Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation between Aggressor and Victim Rows

Gururaj Saileshwar*
gururaj.s@gatech.edu
Georgia Tech
Atlanta, USA

Bolin Wang
bolin@ece.ubc.ca
Univ. of British Columbia
Vancouver, Canada

Moinuddin Qureshi
moin@gatech.edu
Georgia Tech
Atlanta, USA

Prashant J. Nair
prashantnair@ece.ubc.ca
Univ. of British Columbia
Vancouver, Canada

ABSTRACT

Row Hammer is a fault-injection attack in which rapid activations to a single DRAM row causes bit-flips in nearby rows. Several recent defenses propose tracking aggressor-rows and applying mitigating action on neighboring victim rows by refreshing them. However, all such proposals using *victim-focused* mitigation preserve the spatial connection between victim and aggressor rows. Therefore, these proposals are susceptible to access patterns causing bit-flips in rows beyond the immediate neighbor. For example, the *Half-Double* attack causes bit-flips in the presence of victim-focused mitigation.

We propose *Randomized Row-Swap (RRS)*, a novel mitigation action that breaks the spatial connection between the aggressor and victim DRAM rows. This enables RRS to provide robust defense against even complex Row Hammer access patterns. RRS is an *aggressor-focused* mitigation that periodically swaps aggressor-rows with other randomly selected rows in memory. This limits the possible damage in any one locality within the DRAM memory. While RRS can be used with any tracking mechanism, we implement it with a Misra-Gries tracker and target a Row Hammer Threshold of 4.8K activations (similar to the state-of-the-art attacks). Our evaluations show that RRS has negligible slowdown (0.4% on average) and provides strong security guarantees for avoiding Row Hammer bit flips even under several years of continuous attack.

CCS CONCEPTS

• **Security and privacy → Security in hardware**.

KEYWORDS

Fault-Injection Attacks, DRAM, Memory System, Row Hammer

ACM Reference Format:
Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J. Nair. 2022. 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 (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507716

*Gururaj Saileshwar was also affiliated with TU Graz during a part of this work.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507716

1 INTRODUCTION

Row Hammer attacks enable a software-based attacker to inject bit flips in DRAM locations that are not explicitly accessible by it. These attacks pose a serious security threat as they subvert the process isolation guarantees provided by modern operating systems. The phenomenon of Row Hammer (RH) [17] was first demonstrated in 2014, when a frequently accessed DRAM row was shown to cause bit-flips in the nearby rows due to disturbance errors. Subsequently, a large body of attacks [3, 8, 11, 13, 14, 19, 29] has shown that bit-flips injected by RH can be exploited in a variety of ways. An attacker can flip bits in page tables to achieve privilege escalation, *i.e.,* gain kernel privileges and access data stored at arbitrary locations. Furthermore, bit-flips from RH are data-dependent, and this can also be used to stealthily infer data stored in nearby rows [19]. Moreover, the vulnerability to bit-flips due to RH in modern devices has only increased in recent years. For example, the number of activations required on a particular aggressor row to obtain a bit-flip due to RH (termed as the *Row Hammer Threshold*) has reduced by almost 30x in the last seven years, coming down from 139K in DDR3 (in 2014 [17]) to 4.8K for LPDDR4 (in 2020 [16]).

Developing techniques to mitigate RH has been an active area of research in the software and hardware domains. Software-based defenses [3, 5, 18, 35, 36] can protect existing systems with vulnerable DRAM against specific Rowhammer attacks and exploits, but are unable to address the root cause of the vulnerability and often vulnerable to newer attacks [12, 40]. Hardware-based defenses tend to be better suited to address the root-cause of RH, and typically consists of two parts: (1) a *tracking mechanism* to identify the aggressor-rows (rows that receive frequent activations), and (2) a *mitigating-action* that is issued when the aggressor row receives a specified number of activations. While there have been several proposals for identifying the aggressor rows at low-cost (e.g. state-less PRA [15] and PARA [17], TRR [11], CRA [15], TWiCE [20], Graphene [25]), all these proposals employ the same *mitigating action* – specifically, issuing a refresh for the immediate neighbor rows, to restore the charge in the cells of these rows, deemed to be *victim rows*. We term such a mitigation as *victim-focused* mitigation.

Victim-focused mitigation suffers from two shortcomings. First, it requires knowledge of the location of the immediate neighboring rows. This is challenging to know from the memory controller in the CPU as DRAM chips use propriety addressing schemes and this mapping may not get revealed. Second, it assumes that RH attack patterns do not cause any bit-flips in any other rows beyond the ones identified as the victim rows. Unfortunately, attackers continue to develop complex access patterns that can cause bit-flips beyond the immediate neighboring rows, and such access patterns cause bit-flips even in the presence of RH mitigation.

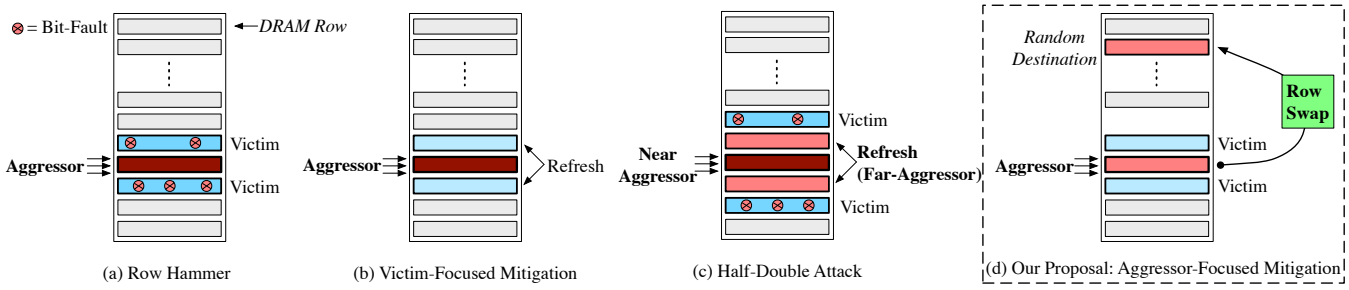

Figure 1: (a) Classical Row Hammer attack (b) Victim-focused mitigation refreshes immediate neighbors (c) Half-Double attack breaks victim-focused mitigation (d) Randomized Row-Swap breaks spatial connection between aggressor and victim row.

We explain the vulnerability of victim-focused mitigation with an example. Figure 1 (a) shows the classical RH pattern that causes bit-flips in the neighboring rows and Figure 1 (b) shows the victim-focused mitigation that issues refresh to the immediate neighboring rows. Figure 1 (c) shows the recent *Half-Double* attack from Google. Half-Double causes a large number of activations to a given row (called Near-Aggressor), which causes mitigative refreshes in neighboring rows (called Far-Aggressor). These mitigating actions aid activations to the Far-Aggressor and cause bit-flips in rows that are at a distance of 2 from the Near-Aggressor. Half-Double showed such an attack to be practical by causing 100+ bit-flips within 64ms. Moreover, mitigating Half-Double by refreshing two neighbors on each side is ineffective as the row at a distance of 3 from the Near-Aggressor could now incur bit-flips. Thus, Half-Double defeats all prior mitigation [11, 15, 17, 20, 25] relying on victim refreshes.

We expect new RH attacks will develop more complex patterns capable of bit-flips at a greater distance, even in the presence of victim-focused mitigation. This is possible because prior mitigation retains the spatial proximity between the aggressor and victim rows even after performing the mitigating action, thereby giving the attacker a large time window to orchestrate complex patterns (for example, Half-Double requires 900K activations on the same aggressor row in 64ms). The key insight in our work is to develop a *new mitigating action* that breaks the spatial connection between the aggressor-row and victim-rows, thereby limiting the time the attacker has to orchestrate the attack pattern at any one location.

We propose *Randomized Row-Swap (RRS)*, an aggressor-focused mitigating action that performs mitigation by swapping the aggressor row with a randomly selected from the same bank within the memory. As RRS limits the time an aggressor row spends around any set of victim rows, it provides a strong defense against both classical RH patterns and more complex access patterns that flip bits in rows beyond immediate neighbors. A key parameter of RRS is the number of activations an aggressor row is allowed before being swapped *(T_{RRS})*, and this parameter is decided by both the *Row Hammer Threshold (T_{RH})* and the possibility of the attacker to randomly discover the new location of the aggressor row. We perform security analysis to determine T_{RRS} and show that T_{RRS} of one-sixth of T_{RH} is sufficient to provide security for several years of continuous attack.

A design with RRS has two key components: *Hot-Row Tracker (HRT)* and *Row-Indirection Table (RIT)*. The HRT is responsible for identifying the rows that exceed a given number of activations (T_{RRS}). The RIT is responsible for tracking the location of the rows that undergo swap. The RIT is consulted on each memory access to determine the physical location for the given request.

We note that unlike prior defenses against RH that focused on tracking mechanisms to identify aggressor rows, the focus of RRS is a new mitigating action after such identification. So, RRS may be implemented with any tracking mechanism used for the HRT. Without loss of generality, we use the recently proposed *Misra-Gries tracker* from Graphene [25] for hot-row tracking in RRS.

