze the security of SRS against the Juggernaut attack pattern.

Goal: For a system with Secure Row-Swap (SRS), create a successful RH attack by causing any specific aggressor row (Row_{aggr}) to incur $\geq T_{RH}$ activations (ACTs).

As illustrated in Section III, Juggernaut is composed of two parts. First, the attacker would attempt to bias any one aggressor row towards higher activation counts during N attack rounds. However, since SRS employs the *swap-only* row indirection, there are no additional latent activations on the *original location* of the aggressor row in each round. Thus, the original location incurs only 1 latent activation (ACT) during the *initial swap* operation of the aggressor row (Row_{aggr}). This is denoted by Equation 11.

$$ACT_{aggr} = 2 \times T_S \tag{11}$$

Since Row_{aggr} already has received ACT$_{aggr}$ activations, the additional activations needed to cause this row to incur Row Hammer (ACT$_{left}$) are represented in Equation 12.

$$ACT_{left} = T_{RH} - ACT_{aggr} \tag{12}$$

Thereafter, the attacker uses the random-guess attack to pick random rows and activate them T$_S$ times. We explained this process in detail in Section III-B. The time for a successful attack can be obtained by plugging Equation 12 into Equation 3.

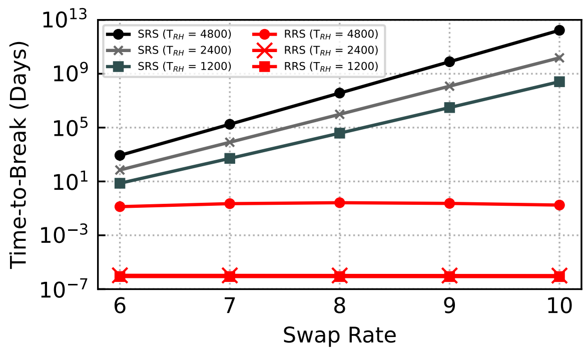

Fig. 10. Time-to-break SRS using the Juggernaut attack pattern. For T$_{RH}$ of 4800, even with a swap rate of 6, SRS has a time-to-break of > 2 years while under continuous attack. In contrast, RRS can be broken in 4 hours.

Figure 10 shows the time-to-break SRS and RRS using Juggernaut as we increase the swap rate and vary T$_{RH}$ values. For a T$_{RH}$ of 4800, even with a swap rate of 6, SRS provides robust security for >2 years against the Juggernaut attack pattern. SRS is more robust at higher *swap rates*. Unfortunately, even at increased swap rates, RRS is highly vulnerable to the Juggernaut attack pattern.

F. Future-Proofing Security by Tracking Swap Counts

To protect against any unknown future attack patterns, we future-proof SRS by adding a per-row swap-tracking counter. We reserve a small portion of the main memory to store these counters. Additionally, we also add a 19-bit on-chip register in the memory controller to count epochs. Similar to prior work, a refresh interval is divided into two epochs [44, 45]. Each counter is composed of two parts. The first part stores an epoch-id. The second part stores the cumulative activation count when a swap occurs – including any latent activations.

Figure 11 shows this design. Let us assume that a counter with 19 bits of epoch-id and 13 bits of activation count. Therefore, it can count up to 8192 activations per row (including latent activations) per epoch. The respective counter for a row is read before a swap operation. If the on-chip epoch register is different from the 19-bit epoch-id, then it indicates a different epoch window. In this case, the activation counts for that row are reset. However, if the epoch-id and the on-chip

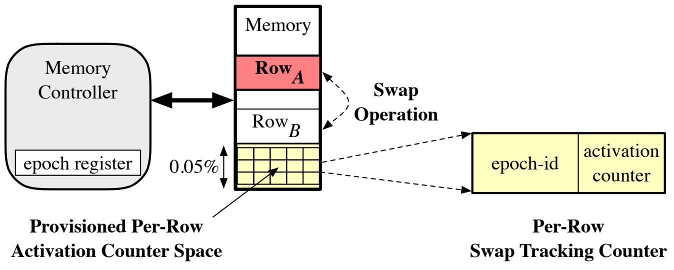

Fig. 11. A memory system with per-row swap-tracking counter. The memory controller stores an epoch register. The main memory reserves 0.05% of its space to store a per-row tracking counter. The respective counter for a row is read and updated before each swap operation.

epoch register have the same value, then activation counts are updated with T$_S$ activations along with any additional latent activation count. Once the on-chip epoch register shows all '1s', it immediately resets all the counters. This involves reading 64 counter rows every 2^{19} epochs (each epoch is 32ms) – incurring a latency of 41μs every 4.6 hours.

In terms of storage, we need only one 32-bit counter per DRAM row. Assuming we have 128K rows per bank, we would need to provision 512KB of space per bank. This represents 0.05% of the total DRAM capacity. These 512KB of counters are stored across sixty-four 8KB DRAM rows accessed only during swap operations. To prevent any recursive look-ups, the counter-rows are tracked using dedicated per-bank on-chip activation counters (similar to prior work [45]).

