AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime

Anish Saxena
Georgia Tech
Atlanta, USA
asaxena317@gatech.edu

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

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

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

***Abstract*—Rowhammer allows an attacker to induce bit flips in a row by rapidly accessing neighboring rows. Rowhammer is a severe security threat as it can be used to escalate privilege or break confidentiality. Moreover, the threshold of activations needed to induce Rowhammer continues to reduce and new attacks like Half-Double break existing solutions that refresh victim rows. The recently proposed *Randomized Row-Swap (RRS)* scheme is resilient to Half-Double as it provides mitigation by swapping an aggressor row with a random row. However, to ensure security, the threshold for triggering a row-swap must be set much lower than the Rowhammer threshold, leading to a significant performance loss of 20% on average, at a Rowhammer threshold of 1K. Furthermore, the SRAM overhead for storing the indirection table of RRS becomes prohibitively large – 2.4MB per rank at a Rowhammer threshold of 1K. Our goal is to develop a scalable Rowhammer mitigation that incurs negligible performance and storage overheads.**

To this end, we propose *AQUA*, a Rowhammer mitigation that breaks the spatial correlation between aggressor and victim rows by dynamically quarantining the aggressor row in a dedicated region of memory. AQUA allows for an effective row migration threshold much higher than in RRS, leading to an order of magnitude less slowdown and SRAM. As the security of AQUA is not reliant on keeping the destination row a secret, we further reduce the SRAM overheads of the indirection table by storing it in DRAM, and accessing it on-demand. We derive the size of the quarantine region required to ensure security for AQUA and show that reserving about 1% of DRAM is sufficient to mitigate Rowhammer at a threshold of 1K. Our evaluations show that AQUA incurs an average slowdown of 2% and an SRAM overhead (for mapping and migration) of only 41KB per rank at a Rowhammer threshold of 1K.

***Keywords*-DRAM, Security, Rowhammer, Isolation**

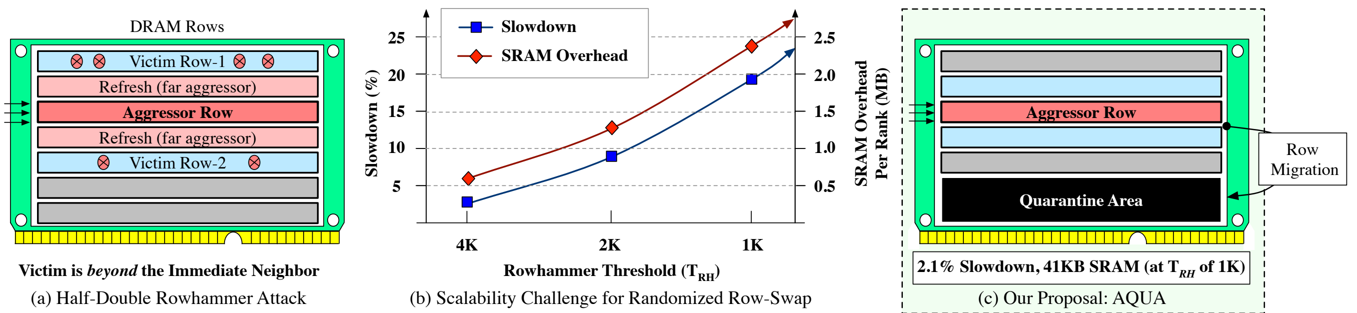
I. Introduction

Technology scaling has been the prime driver for increased DRAM capacity. Unfortunately, while smaller technology nodes offer higher capacities, they also pack DRAM cells more closely. This leads to increased inter-cell interference, wherein activation of a row of DRAM cells can influence the charge of its neighboring rows. This phenomenon is called Rowhammer [17] and it has been shown to be a critical security vulnerability [8]. Several prior work have used Rowhammer to trigger confidentiality breaches [20] and privilege escalation [29] exploits.

Rowhammer requires an attacker to rapidly perform row activations in a limited period of time. The number of row activations required to cause Rowhammer bit flips, called the *Rowhammer Threshold* (T_{RH}), has dramatically reduced. For example, when Rowhammer was first characterized in 2014 [17], T_{RH} was 130K activations, whereas it had reduced to just 4.8K activations in 2020 [15]. T_{RH} is expected to reduce further, as DRAM technology becomes more dense. Therefore, solutions for mitigating Rowhammer must be effective not only at current T_{RH} but also at lower thresholds that are likely to be present in the near future.

Typical hardware-based defenses for mitigating Rowhammer consists of two parts: (1) *Tracking mechanism*, that tracks frequently accessed rows (termed as *aggressor rows*) and (2) *Mitigative action*, which is performed once the aggressor row reaches a specified number of activations. Tracking typically involves using counters that are either stored in SRAM (e.g. Graphene [25], CBT [31], TWiCE [21], TRR [7], Mithril [16]) or DRAM (e.g. CRA [4], [14]). The mitigating action involves refreshing the contents of the rows that are immediate neighbors of the aggressor row. However, a recently developed attack pattern, called Half-Double, influences rows that are a distance-of-2 away from the aggressor row [18]. As shown in Figure 1, Half-Double leverages the victim refresh-based mitigation on rows adjacent to the aggressor to induce bit flips in rows distance-of-2 away. If rows that are a distance-of-1 and a distance-of-2 are issued mitigating refreshes, then the Half-Double attack might even be extended to influence rows that are a distance-of-3 away and so on. Thus, we need mitigating action that is resilient to attacks which exploit the spatial proximity between aggressor and victim rows.

Row migration is an alternative to victim refresh that provides mitigation by breaking the spatial correlation between the aggressor and victim rows. For example, the recently proposed Randomized Row-Swap [28] (RRS) scheme prevents Rowhammer by swapping the aggressor row (once it is flagged by the tracker) with a randomly selected row. If the attacker continues to access the same physical DRAM row, the row is swapped again to ensure that no row in memory reaches the target number of activations. RRS maintains an indirection table to track the mapping of the swapped rows. This table must be stored in SRAM to enable constant-latency lookups and eliminate timing channels that may leak information about the destination row of a swap. Thus, RRS provides security via randomization by ensuring the probability of sufficiently hammering a physical row in memory to induce Rowhammer bit flips is negligibly small.



(a) Half-Double Rowhammer Attack (b) Scalability Challenge for Randomized Row-Swap (c) Our Proposal: AQUA

Figure 1. (a) Half-Double attack [18] exploits victim-refresh to flip bits in rows distance-2 away from the aggressor. (b) Randomized Row-Swap (RRS) [28] breaks spatial correlation between aggressors and victims but incurs significant slowdown and SRAM overhead as Rowhammer threshold reduces from 4K to 1K. (c) AQUA enables scalable mitigation by migrating aggressor rows to a quarantine area in memory with negligible slowdown and SRAM overhead.

To ensure security against the attacker accidentally finding the destination of the row-swap and inflicting more activations on the same physical row (known as birthday paradox attacks), RRS must swap rows at a threshold that is substantially lower (about 6×) than T_{RH}. Thus, migrating rows with randomized-swap forces substantial performance and SRAM overheads, both of which become unacceptable at low T_{RH}, as shown in Figure 1(b).

For example, at T_{RH} of 1K activations, RRS must swap rows after only 166 activations. At this lower effective threshold, the number of rows that need to be swapped rise drastically, leading to significant performance loss, on average of 19.8%, at T_{RH} of 1K. Moreover, the size of indirection tables increases in inverse proportion of the effective threshold and incurs almost 2.4MB of SRAM overhead per rank (at T_{RH} of 1K), which is beyond the limits of practical adoption. For comparison, the recently proposed Hydra tracker uses only 30KB of SRAM per rank (and less than 0.1% DRAM) to track aggressors [26]. With such a tracker, the RRS mitigation alone would account for almost 99% of the overall SRAM overhead to protect against Rowhammer. Finally, despite the significant slowdown and storage overhead, the security guarantee of RRS is *probabilistic* and an attacker can still cause a successful attack on average within 4 years, and if the attacker targets N machines, the time for a successful attack decreases by N.

We observe that one can reduce the overheads of RRS by relying on *isolation* instead of *randomization*, to avoid the artificial lowering of the row migration threshold in RRS. With this insight, we propose an alternative row-migration scheme, *AQUA*. AQUA provides mitigation by dynamically moving the aggressor row to a dedicated region of memory, called the *Row Quarantine Area (RQA)*, as shown in Figure 1(c). If a row is continuously activated while being quarantined, it is moved from one location in the RQA to another location in the RQA. The RQA is sized such that no location is reused within the refresh period (64ms). Thus, row migration with AQUA can be triggered at a threshold determined by T_{RH} (and any inefficiency due to the tracking mechanism), rather than being artificially reduced like in RRS. As AQUA operates at a higher threshold, we observe that its performance loss is almost 10x lower than RRS, due to three reasons: **First**, a smaller number of rows reach the higher threshold within 64ms, requiring fewer mitigations. **Second**, when a frequently accessed row reaches the higher threshold, the mitigation is amortized over a greater number of activations. **Third**, the act of migration to the quarantine area incurs half as much time (one read and one write) compared to swapping two rows (two reads and two writes).

A key parameter of AQUA is the size of the RQA as it determines the maximum number of rows that can be quarantined within 64ms. For security, we need to ensure that a row quarantined in the RQA is not evicted prematurely within the 64ms refresh interval. We analyze the time required to trigger a row migration and then perform a migration to bound the number of entries required in the RQA. We derive that for T_{RH} of 1K activations, we need a RQA with 23K rows, which is approximately 1.1% of the rows in our baseline 16GB memory (two million rows of 8KB each). Thus, AQUA requires negligible DRAM overheads to isolate aggressor rows for the duration of the 64ms refresh interval.