However, designing scalable hardware structures (for the Misra-Gries tracker and RIT) in RRS is challenging with the decreasing Row Hammer threshold (T_{RH}). As T_{RH} has reduced to 4.8K activations [16], we target our RRS implementation to this value. To guarantee no row has 4.8K activations per 64ms, our security analysis suggests a row needs to be randomly swapped each time it has a multiple of 800 activations in 64 ms (T_{RRS} = 800). With 1.36 Million row activations possible in 64ms, up to 1700 rows can reach 800 activations per 64ms and need to be swapped. Therefore, both HRT and RIT must accommodate at least 1700 entries to prevent any attacker-induced evictions, a security threat.

Unfortunately, the prior design [25] for Misra-Gries tracker based HRT (targeted at 10x higher T_{RH}) uses *Content Addressable Memory (CAM)*, which is not scalable beyond few dozens of entries. Moreover, the RIT is latency-critical as it is looked up on the critical path of each DRAM access. To guarantee minimal latency and power overheads, we develop scalable designs for HRT and RIT, with low-latency lookup similar to two-way skewed-associative cache and no premature eviction of entries due to conflicts.

Our analysis shows the slowdown of RRS for benign workloads is less than 1% on average, as they typically have few rows (70 on average) with 800 or more activations in 64ms that incur swaps. In comparison, Blockhammer [37], the only other aggressor-based mitigation, which delays activations for potential aggressor rows, introduces severe denial of service that can frequently stall the application for tens of microseconds as we discuss in Section 8. On the other hand, prior victim-based mitigation [11, 15, 20, 25] are vulnerable to Half-Double and similar future attacks, unlike our design that provides principled mitigation.

Overall, our paper makes the following contributions:

(1) We propose *Randomized Row-Swap (RRS)*, a novel aggressor-focused mitigating action that breaks the spatial connection between aggressor and victim rows, providing strong security against complex Row Hammer attacks.
(2) We design RRS with the state-of-the-art tracking mechanism, and do security analysis to determine the thresholds. RRS incurs negligible performance and power overheads.
(3) To enable our design to operate at a low Row Hammer threshold of 4.8K, we also develop scalable hardware structures that can hold thousands of entries while meeting the latency and eviction requirements of our design.

2 BACKGROUND & MOTIVATION

2.1 Threat Model

We assume an unprivileged attacker that can run code natively on the system. We assume the system has a typical modern Operating System (OS) with virtual memory and page tables providing process isolation. But the system uses DRAM main memory hardware that is vulnerable to bit-flips due to Row Hammer (RH) phenomenon. The attacker can run process(es) under *user* privilege and exploit RH to flip bits in the page-table and achieve privilege escalation or simply flip bits in another program's data to corrupt it. We generically assume an untargeted attack, where the attack succeeds if it causes a bit-flip in any DRAM location (which is harder to defend compared to targeted attacks). We assume the RH bit-flip can occur at any unspecified victim location when a row incurs more activations than the Row Hammer Threshold (T_{RH}) within a refresh interval of 64ms. We use a T_{RH} value of 4.8K as it is the lowest known value for any attack pattern (single-sided, double-sided, or Half-Double attack). Lastly, we assume the attacker does not have physical access to the system and cannot probe the DRAM or the DRAM bus to arbitrarily identify the mapping of addresses to DRAM physical rows.

2.2 DRAM Organization and Timing Parameters

DRAM modules consist of multiple *banks*, which can be operated in parallel and share a common data bus. Internally, the banks are organized as a two-dimensional array of rows and columns. To access data from DRAM, a row must be activated using the *ACT* command, which brings the data into a *row buffer* for the bank. If the memory controller needs to access data in another row of the same bank, it must first clear the row-buffer using the *precharge* command, followed by activation of the new row. DRAM cells leak charge and require periodic refresh operations to maintain data integrity. Memory systems typically use a refresh period of 64ms.

An important DRAM timing parameter is t_{RC} *(Row Cycle Time)*, which indicates the time between consecutive activations in a given bank. The t_{RC} for DDR4 systems is approximately 45ns, which means a bank can encounter up to 1.36 million activations (ACT_{max}) in the refresh window of 64ms if we discount the time spent in refresh.

2.3 Row Hammer and Security Implications

Row Hammer (RH) is a failure mode that occurs when a row undergoes a large number of activations, which cause bit-flips in nearby rows due to charge leakage. The frequently activated row is referred to as the *aggressor* row, and the neighboring rows (candidate for bit-flips) are referred to as the *victim* rows. *Row Hammer Threshold (T_{RH})* denotes the number of activations required on the aggressor row to cause bit-flips in the victim rows. Table 1 shows the T_{RH} for different DRAM generations over the last 7 years. As a given standard can span multiple technology nodes, we use *old* and *new* to distinguish different versions. When the RH was first characterized in 2014, T_{RH} was 139K, whereas it has reduced by an order of magnitude to 4.8K [16] – 9K [12] in 2020.

Table 1: Row Hammer Threshold Over Time

DRAM Generation	RH-Threshold
DDR3 (old)	139K [17]
DDR3 (new)	22.4K [16]
DDR4 (old)	17.5K [16]
DDR4 (new)	10K [16]
LPDDR4 (old)	16.8K [16]
LPDDR4 (new)	4.8K [16] - 9K [12]

Bit-flips caused by RH are a severe security problem. RH gives the attacker a powerful weapon to potentially flip any arbitrary bit in the memory system, and the attacker can use it to flip bits in Page-Tables and cause privilege escalation [8, 11, 13, 29], or use the data-dependent nature of RH to read confidential data [19].

2.4 Proposals for Mitigating Row Hammer

Mitigating Row Hammer is an active area of research in both the hardware and the software community (we discuss related work in detail in Section 8). In this paper, we focus on hardware-based mitigation. Such mitigations can be classified as global mitigation or precise mitigation.

Global-mitigation [21] is performed by increasing the refresh rate of the entire memory to avoid RH. Unfortunately, this is not a viable method for tolerating RH at thresholds below 30K, as we would need to refresh the memory in less than 2.8 ms (which is the minimum time it takes to refresh the memory even if the memory was doing only refresh 100% of the time).

Precise-mitigation [11, 15, 15, 17, 20, 25] consists of: (1) a *tracking mechanism* that identifies the rows that receive frequent activations, (2) a *mitigation policy*, which is invoked when the activation counts for an aggressor row reaches a particular threshold. The tracking policy is based on the T_{RH}, and there are various probabilistic or tracking schemes to identify which rows must be deemed as aggressor rows. However, the mitigation policy typically used in precise method is to refresh the *immediate neighbors* of the aggressor row. We call such mitigation as *victim-focused mitigation*.

Victim-focused mitigation requires identifying the location of the victim row for a given aggressor row. Unfortunately, DRAM chips often use proprietary mapping, and this mapping may not be available within the memory controller. Furthermore, victim-focused mitigation relies on the assumption that the adversary cannot cause bit-flips beyond the immediate neighbors.

2.5 Defeating Victim-Focused Mitigation

RH mitigation techniques are implicitly designed with particular assumptions around the attack patterns and *Blast Radius* (the distance of the victim rows from the aggressor) [22]. A solution developed for a particular attack pattern may become vulnerable when the attacker employs a more complex pattern that can increase the blast radius. This mode of vulnerability poses a continuing risk for current and future solutions that rely on victim-focused mitigation.

Recent work from Google, Half-Double [12], discloses an attack pattern that targets victim rows at a *distance-of-two* away from the aggressor rows. Such an attack pattern, shown in Figure 1 (c), can cause bit-flips even in the presence of victim-focused mitigation techniques. For example, Half-Double is able to cause more than a hundred bit-flips in LPDDR4 modules at a distance of 2 away from the aggressor rows. The key insight in Half-Double was to use the existing RH mitigation itself as a source of activations for the immediate neighbor aiding the attacker activations, such that bit flips are injected at a distance 2 away from the original aggressor row. As DRAM cells scale, it is reasonable to assume that an adversary can target victim rows even farther away from aggressor rows. Therefore, relying on victim-focused mitigation remains an unsafe option for RH mitigation.

2.6 Goal: Enable Aggressor-Focused Mitigation

All existing RH mitigation retains the spatial ordering between the aggressor row and the victim rows, thus giving the attacker a large time window to orchestrate complex pattern. The insight in our work is to reduce the time that a given aggressor row spends in a neighborhood, thereby breaking the spatial correlation between the aggressor and the victim. Such *aggressor-focused* mitigation can avoid the security risks emanating from complex access patterns. We develop a simple and practical aggressor-focused mitigation that perform row swap between the aggressor row and another randomly selected row within the same bank. We discuss our experimental methodology before presenting our solution.

3 EVALUATION METHODOLOGY

Simulation Framework. We use USIMM [7], a detailed memory system simulator, which models the DDR protocol, refresh, and scheduling policies, and was used in the Memory Scheduling Championship [1]. We modified USIMM to enforce the DDR4 protocol. We implement the HRT and RIT in RRS within the memory controller. We report the performance and related metrics from USIMM, the DRAM power based on USIMM's power models, and SRAM power from Cacti 6.0 [24] with 32 nm technology.

Our memory controller in USIMM uses a First-Come-First-Serve (FCFS) scheduling policy. Table 2 shows the configuration for our baseline system. We use a DRAM configuration with 16 banks per rank and 1 rank per channel (similar to prior works [25, 37]) and 2 channels. Each bank has 128K rows of 8KB each and 1.36 million activations possible per bank in 64ms.