G. SRS: Performance and Scalability

Figure 12 compares the performance of SRS with RRS. SRS shows a similar slowdown as RRS. This is because, while SRS prevents the Juggernaut attack, it still incurs the same memory bandwidth overheads as RRS. The memory bandwidth overheads are dictated by the swap rate. As the swap rate of SRS and RRS are the same, they do not scale well towards lower values of T$_{RH}$. SRS and RRS show a variation in performance occurs due to the sub-optimal schedules of the lazy eviction mechanism and place-back operations.

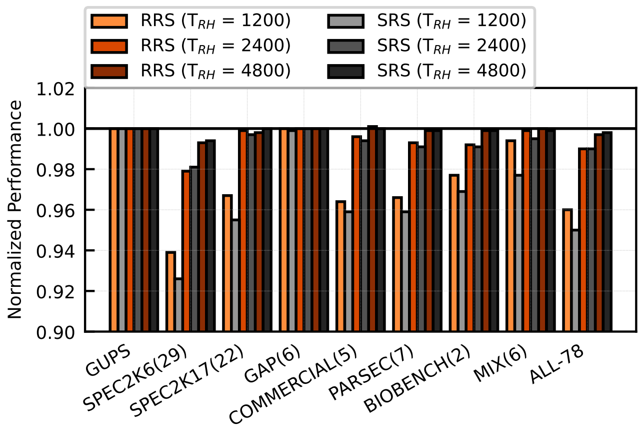

Fig. 12. The normalized performance of SRS and RRS compared to an not-secure baseline. Overall, SRS and RRS show similar slowdowns across different values of T$_{RH}$. The variation in performance occurs due to the sub-optimal schedules of the lazy eviction mechanism and place-back operations.

V. Scalable and Secure Row-Swap

A. Overview and Intuition

Scalable and Secure Row-Swap (Scale-SRS) aims to reduce the swap rate and mitigate the memory bandwidth overheads from swaps while providing years of security. To this end, Scale-SRS uses the observation that, even during an attack, the original locations of only a few aggressor rows receive multiple swaps. RRS and SRS increase the swap rate of the entire memory system *only* to take care of these outlier rows. Instead of designing for the worse-case outlier rows, Scale-SRS designs for the common case. To this end, Scale-SRS detects the outlier rows and stores them in the Last Level Cache (LLC). Fortunately, even during an attack, there tend to be only a few outlier rows every few hours or days. Thus, the LLC observes a minor capacity loss only for one refresh interval that occurs every few hours or days (in the worst case).

B. Improving Scalability by Reducing Swap Rates

Even during an attack, there are only a few such locations that stand out as outliers. This is because, within a refresh window, there are only a finite number of activations ($ACT_{max} = 1.36$ million) are possible. Assuming a T$_S$ = 1200, the attacker can only activate up to 1134 ($\frac{ACT_{max}}{T_S}$) rows T$_S$ times. Furthermore, if a T$_{RH}$ is 4800, then the attacker would need to land on the original location of any one of these rows 3 times.

Fortunately, the memory bank tends to have several rows – say between 64K-128K rows. Even during an attack, only a small fraction of these rows (1134 rows) are swapped, and they have 64K-128K locations they could be swapped into. Thus, in most refresh intervals, the original location of any attacked rows would not have been chosen more than 3 times. The intervals wherein the row is chosen more than 3 times are outliers. These occur only every few hours or days. Figure 13 shows the time to appear for these outlier rows with varying swap rates. For this analysis, we assumed a T$_{RH}$ of 4800.

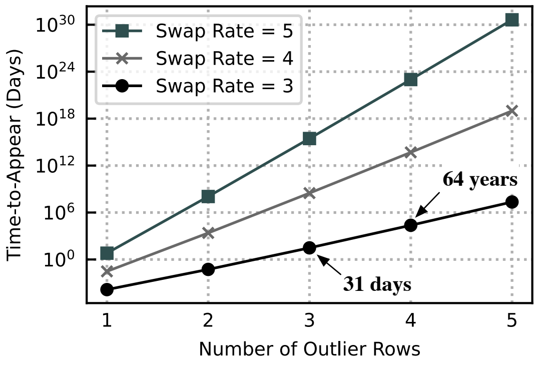

Fig. 13. The time-to-appear (in days) for outlier rows with varying swap rates for T$_{RH}$ of 4800. Even at a lower swap rate of 3, it takes at least *64 years* for 4 outlier rows with >3 swaps to simultaneously appear within a bank. Additionally, only one 64ms refresh window every 31 days showcases 3 outlier rows – thus, these outliers are very rare.

Without loss of generality, this paper chooses a swap rate of 3. We observe that three rows (as shown in Figure 13) are chosen only three times in a 31-day window[4]. We use the per-row swap-tracking counters to identify such events. If any per-row swap-tracking counter value is $\geq 3 \times T_S$, we classify its respective row as an outlier and pin it within the LLC.