AQUA also relies on an indirection table to store the location of the quarantined rows. But as AQUA uses a higher threshold than RRS, its indirection table incurs an SRAM overhead of only 172KB per rank (12× lower than RRS). Moreover, since AQUA's security does not rely on keeping destination of migrated rows secret, it can store the indirection table in DRAM, cache it as needed in on-chip SRAM, and even avoid unnecessary lookups using a small 16KB SRAM filter. Overall, such a hybrid design reduces the SRAM overheads to identify row locations to only 32KB per rank with negligible performance impact.

Overall, our paper makes the following contributions:

- We present AQUA, a scalable Rowhammer mitigation mechanism that quarantines aggressor rows at runtime in a dedicated region of memory.
- We analyze the size of the Row Quarantine Area (RQA) required to ensure the security of AQUA and show that provisioning only 1.1% of DRAM is sufficient.
- We reduce the SRAM overheads required to track the location of quarantined rows to only 32KB per rank (at T_{RH} of 1K) by storing the indirection table in DRAM.

Our evaluations with gem5 show AQUA that incurs an average performance loss of 2.1% (at T_{RH} of 1K). AQUA compares favorably with other recent schemes, as it requires only 1.1% DRAM overhead, unlike CROW [9] which incurs up to 1000%), and has a worst-case slowdown of 3×, unlike Blockhammer [36] which can have up to 1280× slowdown.

II. Background and Motivation

A. Threat Model

We assume an attacker that has the capability to run code natively on a system with user-level privileges. The targeted system uses DRAM that is vulnerable to Rowhammer bit flips. The attacker knows (or is capable of discovering) the mapping of DRAM rows to surgically hammer a single row or a set of nearby rows, to launch a double-sided or Half-Double attack. Moreover, the attacker is able to evict addresses from the cache to hammer the DRAM with the desired intensity. We assume a Rowhammer bit flip can occur at any victim location when a row incurs more activations than the Rowhammer Threshold (T_{RH}) in a refresh interval of 64ms. We assume an untargeted attack where the goal for the attacker is to activate any single physical row of the DRAM more than T_{RH} times in 64ms.

B. Background on DRAM Organization

DRAM modules are logically divided into ranks, banks, rows, and columns. Access to different banks can be time-multiplexed by the memory controller. Moreover, consecutive accesses to the same row are fast compared to different rows in a bank. This is because to open a different row, the previous row must be closed using the *precharge* command, followed by an *activation* command to open the new row. The DDR4 standard [13] specifies the minimum time between activating different rows (ACT-to-ACT delay) within the bank as t_{RC} (Row Cycle Time), which is typically 45ns.

The rows in memory lose charge over time and are refreshed periodically. The time window within which a given row must be refreshed is usually 64ms (t_{REFW}). To maintain quality-of-service, rows are refreshed in small groups internally by the DRAM, and the memory controller must send a refresh command every 7.8 μs (t_{REFI}), and then wait for 350ns (t_{RFC}) to allow rows to be refreshed. The maximum number of activations to a bank (ACT_{max}) are bounded at $ACT_{max} = t_{REFW}(1 - t_{RFC}/t_{REFI})/t_{RC} = 1360K$. Thus, the attacker has a budget of up to 1360K activations within 64ms to cause Rowhammer bit flips in a single bank.

C. DRAM Vulnerability to Rowhammer

The root-cause of Rowhammer based bit-flips are data-disturbance errors that occur when a row is activated frequently and leaks sufficient charge from neighboring rows. The row receiving frequent activations is called the *aggressor row* and the row with bit flips is called the *victim row*. The number of activations to the aggressor row(s) required to induce a bit flip is called the *Rowhammer Threshold (T_{RH})*.

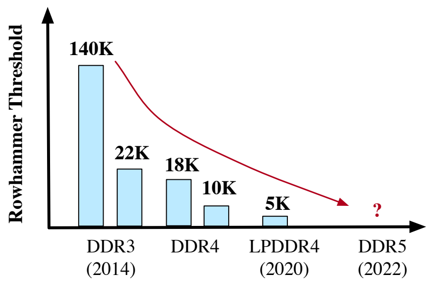


Figure 2. Rowhammer Threshold (T_{RH}) over time

Rowhammer was publicly demonstrated in 2014, where Kim *et al.* [17] flipped bits in DDR3 memory by inflicting about 139K activations to the same row within 64ms. As technology shrinks, the threshold to induce Rowhammer bit flips reduces. As shown in Figure 2, the Rowhammer threshold decreased by almost 30× to 4.8K for LPDDR4 memory in 2020 [15]. Therefore, solutions for mitigating Rowhammer must not only be effective for current T_{RH}, but also for lower T_{RH} likely to be present in the near future.

Several exploits [8], [12], [22], [29], [33], [35] have demonstrated that attacker programs can use Rowhammer bit-flips to gain kernel privileges [38], escape browser sandboxing [6], and leak secret data [20]. Moreover, at low T_{RH}, even benign applications can have many rows with activations beyond T_{RH} [23], [28]. Thus, Rowhammer is not just a security threat, but also a reliability problem.

D. Mitigation via Victim Refresh: Pitfalls

Hardware-based mitigations of Rowhammer typically use dedicated counters in *SRAM* (e.g. Graphene [25], CBT [31], TWiCE [21], TRR [7], Mithril [16]), in *DRAM* (*e.g.,* CRA [14], Panopticon [4]), or in both *SRAM* and *DRAM* with a hybrid design (*e.g.,* Hydra [26]) to track potential aggressors. Such trackers provide guaranteed detection of rows that exceed the Rowhammer threshold. Once an aggressor row is identified, a popular mitigation approach is to do a *victim refresh*, *i.e.,* refresh the rows neighboring the aggressor row.

Victim refresh suffers from two shortcomings. First, to implement victim refresh, the memory controller needs to precisely identify the neighbors of any given row. Unfortunately, DRAM vendors often use proprietary internal row mapping and this mapping is not exposed to the CPU, making the task of identifying the neighboring rows difficult for the memory controller. Second, the act of victim refresh can itself lead to newer attacks. For example, the recently disclosed Half-Double attack [2], [18] from Google uses the victim refreshes to cause bit flips at a distance-of-two from the aggressor rows. As victim refreshes retain the spatial correlation between the aggressor and victim rows, it enables the attacker a long period (64ms) to focus a large number of activations to a single location and launch more effective attacks.

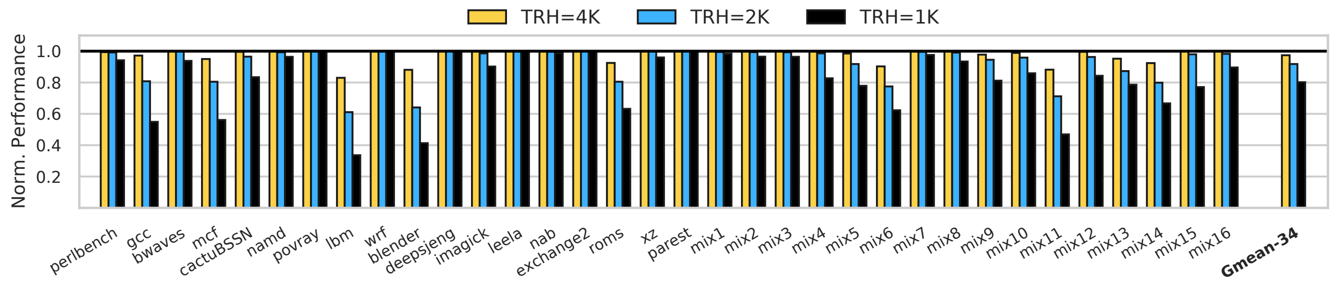


Figure 3. Performance of RRS workloads as the Rowhammer threshold decreases from 4K to 2K to 1K. At 4K threshold, RRS has low performance loss (on average at 2.7%), but it increases dramatically at 2K to 8.2% and at 1K to 19.8%.

E. Mitigation via Row Migration

Row migration is a mitigation that overcomes both the pitfalls of victim refresh: it can be enabled without the knowledge of internal mappings of DRAM and it breaks the spatial connection between aggressor and victim rows, thus significantly reducing the time for the attacker to focus on a given neighbourhood. The recently proposed *Randomized Row-Swap (RRS)* [28] provides mitigation by swapping a given aggressor row with a randomly selected row, once the aggressor has accrued a threshold number of activations. RRS uses a *Row Indirection Table (RIT)* to keep track of the row locations. As the security of RRS is dependent on keeping the destination of the swapped row a secret, the RIT is stored entirely in SRAM[1] to avoid any leakage due to side-channels via DRAM accesses.

F. Scalability Challenges for RRS

RRS uses randomization for security. When an aggressor row is randomly swapped with another destination row, an attacker tries to discover the new destination of the row to continue the attack on the same physical row. With RRS, an attacker has a small probability of randomly discovering the physical location of a previously attacked row (via the *birthday paradox*). To defend against this, the threshold for swapping a row (T_{RRS}) needs to be much lower than T_{RH}. For example, RRS advocates T_{RRS} to be one-sixth of T_{RH}.