Workloads and Figure-of-Merit. We evaluate our design across SPEC2006 [9], SPEC2017 [33], GAP [26], BIOBENCH [2], PARSEC [4] and COMMERCIAL [7] benchmarks. The SPEC2006, SPEC2017, and GAP benchmarks traces are extracted using Intel Pintool for representative regions of execution, while COMMERCIAL, BIOBENCH, and PARSEC benchmark traces are used from the USIMM distribution. We executed each benchmark for 1 Billion instructions per core. We also create 6 mixed workloads by combining randomly selected benchmarks. We run the workloads in rate mode and continue executing these benchmarks until all cores complete 1 billion instructions each. For brevity, we show detailed results only for workloads that encounter at least one row with 800+ activations within a 64ms time window and report averages for all 78 workloads. Table 3 shows the memory footprint, Misses Per 1000 Instructions (MPKI), and the average number of rows that encounter > 800 row activations (ACT-800+) within 64ms window.

Table 2: 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

Table 3: Workloads Characteristics (with Rows ACT-800+)

Workload	Footprint (GB)	MPKI	Rows ACT-800+
hmmer	0.01	0.84	1675
bzip2	2.41	5.57	1150
h264	0.05	0.52	1136
calculix	0.16	1.12	932
gcc	0.09	4.42	818
zeusmp	0.55	2.00	405
astar	0.04	1.04	352
sphinx	0.13	12.90	242
mummer	2.17	19.13	192
ferret	0.79	5.67	132
gobmk	0.2	1.17	79
blender_17	0.24	1.53	53
freq	0.59	2.89	44
stream	0.63	3.48	41
gcc_17	0.36	0.55	38
swapt	0.76	3.52	37
black	0.55	3.08	37
comm1	1.55	5.93	19
xz_17	0.64	5.12	12
comm2	3.37	6.14	8
omnetpp_17	1.55	9.81	7
fluid	0.99	2.70	7
omnetpp	1.1	17.24	5
face	1.1	7.18	3
mcf	7.71	107.81	2
gromacs	0.06	0.58	1
comm5	0.67	1.48	1
comm3	1.77	2.84	1

4 RANDOMIZED ROW-SWAP

We propose *Randomized Row-Swap (RRS)*, an aggressor-focused mitigating action that breaks the spatial correlation between the aggressor and the victim. In this section, we provide an overview of RRS, explain the structures required for RRS, and present some key results for the number of rows swapped and the slowdown. We perform security analysis to determine the parameters for our design in Section 5.

4.1 Overview of RRS

Figure 2 shows the overview of RRS. The two main structures for RRS are the *Hot-Row Tracker (HRT)* and the *Row-Indirection Table (RIT)*. The HRT keeps track of rows that receive frequent activation and identifies rows that have received more than a given number of activations. These rows are candidates for undergoing row-swap. The RIT keeps track of the rows that have undergone swap and their destination locations. Each access first checks the RIT to see if it should be sent to the original location or the remapped location.

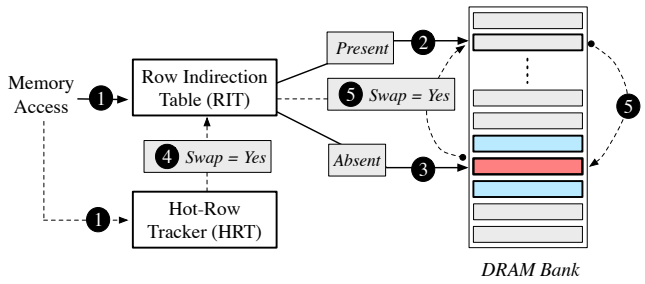

Figure 2: Overview of the Randomized Row Swap (RRS). The Row Indirection Table (RIT) is checked to determine if the access should go to original or remapped location. The Hot-Row Tracker (HRT) identifies rows that must undergo swap.

Figure 2 also shows the flow of events for RRS when a memory access occurs. ❶ Each memory access indexes the RIT and the HRT in parallel. ❷ If the request is present in the RIT, then it is redirected into the swapped location. ❸ If the request is absent in the RIT, then it is directed into its original location. ❹ The HRT could deem the request to be a potential aggressor and recommend swapping the requested row. ❺ In such an event, the RIT swaps the original address of the memory access with a randomly chosen destination from the same bank (while avoiding the random selection of rows that are already being tracked by the RIT and the HRT).

We define the refresh window of 64ms to be an *Epoch*. The HRT is reset at the end of every epoch to ensure that row accesses only within the current epoch are used for determining the eligibility for swaps. We do not do a bulk reset for the RIT as this would cause a torrent of un-swap operations for all the RIT entries. Instead, we let the RIT drain out lazily by replacing entries installed in a previous epoch (currently invalid) as and when new entries are inserted (entries installed in the current epoch are not evicted).

A key parameter for RRS is the threshold (T_{RRS}) number of activations allowed for a given row before it becomes eligible. This threshold must be kept lower than the Row-Hammer threshold to ensure security (more details in Section 5). This threshold determines the number of entries in the HRT and the maximum rate at which the rows can undergo swap, which determines the number of entries in the RIT. In our paper, we target a Row-Hammer threshold of 4.8K and our analysis (in Section 5) shows that T_{RRS} of 800 is sufficient for security.

In the next few sections, we describe the design of the HRT and RIT, the row-swap operations and its impact on performance.

4.2 Hot-Row Tracker (HRT)

The HRT is responsible for identifying rows that have activation counts equal to (or multiple of) (T_{RRS}). We note that RRS is a mitigating action and not a specific tracking technique, therefore it can be implemented with any tracking mechanism.[1] Without loss of generality, we implement HRT with the state-of-the-art Misra-Gries Tracker (proposed in Graphene [25]) as it is *guaranteed* to track all rows with activation counts greater than T_{RRS} using just $E = \frac{ACT_{max}}{T_{RRS}}$ entries, where ACT_{max} is the maximum number of activations within the 64ms window.

Figure 3 describes the operation of a 3-entry Misra-Gries Tracker (tracker). Each entry contains row addresses and *estimated* access count. When Row-A arrives, as it is present in the tracker, the access count is incremented from 6 to 7. Next, when Row-B arrives, as it is not present in the tracker, the minimum value of all the access counters is compared to the spill-counter. As the minimum access counter value is 3 and the spill counter value is 2, only the spill-counter is incremented. Thereafter, when Row-C arrives, as it is also not present in the tracker, the minimum value of all the access counters is compared to the spill-counter. As the minimum access counter value is equal to the spill-counter, the entry with minimum access counters (Row-X) is replaced with Row-C and its access count is incremented to 4. For more detailed operation and bounds we refer the reader to [25].

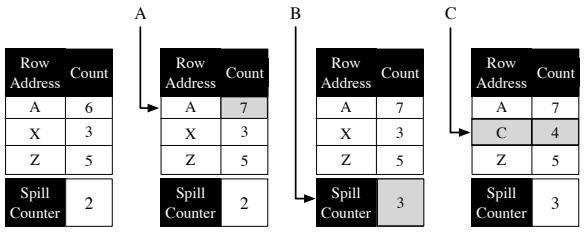

Figure 3: Operation of Misra-Gries Tracker with 3-entries. If the requested address is present, increment the count. Else, increment the spill-counter (if no ties for count and spill-counter) or install the address (with count=spill-counter+1).

The tracker can identify all rows that exceed T_{RRS} within the current epoch and issue a swap for a row whenever the access count associated with the row crosses an integer multiple of T_{RRS}. So an attacker could attempt to cause a large number of random

[1] In fact, one could have a probabilistic version of RRS, similar to PARA [17], where the row-swap is triggered with probability p on each row activation. Unfortunately, the rate of swap with such state-less methods is much higher than with a tracker, making them unsuitable for low Row-Hammer Threshold (such as 4K). These designs would be viable if the Row-Hammer Threshold were more than an order of magnitude higher.

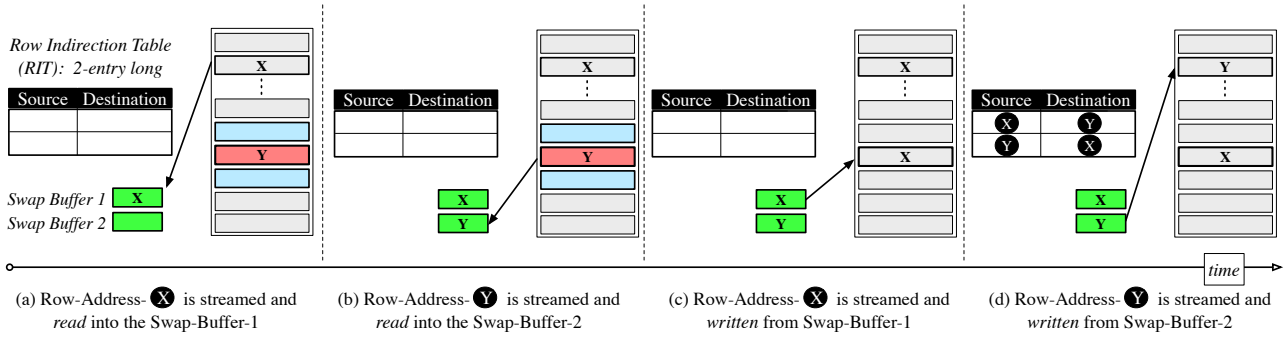

Figure 4: An overview of the row-swap for Row-X and Row-Y using the two Swap-Buffers. The RIT is updated at the end.

swaps to ensure some physical rows get multiple swaps and cross T_{RH} activations. Our analysis in Section 5 studies such attacks and determines $T_{RRS} = 800$ is sufficient for a T_{RH} of 4.8K.