C. Provisioning Space in the Last Level Cache

Assuming a T$_{RH}$ of 4800, the LLC needs to be equipped to store a maximum of 3 DRAM rows in a single bank attack (occurring once every 31 days). As each row is 8KB and an adversary targets a single bank per channel (to maximize attack bandwidth), we may need up to 3×8×1×2 = 48KB of space in the LLC. This accounts for only 0.05% of 8MB LLC.

We also analyze the multiple bank attack, as it might increase the capacity overhead in LLC. Assuming years of continuous attack, up to 3 outlier rows can appear in 11 banks per channel, which requires LLC to store 66 DRAM rows. For an 8MB LLC, this translates to a 6.5% lower capacity. However, as the multiple bank attack degrades the attack efficiency (as explained in Section III-C), *this scenario now occurs only once every **2.6 years** and only lasts for **one refresh interval (64ms)**. Thus, on average, pinning rows in LLC has a negligible impact on performance.*

As the LLC employs its own address mappings into its sets, it cannot simply pin DRAM rows. It could be likely that these rows could map the same set and thereby conflict with each other. To prevent this, Scale-SRS employs a small buffer, called pin-buffer, in front of the LLC to indicate the pinned physical addresses and redirect them into their new set locations. For instance, we would need a 66-entry buffer that stores the addresses of 66 DRAM rows. For an 8KB row, each entry would be 35 bits long (48-bit physical address - 13-bits).

Each pin-buffer entry points to a fixed set. For instance, the first entry would point to set 0. Assuming 64 Byte cache lines and an 8-way cache, we would need 16 contiguous sets to store this row. Thus, the second entry would now point to set 16, and so on. All accesses into the LLC flow through the pin-buffer, preventing any new cacheline from evicting these entries. These entries are cleared, and their respective rows are evicted once the refresh interval ends. In most 64ms refresh intervals, the pin-buffer does not contain any rows.

VI. Evaluation Methodology

Simulation Framework: We use a detailed memory system simulator USIMM [2, 9], which is modified to enforce the DDR4 protocol. The Misra-Gries tracker and the RIT are modeled as a Collision Avoidance Table (CAT) structure [51] within the memory controller. We report the performance and other related metrics from the USIMM memory model.

Table III shows the baseline system configuration. We use a DRAM configuration with 16 banks per rank and 1 rank per channel (similar to the prior work [51]) and 2 channels. Each bank has 128K rows of 8KB each and 1.36 million activations

[4]The expected number of rows with 'k' swaps for a DRAM bank that has 'R' rows (R_K) is $R \times p_{k,T_S}$. The probability of having 'M' rows with 'k' swaps ($p_{M,k}$) can be calculated with the Poisson distribution as $\frac{e^{-R_K} \times R_K^M}{M!}$

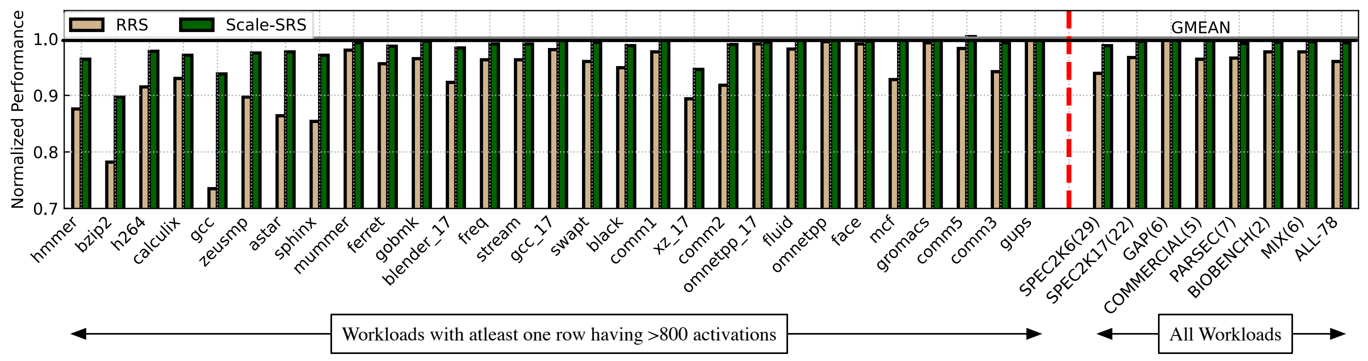

Fig. 14. The normalized performance of Scalable and Secure Row Swap (Scale-SRS) and Randomized Row Swap (RRS) compared to a not-secure baseline at T_{RH} of 1200. Scale-SRS and RRS incur an average slowdown of only 0.7% and 4% respectively, with several benchmarks in RRS incurring >10% slowdown.

possible per bank in the 64ms refresh interval. To emphasize the scalability of Scale-SRS, we evaluate against a T_{RH} of 1200 activations. We also perform sensitivity studies for T_{RH} values of 512, 2400, and 4800 activations.