Figure 3 shows the slowdown due to RRS as T_{RH} is varied from 4K to 1K. We note that while RRS has a negligible slowdown at 4K (on average, 2.7%), it becomes significantly high at 1K (on average, 19.8%). At T_{RH} of 1K, the effective threshold for row swaps (T_{RRS}) becomes 166 activations. This causes a significant slowdown for two reasons: First, as seen in Table II, workloads tend to have significantly more rows that encounter 166 activations within 64ms (than say 1K activations)), and a much greater number of rows trigger mitigative action. Second, the overhead of row-swap is amortized over only 166 activations instead of 1K activations.

[1]An alternative approach for row-migration is Row-Clone [30], which migrates rows to alternative locations in the same sub-array (of 512 rows) using in-DRAM logic. However, CROW [9] an early solution for mitigating Rowhammer, that relies on Row-Clone for row-migration, would need to provision each subarray with 1060% extra rows to be secure at a T_{RH} of 1K. We discuss the shortcomings of this approach in Section VII.

The reduction in the effective threshold for RRS also causes high SRAM overhead for the RIT. As T_{RH} reduces from 4K to 1K, effective threshold reduces from 800 to 166, and the SRAM overhead for the RIT increases from 0.65MB per rank to 2.4MB per rank.

G. Goal: Scalable Row-Migration

The goal of this paper is to develop a scalable and practical row-migration scheme that incurs negligible slowdown and SRAM overheads, even at lower thresholds (T_{RH} of 1K). Furthermore, the solution must be compatible with commodity memory systems and should not require any changes to the memory array or interfaces. Our key insight is to rely on *isolation* instead of *randomization* to enable such a solution. We describe our methodology before our solution.

III. Evaluation Methodology

We use gem5 [24], a cycle-level simulator to perform multi-core simulations in the Syscall Emulation (SE) mode with an accurate out-of-order core and DDR4 memory model. We use the DDR4 2400MT/s 8 Gbit memory configuration which models the Micron MT40A2G4 [10]. We assume a per-bank Misra-Gries tracker [25] in the memory controller. Table I shows the configuration for our baseline system.

Table I
Baseline System Configuration

Out-of-Order Cores	4 cores at 3GHz
ROB size	192
Fetch and Retire width	8
Last Level Cache (Shared)	4MB, 16-Way, 64B lines
Memory size	16 GB – DDR4
Memory bus speed	1.2 GHz (2400 MT/s)
t_{RCD}-t_{CL}-t_{RP}-t_{RC}	14.2-14.2-14.2-45 ns
t_{CCD_S}, t_{CCD_L}	3.3 ns, 5 ns
Banks x Ranks x Channels	16×1×1
Rows per bank	128K
Size of row	8KB

We evaluate our design with 18 SPEC2017 [1] *rate* workloads and 16 mixed workloads, each a set of four random SPEC2017 workloads. We fast-forward the workloads by 25 billion instructions to reach regions of interest and simulate for 250 million instructions. Table II shows the Misses Per 1K Instructions (MPKI) and average number of rows with 166+, 500+, and 1000+ activations per 64ms epoch.

Table II
WORKLOADS CHARACTERISTICS: MPKI AND AVERAGE ROW ACTIVATIONS OF 166+, 500+, 1000+ PER 64 MS.

Workload	MPKI	Rows		
		ACT-166+	ACT-500+	ACT-1K+
lbm	20.9	6794	5437	0
blender	14.8	6085	3021	572
gcc	6.32	4850	1836	111
mcf	7.02	4819	835	393
cactuBSSN	2.57	2515	0	0
roms	4.37	1150	191	11
xz	0.41	655	0	0
perlbench	0.74	0	0	0
bwaves	0.21	0	0	0
namd	0.38	0	0	0
povray	0.01	0	0	0
wrf	0.02	0	0	0
deepsjeng	0.25	0	0	0
imagick	0.27	0	0	0
leela	0.03	0	0	0
nab	0.54	0	0	0
exchange2	0.01	0	0	0
parest	0.1	0	0	0
Average	3.5	1665	694	57

IV. AQUA: QUARANTINING AGGRESSORS

AQUA provides mitigation by migrating the aggressor rows to a dedicated quarantine region in memory. AQUA provides the guarantee that no physical row can be activated more than a specified threshold, thus providing principled security while incurring negligible overheads in terms of performance and storage. Next, we provide an overview of AQUA, before describing its structures, analyzing the size of the quarantine area, and evaluating its performance impact.

A. *Overview of AQUA*

Figure 4 provides an overview of AQUA. AQUA reserves a small fraction of the memory space to form the *Row Quarantine Area (RQA)*. The RQA is not visible to software or the operating system, and is accessible only by the memory controller. To identify rows requiring mitigation (quarantining), AQUA uses a *Aggressor-Row Tracker (ART)* to track frequently accessed rows. To track quarantined rows, AQUA has two tables: a *Forward-Pointer Table (FPT)* that stores the location in the RQA for quarantined rows and a *Reverse-Pointer Table (RPT)* which identifies the memory row stored in a given RQA location. On a memory access, the memory controller checks the FPT to determine if the access is to be sent to the original location in memory or the RQA. If the row is quarantined, the FPT provides the location within the RQA where the access must be routed.

Figure 4-③ outlines the events that occur when an aggressor row is quarantined. When the ART identifies an accessed row as an aggressor, an unused RQA row is identified as the destination of the migration. Then, the contents of the original row location (or its current location

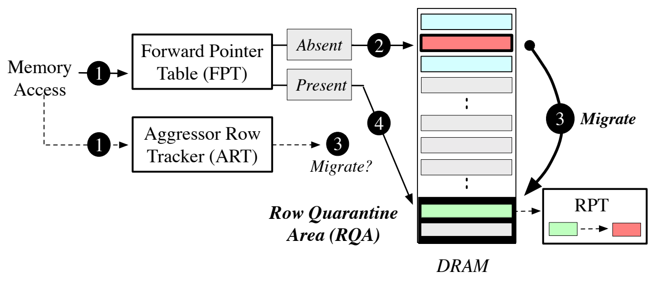

Memory Access
1
Forward Pointer Table (FPT)
Absent
2
Present
4
1
Aggressor Row Tracker (ART)
3
Migrate?
3
Migrate
Row Quarantine Area (RQA)
RPT
DRAM

Figure 4. Overview of AQUA. The Forward-Pointer Table (FPT) determines if the access should go to the original or the quarantined location. The Aggressor-Row Tracker (ART) identifies rows that must get quarantined.

in the RQA if the row is presently quarantined) are copied to the destination, and the FPT and RPT are updated.

We define the refresh interval of 64ms to be an epoch. The ART is reset at the end of every epoch to ensure that only the row access counts of the current epoch determine the eligibility for getting quarantined. We do not reset the FPT and RPT at that time, as this would require bulk eviction of all the entries in the RQA, causing significant latency overheads. Instead, we lazily drain out entries from the past epoch when new entries are brought in, while ensuring that an RQA entry is never reused in the same epoch. Thus, each physical row is guaranteed to not have more than the threshold number of activations within the epoch.

B. *Tracking Aggressor Rows*

The ART is responsible for identifying when activations for a row exceed a threshold of activations within an epoch and invoking a row-migration. Since the design of the tracker itself is orthogonal to our mitigation technique, we design AQUA to be compatible with any hardware-based ART. For our analysis, we assume the ART uses a per-bank Misra-Gries tracker in SRAM, as used in Graphene [25] and RRS [28]. However, AQUA is also compatible with the recent storage-optimized Hydra tracker [26]). The ART is indexed with the physical row address, obtained after consulting the FPT, and updated on each DRAM activation.

The periodic reset of the tracker can cause a vulnerability whereby the attacker can do a significant number of accesses just before and after the reset, each below T_{RH} individually, but exceeding T_{RH} activations in total within 64ms. To account for the loss of state due to reset, the effective threshold for the Misra-Gries tracker is set to be half of T_{RH}, in order to identify all rows that reach T_{RH} activations within 64ms. We use a default T_{RH} of 1K, therefore, ART is designed to initiate row-migration whenever a row reaches a multiple of 500 activations.

C. *Maintaining Location Information*

AQUA maintains the location information using two tables: FPT and RPT. The number of entries in both these structures is determined by the size of the row-quarantine area (RQA).

For our 16GB memory containing 2 million rows, our analysis shows that the RQA must have at least 23K entries to provide security at T_{RH} of 1K. Therefore, both FPT and RPT must be designed to store at least 23K entries.

RPT is a direct-mapped structure with one entry for each row in the quarantine area. Each entry contains a valid bit and a 21-bit reverse pointer pointing to the original location of the given row in memory. Thus, for storing 23K entries, the RPT is required to be 64KB in size.

An entry in FPT is provisioned only for the rows that are mapped to the quarantine area. Each entry in the FPT contains a valid bit, a tag for identifying the row, and a 15-bit forward-pointer identifying the location of the row in the quarantine area. However, the entries in FPT can come from arbitrary locations in memory, and the FPT must be able to hold such entries without any set-conflicts. Therefore, we design FPT as an over-provisioned *collision-avoidance table (CAT)* adopted from RRS [27], [28], with 32K entries, for storing 23K valid entries. Thus, the FPT size is 108KB.