4.3 Row Indirection Table (RIT)

When the access count for a row in the HRT crosses (an integer multiple) of T_{RRS}, our design initiates a row-swap to prevent further damage. For rows that undergo swap, any future access must be routed to the new destination. This is facilitated by the *Row Indirection Table (RIT)*. When two rows (say Row-X and Row-Y) get swapped, the tuple ⟨X,Y⟩ is stored in the table. To enable quick lookups, we store two entries: one indexed by X and provides Y, and the other indexed by Y and provides X, as shown in Figure 4(d).

We do not reset the RIT at the end of the epoch as this can cause a torrent of row-swaps corresponding to all the valid entries in the RIT. Therefore, we use lazy eviction out of the RIT as other entries get installed. When an entry is evicted out of the RIT the dual entry corresponding to the same tuple is also evicted. Upon eviction from RIT, the rows are un-swapped. The RIT must ensure that entries that are installed in an epoch are retained in the RIT at least until the end of the Epoch. To achieve this, we add a *lock-bit* with each RIT entry that indicates that the entry must not be selected for eviction. When an entry is installed the lock-bit of that entry is set. At the end of the epoch, the lock-bit of all entries are reset, so that these entries are eligible for eviction. The RIT is sized with enough capacity to install the maximum number of row-swaps that can occur in an epoch (as dictated by T_{RRS}).

4.4 Support for Row Swapping

When a row is identified to undergo swap, the destination row for the swap operation is picked randomly from all the rows in the bank. We exclude rows tracked by the HRT (as these rows may be swapped soon) and the RIT (as these rows are already under swap) from being the random destination. We observe that a typical memory bank has a large number of rows (128K rows in our 16GB DIMM), but only a few rows can receive sufficient accesses to be eligible for swap (a maximum of 1700 rows for T_{RRS}=800 in a window of 64ms). Thus, more than 98% of the rows are suitable (rows with less than T_{RRS} activations) to be chosen as random swap destinations; if our first random choice is present in the RIT or HRT another randomized location is re-generated and checked. The probability of more than 1 such re-generation is < 1%.

The random swap destinations are generated using a hardware pseudo-random-number-generator (PRNG). This is accomplished by a low-latency cipher (64-bit PRINCE cipher has < 2ns latency [28]) in CTR-mode with a 64-bit cycle counter as input. The output of the cipher provides cryptographically-secure random values at negligible overheads.

To facilitate row-swap, we equip each channel with two SRAM-based Swap-Buffers, each having the same as the DRAM row (8KB in our study). Figure 4 shows the row-swap for Row-X with Row-Y. To perform row-swap, the content of Row-X is streamed and stored into Swap-Buffer-1 (Figure 4(a)), then the content of Row-Y is streamed and stored into Swap-Buffer-2 (Figure 4(b)), then the content of Swap-Buffer-1 is streamed and written to Row-Y (Figure 4(c)), and finally the content of Swap-Buffer-2 is streamed and written to Row-X (Figure 4(d)). The RIT is updated with ⟨X,Y⟩.

To perform row-swap efficiently, we leverage streaming accesses from DRAM, whereby accesses to the same row can be serviced quickly (one 64 byte line every 4 DRAM bus cycles, after the first line) once the row has been activated. To transfer the 8KB row (128 lines), we would need 512 bus cycles (at 1.6GHz bus frequency for our DDR4-3200 memory) after the activation time (45 ns ACT-to-ACT delay). For our system, it takes approximately 365 nanoseconds to transfer the row between DRAM and the Swap-Buffer. As a row-swap requires four transfers, it takes approximately 1.46 microseconds to do a row-swap. As the bus is shared by all banks in the channel, no memory request for the given channel can be serviced during the row transfer operation (fortunately, the frequency of row-swap is negligibly small for benign workloads).

An install of a new tuple in the RIT may evict a tuple and trigger an *un-swap* of the rows in that tuple (the operations of row-unswap is identical to row-swap). Thus, marking a row for swap may trigger two back-to-back swap operations and incur a total latency overhead of approximately 2.9 microseconds (our design provides two registers to store the evictions out of RIT). An adversary can also do enough activations for a swapped-row in RIT to trigger a *re-swap* operation, in which the existing tuple ⟨X,Y⟩ in RIT get swapped with two different rows (⟨X,A⟩ and ⟨Y,B⟩), requiring a total latency overhead of 2.9 microseconds. Finally, in an extreme case, the re-swap operation might also require evicting a tuple, that was installed in the RIT in the previous 64ms: this can incur a worst-case latency overhead of 4.4 microseconds. This latency can be avoided by periodically draining some of the RIT entries.

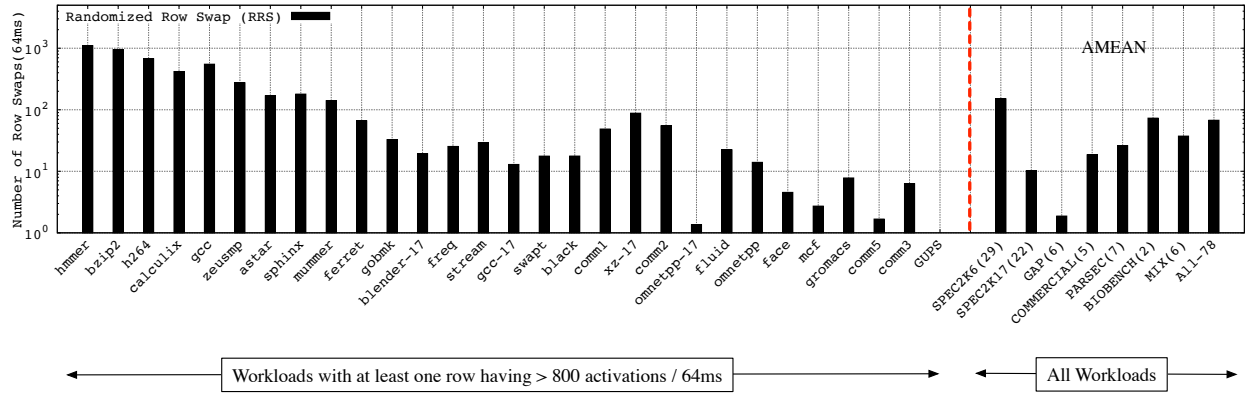

Figure 5: Average number of row-swaps per 64ms (detailed results shown for only for the 28 workloads that have at-least one row-swap, other 50 workloads do not encounter row-swap). Bars on right are arithmetic mean over respective workload suites.

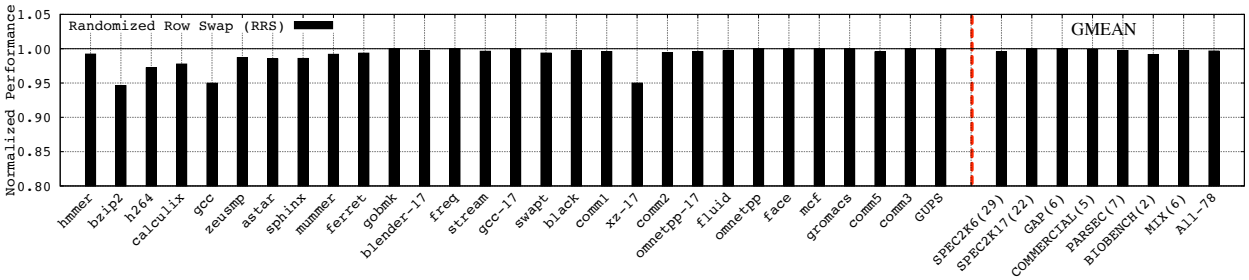

Figure 6: Performance of RRS normalized to the baseline. Bars on the right are geometric mean over respective workload suites.

4.5 Threshold and Impact on Structure Sizes

The threshold T_{RRS} determines the size of the structures and the maximum rate at which row-swap can be performed. We need T_{RRS}=800 for a Row-Hammer threshold of 4.8K to guarantee sufficient security, as we show in Section 5. To support this threshold, the Misra-Gries tracker must contain 1700 entries (ACT_{max} = 1.36 million activations). A row can be selected for row-swap by the tracker at most once every 800 activations, therefore the RIT would observe at most 1700 swap requests and at most 1700 new RIT-tuple installs. If a swap is requested for a row which was already swapped in the current epoch, such re-swaps require two RIT-tuples (one for swapping the row, and one for keeping its previous location also swapped with a new destination). So, the maximum number of RIT-tuples that can be used in an epoch is 3400 and we size the RIT with a capacity of 3400 tuples. We discuss scalable hardware implementations for the Misra-Gries tracker and RIT in Section 6. For this section, we assume scalable implementations exist and analyze the number of row-swaps and slowdown.