TABLE III
BASELINE SYSTEM CONFIGURATION

Cores (OoO)	8
Processor clock speed	3.2GHz
ROB size	192
Fetch and Retire width	4
Last Level Cache (Shared)	8MB, 16-Way, 64B lines
Memory size	32 GB – DDR4
Memory bus speed	1.6 GHz (3.2GHz DDR)
T_{RCD}-T_{RP}-T_{CAS}	14-14-14 ns
T_{RC}, T_{RFC}, T_{REFI}	45ns, 350 ns, 7.8μs
Banks x Ranks x Channels	16 x 1 x 2
Rows per bank	128K
Size of row	8KB

Workloads: We evaluate Scale-SRS across SPEC2006 [12], SPEC2017 [57], GAP [48], BIOBENCH [3], PARSEC [5], and COMMERCIAL [9] benchmarks. We use Intel Pintool [31] to extract the SPEC2006, SPEC2017, and GAP benchmarks for representative regions. The COMMERCIAL, BIOBENCH, and PARSEC benchmark traces are obtained from the USIMM distribution. We executed each benchmark for 1 Billion instructions per core. We also create 6 mixed workloads by randomly combining benchmarks. We execute the workloads in rate mode and continue simulating the individual benchmarks until all cores complete 1 billion instructions each. For conciseness, we show detailed results only for workloads that encounter at least one row with 800+ activations within a 64ms time refresh window and report averages for all 78 workloads.

VII. Results and Analysis

A. Performance

Figure 14 shows the normalized performance of Scale-SRS and RRS with respect to a baseline that does not employ RH mitigation. To emphasize the scalability of Scale-SRS, we use an aggressively low T_{RH} of 1200. Workloads such as `hmmer`, `bzip2`, `gcc`, `zeusmp`, `astar`, `sphinx`, and `xz_17` have greater than 10% slowdown while employing RRS. In the worst case, `gcc` has a 26.5% slowdown due to frequent swaps in RRS. On average, across 78 workloads, Scale-SRS has a slowdown of only 0.7%, whereas RRS has a slowdown of 4%.

B. Sensitivity to Varying RH Thresholds

Figure 15 shows the performance sensitivity of Scale-SRS and RRS as T_{RH} varies from 4800 to 512. Even when T_{RH} drops, Scale-SRS minimizes its performance overhead since it employs a relatively lower swap rate. On the contrary, RRS incurs higher performance overhead as RRS caters to the outlier rows, which makes it swaps (and unswaps) rows at a relatively higher rate. Even at a T_{RH} of 512, Scale-SRS shows an average slowdown of only 4%, whereas RRS shows an average slowdown of 14%.

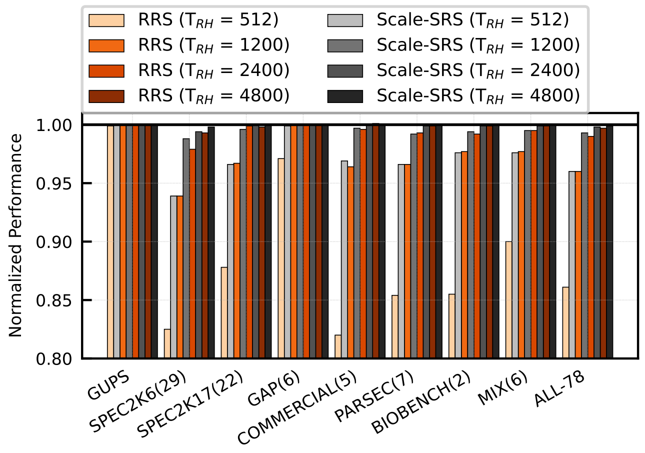

Fig. 15. The normalized performance of SRS and RRS as the value of T_{RH} varies from 4800 to 512. Even at a T_{RH} of 512, Scale-SRS shows an average slowdown of only 4%, whereas RRS shows an average slowdown of 14%.

C. Impact of Aggressor Row Tracker

Figure 16 shows the performance sensitivity of Scale-SRS and RRS if they use the Hydra tracker instead of the Misra-Gries Tracker. We vary T_{RH} from 4800 to 512. Even at a T_{RH} of 512, Scale-SRS with Hydra has an average slowdown of only 5.9%, whereas RRS has an average slowdown of 26.8%. Hydra stores its activation counters in the memory. Thus, despite using a counter cache, RRS with Hydra tends to access the memory frequently at lower T_{RH} values.

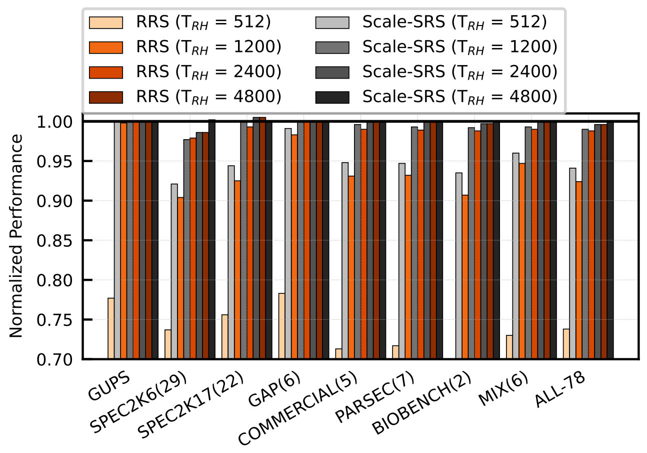

Fig. 16. The normalized performance of Scale-SRS and RRS while using the Hydra tracker. Even at a T_{RH} of 512, Scale-SRS with Hydra has an average slowdown of only 5.9%, whereas RRS has an average slowdown of 26.8%.

