Simply-Track-And-Refresh: Efficient and Scalable Rowhammer Mitigation†

Eduardo Ortega*, Tyler Bletsch*, Biresh Joardar‡, Jonti Talukdar*, Woohyun Paik*, Krishnendu Chakrabarty‡‡
*Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA
‡Department of Electrical and Computer Engineering, University of Houston, Houston, TX, USA
‡‡School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ, USA

***Abstract* – Rowhammer is a memory vulnerability that can compromise system-level security. Rowhammer occurs when a DRAM row is accessed repeatedly, potentially causing bit-flips for neighboring rows. The threshold for Rowhammer has decreased from 139K accesses in 2014 to 3.2K in 2022. This threshold is projected to decrease further. Many existing solutions are not scalable, incur high overhead, or fail to offer protection in realistic scenarios. We propose Simply-Track-And-Refresh (STAR) as an effective and scalable Rowhammer mitigation. We compare STAR's performance overhead to recent solutions, HYDRA and AQUA. At ultra-low thresholds (500), STAR introduces 9.5x/31.7x lower average execution time overhead than HYDRA/AQUA. In addition, STAR introduces up to 4.3x lower area overhead and up to 3.3x lower power consumption compared to HYDRA and AQUA. We present proof of correctness, area and power consumption results derived using CACTI, and evaluation results from the PARSEC, SPLASH-2, SPEC2006, SPEC2017, and PAMPAR benchmark suites.**

***Keyword* – Rowhammer, Memory Integrity, Scalable Security**

I. INTRODUCTION & MOTIVATION

Rowhammer is a well-known memory vulnerability that remains an active threat to system-level security [1][2][8][9][30]. Rowhammer occurs when a DRAM row is accessed multiple times, which causes bit flips in physically adjacent DRAM rows (referred to as victim rows). This memory vulnerability has been demonstrated to be a security concern across multiple generations of DRAMs [2][17]. Rowhammer specifically causes bit flips by repeatedly accessing row(s) in the same DRAM bank. These frequently accessed rows are referred to as the aggressor rows. The minimum number of aggressor accesses to flip bits is referred to as "hammer count first" (HC_{first}). A major concern in system security is that the HC_{first} threshold has been decreasing since this issue was first reported. It is notable that HC_{first} has decreased from 139K in 2014 to 4.8K in 2020, and 3.2K in 2022 [1][17][7]. In addition, it is projected that HC_{first} will continue to decrease due to technology scaling. A lower Rowhammer threshold will make the attack easier to achieve and more likely to occur [2][3][17].

In each DRAM bank, a new row can be accessed every 45 ns (often referred to as t_{RC} for DDR4 and DDR5) [20][22]. A particular DRAM row will experience an automatic refresh every 64 ms (t_{REFW}). Overall, within a 64 ms refresh window period, we can access a maximum of ~1360k rows per bank. Only HC_{first} (out of ~1360K) DRAM row accesses need to be to the same row to cause a bit flip, i.e., there are $\sim 1360\text{K}/HC_{first}$ potential Rowhammer aggressors possible in the worst case. With lower HC_{first}, an attacker can target a larger number of rows per bank. For example, with HC_{first} of 3.2K, an attacker can target over 425 rows per bank simultaneously. In addition, lower HC_{first} enables the attacker to create various permutations of malicious accesses through choice of row and timing of access. It is typically assumed that Rowhammer attacks cause bit flips in the victim row(s) immediately adjacent to the row being hammered (aggressor). However, new attack methods have shown that Rowhammer can occur in rows that are farther away. The distance between the farthest victim and aggressor row is defined as the 'attack radii'. Half-double attack can cause bit flips at attack radii of two, i.e., the victim is two rows away from the aggressor.