4.6 Results: Number of Row-Swaps in 64ms

A typical row-swap (including the un-swap due to RIT evictions) incurs a latency of 2.9 microseconds. Over a window of 64ms, there are 22K row-swaps possible per channel. Figure 5 shows the average number of row-swaps incurred for each workload over 64ms with RRS. Note that y-axis is log-scale. The average number of row-swaps across all 78 workloads is 68 (or 34 per channel), indicating only about 0.1ms of the 64ms is spent on row-swap. Only `hmmer` and `bzip2` have close to 1000 row-swaps as these workloads continuously access a working-set slightly larger than the last-level cache. Workloads with large footprint (`mcf` and `GAP`) have less than 5 row-swaps as their accesses are spread over many rows. Thus, the workloads with large footprint size may not necessarily correlate with higher number of swaps, as it would depend on how focused the activations are on individual rows.

4.7 Results: Performance Impact of RRS

RRS impacts performance in two ways: (1) the latency of RIT is incurred on each memory access (2) the memory channel can be busy doing row-swap and is unavailable for service. We add a 4-cycle latency for RIT access. Figure 6 shows the normalized performance of RRS. The average slowdown (measured over all 78 workloads) is 0.4%. The slowdown depends on both the number of row-swaps (Figure 5) and the MPKI (Table 3). For example, both `hmmer` and `bzip2` have similar number of row swaps, however, `bzip2` has an MPKI of 5.6 whereas `hmmer` as an MPKI of 0.84, therefore `bzip2` has a much higher overhead than `hmmer` as it is more memory intensive. Among the workloads with the highest slowdown of around 5%, `bzip2` and `gcc` have high swap counts, while `xz-17` has a relatively high MPKI. Overall, as the number of swaps is low in benign workloads, RRS has a low performance impact.

5 SECURITY ANALYSIS

In this section, we analyze the security of our design. We first discuss the assumptions of our solution and the invariant properties of our defense. We then describe an optimal strategy an adversary might use to launch a Rowhammer attack against our defense and finally determine the design parameters (RRS Swap Threshold or T_{RRS}) required to mitigate such attacks.

5.1 Assumptions

The only assumption for our defense is the following:

> A successful row hammer attack (using any attack pattern) requires activating at least one row more than T_{RH} (4.8K) times within a refresh period.

We make no assumptions regarding the attack pattern used or the placement of the attacker rows with respect to victim rows to keep our defense robust to current and new attacks. By aiming to prevent an attacker from reaching T_{RH} (4.8K) activations on any row, our defense encompasses all current attacks, including single-sided and double-sided rowhammer (which require at least 4.8K activations per row [16]), and even the recent half-double attack (which requires at least 296K activations on one row [12]).

5.2 Invariants of our Design

For simplicity, in this section, we will denote the threshold after which a swap occurs in RRS (T_{RRS}) simply by T. Our RRS design with a swap threshold of T has the following invariants:

> **Invariant 1:** The Misra-Gries tracker is guaranteed to detect any row before it crosses a threshold of T activations within a tracking window, and also before it crosses a multiple of the threshold, i.e. $n * T$ activations within the window.

> **Invariant 2:** Each time a row reaches a threshold number of row activations (T or $n * T$) within a tracking window, it is guaranteed to be swapped with a randomly selected row in the bank, with less than T activations within the same tracking window.

The proof of *Invariant 1* directly follows from the guarantees of the Misra-Gries (MG) algorithm (proved in Graphene [25]), that as long as the number of counters tracking the activations (N) satisfy the inequality $N > W/T - 1$, where W is the total number of row activations possible in a tracking window, all rows with $\geq T$ activations will have a counter value $\geq T$. Similarly, all rows with $\geq n * T$ activations within the tracking window would have a counter value $\geq n * T$. Thus, rows which will cross an activation threshold of T (or a multiple of T) within the tracking window are a subset of rows whose counter values in the MG table reach T (or a multiple of T) and can be preemptively identified when the MG counter value reaches the threshold T.

The proof of *Invariant 2* is based on the row-swapping guarantees of RRS. Whenever the Misra-Gries tracker finds a row's counter equals T or $n * T$, implying that the row's activation count could cross the same threshold (from *Invariant 1*), the tracker signals RRS to initiate a swap of this row with a randomly selected row. If the selected row is already present in the RIT, indicating that it is already swapped, another row is randomly chosen till an unswapped row is selected. This guarantees that the destination of the swap has less than T activations in the current tracking window, as otherwise it would already be swapped and be in the RIT.

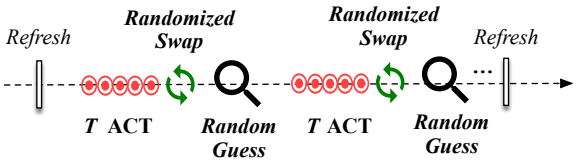

Figure 7: Attacker strategy to maximize the number of row activations (ACT) within a refresh period for a row. Each time a attacker row reaches a multiple of T ACTs in a tracking window, a randomized row swap occurs. This forces the attacker to randomly choose another row repeatedly after each T activations, in the hope that it discovers a previously swapped row, so that its activations can continue.

5.3 Row Hammer Prevention with RRS

The goal of a successful attack is to reach T_{RH} activations on a single row. However, Invariant 2 guarantees that an adversary can only activate a row at most T times (smaller than T_{RH}), before it is swapped with a randomly selected row in the bank, which has not been swapped in the current window. Thus, the physical location accessible to the attacker after a swap always has less than T activations in the current window: so it is not beneficial to keep attacking this row. Instead, it is beneficial to attack the physical row that already has T activations. But, the attacker does not know which row now maps to that physical location after the swap.

So, the best possible strategy for the attacker is to randomly choose a row address in the same bank, hoping that its current physical location has already received more than T activations in the current window (its physical location was previously involved in a swap), and activate it an additional T times. This attack strategy is shown in Figure 7, where the attacker repeatedly chooses a random row within the same bank, activates it T times till it gets swapped, then repeats this with another random row from the bank, for the entire window of 64 ms. This attack has a reasonable chance of activating physical rows that have already been swapped (similar to birthday paradox), as the space of randomization is only within 128K rows. So, a physical row could receive several multiples of T activations and may eventually cross T_{RH} activations within 64ms.

5.3.1 Statistically Modelling Rowhammer Attacks on RRS. For security against Rowhammer attacks, we seek the probability that any physical row receives T_{RH} activations within a refresh window of 64ms. We assume that the RRS row swap threshold (T) is chosen such that T_{RH} is an integer multiple of T, *i.e.*, $T_{RH} = k * T$. To achieve $\geq k * T$ activations on any physical row within 64ms, it must undergo at least k swaps. So for each step of the attack shown in Figure 7, it is beneficial for the attacker to make exactly T activations (no more) to make it swap, and then focus the attack on another random row within the same bank. Thus, the attack

effectively picks a random row, causes it to swap, then repeats this with another random row in the bank successively for the entire refresh period. We model this attack behavior statistically.

Bucket and Balls Model: We model this attack using the bucket and balls model and obtain the probability of achieving at least k swaps on any physical row within 64 ms (to cross $k * T$ or T_{RH} activations). Let A be the number of maximum activations possible for any bank within a period of 64ms. For our study, A equals 1.36 million. With RRS, the bank can be busy doing row-swap for a subset of the 64ms, so the bank is available for doing row activations for only a fraction of the 64ms: we call this fraction for which the bank is available for activations as the *Duty Cycle (D)*. Thus, with RRS, a bank can undergo $A \cdot D$ activations in 64ms. For an attack focusing only on one bank, we estimate D to be 0.925 (the bank is busy for 2.9 us every $T = 800$ activations), and thus a total of $A \cdot D = 1.26$ million activations are possible for a bank within 64ms.

We can represent each round of attack (T activations on a row) as an event of throwing a ball randomly into N buckets, the number of rows per bank ($N = 128K$). The attacker will be able to throw a total of $B = A \cdot D/T$ such balls during the period of 64ms. We calculate the probability of a row having k swaps in 64 ms as the probability ($p_{k,T}$) of a bucket having k balls per bucket after B balls are randomly thrown in N buckets, for a given T.

Calculating Probabilities using Bernoulli Trials: $p_{k,T}$ can be calculated as the probability of k successes in a Bernoulli trial over B trials, where each trial has a success probability of $p = 1/N$. From the probability distribution of Bernoulli trials, we calculate the probability of a row having k swaps ($p_{k,T}$), as

$$p_{k,T} = {}^{B}C_k * p^k * (1-p)^{(B-k)} \quad (1)$$

The expected number of rows with k swaps in 64ms is, $N_k = N * p_{k,T}$. For cases where $N_k \ll 1$, the expected number of attack iterations (AT_{iter}) to obtain a single row with k swaps is,

$$AT_{iter} = 1/(N * p_{k,T}) \quad (2)$$

Thus, the expected number of attack iterations (each iteration spans a refresh window of 64ms) before a row crosses Row Hammer threshold ($T_{RH} = k * T$) activations leading to a successful attack, is calculated by combining Equation (1) and Equation (2), and using $B = A \cdot D/\ T$. We thus obtain the expected number of attack iterations (AT_{iter}) for a successful attack as,

$$AT_{iter} = 1/(N * {}^{B}C_k * p^k * (1-p)^{(B\ -\ k)}) \quad (3)$$

Here, k can be replaced by $k = T_{RH}/T$ to parametrize it for a particular Row Hammer threshold and the time required for a successful attack is $AT_{time} = 64\text{ ms} * AT_{iter}$.