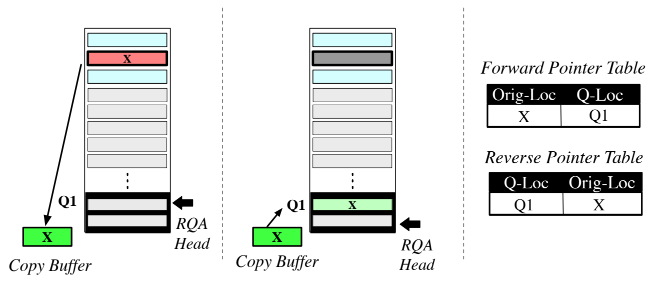
Both the FPT and RPT are global structures and are maintained at the memory controller for each rank. We store both the FPT and RPT in SRAM. Thus, the mapping structures for AQUA incur a combined storage overhead of 172KB, which is almost 12× lower than the 2.4MB SRAM required to store the row mappings of RRS.

D. Process of Quarantining Aggressor Row

When the ART identifies a row for quarantining, we need to specify the destination in the RQA where the incoming row will be stored. We architect RQA logically as a circular buffer, where the incoming entry is always stored at the oldest install location. We maintain a pointer (RQA-Head-Pointer) that identifies the destination location within the RQA.

To facilitate row-migration, we provision the channel with a copy-buffer, which is sized the same as the DRAM row (8KB in our study). Figure 5 describes the process of quarantining Row-X, which is not currently quarantined. Row-X is streamed into the copy buffer in the memory controller and then streamed out from the copy-buffer to the Row-Q1 in DRAM. After the quarantining operation, the FPT is updated with the tuple <X, Q1> and the RPT entry at index Q1 is updated to store X. The RQA-Head-Pointer is then incremented to point to the next location in the RQA.

To perform row-migration efficiently, we leverage streaming accesses from DRAM, whereby accesses to the same row can be serviced quickly (one 64 byte line every 5ns, after the first line) once the row has been activated. To transfer the 8KB row (128 lines), we would need 640ns after a row activation time of 45 ns (ACT-to-ACT delay). Thus, for our system, it takes approximately 685ns to transfer the row between DRAM and the copy-buffer. For moving the row to the quarantine area, we need one row read and one row write, which incurs a total latency of 1.37μs.



(a) Row-X is *read* (b) Row-X is *written* to Row-Q1 (c) Tables are updated

Figure 5. An overview of the quarantine process with AQUA. Row-X needs to be moved to the location Q1 in RQA, pointed to by the RQA-Head-Pointer. Subsequently, FPT and RPT entries are updated in the DRAM.

We note that if a row receives frequent activations, it may trigger another row-migration, while still being in the quarantine area. In this case, we copy the contents of the row from the previous location in the RQA to the new location within the RQA. Such an internal migration still incurs a latency of 1.37μs, as the process is similar to moving a new row into the RQA and needs one row-read and one row-write. The FPT entry and RPT entry for new RQA location are updated, and the original RQA entry is invalidated.

When row migration is triggered, it is possible that the destination in the RQA stores valid data of a row quarantined in the previous epoch (as indicated by the valid bit of the associated RPT entry). Here, we first move out the row from the RQA to its original location (invalidating the associated FPT entry) and then move the new data in the RQA location. AQUA incurs a latency of 1.37μs for moving out the old row and 1.37μs for moving in the new row, for a total latency of 2.74μs. We note that the latency for moving out a row from the RQA can be removed from the critical path of row access by periodically draining old entries.

E. Bounding the Size of Quarantine Area

A critical parameter in AQUA is the size of the quarantine area. If the size is too small, the quarantined rows will be evicted from the quarantine area within 64ms, potentially leading to security problem. If the size is too large, there is an unnecessary loss of DRAM memory and this requires larger tracking structures (FPT and RPT). In this section, we determine the minimum size of the quarantine area to ensure security, considering worst-case adversarial access patterns.

The size of the quarantine area depends on two factors: (1) the time (t_{AGG}) incurred in performing enough activations to an aggressor row to trigger a row-migration, and (2) the time (t_{mov}) incurred in performing row migration to the quarantine area, which keeps the channel busy and makes it unavailable for performing other DRAM accesses.

Let the threshold for initiating a row quarantine operation be A activations, then the time (t_{AGG}) to trigger a row-migration can be computed as shown in Equation 1.

$$t_{AGG} = A \cdot t_{RC} \qquad (1)$$

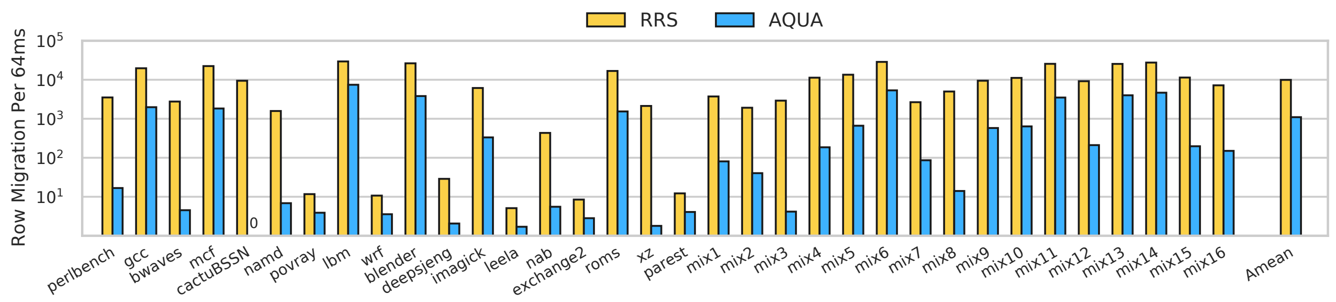


Figure 6. Number of row migrations performed per 64ms for AQUA. On average, AQUA performs 1099 row migrations per 64ms, which is 9× lower than RRS, which performs 9935 row migrations per 64ms on average.

If there are B banks, then the attacker can concurrently attack all B banks and cause B rows to initiate row-migration within the t_{AGG} period. Each of these rows would incur a latency of t_{mov} for the quarantine operation. Thus, the highest rate at which rows can enter the quarantine region is, B rows within a duration of t_B, as shown in Equation 2.

$$t_B = t_{AGG} + B \cdot t_{mov}. \tag{2}$$

Therefore, the maximal number or row migrations (R_{max}) to the quarantine area within t_{REFW} is given by Equation 3.

$$R_{max} = t_{REFW} \cdot \frac{B}{t_{AGG} + B \cdot t_{mov}} \tag{3}$$

The quarantine area must be provisioned for at least R_{max} rows. Thus, for a Rowhammer threshold of 1K, $A = 500$, and with 16 banks per each DDR4 rank, the quarantine region is sized to $R_{max} = 23,053$ rows or 180MB per rank. Thus, for our 16GB memory and a Rowhammer threshold of 1K, the DRAM overhead of AQUA is only 1.1%.

Using Eq. 3, we can also estimate the quarantine region size at decreasing Rowhammer thresholds, as shown in Table III. Overall, even if the effective threshold is reduced significantly, to say 125 activations, the size of the quarantine area needed for AQUA remains less than 2%. We note that even at an impractical T_{RH} of 2 (an effective threshold of 1), the rate of row migration is at most one migration on every access and requires DRAM overhead of at most 2.2%.

Table III
SIZE OF QUARANTINE AREA AS ROWHAMMER THRESHOLD IS VARIED

Effective Threshold (A)	R_{max} (Rows)	Quarantine Size (MB)	DRAM Overhead
1000	15,302	120 MB	0.7%
500	**23,053**	**180 MB**	**1.1%**
250	30,872	241 MB	1.5%
125	37,176	290 MB	1.8%
50	42,367	331 MB	2%
1	46620	364 MB	2.2%

F. Results: Number of Mitigations

Our default design of AQUA targets a T_{RH} of 1K. The primary overhead of row migration is due to the mitigative action that involves transferring rows from one location to another. As RRS performs mitigation via row-swap, it needs to migrates two rows and it uses an effective threshold of 166 (one-sixth of T_{RH}). Whereas, AQUA performs mitigation via a row-copy, so it needs to migrate only one row (from the current location to the quarantine location) and it uses an effective threshold of 500 (one-half of T_{RH}).

Figure 6 shows the number of row migrations incurred by RRS and AQUA. On average, AQUA requires 9× fewer row migrations than RRS. While most workloads require only a few mitigations (tens to hundreds) per 64ms, some workloads, such as `lbm` and `blender` require thousands of mitigations. Moreover, some workloads where no row exceeds 500 activations (refer to Table II) suffer from spurious mitigations (e.g., `imagick`) because Misra Gries is not an exact tracker – newly installed rows (without an ACT) have their estimated count starting from spill-counter value (non-zero value) causing unnecessary mitigations.

Note that the cost of each mitigation is not equal between AQUA and RRS. AQUA relies on migrating one row to the quarantine area (one row-read and one row-write) whereas mitigation in RRS (swap) typically requires two row migrations (two row-reads and two row-writes). Furthermore, if a row receives a lot of activations and requires mitigation again in 64ms, the cost for AQUA remains one row migration, whereas RRS requires four migrations (swap of both entries of the existing row-pair <X, Y> to get <X, A> and <Y, B>), a 4× overhead compared to AQUA. We observe some rows are hammered continuously in `lbm` and it incurs significantly more row migrations with RRS than AQUA. Finally, we provide an analytical model of row migration overhead of RRS compared to AQUA in Appendix A.

Thus, AQUA not only provides a more principled isolation-based security (compared to the probabilistic guarantees of RRS), but also requires fewer mitigations and each mitigation has a lower cost than RRS.