Most mitigation schemes utilize refreshes. However, refreshes require accessing the victim row. We can use this mitigation mechanism as a new way to cause Rowhammer at rows that are far away. For instance, let us assume $HC_{first} = 1000$. As mentioned earlier, a single row r can be accessed 1360K times, which will cause bit flips in the rows $r \pm 1$. To prevent bit flips, we must refresh (access) the rows $r \pm 1$ at least 1360 times (1360K/ HC_{first}). However, due to the mitigating refresh to rows $r \pm 1$, the number of accesses is higher than HC_{first} and will therefore cause bit flips in the rows $r \pm 2$. As illustrated by this example, it is possible to cause bit flips at an attack radius of two by cleverly using the mitigation mechanism itself. Existing refresh-based Rowhammer mitigations do not consider this problem and are vulnerable to potential misuse. Hence, we must take into consideration that refresh mitigation may be used maliciously for Rowhammer [7]. This kind of attack is more serious at ultra-low HC_{first} (below 1K) as more refreshes will be triggered causing bit flips

†This work was supported by the National Science Foundation by grant no. CNS-2011561 and the Semiconductor Research Corporation under contract no. 3199.001.

in rows far from the aggressor. This mitigation-enabled exploit must be taken into consideration for Rowhammer defenses.

Previous countermeasures from industry practitioners have recently been shown to be ineffective against Rowhammer [10][11][19]. Several other solutions have been shown to be expensive at ultra-low HC_{first} , i.e., they introduce high overhead as the trigger threshold decreases [4][5][6]. To address Rowhammer attacks at ultra-low HC_{first}, we propose Simply-Track-And-Refresh (henceforth referred to as STAR). STAR is an effective and efficient tracking method for scalable Rowhammer mitigation. One of the key features of STAR is that we only count until specified mitigation threshold that is less than HC_{first}. This enables STAR to use smaller counters compared to other existing solutions [4] which leads to lower implementation overhead. In addition, STAR utilizes associative storage for faster lookup and uses proactive refreshes to prevent bit-flips. STAR utilizes a combination of deterministic and probabilistic refresh-based mitigations schemes to address the Rowhammer vulnerability. At ultra-low Rowhammer thresholds (500), STAR has an average execution time overhead of 0.53% when 43 different applications are considered, and it outperforms two recently proposed Rowhammer mitigation schemes.

The remainder of the paper is organized as follows. In Section II, we discuss related prior work on Rowhammer attacks and Rowhammer defenses. In Section III, we review our threat model and the proposed countermeasure (STAR). In addition, we review the mitigation strategy and provide proof of correctness. In Section IV, we discuss our experimental methodology and design parameters for STAR. In Section V, we describe our experimental setup and results, and present an analysis based on our results. We compare the performance, power consumption, and area overhead of the proposed solution with two recently proposed Rowhammer defenses (HYDRA and AQUA). Section VI concludes the paper.

II. Related Prior Work

A. Rowhammer Attacks

A Rowhammer attack can be conducted in different ways. The simplest attack, single-sided, occurs when a single aggressor row is accessed HC_{first} number of times. A double-sided Rowhammer attack occurs when two aggressor rows sandwich the victim; each aggressor row is accessed $HC_{first}/2$ times to achieve bit flips. These attack methods are used in other open-source Rowhammer frameworks such as TRRespass and BLACKSMITH [11][19]. TRRespass uses multiple interlacing aggressors to attack multiple victim rows simultaneously [19]. BLACKSMITH attacks by hammering different rows in a semi-random fashion [11]. Another attack, half-double, occurs when the attacker attempts to flip a bit in row *r* by accessing row *r*+1 (the "near aggressor") at low frequency and row *r*+2 (the "far aggressor") at high frequency [28]. Half-double Rowhammer attacks cause bit flips at an "attack radius" of 2 from the far aggressor. Another recent Rowhammer attack scheme called feinting [7] utilizes decoy aggressor rows to slowly build up the number of accesses required to flip bits. Interestingly, an attack method can even be deployed to exploit the additional refreshes introduced by a Rowhammer defense system itself to cause Rowhammer at adjacent rows next to mitigated victims [7]. We refer to this exploit as *refresh-assisted Rowhammer*. At lower HC_{first} these attacks become more dangerous as there are more potential victim rows and more ways an attacker can build the attack. Previous refresh-based Rowhammer defenses do not address this attack as described in the next section. In this paper, we aim to address all attack scenarios at lower cost (e.g., power consumption, performance impact, area overhead).