D. Storage Analysis

Table IV shows the required SRAM-based on-chip storage for RRS and compares that to Scale-SRS. A key difference between RRS and Scale-SRS is the reduced swap rate of 3. This enables Scale-SRS to reduce the size of the RIT.

Scale-SRS requires one additional 8KB place-back buffer per bank. Additionally, it also uses a 19-bit epoch register and a pin-buffer. The size of the pin-buffer depends on the number of outlier rows – which is determined by T_{RH}. The LLC overhead from pinning rows occurs only once every few thousand 64ms refresh intervals. Thus, it has a negligible impact on performance and is not shown in Table IV. Overall, Scale-SRS has about 3.3× less storage overhead compared to RRS at a T_{RH} of 1200.

TABLE IV
STORAGE OVERHEAD PER BANK

Structure	$T_{RH} = 4800$		$T_{RH} = 2400$		$T_{RH} = 1200$	
	RRS	**Scale-SRS**	**RRS**	**Scale-SRS**	**RRS**	**Scale-SRS**
RIT	35 KB	9.4KB	130KB	35KB	250KB	67.5KB
Swap-Buffer	1 KB	1 KB	1KB	1KB	1KB	1KB
Place-Back Buffer	-	8KB	-	8KB	-	8KB
Epoch Register	-	19 bits	-	19 bits	-	19 bits
Pin Buffer	-	289 bytes	-	420 bytes	-	420 bytes
Total	36 KB	18.7KB	131KB	44.4KB	251KB	76.9KB

E. Power Analysis

Scale-SRS incurs power overheads from extra operations such as row swaps and accesses to on-chip structures. Table V shows the power consumed by DRAM (obtained from USIMM [9]) and the SRAM structures (obtained using Cactii [33] in the 32 nm technology) in Scale-SRS and RRS. Compared to RRS, due to smaller-sized SRAM structures, Scale-SRS incurs 23% lower on-chip power. Scale-SRS also reduces the DRAM power as it reduces the effective swap rate.

TABLE V
EXTRA POWER CONSUMPTION PER CHANNEL (T_{RH} = 4800)

Type of Power Overhead	RRS	Scale SRS
DRAM Power Overhead (Row-Swap)	0.5%	0.2%
SRAM Power Overhead	903 mW	703 mW

VIII. DISCUSSION

1. Internal Chip Address versus Physical Address:
We have demonstrated Scale-SRS and RRS using physical addresses supplied by the OS. However, it is possible that the chip rows are larger. In such scenarios, the memory controller can use the chip row addresses for the RIT and swap these rows. While this requires knowledge of the internals of DRAM, this does not change our technique or the security analysis.

2. Implementing Scale-SRS Near-Memory or In-Memory:
While we have demonstrated Scale-SRS on the CPU-based memory controller, it does not prevent us from implementing this as near-memory or in-memory (within DRAM chips [8, 37]). This can help new interfaces such as CXL [13].

3. Juggernaut Attack with Open-Page Policy:
Using an open-page policy [19] for the memory controller could reduce the attack potency of Juggernaut. This is because keeping the page open can reduce the number of row activations and thereby decrease the maximum number of possible attack rounds. For instance, using open page policy at a T_{RH} of 4800 and a swap rate of 6, the time-to-break RRS using Juggernaut increases from 4 hours to 10 days. However, the advantages of using open page policy *disappear as T_{RH} decreases*. At lower T_{RH} values, Juggernaut is powerful *regardless of page policies*. For example, if $T_{RH} \leq 3300$, Juggernaut can break RRS in under 1 day, even with the swap rate of 10. Thus, developing a new protection method against Juggernaut, such as our Scale-SRS, is essential to enable the adoption of randomized-based defense in the future DRAM generations (with lower T_{RH}).

4. Possible Storage Overhead Reduction of Scale-SRS:
Although Scale-SRS has much less SRAM-based storage overhead than RRS, there is still room for storage overhead reduction. One way is to add a bit to every RIT entry to distinguish between the original and the reverse mapping. This would prevent the need for a mirrored part of the RIT and can reduce its storage overhead by almost 2×.

5. Juggernaut and Scale-SRS in Future DRAM Generations:
The T_{RH} value will highly likely drop further in future DRAM generations, making them more vulnerable to RH-based attacks such as Juggernaut and half-double. Thus, future DRAM generations would involve more features to mitigate Row Hammer. For instance, recently introduced DDR5 devices perform refresh operations 2× more frequently than DDR4 [35, 36]. However, even in DDR5 devices, Juggernaut can break RRS in under 1 day regardless of the swap rate if $T_{RH} \leq 3100$. This demonstrates the potency of the Juggernaut attack even for future DRAM generations. This also highlights the necessity of new protection methods such as Scale-SRS.