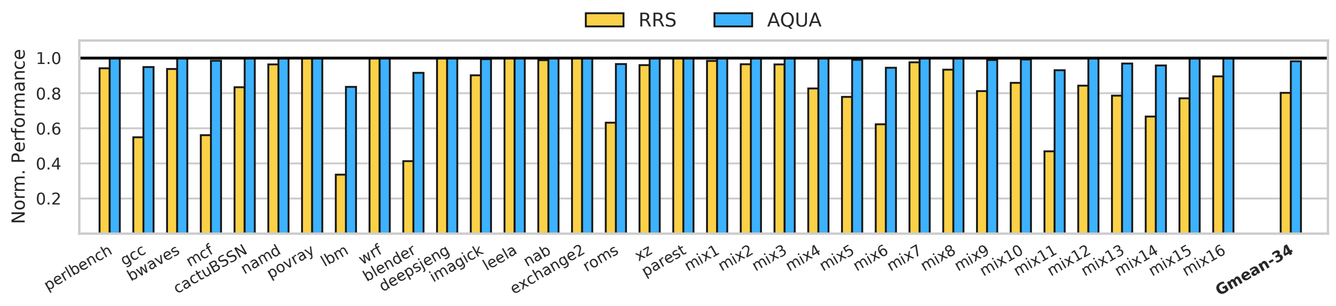


Figure 7. Performance of AQUA normalized to the baseline. *Gmean-34* denotes the geometric mean over all the 34 workloads. On average, AQUA incurs an performance loss of 1.8%, which is almost an order of magnitude less than the 19.8% performance loss with RRS (at T_{RH} of 1K).

G. *Results: Performance Impact of AQUA*

The slowdown from row-migration schemes comes from two sources: First, row migration makes the channel unavailable for servicing any memory request until the migration is complete. Second, each access needs to first lookup a table (FPT for AQUA and RIT for RRS) to decide the location to access for the given request. Fortunately, these tables are small and incur a latency of 3 to 4 cycles. Therefore, the slowdown is dominated by row migrations for both designs.

Figure 7 shows the performance of AQUA and RRS normalized to the baseline. AQUA has an average slowdown of 1.8% which is correlated with the time spent in performing row migrations. Workloads such as `wrf` and `parest` that do not encounter any aggressor rows are unaffected by our mitigation. Other workloads, such as `cactuBSSN` have a lot of rows with 166+ activations that require thousands of mitigations with RRS, have only a few rows above 500+ activations and incur no slowdown with AQUA.

Workloads such as `lbm` and `blender` encounter the most number of mitigations and the performance loss with RRS grows considerably in such cases. For instance, in case of `lbm`, RRS incurs a slowdown of almost 3×. Whereas, the slowdown with AQUA is less than 20% for lbm. Overall, AQUA has an order of magnitude lower performance loss than RRS, making it more practical for adoption, particularly as T_{RH} drops in future DRAM modules.

V. AQUA with Memory-Mapped Table

The quarantine area for AQUA incurs negligible DRAM space (1.1%). However, the structures for AQUA (FPT and RPT) still incur 172KB of SRAM overheads for each rank. In this section, we show how the SRAM overhead required for these structures can be further reduced by almost 4×.

We propose a *Memory-Mapped Table* design for AQUA that reduces the SRAM overhead of mapping tables by storing them in DRAM and caching the recent entries on-chip.[2]

[2]We note that such a memory-mapped design is not viable for RRS, as it relies on secrecy of the swapped location and variable access latency can reveal the destination of a swapped location.

Storing the FPT and RPT in DRAM presents several challenges. First, unlike the in-SRAM variant, which requires fully associative FPT, we want to avoid an in-DRAM FPT that requires multiple lookups. Second, as only a small fraction of the rows are ever quarantined, we would like to avoid accessing the FPT for non-quarantined rows. Third, we would like to avoid incurring memory accesses for the FPT even for rows that are quarantined, and ideally only perform memory accesses for FPT and RPT when row migrations are performed. Our design solves these challenges.

A. *Design Overview*

Figure 8 shows an overview of AQUA with memory-mapped tables. To simplify our design, we provision an FPT entry for each row in memory, requiring only 4MB of DRAM for our 16GB memory with 2 million rows. Similarly, we store the RPT as is in DRAM (0.1 MB of DRAM space). Both tables are accessed using extra DRAM reads and writes. RPT is accessed infrequently (only on row migrations), so we optimize the latency of accessing the FPT, which is on the critical path of every memory access.

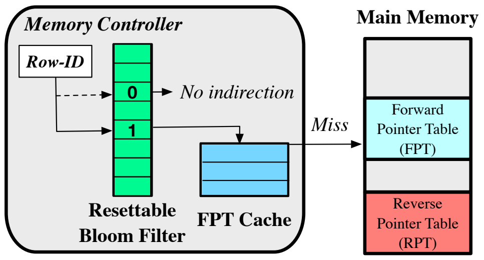


Figure 8. An overview of AQUA with Memory-Mapped Tables. The bloom filter identifies rows that are possibly quarantined, the FPT-Cache stores recent FPT entries, and the FPT and RPT are stored in DRAM.

Our design contains an *FPT-Cache* that stores the recently accessed FPT-entries. A miss in the FPT-Cache would require a DRAM access to get the FPT-entry. Even at T_{RH} of 1K, only a small fraction of the memory rows are quarantined (about 1%), so for a vast majority of the rows, this access is not needed. Leveraging this insight, our design contains a *Resettable Bloom-Filter* to quickly identify if the row is

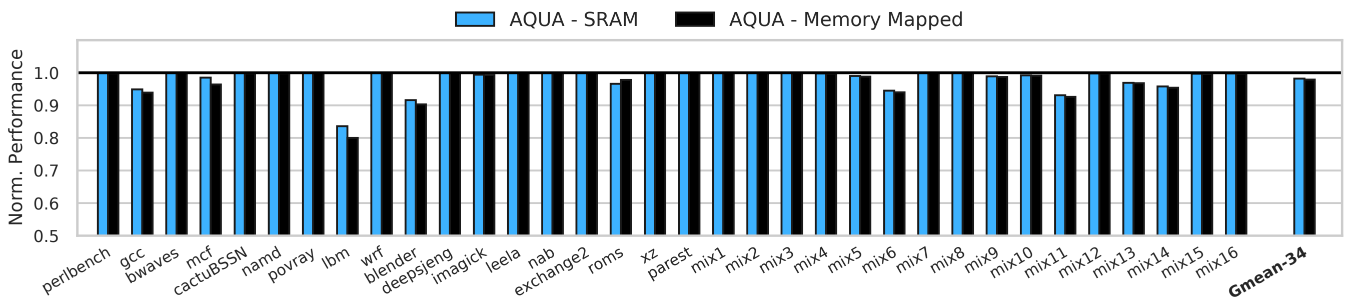


Figure 9. Performance of AQUA with SRAM and memory-mapped tables normalized to the baseline. On average, AQUA with SRAM tables incurs an average performance-loss of 1.8%, whereas with memory-mapped tables it is 2.1%.

quarantined. A zero value in the bloom filter implies that the row is not quarantined and we can access the original location. A non-zero value in the bloom filter means that row is *possibly* quarantined, and we access the FPT-Cache, and DRAM on a miss in the FPT-Cache, to confirm and obtain the mapping. Our default design uses a 128K-entry bloom-filter (16KB SRAM) and a 4K-entry FPT-Cache (16KB SRAM).

B. Resettable Bloom Filter

While bloom filters are efficient at identifying the presence of an item, it is challenging to remove an entry from the bloom filter. We want a resettable bloom-filter design, without resorting to counting bloom filters as it would incur almost $6\times$ more SRAM. We develop a design that enables resettable bloom-filters while having a single bit per entry.

We note that when a memory-mapped FPT is accessed, we get 64-bytes containing FPT entries for 32 rows. We map a group of G rows that belong to the same 64-byte line of the memory-mapped FPT cacheline to the same bit in the bloom filter. If any FPT entry in the group is valid, the bloom filter indicates a 1. When an FPT entry is invalidated, we reset the bloom-filter entry of the group only if all the remaining FPT entries in the group are also invalid (otherwise the bloom filter entry remains set). The bloom filter contains 128K entries and the memory contains 2M rows, so each group is formed over 16 FPT entries – upper half or lower-half of the 64-byte line of the memory-mapped FPT.

C. FPT-Cache Organization

The FPT-Cache is designed as a 16-way set-associative structure and uses RRIP replacement policy [11]. Each entry in the FPT-Cache contains a valid bit, tag, RRIP replacement bits, and a 2-byte FPT-entry. As the FPT-Cache is much smaller (4K entries) than the number of rows in memory, we avoid thrashing of the FPT-Cache by storing FPT-entries only for the rows that are currently quarantined. Thus, the FPT-Cache caters to a much smaller number of rows: at most 23K rows instead of all 2 million rows.

If a row finds a false positive in the bloom filter, then the FPT-Cache is guaranteed to be a miss, and the request will need to lookup the FPT entry in DRAM. We may need to do this for almost 16% of the accesses (bloom filter false-positive rate). We provide a simple optimization that avoids these lookups for 99% of the rows.

D. Filtering Singleton Groups

We have 128K groups with a 128K-entry bloom-filter. Even if 23K rows are quarantined, the likelihood that a group has more than 1 quarantined row is negligibly small. For example, on average, we expect, 84% groups to have no quarantined rows, 14.7% groups to have exactly one quarantined row, and only 1.3% groups to have multiple quarantined rows. So, majority of the unnecessary FPT lookups occur for groups that only have a single quarantined row. We call such groups that have exactly one quarantined row as a *singleton* group.