B. Previous Rowhammer Defenses

Target Row Refresh (TRR) is a mitigation strategy adopted in commercial DRAMs to prevent Rowhammer attacks [10][32]. However, attacks on TRR, such as TRRespass and BLACKSMITH, have shown that this mitigation technique is not adequate [11][19]. Pro-TRR [7] is an improvement to TRR that adopts a proactive stance to mitigate Rowhammer. However, Pro-TRR cannot provide guaranteed security against Rowhammer [7]. Other solutions, Graphene and Blockhammer, have been proposed to defend against Rowhammer [4][5]. These solutions are implemented in the memory controller and do not require modification to the DRAM. However, they do not scale effectively at ultra-low HC_{first}, e.g., HC_{first} of 1K and lower [3][6]. With a lower HC_{first} threshold, Graphene's table size requirement increases exponentially. Graphene utilizes the Misra-Gries algorithm which is a highly effective tracking scheme but requires high overhead for low tracking thresholds [4][34]. At ultra-low thresholds, ($HC_{first} < 1000$), Graphene requires high area overhead as it needs to store large amounts of tracking information in associative storage for its Misra-Gries implementation [3][4]. Similarly, Blockhammer leads to an average execution time overhead of 36% at $HC_{first} = 1000$ [6].

Recent Rowhammer mitigation strategies such as HYDRA and AQUA have been designed for ultra-low HC_{first} [3][6]. These solutions require tracking tables in both the DRAM module and the memory controller. For example, a large Row Count Table is needed for HYDRA and a dedicated Row Quarantine Zone is needed inside the DRAM for AQUA [3][6]. These solutions are associated with higher design complexity as require coordinated changes to the memory controller and the DRAM. Thus, these other solutions are therefore less likely to be adopted in practice. In conjunction, mitigation from HYDRA is refresh-based and mitigation from AQUA is aggressor row isolation (referred to as row migration) [3][6]. While AQUA's row migration inherently addresses the refresh-assisted Rowhammer, HYDRA does not consider this Rowhammer variant within scope [3][6]. A machine learning-based solution has been proposed in [18]. However, this solution cannot guarantee detection as machine learning models rarely achieve 100% detection accuracy. In addition, the detection accuracy decreases significantly at low HC_{first}. We address these shortcomings of existing techniques by proposing STAR, which is implemented in the memory controller and can track Rowhammer attacks at very low overheads.

III. PROPOSED COUNTERMEASURE: STAR

A. Threat Model

We assume that the attacker has knowledge of the type of DRAM used in the targeted system, i.e., the attacker knows the make or model of the DIMMs used in the targeted victim system. We assume that the attacker knows the virtual to physical memory mappings by using techniques such as [27] and [31]. We also assume that the attacker can execute benign code that can attempt the Rowhammer vulnerability at any time for an unlimited amount of time [30]. However, we make no assumptions about whether the attacker has physical access to the target victim system, i.e., having physical access is not prerequisite for a Rowhammer attack [33]. An attacker's aim is to use Rowhammer to raise user privileges, steal IP or execute a system-wide takeover. An attack is successful if an attacker can access a row enough number of times within the refresh period of 64 ms to cause a bit flip. If the attacker can access a row enough times to execute an attack listed previously, e.g., single-sided, double-sided, feinting etc., then the attack is deemed to have been successful.