Furthermore, Scale-SRS has better scalability (*i.e.*, better performance and less storage overhead) than RRS at lower T_{RH} values. This enables Scale-SRS to be commercially viable as a defense line against RH attacks (known and unknown) for present and future DRAM generations.

IX. Related Work

A. Aggressor-Focused Mitigation

We have already described and analyzed the most closely related state-of-the-art aggressor-focused mitigation, Randomized Row-Swap (RRS), in Section II-F. Besides RRS, BlockHammer (BH) [59] is another aggressor-focused mitigation. BH exploits dual counting bloom filters to track potential aggressor rows and uses a throttling-based approach for such rows. Unfortunately, BH is vulnerable to denial-of-service (DoS) attacks. For instance, at a T_{RH} of 4800, memory requests would be delayed by approximately $20\mu s$ per activation. BH also requires complex memory scheduling policies. In comparison to BH, Scale-SRS is more efficient and has no DoS concerns. A recent work, AQUA [53], improves the performance and storage overhead of RRS by exploiting isolation instead of randomization. Specifically, AQUA reserves a dedicated region of DRAM as the quarantine region and migrates the aggressor rows into the quarantine region when the migration threshold is reached. As compared to AQUA, Scale-SRS does not need a dedicated quarantine region and relies on randomized row movement.

B. Victim-Focused Mitigation

Victim-focused mitigation (VFM) prevents RH by performing targeted refreshes on victim rows. This can be done either probabilistically (PRA [20], PARA [24], PRoHIT [56], MRLoC [60], HammerFilter [22]) or by tracking accesses to particular rows (CRA [20], CBT [55], TWiCe [28], Graphene [44], Hydra [45]). While it is effective to prevent classic RH attacks that target victims that are immediate neighbors, they are susceptible to attack patterns, such as the *half-double* attack [16, 25], that target distant neighbors. One way that VFM may adapt to defend against *half-double* is to account for neighbor refreshes in the activation counts of the tracker. However, this requires VFM to know the proprietary internal DRAM row mappings and accurate theoretical modeling of the *half-double* and blast-radius effects. To the best of our knowledge, these effects are not yet fully known.

Mithril [23] and ProTRR [32] suggest using the newly introduced Refresh Management (RFM)-based RH mitigations. These solutions are implemented inside DRAM chips and coordinate with the memory controller using the RFM interface. This approach solves the limitations of prior VFM methods (such as requiring proprietary internal DRAM row mappings or an additional interface to communicate with the memory controller). ProTRR also shows how to prevent the *half-double* attack. However, as T_{RH} becomes lower and blast-radius increases due to DRAM technology scaling, implementing these methods inside DRAM chips tends to become infeasible due to their high storage overhead.

C. ECC-Based Defenses

ECC memories can correct a small number of bit-flips [10, 39, 41, 42, 46]. Such an approach can be used to correct the bit-flips from RH. However, ECCploit [11] shows that an attacker can still cause RH by overcoming ECC protection. Synergy [49] and SafeGuard [14] provide integrity protection and can detect RH without recovering corrupted data.

D. Software-Based Defenses

Software-based defenses often require information about DRAM properties that may be proprietary or not readily accessible to software [4, 6, 26, 58]. Additionally, these solutions often incur severe performance overheads, demand intrusive modifications to system software, and only tend to be effective for certain types of attacks.

For example, ANVIL [4] employs CPU performance counters to identify RH attacks and perform refreshes to the immediate victim rows. GuardION [58] prevents RH attacks by putting a guard row between data of different security domains. In ZemRAM [26] and RIP-RH [6], isolation is provided by locating the kernel space and user space(s) in isolated parts of DRAM. Unfortunately, these solutions require proprietary internal DRAM mappings information. Other solutions, such as CATT [7], which carries out profiling of cells and blacklists pages that contain vulnerable cells to RH, can cause significant loss of memory capacity at lower T_{RH}.

X. Conclusion

As DRAM-based systems are becoming increasingly susceptible to Row Hammer (RH) attacks, a recent work called Randomized Row-Swap (RRS) proposed proactively swapping aggressor rows to break spatial correlations with victim rows. Our paper shows that RRS neither secure nor scalable. We propose Juggernaut that breaks RRS in under *1 day* regardless of the swap rate. Juggernaut uses latent activations in RRS to make a row vulnerable to RH. To overcome this, we propose the Scalable and Secure Row-Swap (Scale-SRS). Scale-SRS avoids latent activations and prevents Juggernaut. It also enables scalable RH mitigation by allowing the use of a much lower swap rate than RRS. Overall, even at an RH threshold of 1200, Scale-SRS has a 0.7% slowdown while requiring 3.3× less on-chip storage compared to RRS, which has a 4% slowdown.

Acknowledgements

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

APPENDIX

A. Abstract

This artifact covers two aspects of the results from the paper: (1) Security analysis of our Juggernaut attack against Randomized Row-Swap (RRS) and (2) Performance analysis of our Scalable and Secure Row-Swap (Scale-SRS) and RRS.