For a singleton group, an access to the quarantined row typically finds the entry in the FPT-Cache. However, accesses to all other rows in the group still needs a lookup. We filter such accesses by extending the FPT-Cache entry to include a *singleton* bit. It is set if the group has exactly one valid FPT-entry. We change the FPT-Cache indexing such that all entries of a group map to the same set. When a row looks up the FPT-Cache and gets a miss, we lookup the FPT-Cache again to check if any entry of the group is present in the same set. On such a hit, if the singleton bit of that entry is set, we know that no other row in that group has a valid FPT-entry and skip the FPT lookup. This optimization avoids DRAM access for FPT-entries for 99% of the accesses.

E. Performance with Memory-Mapped Tables

Figure 9 shows the normalized performance of AQUA with SRAM and with memory-mapped tables. The difference between the performance of these two designs is negligible. AQUA with SRAM has an average performance loss of 1.8%, and with memory-mapped tables it becomes 2.1%.

Figure 10 classifies each FPT access into four categories: (1) bloom-filter bit is reset, so no need to lookup the FPT, (2) hit in the FPT-Cache, (3) miss in the FPT-Cache but singleton bit set, and (4) DRAM access. The bloom-filter is effective at filtering out 92.2% of the accesses, whereas the FPT-Cache provides hits to 7.3% of the accesses. The singleton optimization avoids DRAM accesses for 0.4% of requests. Overall, only few ($< 0.1\%$) accesses go to DRAM.

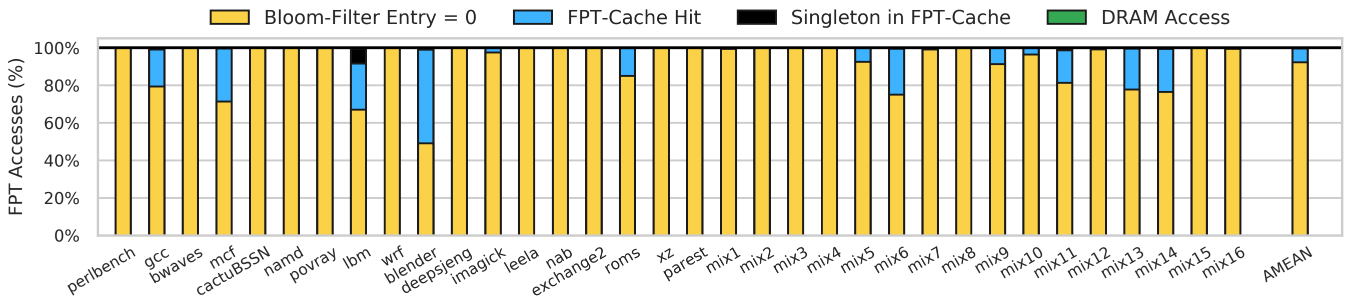


Figure 10. Breakdown of FPT-Lookups into (1) Bloom-filter bit is reset (average: 92.2%), (2) Hits in the FPT-Cache (average: 7.3%), (3) Singleton in FPT-Cache (average: 0.4%), and (4) DRAM Access (average: 0.02%).

F. Sensitivity to Threshold and Structures

We use a default Rowhammer threshold of 1K. Figure 11 shows sensitivity of AQUA to threshold. As threshold drops from 2K to 500, the performance loss changes from 0.2% to 2.1% to 6.8%. We also analyzed the sensitivity of AQUA to size of the bloom-filter and FPT-Cache. As the size of bloom-filter is varied from 8KB to 16KB to 32KB, the performance loss decreases from 2.3%, 2.1%, to 2%. The loss remained 2.1% as FPT-Cache is varied from 8KB to 32KB.

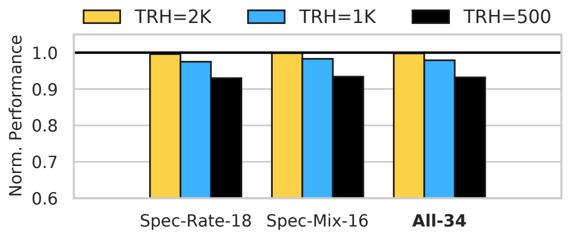


Figure 11. Sensitivity to Rowhammer Threshold.

G. Storage Overhead

With memory-mapped tables, AQUA reduces the SRAM overhead required for the tables from 172KB to approximately 32KB. Overall, AQUA requires 16KB for the bloom-filter, 16KB for the FP-Cache, 8KB for the copy-buffer, and approximately 0.6 KB to store the FPT entries for the DRAM-rows storing the FPT and RPT (to avoid recursive lookups). Thus, the SRAM overhead of AQUA for mapping and migration is only 41KB. In comparison, RRS incurs an SRAM overhead of 2.4MB. The SRAM overhead of ART is orthogonal and hence we discuss it in Appendix B.

AQUA requires DRAM space for the quarantine area (23K rows or 180MB), for storing the FPT (4MB) and the RPT (0.1MB). So, the total DRAM overhead of AQUA is 185MB, which is only 1.13% of the baseline 16GB memory.

H. Power Analysis

AQUA incurs DRAM power overhead for row-migrations and to access the memory-mapped tables, and SRAM power for the bloom-filter and FPT-Cache. For DRAM power, we use the DDR4 power model in gem5. AQUA increases DRAM power by 0.7% (8.5mW). For SRAM power overheads, we use CACTI 7.0 [3] with 22nm technology. AQUA SRAM structures incur 13.6 mW: 5.4 mW for bloom-filter, 5.4 mW for FPT-Cache, and 2.8 mW for copy-buffer.

VI. Security Analysis

We define T_{RH} as the minimum number of activations to **at least** one row within 64ms which causes a bit flip via any attack pattern (single-sided, double-sided, many-sided or Half-Double [18] or a future attack pattern). So for security of AQUA, our only assumption is:

A successful Rowhammer attack requires activating **at least** one row more than T_{RH} times within a refresh period.

A. Security Guarantee of AQUA

For security against Rowhammer, AQUA guarantees the invariant: *no physical row receives T_{RH} activations in a 64ms refresh period.* This guarantee rests on three properties:

P1 The AQUA tracker issues a mitigation every $T = T_{RH}/2$ activations to a row. The Misra-Gries based tracker (used in Graphene [25] and RRS [28]) in AQUA satisfies this property. As the tracker can be reset, up-to two tracking epochs can span the refresh period of a row. So a row can have at most $T_{RH} - 1$ activations in a refresh window before a mitigation is issued for it by the tracker.

P2 For a row migrated to the quarantine region, it can be moved back to its original location only in the next epoch. Since at most two tracking epochs span a refresh period for a row, and each tracking epoch only allows at most $T_{RH}/2 - 1$ activations at its original location, even for such cases, the original location never receives more than T_{RH} activations in a refresh period.

P3 The tracker is indexed with the address of the accessed DRAM row after consulting the FPT. So, the physical location in the quarantine region will also be identified by the tracker before it reaches T_{RH} activations in a refresh period. At that time, AQUA would perform another migration and the previous quarantine row remains unused for the remainder of the refresh period. Thus, no physical row even in the quarantine area ever receives T_{RH} activations in a refresh period.

B. *Security for AQUA's Tables*

Confidentiality: We note that the security of AQUA is not dependent on keeping the location of rows in the quarantine region a secret. Thus, confidentiality of AQUA's tables is not a concern. While the variable latency in accessing memory-mapped tables can reveal if a row is quarantined or not, AQUA's security does not rely on keeping this secret.

Integrity: AQUA stores its FPT and RPT tables in the main memory which allows an attacker to attempt a Rowhammer attack on the memory-mapped tables by inducing repeated accesses to the tables (like PTHammer [38]). AQUA avoids this by quarantining even the rows storing the tables if they get accessed beyond the threshold. The FPT-entries and RPT-entries associated with rows storing AQUA's tables are always stored in SRAM to avoid recursive lookups. The SRAM overhead is negligibly small: 512-bytes for FPT rows and 32-bytes for RPT rows.

C. *Potential for Denial of Service*

An attacker may induce repeated AQUA mitigations to slow down the memory system and impact availability. In the worst-case, an attacker can cause a new row to be quarantined every 500 activations (for $T_{RH} = 1K$), *i.e.*, once every $500 \cdot 45ns = 22.5\mu s$. If quarantine operation also requires an eviction, the total time to transfer both rows would be $2 \cdot 1.37\mu s = 2.74\mu s$. However, the attacker could also attack all the 16 banks in parallel, causing 16 mitigations every $22.5\mu s$, the memory system would be busy for $16 \cdot 2.74\mu s = 43.8\mu s$ every $22.5\mu s$. Therefore, the worst-case slowdown for AQUA is 2.95×. This worst-case slowdown is similar to the slowdown that may be observed due to row-buffer conflicts even in current systems.

VII. Related Work

Mitigating Rowhammer is an active area of research. In this section, we describe closely related prior mitigations.

A. *Mitigation via Victim-Refresh*

Prior proposals [14], [17], [25], [31], [32], [37] employ refreshing victims as the mitigation, where rows adjacent to the aggressor row restore their charge to prevent Rowhammer. Table IV compares AQUA with victim-refresh. While victim-refresh mitigates the classic Rowhammer attacks, it is vulnerable to complex patterns, such as Half-Double [18]. AQUA is robust to both classic and complex patterns. Furthermore, AQUA does not require knowledge of internal DRAM mappings.