5.3.2 Determining the RRS Threshold (T) to Mitigate Rowhammer. Table 4 shows the number of attack iterations (AT_{iter}) for a successful attack calculated using Equation (3) and the time required (AT_{time}) based on 64 ms per attack iteration. We show the attack time in Table 4 for different values of T for which $k * T = 4800$ (our value for T_{RH}). As T (the threshold for a swap in RRS) reduces, the time for a successful attack and the security of RRS design increases as the mitigation (swap) gets invoked more frequently. However, there is a security vs performance trade-off, as a smaller T would incur higher performance and storage overhead, due to more frequent swaps and larger tracking structures. Hence, we choose T such that the system is protected for at least a year of continuous attack: with $T = 800$, the expected time for a successful attack is 3.8 years. A lower swap threshold T could also be used if a higher level of security is desired or if the Rowhammer threshold itself reduces in the future. Alternatively, a higher level of security can also be obtained with the same T, by co-designing this with attack-detection[2], which can be explored in future work.

Table 4: Number of Attack Iterations (AT_{iter}) and Attack Time (AT_{time}) needed to cause $T_{RH} = 4800$ activations on a row

RRS Threshold (T)	Attack Iterations (AT_{iter})	Time (AT_{time})
960 ($k = 5$)	9.3×10^6	6.9 days
800 ($k = 6$)	1.9×10^9	3.8 years
685 ($k = 7$)	3.8×10^{11}	762 years

We also analyze an all-bank attack, where the adversary tries to cause failures in all the 16 banks. While this theoretically provides the adversary 16 times greater chance for success, the swaps caused by attacking all the banks reduce the amount of time available for each banks to do row activations. For this attack, the duty cycle (D) equals 0.55, which means the actual time for successful attack becomes even larger than the single-bank attack (for example, for k=6, the attack time for the all-bank attack increases from 3.8 years to 5.1 years). Therefore, we focus on the single-bank attack.

5.4 Security of Row Swap Structures

Security of Row-Indirection-Table: The security of our design hinges on the rows swapped in the current tracking window remaining swapped for the entirety of that window. The row indirection table guarantees this by ensuring its entries are not victimized until after the completion of the tracking window when they were originally installed. Moreover, the attacker cannot overflow the row-indirection table to force a premature unswap operation for swapped rows, as the table is sufficiently sized to accommodate the maximum possible rows that may be swapped rows in a tracking window (bounded by the number of entries in Misra-Gries tracker).

Latency Implications of RIT Lookups and Installs: The two RIT operations in RRS are RIT lookups on a DRAM access and RIT installs on a Row-Swap. The RIT lookup has a constant latency: so it does not introduce any information leakage via any timing variation. The Row-Swap may have a higher latency if the row to be swapped is already present in the RIT. But this does not leak any useful information to the adversary, as the destination of the row after the swap is unknown. The mitigation of swapping a given row with a random row occurs within the same bank. We ensure that the row buffer of the bank is closed after swap and no accesses are allowed during swap. Thus, the attacker cannot deduce the location of the destination-row (the attacker can only infer that a swap occurred), thus ensuring no security impact to our design.

[2]A trivial mechanism to detect an attack on RRS is to count the number of swaps in 64 ms for each swapped row as a successful attack requires repetitive swaps in 64 ms on one row. When an imminent attack on RRS is flagged, a preemptive refresh of the entire DRAM can prevent the attack, thus providing higher security than RRS alone.

6 ARCHITECTING SCALABLE STRUCTURES

For T_{RRS} of 800, we need the Misra-Gries tracker to have 1700 entries and the RIT to store 3400 tuples (6800 entries). Unfortunately, the prior design [25] for Misra-Gries tracker (targeted T_{RH} of 25K) uses *Content Addressable Memory (CAM)*, which is not scalable beyond a few dozens of entries. Furthermore, RIT is a latency-critical structure that is looked up on every memory access, so it requires low lookup latency. Finally, to ensure security, both the tracker and RIT cannot evict entries (due to conflict misses) that are installed in the current epoch, so we cannot use conventional set-associative structures. In this section, we develop scalable hardware primitives that have lookup latency similar to set-associative structures, while still guaranteeing conflict-free storage for a given number of items. We first describe the generic structure and establish bounds, then discuss implementation of RIT and Misra-Gries tracker.

6.1 Collision Avoidance Table

Our solution *Collision Avoidance Table (CAT)* is inspired from recent work on practical fully associative cache, MIRAGE [28]. CAT combines a set-associative structure and multiple randomizing hashes, as shown in Figure 8. Let us say we want to store C items. CAT has two tables $T1$ and $T2$, which are set-associative and indexed by independent hashes $H1$ and $H2$ (constructed using a low latency cipher with different keys [28]). The tables have S sets and D demand ways such that $D = C/2S$. We over-provision CAT so that each set has E extra ways, for a total of $(D+E)$ ways per set. Given sufficient over-provisioning, CAT installs always occur in invalid entries, without needing to evict any entry from the same set [28].

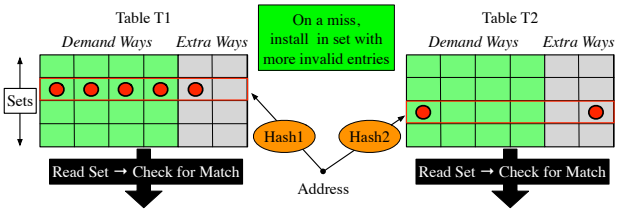

Figure 8: Overview of Conflict-Avoidance Table (CAT) offering set-associative look-ups and conflict-free storage.

To lookup CAT, the T1 and T2 are indexed with two independent hash functions. If the entry is found in either set, a hit signal is provided, and any metadata associated with the entry can be updated. Thus, look-ups are similar to set-associative tables.

To install an entry, we first lookup the set corresponding to the address in both T1 and T2, and select the set that has the most number of invalid entries as the destination. Picking the lower loaded set in this manner provides a balanced distribution of invalid entries across sets; this ensures installs can continue to happen in invalid entries, while avoiding conflicts with a high probability given sufficient over-provisioning of entries (see Section 4 of MIRAGE [28] for a more rigorous analysis and an analytical model). If the install would cause CAT capacity to exceed the target capacity of C, then a random entry from CAT (that can be evicted) is selected for eviction, thus maintaining a capacity of at-most C entries. Next, we show how many extra ways (over-provisioning) are needed to ensure the conflict-avoidance property.

6.2 Setting Bounds on Conflict with CAT

We deem a conflict in CAT when an install finds that both sets have zero invalid lines. We are interested in determining the number of installs required to cause a conflict in CAT. Let us assume that we have 64 sets and the number of demand ways is 14. We add extra 1-6 ways to each set to handle conflicts. Figure 9 shows the number of installs required to get a conflict with CAT. We generate the data for 1 - 4 extra ways using a Monte Carlo simulation of a buckets and balls model of the CAT and the data for 5 and 6 extra ways is based on the continued squaring behavior demonstrated in the analytical model from MIRAGE (Equations 6 and 7) [28]. With 6 extra ways, we need 10^{30} installs, and at a rate of 1 install per 10 microseconds, it would take CAT 10^{18} years (more than the lifetime of the universe) to encounter a conflict. Thus, with 6-extra ways, we claim that CAT is conflict-free. Nonetheless, if ever a conflict occurs, CAT can rely on Cuckoo relocation to check if each of the 20 entries in both sets can be moved to their alternative set in the other table, similar to MIRAGE-Lite [28].

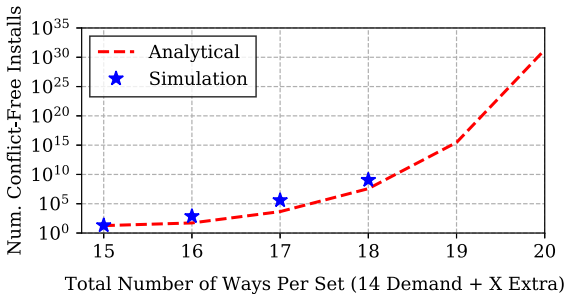

Figure 9: Number of installs required to cause a conflict in CAT with 64-sets (numbers are similar for 256 sets).

6.3 Designing Scalable RIT with CAT

Our design needs RIT with 3400 tuples, or 6800 entries. This can be done with CAT by having 2 tables, each with 256 sets and slightly less than 14 demand ways. So with 20 total ways per set, we get 6 extra ways – enough for the conflict-free property. Each entry has the source and destination row-ids, valid bit, and lock bit. During eviction, we only pick a valid entry for which lock-bit is 0. When an RIT entry is evicted, we also evict the other pair of the tuple.

6.4 Designing Scalable Tracker with CAT

The Misra-Gries tracker with 1700 entries can be achieved with CAT having 2 tables, with 64 sets and slightly less than 14 demand ways, so with 20 ways per-set, we get the required 6 extra ways. Each entry has a row-id and access counter. For the Misra-Gries algorithm, we also need the ability to check if spill counter is equal to the minimum access counter. To avoid fully associative search for counter values, we append each set with *SetMin* counter, which tracks the minimum value of the access counter in the set. On access, install, and invalidation in a set, the SetMin is recomputed. So, the spill-counter is first only checked with 64 SetMin counters, and only then checked with the set where there is a match. These accesses are performed in the shadow of a memory access as they are not on the critical path.