For the security analysis, a Bins and Buckets model of the Juggernaut attack is provided as a C++ program. Our program is based on event-driven Monte Carlo simulations for faster simulations. We provide scripts to compile our simulators and to recreate the results shown in Figure 6.

For the performance analysis, we provide the C code for the implementation of Scale-SRS and RRS, which is encapsulated within the USIMM [9] memory system simulator. The Scale-SRS and RRS structures and operations are implemented within the memory controller module in our artifact. We provide scripts to compile our simulator and run the baseline, Scale-SRS, and RRS for all the workloads and plot the results in Figure 14.

B. Artifact Check-List

1) Security Evaluations:

- **Algorithm:** Implementation of event-driven Monte Carlo Simulations of the Juggernaut attack in C++.
- **Compilation:** Tested with g++ (versions 9.4.0, 11.3.0), but should compile with most standard compilers.
- **Run-time environment:** Tested on Ubuntu 20.04 and 22.04, but should broadly run on any Linux distribution.
- **Hardware:** Running all simulations with 100,000 iterations for Row Hammer thresholds of 4800, 2400, and 1200 requires a single-core CPU.
- **Metrics:** Attack Time (seconds and days).
- **Output:** Results shown in Figure 6.
- **Experiments:** Instructions to run the experiments and parse the results are available in the README file.
- **How much time is needed to complete experiments (approximately)?:** 3 minutes with a single-core Intel Xeon CPU.
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7445036

2) Performance Evaluations:

- **Algorithm:** Implementation of Scale-SRS and RRS structures and operations in C.
- **Program:** The artifact assumes memory-access traces are available (filtered through an L1 and L2 cache model) for all of the benchmarks. This can be generated with any tracing tool (like Intel Pin [31] v2.12). We tested the artifact with benchmarks from SPEC-2006, SPEC-2017, PARSEC, BIOBENCH, and GAP suites.
- **Compilation:** Tested with gcc (version 11.3.0), but should compile with most standard compilers.
- **Run-time environment:** Tested on Ubuntu 22.04, but should broadly run on any Linux distribution.
- **Hardware:** Running all 78 benchmarks in parallel (78 simultaneous instances of the simulator) requires a CPU with a sufficient number of cores (64+) and memory (128GB+).
- **Metrics:** Normalized Performance (IPC).
- **Output:** Performance results shown in Figure 14.
- **Experiments:** Instructions to run the experiments and parse the results are available in the README file.
- **How much time is needed to complete experiments (approximately)?:** 15 hours on Intel Xeon CPU if all 78 benchmarks are run in parallel (7-8 hours for baseline and RRS each on our system).
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7445036

C. Access to the Artifact

The code is available at https://github.com/STAR-Laboratory/scale-srs

D. System Requirements and Dependencies

1) Requirements for Security Evaluations:

- **Software Dependencies**: C++, Python3, g++ (tested to compile successfully with the version: 9.4.0 and 11.3.0), and Python3 Packages (pandas and matplotlib).
- **Hardware Dependencies**: A single-core CPU desktop/laptop will allow 100,000 iterations of Monte Carlo simulations in 1-3 minutes.
- **Data Dependencies**: Several input values, such as the number of attack rounds and the success probability of attack in a single refresh interval (p_{k,T_S}) in Equation 8, are required to run the simulation. We generated these values following the equations in Section III-B and included the values in 'scale-srs/security_analysis/montecarlo-event/simscript/input'.

2) Requirements for Performance Evaluations:

- **Software Dependencies**: Perl (for scripts to run experiments and collate results) and gcc (tested to compile successfully with the version: 11.3.0).
- **Hardware Dependencies**: For running all the benchmarks, a CPU with lots of memory (128GB+) and cores (64+).
- **Trace Dependencies**: Our simulator requires traces of memory accesses for benchmarks (filtered through an L1 and L2 cache). We generate these traces using an Intel Pin [31] (version 2.12). However, traces extracted in the format described at the end of the README file by any methodology (*e.g.*, any Pin version) would be supported.

E. Installation and Experiment Workflow

1) Security Evaluations: The `run_artifact.sh` in the scale-srs/security_analysis/montecarlo-event folder performs all the steps required to compile, execute, collate results, and generate the results shown in Figure 6.

- **Compiles the code** using the Makefile in the scale-srs/security_analysis/montecarlo-event folder.
- **Executes the simulations** for all Row Hammer threshold values, first for `4800`, then for `2400`, and finally, for `1200`.
- **Collates the results** for all benchmarks and provides the normalized performance.
- **Reproduce the Figure 6.**

2) Performance Evaluations: The `run_artifact.sh` in the scale-srs/perf_analysis folder performs all the steps required to compile, execute, collate results, and generate the results shown in Figure 14.

- **Compiles the code** using the Makefile in the scale-srs-/perf_analysis/src folder.
- **Executes the simulations** for all benchmarks in parallel (assuming the trace files are available), first for the `baseline`, then for the `Scale-SRS`, and finally, for the `RRS` configuration.
- **Collates the results** for all benchmarks and provides the normalized performance.
- **Reproduce the Figure 14**.