B. *Alternatives to Victim-Refresh*

We compare AQUA to three recent schemes: RRS [28], CROW [9], and Blockhammer [36], each of which relies on mitigating action that is different from victim-refresh.

RRS [28] periodically swaps aggressor rows with a random row in memory. Because its security relies on the adversary not being able to guess the destination in DRAM, rows must be swapped every $T_{RH}/6$ activations. This results in frequent row migrations. Overall, RRS incurs performance loss of 20% and 2.6MB of SRAM at T_{RH} of $1K$.

Table IV
Comparison of AQUA with Victim-Refresh

Attribute	Victim-Refresh	AQUA
Slowdown	<0.2%	2.1%
Mitigates Classic Rowhammer (Neighboring Row Bit Flips)	✓	✓
Mitigates Complex Patterns (Far Aggressors of Half-Double)	✗	✓
Works Without Knowing DRAM Mapping	✗	✓

CROW [9] redesigns the DRAM subarray to provision extra *copy rows* (8 copy rows for a subarray of 512 rows) and uses these copy-rows to improve performance, energy-efficiency and reliability. It proposes to improve the reliability against Rowhammer by moving victim rows to the copy-rows.[3]

As CROW relies on Row-Clone [30] to perform row migration, the rows can be copied only within the sub-array of a bank. As an attacker can launch a focused attack targeting a single subarray, the defense needs to provision sufficient copy-rows in each subarray to accommodate all aggressors to ensure security. CROW [9] provisions 8 copy rows per subarray which can be overwhelmed with 4 or more aggressors rows: so it is only secure above T_{RH} of $340K$.

Table V shows the Rowhammer threshold tolerated by CROW as the number of copy-rows increase from 8 to 512 per sub-array of 512 rows. Even if CROW dedicates an additional 100% of DRAM for copy-rows, it is only secure above a Rowhammer threshold of 5.3K, whereas current memories have already been shown to have lower Rowhammer thresholds (4.8K) [15].

Table V
Rowhammer Threshold Tolerated by CROW as Copy-Rows are Increased for a Subarray of 512 Rows

Copy-Rows	DRAM Overhead	Aggressors	T_{RH} Tolerated
8 (default)	1.6%	4	340K
32	6.3%	16	85K
128	25%	64	21.3K
512	100%	256	5.3K

CROW also requires significant changes to the sub-array design (to include copy-rows) and changes to the memory interfaces to access copy-rows. In contrast, AQUA works with commodity DRAM – without changes to DRAM arrays or interface – and incurs just 1.1% DRAM overhead.

[3]We note that the description of using CROW to tolerate Rowhammer is limited to a small subsection [9]. It was not evaluated for any particular Rowhammer threshold and does not have a security analysis. We show that CROW incurs prohibitive storage overheads to ensure security for current Rowhammer thresholds.

Table VI
COMPARISON OF DIFFERENT ROWHAMMER MITIGATION SCHEMES AT T_{RH} OF $1K$ (SIGNIFICANT DRAWBACKS ARE IN **BOLD**.)

Metric	Blockhammer	CROW	CROW-Agg	RRS	AQUA
SRAM for Mapping Tables	N/A	**26 MB**	32 KB	**2.4 MB**	41KB
DRAM Storage Overhead	0%	**1060%**	**530%**	0%	1.1%
Normalized Perf. Loss (Avg)	**36%**	<0.1%	<0.1%	**19.8%**	2.1%
Worst-Case Slowdown	**1280×**	<1%	<1%	11×	3×
Commodity DRAM	Yes	**NO**	**NO**	Yes	Yes

Blockhammer (BH) mitigates Rowhammer by limiting the access rate to the aggressor rows, such that no row can have more than a specified number of accesses within 64ms. While such a design is effective at high Rowhammer thresholds (T_{RH} of 32K), it can inject unacceptably high delays at lower thresholds (T_{RH} of 1K). For example, consider a pattern that continuously accesses two conflicting rows within a bank. Such conflicting pattern takes 100ns per round and 640K rounds can be performed within 64ms. BH, however, limits the access rate to just 500 activations per row in 64ms (T_{RH} of 1K). Thus, such a pattern can only do 500 rounds in 64ms, incurring a worst-case slowdown of 1280×. Overall, BH is susceptible to significant denial-of-service attacks at T_{RH} of 1K. This concern is also applicable to regular workloads, as many workloads have thousands of aggressor rows crossing 500 activations in 64ms, as shown in Table II. In contrast, AQUA has a worst-case slowdown under pathological access pattern of only 2.95x, as described in Section VI-C.

Table VI compares the different mitigation schemes with regards to SRAM overhead required for mapping tables, DRAM overhead, slowdown (average and worst-case) and compatibility with commodity DRAM. As all designs require a tracker, we do not focus on tracker overheads.

For Blockhammer, we obtain the slowdown with an implementation in our gem5 setup using a blacklisting threshold of 256 (as per Table 7 in [36]) and an ideal tracker. For CROW, we also evaluate an aggressor-focused mitigation (CROW-Agg) that moves the aggressor row, like AQUA, and uses AQUA's memory-mapped design for the mapping tables. We estimate the average case slowdown for CROW and CROW-Agg to be negligible as the mitigative action uses in-DRAM copying with Row-Clone [30].

Blockhammer incurs significant performance overhead and is susceptible to denial-of service. CROW and CROW-Agg incur unacceptable DRAM storage overheads and require changes to the DRAM sub-array design. RRS requires impractical SRAM overhead and slowdown. Whereas, AQUA has low SRAM and DRAM overheads, low performance loss, and is viable with commodity DRAM, thus making it appealing for commercial adoption.

C. Memory Isolation via Software Support

AQUA provides Rowhammer mitigation by quarantining the aggressor row in a dedicated area in memory, thus isolating the aggressor from the neighborhood of victim rows. Memory isolation can also be provided by software-based approaches. For example, GuardION [34] inserts a guard row between data of different security domains. However, with a single guard-row, this solution can be broken with Half-Double attack. Such solutions also require the software to identify specifically which rows need to be protected (which is difficult without knowing the internal DRAM row mappings), whereas AQUA can protect all memory rows.

ZebRAM [19] and RIP-RH [5] provide isolation by partitioning DRAM and mapping kernel and user space(s) to different DRAM partitions. However, such solutions require knowledge of the DRAM row mappings and are also vulnerable to attacks like PTHammer [38] which can hammer and induce bit flips in kernel memory via page table walks. AQUA is immune to such attacks as it limits the activations possible on any single physical row.

VIII. CONCLUSION

We present AQUA, a scalable and secure Rowhammer mitigation mechanism that quarantines aggressor rows. AQUA is compatible with any tracking mechanism and performs a row migration of the aggressor to a dedicated quarantine region in the memory. We also propose memory-mapped tables for AQUA and reduce the SRAM overhead required for the mapping tables from 172KB to only 32KB, which is almost two orders of magnitude lower than RRS, which incurs an overhead of 2.4MB. Moreover, AQUA incurs a slowdown of only 2.1%. Finally, the quarantine region in AQUA incurs a negligible DRAM overhead of 1.1%. Thus, AQUA is a scalable and practical solution to mitigate Rowhammer.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for feedback. We also thank the members of the MICRO Artifact Evaluation committee for their diligence. 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.

APPENDIX A.
ANALYTICAL MIGRATION MODEL

The total overhead of row migration schemes depends on two factors: first, how frequently a row migration is incurred and second, the cost of each row migration. In this section, we develop an analytical model to analyze the relative row-migration overhead of AQUA as compared to RRS.

If a row incurs less than $T_{RH}/6$ activations, neither RRS nor AQUA perform any migrations. Let f be the fraction of rows that incur $T_{RH}/2$ activations compared to the rows that incur $T_{RH}/6$ activations. For simplicity, we assume that a row incurs either $T_{RH}/6$ activations or $T_{RH}/2$ activations. For a given f, exactly f (fraction of) rows incur a mitigation in AQUA. Whereas in RRS, f rows incur 3 mitigations each while $(1-f)$ rows incur 1 mitigation in RRS.

Figure 12 plots the relative number of row migrations in RRS compared to AQUA, r. In the best case, RRS has $r = 6\times$ overhead compared to AQUA because: (1) each mitigation launched by AQUA corresponds to 3 mitigations launched by RRS, and (2) each mitigation in AQUA performs one row migration while that in RRS performs two row migrations. Across our 34 benchmarks, the average relative row migration overhead in RRS is $r = 9\times$. Note that the estimated row migration overhead derived from our analytical model matches well with the row migration overhead obtained experimentally for the workloads, as shown in Figure 6.

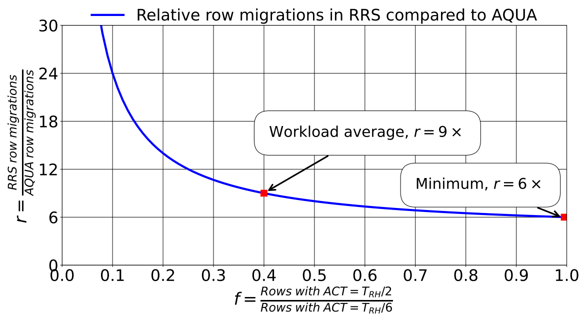


Figure 12. The relative number of row migrations, r, performed by RRS compared to AQUA. f is the fraction of rows that incur $T_{RH}/2$ activations from the set of rows that have at-least $T_{RH}/6$ activations. AQUA is guaranteed to incur at-least 6x lower migration overheads than RRS.