7 RESULTS AND ANALYSIS

7.1 Storage Analysis

RRS requires storage for the tracker, RIT and per-channel swap-buffers. Table 5 shows the storage overhead of RRS. We assume a 17-bit rowid, with the set-associative structures (tracker and RIT) storing the tag as the rowid without the set-index bits. In total, the SRAM overhead of RRS is 42.9KB per bank and 686KB per rank.

Table 5: Storage Overhead Per Bank

Structure	Entry-Size	Entries	Cost
RIT	28-bits (valid+lock+src+dest)	2x256x20	35KB
Tracker	22-bits (valid+row+counter)	2x64x20	6.9KB
Swap-Buffers	16KB (amortized over 16 banks)	1/16	1KB
Total			42.9KB

7.2 Power Analysis

RRS incurs a power overhead due to the extra DRAM accesses for the row-swap operations and for the additional SRAM structures (RIT and tracker). Table 6 shows the extra power consumed in RRS per rank. We measure the DRAM power from USIMM [7]) and observe that on average, RRS has a negligible DRAM power overhead of 0.5%. It is proportional to the additional row-swap operations performed: workloads with more frequent row swaps. We also measure that the RRS SRAM structures consume 903mW per rank in 32nm technology using Cacti 6.0 [24], however, this is likely to reduce with smaller more modern technology nodes.

Table 6: Extra Power Consumption in RRS Per Rank

Type of Power Overhead	Average
DRAM Power Overhead (Row-Swap)	0.5%
SRAM Power Overhead (RRS Structures)	903 mW

7.3 Performance Sensitivity to RH-Threshold

Figure 10 shows the performance of RRS as T_{RH} is varied from 0.25× to 4× of our default threshold 4.8K. We adapt the parameters of our design for each threshold to maintain security. On average, we observe 4.5% slowdown at T_{RH} of 1.2K (0.25×), 2.2% slowdown at T_{RH} of 2.4K (0.5×), 0.4% slowdown at T_{RH} of 4.8K (1×), and almost no slowdown at 9.6K (2×) and 19.2K (4×).

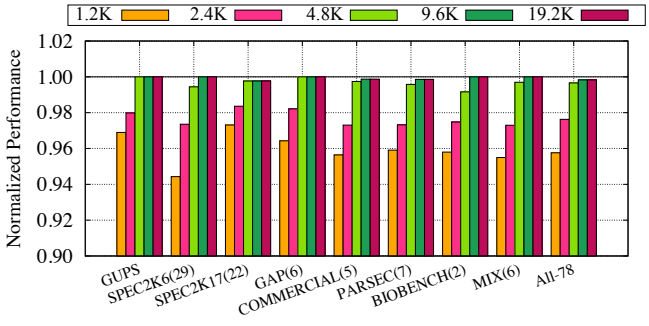

Figure 10: Performance of RRS across RH-Threshold.

8 RELATED WORK

In this section, we describe the closely related works, and compare and contrast when applicable.

8.1 Aggressor Focused Mitigation

To the best of our knowledge, the only other example of aggressor-focused mitigation, similar to RRS, is BlockHammer [37].

BlockHammer (BH) identifies aggressor rows using counting bloom filters and uses a delay-based mitigation for such rows. For rows that map to bloom filter entries with counts greater than a given threshold (e.g 512 or 1K), *i.e.*, the *blacklisting threshold*, activations are delayed such that that the total activations for such rows never crosses T_{RH} in a refresh period. Similar to RRS, this solution has the benefit of mitigating Rowhammer without requiring knowledge of the DRAM-row mappings or adding any new DRAM commands. Unfortunately, the inserted delays in Blockhammer can be quite large at lower values of T_{RH}. For example at T_{RH} of 4.8K, we would need to delay memory requests for approximately 20 microseconds per activation. Such large delays can make the system susceptible to denial-of-service concerns.

Performance. Figure 11 shows the performance S-curve across 78 workloads for RRS and BlockHammer, for a T_{RH} of 4.8K. We analyze BlockHammer with two values for the *blacklisting threshold*, 512 and 1K, as recommended in [37] for low T_{RH}. BlockHammer has significant slowdowns for several applications, with up to a 21.7% slowdown in the worst-case (10-25 workloads have more than 5% slowdown); the average slowdown is close to 2%. This is because hot rows (with frequent activations), and other rows mapping to the same bloom filter entry which get blacklisted, have their activations delayed for the entire refresh window of 64 milliseconds. In comparison, RRS has more robust performance with a worst-case slowdown of 7.6% (with only 3 workloads over 5%) and an average of 0.4%, as the mitigation overheads are only incurred for the duration of a row-swap (few microseconds).

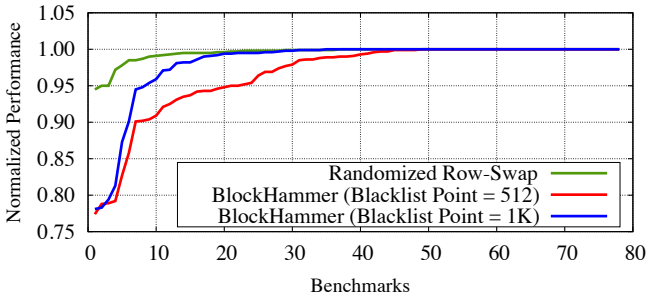

Figure 11: Performance S-Curve for RRS and Block-Hammer (with a blacklist point of 512 and 1K) over all 78 workloads.

Denial of Service. As BlockHammer could delay every single activation for rows mapping to blacklisted bloom filter entries, it is susceptible to severe denial of service. For instance, attacks that continuously activate few rows can make BlockHammer incur slowdown of close to 200x (20 microsecond delay for DRAM access that takes 100ns). Moreover, with an attack that hammers DRAM rows using OS activity, such as in PTHammer [40], the delayed

OS memory accesses could cause cascaded delays in all OS operations, leading to system-wide denial of service. In comparison, RRS incurs a worst-case latency overhead of row-swaps once per 800 activations (800 × 45ns = 36 microseconds) to a bank; so, it has a much lower worst-case slowdown (of approximately 2x). The RRS slowdown under attack can be reduced even further with DRAM-based techniques for faster copying of rows, such as RowClone [30], which could considerably reduce the row-swap latency.

Storage and Scalability. BlockHammer requires a storage overhead of 100-200KB per rank for SRAM and CAM structures for a T_{RH} of 4.8K, which is lower than RRS which requires 690KB of SRAM per rank. This is in part because RRS uses scalable conflict-avoiding designs for its tables. However, BlockHammer faces practical challenges in scalability: as T_{RH} scales to 1K, the delay for blacklisted rows even in benign applications can reach almost 100 microseconds, close to SSD latencies, making it impractical. Additionally, BlockHammer requires changes to memory scheduling policies and operating system to limit performance impact under attack scenarios Whereas, RRS does not suffer from such performance problems, even at lower thresholds, as shown in Section 7.3.

8.2 Victim-Focused Mitigation

Victim-focused mitigation (VFM) restores the charge of victim cells in rows neighboring aggressor rows by issuing targeted refresh. Prior hardware-based proposals for row-hammer mitigation differ primarily on when the victim refresh is issued, and this is done either probabilistically (PRA [15], PARA [17], MRLOC [39], ProHIT [32]) or counting accesses to specific rows (CRA [15], CBT [31], TWiCe [20], Graphene [25]). Table 7 compared RRS with VFM, where VFM is implemented with idealized tracking (100% accuracy and no overhead). Both RRS and VFM incur negligible slowdown. While VFM can mitigate classic row-hammer attacks that target immediate neighbors, it fails with complex patterns such as Half-Double [12], that causes bit-flips in rows farther away. RRS is robust to both classic patterns and complex patterns. Furthermore, unlike VFM, RRS does not require any knowledge of internal DRAM mappings to identify specific victim rows.

Table 7: Comparison of RRS with Victim-Focused Mitigation

Attribute	Victim-Focused	RRS
Slowdown	<0.1%	0.4%
Mitigates Classic Rowhammer (Neighboring Row Bit-Flips)	✓	✓
Mitigates Complex Patterns (Far Aggressors of Half-Double [12])	✗	✓
Works Without Knowing DRAM Mapping	✗	✓

8.3 ECC-Based Defenses

If the number of bit-flips due to RH are small, then they can potentially be corrected by ECC memories. However, a recent work, ECCploit [8], shows that an attacker can even overcome ECC protection to still cause RH. Integrity protection, as provided by Synergy [27] and a contemporary work, SafeGuard [10], can detect RH bit-flips. However, such schemes cannot correct the arbitrary bit-flips possible with RH, thus potentially leading to data loss.

8.4 Software-Only Defenses

Unlike hardware-based solutions that can only be deployed in future generations, software-based defenses can practically prevent Rowhammer attacks on existing systems. However, such solutions [3, 5, 18, 35] often require knowledge of DRAM properties that may be proprietary or not easily available to software.