F. Evaluation and Expected Results

1) Security Evaluations: The artifact provides the `get_results_4800.py, get_results_2400.py, and get_results_1200.py` files in the scale-srs/security_analysis/montecarlo-event/simscript folder. This script allows the collation of the results, and the commands to collate the successful attack time of Juggernaut against RRS are provided in the `run_artifact.sh` in the scale-srs/security_analysis/montecarlo-event folder and the README file. After the completion of the `run_artifact.sh`, the successful attack time for Row Hammer thresholds of 4800, 2400, and 1200 can be obtained as the `aggregate_trh_4800`, `aggregate_trh_2400`, and `aggregate_trh_1200` in the scale-srs/security_analysis/montecarlo-event/results folder. Also, the regenerated Figure 6 can be obtained as the `Figure6.pdf` file in the scale-srs/security_analysis/montecarlo-event/graph folder. The sample results files for all of the used Row Hammer threshold values are provided in the scale-srs/security_analysis/montecarlo-event/result folder.

2) Performance Evaluations: The artifact provides the `plot.sh` file in the scale-srs-/perf_analysis/simscript folder. This script allows the collation of the results, and the commands to collate the IPC are provided in the `run_artifact.sh` in the scale-srs/perf_analysis folder and the README file. After the completion of the `run_artifact.sh`, the normalized performance for all benchmarks can be obtained as the `data.csv` file in the scale-srs-/perf_analysis/simscript folder. Also, the regenerated Figure 14 can be obtained as the `Figure14.pdf` file in the scale-srs-/perf_analysis/graph folder. The sample results files for the baseline, Scale-SRS, and RRS configurations for all the benchmarks are provided in the scale-srs-/perf_analysis/output folder of the artifact.

G. Methodology

Submission, reviewing and badging methodology:

- https://ctuning.org/ae/reviewing.html
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

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

[2] "3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC)," 2012.

[3] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2005, pp. 2–9.

[4] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81.

[6] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Rip-rh: Preventing rowhammer-based inter-process attacks," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 561–572.

[7] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 117–130.

[8] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 568–580.

[9] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[10] C. Chou, P. Nair, and M. K. Qureshi, "Reducing refresh power in mobile devices with morphable ecc," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 355–366.

[11] 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.

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

[13] CXL Consortium, "Compute Express Link: The Breakthrough CPU-to-Device Interconnect," https://www.computeexpresslink.org/, 2022.

[14] A. Fakhrzadehgan, Y. Patt, P. J. Nair, and M. Qureshi, "Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection," in *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022.

[15] 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.

[16] Google, "Half-Double: Next-Row-Over Assisted RowHammer," https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf, 2021.

[17] 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.

[18] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

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

[20] 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.

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

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

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

[24] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

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

[26] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and compatible software protection against rowhammer attacks," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 697–710.

[27] 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.

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

[29] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci, "Stop! hammer time: rethinking our approach to rowhammer mitigations," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 88–95.

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

[31] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, jun 2005.

[32] M. Marazzi, P. Jattke, S. Flavien, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[33] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[34] O. Mutlu, A. Olgun, and A. G. Yağlıkçı, "Fundamentally understanding and solving rowhammer," *arXiv preprint arXiv:2211.07613*, 2022.

[35] P. Nair, C.-C. Chou, and M. K. Qureshi, "A case for refresh pausing in dram memory systems," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 627–638.

[36] P. J. Nair, C.-C. Chou, and M. K. Qureshi, "Refresh pausing in dram memory systems," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, feb 2014. [Online]. Available: https://doi.org/10.1145/2579669

[37] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 309–319.

[38] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield:

Architectural framework for assisting dram scaling by tolerating high error rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 72–83.

[39] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently protecting stacked memory from large granularity failures," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 51–62.

[40] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Faultsim: A fast, configurable memory-reliability simulator for conventional and 3d-stacked systems," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, dec 2015. [Online]. Available: https://doi.org/10.1145/2831234

[41] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently protecting stacked memory from tsv and large granularity failures," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, jan 2016. [Online]. Available: https://doi.org/10.1145/2840807

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

[43] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *2020 53rd Annual IEEE/ACM MICRO*. IEEE, 2020, pp. 1–13.

[44] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. Ho Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, Oct. 2020, pp. 1–13.

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

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

[47] D. A. Roberts and P. J. Nair, "Faultsim: A fast, configurable memory-resilience simulator," in *The Memory Forum: In conjunction with ISCA-41*, Jun 2014.

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

[49] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 454–465.

[50] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1379–1396.

[51] 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.

[52] S. Saroiu and A. Wolman, "How to configure row-sampling-based rowhammer defenses," *DRAMSec 2022*, 2022.

[53] 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.

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

[55] 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.

[56] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

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

[58] V. Van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "Guardion: Practical mitigation of dma-based rowhammer attacks on arm," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 92–113.

[59] A. G. Yağlıkçı, 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.

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