Appendix B. Tracker Storage Overhead

Like all rowhammer mitigations, AQUA needs a tracker to identify aggressor rows exceeding a threshold. Although AQUA uses the Misra-Gries (MG) tracker by default, the choice of tracker is orthogonal to AQUA's design, making it compatible with any aggressor tracker. Table VII shows the total storage overhead per rank for AQUA (including the tracker overhead) compared with prior mitigation RRS [28].

For a 16GB DRAM at T_{RH}=1K, AQUA incurs an SRAM overhead for its structures (mapping tables and copy buffer) of 40.6 KB (as discussed in Section V-G). With the default MG tracker, AQUA-MG incurs an SRAM overhead of 437 KB including the MG tracker SRAM overhead of 396 KB. In comparison, RRS (RRS-MG) requires 2870 KB.

However, AQUA (and RRS) can also be combined with a recent storage-optimized tracker design, Hydra [26] that uses hybrid tracking using SRAM and DRAM to reduce the SRAM overhead of tracking. Hydra requires 28.3KB SRAM (at the cost of 0.02% DRAM capacity loss). AQUA with the Hydra tracker (AQUA-Hydra) would incur 71KB SRAM overhead, while RRS-Hydra would incur a total SRAM overhead of 2502 KB.

Table VII
SRAM overheads of RRS and AQUA including trackers (with Misra-Gries [25] and Hydra [26] trackers).

Structure	RRS-MG	AQUA-MG	RRS-Hydra	AQUA-Hydra
Tracker	396 KB	396 KB	28.3 KB	30.3 KB
Mapping Table(s)	2.4 MB	32.6 KB	2.4 MB	32.6 KB
Buffer(s)	16 KB	8 KB	16 KB	8 KB
Total	2,870 KB	437 KB	2,502 KB	71 KB

Appendix C. Artifact Appendix

A. Abstract

This artifact presents the code and methodology for AQUA, our Rowhammer mitigation, along with the Misra-Gries aggressor row tracker (ART) as well as Randomized Row-Swap (RRS), another Rowhammer mitigation. We provide the C++ code for AQUA, Misra-Gries, and RRS implemented within the gem5 [24] simulator. The data structures and algorithms are implemented within the memory controller in the simulator source code. The Rowhammer defense functionality is tunable using the command line. We provide scripts to compile the simulator, and run the baseline, AQUA with SRAM tables and memory-mapped tables, and RRS for all the SPEC 2017 workloads we studied in this paper. We also provide scripts to parse the results and plot the graphs to recreate Figures 3, 6, 7, 9, 10, and 11.

B. Artifact check-list (meta-information)

- **Algorithm**: Misra-Gries tracking mechanism, Randomized Row-Swap mitigation, and Aqua mitigation.
- **Program**: 18 SPEC-2017 workload binaries checkpointed using gem5 simulator and simulated with gem5's Syscall-Emulation (SE) mode.
- **Compilation**: Tested with gcc v6.4.0 with scons v3.0.5 software construction tool.
- **Binary**: SPEC-2017 workloads- perlbench, gcc, bwaves, mcf, cactuBSSN, namd, povray, lbm, wrf, x264, blender, deepsjeng, imagick, leela, nab, exchange2, roms, xz, and parest.
- **Data set**: SPEC-2017 rate workloads mentioned above are running in ref mode.
- **Run-time environment**: Tested on RHEL 7.9 with Linux kernel 3.10 running on x86_64 server.
- **Hardware**: Requires many-core server. Our machine has 72 cores and 512GB DRAM. The experiments take 3-4 days to complete as there are about 350 gem5 instances required to reproduce all graphs.
- **Execution**: gem5 simulations.
- **Metrics**: Normalized IPC (performance metric) and number of row migrations.
- **Output**: Recreating Figures 3, 6, 7, 9, 10, and 11.

- **Experiments**: Instructions to run the experiments, parse results, and plot graphs are available in the README file.
- **How much disk space required (approximately)?**: About 10-20 GB excluding the SPEC2017 benchmark installation.
- **How much time is needed to prepare workflow (approximately)?**: SPEC-2017 installation takes between 2-6 hours and gem5 simulator compilation takes 30-60 minutes on a server system.
- **How much time is needed to complete experiments (approximately)?**: Approximately 3 days on a 72 core system. There are 350 parallelizable gem5 execution instances that each take between 6 to 24 hours.
- **Publicly available?**: Yes.
- **Code licenses (if publicly available)?**: Unlicense License.
- **Data licenses (if publicly available)?**: None.
- **Workflow framework used?**: gem5 simulator and SPEC-2017 benchmarks.
- **Archived (provide DOI)?**: The artifact is available at https://doi.org/10.5281/zenodo.6998384.

C. Description

1) How to access: The code and instructions to run the artifact are available at https://doi.org/10.5281/zenodo.6998384 and at the GitHub repository https://github.com/Anish-Saxena/aqua_rowhammer_mitigation.

2) Hardware dependencies: Running all of the simulations requires a cluster with many cores (ours has 72 cores) and large memory (ours has 512GB DRAM) – these allow running all the 34 workloads in parallel per configuration ($\sim$10 configurations including checkpointing and baseline).

3) Software dependencies: Gcc for compilation, Perl to parse results, Python with Jupyter Notebooks, and matplotlib to plot results. Additionally all the dependencies for Gem5 itself are needed. An installed copy of SPEC CPU-2017 workloads is required.

D. Installation

The `scons` software construction tool is used to compile the gem5 simulator. The SPEC-2017 workloads must be installed separately as we cannot provide them due to SPEC's restrictive license.

E. Experiment workflow

The README provides detailed instructions required to reproduce the results from the paper. These include:

- Compilation of gem5 simulator on the platform and required software dependencies.
- Creating gem5 checkpoints for the workloads.
- Executing the simulations (baseline, AQUA, and RRS).
- Parsing the simulation results and plotting the graphs to create relevant figures.

F. Evaluation and expected results

The artifact provides the scripts to collate results and gather the normalized performance (IPC) and row migration metrics that are used to plot figures. The relevant commands are provided in the artifact README. Moreover, the python scripts to plot the graphs, encapsulated in Jupyter Notebooks, is provided as well. The expected result from this artifact is to recreate Figures 3, 6, 7, 9, 10, and 11.

G. Experiment customization

Scripts to conduct test runs before launching full checkpointing and simulation runs are provided in the artifact. Although customization is not expected, it can be done in interest of limited time or resources by changing a few parameters in the scripts in the artifact.

H. Troubleshooting Steps Useful for Debugging

1) Verify gem5 experiment completion: Use the `count_successful_exps.sh` script in `scripts/` directory which counts successful runs for each multi-core configuration. If the number outputted differs from 34 for any configuration, it would mean some experiments failed for that configuration. Note that 34 is the sum of 18 SPEC-17 and 16 SPEC-MIX workloads.

2) Verify stat file generation: Compare the stats generated using the stats present in the GitHub repository in the `scripts/stats_scripts/data` directory. For each Figure, first compare the representative mean (the last line in the stat file) in both files, before comparing individual stats. The stat file `rrs_scalability.stat` corresponds to stats used in Figure 3, for example.

I. Methodology

Submission, reviewing and badging methodology:

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

References

[1] "Spec cpu2017 benchmark suite," in *Standard Performance Evaluation Corporation*. [Online]. Available: http://www.spec.org/cpu2017/

[2] (2021) "Half-Double": Next-Row-Over Assisted RowHammer. https://github.com/google/hammer-kit/blob/main/20210525_half_double.pdf.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[4] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, 2021.

[5] 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.

[6] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript," in *USENIX Security 21*, 2021.

[7] 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.

[8] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 300–321.

[9] H. Hassan, M. Patel, J. S. Kim, A. G. Yaglikci, N. Vijaykumar, N. M. Ghiasi, S. Ghose, and O. Mutlu, "Crow: A low-cost substrate for improving dram performance, energy efficiency, and reliability," in *ISCA*, 2019, pp. 129–142.

[10] M. T. Inc., "Ddr4 sdram datasheet (mt40a2g4)," 2015. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[11] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, jun 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815971

[12] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Rowhammering in the Frequency Domain," in *43rd IEEE Symposium on Security and Privacy'22 (Oakland)*, 2022, https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf.

[13] JEDEC, "Ddr4 sdram standard (jesd79-4b)," 2017.

[14] 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.

[15] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *ISCA*. IEEE, 2020, pp. 638–651.

[16] M. J. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. J. Ham, J. W. Lee, and J. H. Ahn, "Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh," *arXiv preprint arXiv:2108.06703*, 2021.

[17] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ISCA*, 2014.

[18] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering from the next row over," in *USENIX Security Symposium*, 2022.

[19] 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 - (OSDI 18)*, 2018, pp. 697–710.

[20] 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.

[21] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: preventing row-hammering by exploiting time window counters," in *ISCA*, 2019.

[22] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing rowhammer faults through network requests," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, 2020, pp. 710–719.

[23] 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*, 2022, pp. 670–684.

[24] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

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

[26] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 699–710.

[27] 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)*, Aug. 2021, pp. 1379–1396.

[28] 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.

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

[30] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 185–197.

[31] 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.

[32] 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.

[33] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, New York, NY, USA, 2016, p. 1675–1689. [Online]. Available: https://doi.org/10.1145/2976749.2978406

[34] 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.

[35] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, 2020.

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

[37] 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.

[38] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 28–41.