For instance, **ANVIL** [3] uses CPU performance counters to detect RH attacks and issues refresh to the immediate victim rows. **GuardION** [35] inserts a guard row between data of different security domains. **ZebRAM** [18] and **RIP-RH** [5] provide isolation by keeping the kernel space and user space(s) in isolated parts of DRAM. However, all such solutions require knowledge of the DRAM-internal row mappings which may not be easily available to software. Other solutions like **CATT** [6] that perform testing of cells and blacklists pages with cells vulnerable to RH, can lead to considerable loss of memory capacity at lower RH thresholds.

Software-based solutions may also be unable to fully mitigate the root cause of the Rowhammer vulnerability in DRAM. For example, a single guard-row in GuardION or refresh of immediate neighbors in ANVIL can be broken with Half-Double attack [12]. Despite isolation between user and kernel spaces in ZebRAM and RIP-RH, bit-flips in kernel memory from user-space code continues to be possible with attacks like PTHammer [40], which hammer kernel memory with frequent page-table walks initiated from user space.

Monotonic pointers [36] prevents privilege escalation exploits using Rowhammer targeting page-tables – a typical class of Rowhammer attacks on vulnerable systems. It provides this protection by mapping page-tables to DRAM cells where bit flips only occur from $1 \rightarrow 0$ and stores them only in higher physical address ranges. This prevents any page-table entries from becoming self-referential after bit-flips, which is a key requirement for privilege escalation exploits attacking page-tables, thus preventing such exploits. However, this defense is unable to prevent non-page-table privilege escalation exploits like those exploiting op-code tampering in sensitive binaries with Rowhammer [13], or bit-flips in other user programs that affect correctness [38] and cause confidentiality breaches [19]. In contrast, RRS mitigates the root-cause of Rowhammer attacks, *i.e.*, charge leakage due to inter-cell interference, by breaking the spatial co-relation between aggressor and victim rows, thus preventing all current attacks (including Half-Double) and related exploits.

9 CONCLUSION

Row-Hammer (RH) bit-flips continue to be a serious security threat leading to privilege escalation and break of confidentiality. While classic RH patterns affected only nearby rows, prior works built defenses using victim-focused mitigation that refreshes immediate neighbor rows. However, the RH threshold has reduced significantly in recent years, and attackers are developing complex access patterns that can cause flips in many rows beyond immediate neighbors. In this paper, we design an effective RH defense by breaking the spatial connection between the aggressor and victim rows. Our proposal of *Randomized Row-Swap (RRS)* swaps the aggressor row with a random row in memory, thereby severely limiting the time the attacker has to mount an attack on the same neighborhood. We show that RRS incurs negligible slowdown (0.4%) and provides strong security of several years even under continuous attack.

ACKNOWLEDGEMENTS

We thank our shepherd, Yanjing Li, and the anonymous reviewers of ASPLOS 2022 for their comments and feedback. We also thank the members of the CoreSec group at TU Graz for their helpful feedback. We also thank the members of the ASPLOS Artifact Evaluation committee for their diligence. We express our thanks to the entire team of the Advanced Research Computing (ARC) center at UBC [34]. The computing resources at ARC center were instrumental for this paper. This work was supported in part by SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP), the Natural Sciences and Engineering Research Council of Canada (RGPIN-2019-05059), and a gift from Intel.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact presents the code and methodology to simulate Randomized Row-Swap (RRS), our defense against Rowhammer attacks. We provide the C code for the implementation of RRS which is encapsulated within the USIMM [7] memory system simulator. The 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 and RRS for all the workloads we studied in this paper. We also provide scripts to parse the results and collate the performance results shown in Figure 6.

A.2 Artifact Check-List

- **Algorithm:** Implementation of RRS structures and operations in C.
- **Program:** Memory-access traces with information for non-memory instructions (filtered through an L1 and L2 cache model) for any benchmark. This can be generated with any tracing tool (like Intel Pin [23] v2.12). We tested the artifact with benchmarks from SPEC-2006, SPEC-2017, PARSEC, BIOBENCH, and GAP suites.
- **Compilation:** Tested with gcc (versions 4.8.5, 6.4.0, 8.4.0), but should compile with most standard compilers.
- **Run-time environment:** Tested on Linux RHEL Server 7.9, but should broadly run on any Linux distribution.
- **Hardware:** Running all the 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 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)?:** 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.5800077

A.3 Description

A.3.1 Link. The code is available at https://github.com/gururaj-s/randrowswap and https://doi.org/10.5281/zenodo.5800077.

A.3.2 Hardware Dependencies. The artifact requires a CPU with a sufficient number of cores (64+) and memory (128GB+) to run all the benchmarks (78) in parallel for one configuration.

A.3.3 Software Dependencies. Perl (for the scripts to run experiments and collate results) and *gcc* (tested to compile successfully with versions: 4.8.5, 6.4.0, 8.4.0).

A.4 Installation and Experiment Workflow

The `run_artifact.sh` performs all the steps required to reproduce the results from the artifact:

- **Downloads the trace files**.
- **Compiles the code** using the makefiles in the `src_baseline` and `src_rrs`) and also runs the benchmarks.
- **Executes the simulations** for all 78 benchmarks in parallel, first for the `baseline` and then for `rrs` configuration.
- **Collates the results** for all 78 benchmarks, and provides the normalized performance.

A.5 Evaluation and Expected Results

The artifact provides the `getdata.pl` in the `simscript` folder. The perl script allows collation of the results and the commands to collate the IPC are provided in the `run_artifact.sh` and the README file. After the completion of the `run_artifact.sh`, the normalized performance for all benchmarks can be obtained as shown in Figure 6. The sample results files for the baseline and rrs configurations for all the benchmarks are provided in the `output` folder of the artifact.

A.6 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

REFERENCES

[1] 2012. 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC). https://www.cs.utah.edu/~rajeev/jwac12/.

[2] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. 2005. Biobench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 2–9. https://doi.org/10.1109/ISPASS.2005.1430554

[3] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices* 51, 4 (2016), 743–755. https://doi.org/10.1145/2954679.2872390

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[5] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 561–572.

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 117–130. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

[7] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[8] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 55–71. https://doi.org/10.1109/SP.2019.00089

[9] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmark Suite. http://www.spec.org/cpu2006/

[10] Ali Fakhrzadehgan, Yale Patt, Prashant J. Nair, and Moin Qureshi. 2022. 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.

[11] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 747–762. https://doi.org/10.1109/SP40000.2020.00090

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

[13] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 245–261. https://doi.org/10.1109/SP.2018.00031

[14] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.

[15] Dae-Hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. 2014. Architectural support for mitigating row hammering in DRAM memories. *IEEE CAL* 14, 1 (2014), 9–12. https://doi.org/10.1109/LCA.2014.2332177

[16] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. 2020. 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, 638–651. https://doi.org/10.1109/ISCA45697.2020.00059

[17] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[18] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 697–710. https://www.usenix.org/conference/osdi18/presentation/konoth

[19] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 695–711. https://doi.org/10.1109/SP40000.2020.00020

[20] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *Proceedings of the 46th International Symposium on Computer Architecture*. 385–396. https://doi.org/10.1145/3307650.3322232

[21] Lenovo. 2015. Row Hammer Privilege Escalation - Lenovo Security Advisory: LEN-2015-009. https://support.lenovo.com/us/en/product_security/row_hammer.

[22] Kevin Loughlin, Stefan Saroiu, Alec Wolman, and Baris Kasikci. 2021. Stop! Hammer time: rethinking our approach to rowhammer mitigations. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 88–95. https://doi.org/10.1145/3458336.3465295

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (jun 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[24] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[25] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W. Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Athens, Greece, 1–13. https://doi.org/10.1109/MICRO50266.2020.00014

[26] K. Asanovic S. Beamer and D. Patterson. 2015. The GAP benchmark suite. In *arXiv preprint arXiv:1508.03619*.

[27] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. 2018. Synergy: Rethinking secure-memory design for error-correcting memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 454–465. https://doi.org/10.1109/HPCA.2018.00046

[28] Gururaj Saileshwar and Moinuddin Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1379–1396. https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[29] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015), 71.

[30] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197. https://doi.org/10.1145/2540708.2540725

[31] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigating wordline crosstalk using adaptive trees of counters. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 612–623. https://doi.org/10.1109/ISCA.2018.00057

[32] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM stronger against row hammering. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[33] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017 Benchmark Suite. http://www.spec.org/cpu2017/

[34] UBC Advanced Research Computing. 2019. UBC ARC Sockeye. https://doi.org/10.14288/SOCKEYE

[35] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 92–113. https://doi.org/10.1007/978-3-319-93411-2_5

[36] Xin-Chuan Wu, Timothy Sherwood, Frederic T. Chong, and Yanjing Li. 2019. Protecting Page Tables from RowHammer Attacks Using Monotonic Pointers in DRAM True-Cells. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 645–657. https://doi.org/10.1145/3297858.3304039

[37] A Giray Yağlikçi, Minesh Patel, Jeremie S Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, et al. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–358. https://doi.org/10.1109/HPCA51647.2021.00037

[38] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1463–1480. https://www.usenix.org/conference/usenixsecurity20/presentation/yao

[39] Jung Min You and Joon-Sung Yang. 2019. MRLoc: Mitigating Row-hammering based on memory Locality. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[40] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 28–41. https://doi.org/10.1109/MICRO50266.2020.00016