B. STAR Tracking Algorithm

To enable our Rowhammer defense, we illustrate the following tracking algorithm for STAR as shown in Figure 1. We start with an associative table of counters per bank and the specified tracking threshold, HC_{thr}. We discuss the issue of the number the counters later in Section IV. In addition, we discuss the specified tracking threshold (HC_{thr}) later in this section. Each entry in a table is made up of 'tag' and 'data' tuples. Each tag represents the row being accessed, and each data acts as the tag's counter (access counter). The algorithm starts when a new DRAM Row request occurs (Figure 1). When a new DRAM row request occurs, we check if the row request is being tracked. If the row request is being tracked, then we proceed to check its associated counter. If the counter value (data) is equivalent to HC_{thr}, then refresh mitigation (on victim rows) is queued and the tag is evicted. If the associated counter is not equivalent to HC_{thr}, we simply increment the data by 1.

If the DRAM row request is not already in the counting table, then we proceed to check if the table is full. If the table is not full, then we insert the new row (tag) into the table and initialize the data (counter) to 1. If the table is full, we implement a data (counter) value lookup to find the associated tag with the max data value. Once found, we queue refresh mitigation on the associated tag's victims. The associated tag is evicted from the table and replaced by the current row request. After executing the tracking algorithm, the table waits for the next row access event to re-execute the tracking algorithm.

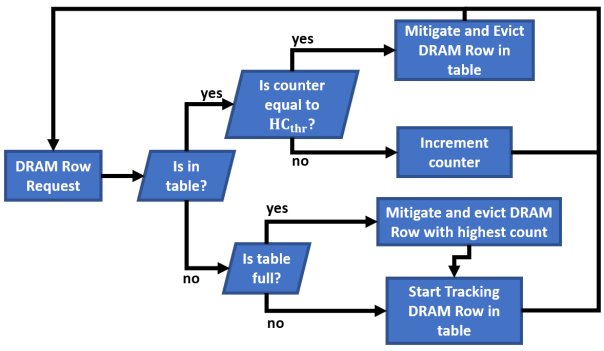


Figure 1: Flowchart for the STAR tracking algorithm.

C. STAR Mitigation Strategy

As shown in previous work, HC_{first} may vary from 3.2K to 16K [7][10] and the lower end of this range is projected to decrease further. For STAR, we consider HC_{first} values ranging from 16K to as low as 500 to assess and demonstrate the overall efficacy of STAR. Note that the value of HC_{first} may vary based on manufacturer [2][36]. In addition, HC_{first} is a DRAM hardware metric that system integrators can determine using open-source Rowhammer frameworks such as BLACKSMITH [11]. While the specific HC_{first} value depends on the DRAM, there is a clear trend that these Rowhammer thresholds are getting smaller [2]. Hence, the purpose STAR's mitigation strategy is to ensure scalability, whereby low-cost Rowhammer detection can be achieved even for small values of HC_{first} .

STAR monitors row accesses within the memory controller and refreshes rows being targeted by potential Rowhammer events. The input to STAR is the current row access request (potential aggressor rows). The output is a set of control signals to queue refresh operations if a potential Rowhammer attack is detected. STAR's mitigation is triggered based on a tracking threshold (referred to as HC_{thr}), where $HC_{thr} = \lfloor HC_{first}/4 \rfloor$. We discuss this computed value of HC_{thr} in Theorem 1 later in this section. HC_{thr} enables early detection of attacks such as single-sided, double-sided, and feinting. Once detected, these attacks can be mitigated with targeted adjacent row refreshes to the immediately adjacent rows. However, a targeted row refresh to the adjacent victim row is not sufficient to prevent half-double attacks. In addition, as discussed earlier, a scheme that simply refreshes adjacent rows can be vulnerable to exploits that utilize these additional refreshes for a Rowhammer attack.

In the case of half-double attack, the attacker accesses the row r with high frequency and rows $r \pm 1$ less frequently. The number of accesses to $r \pm 1$ is generally very low and remains undetected. The higher number of accesses to row r can confound the mitigation mechanism and lead to a refresh on rows $r \pm 1$, leaving $r \pm 2$ unattended and vulnerable to bit flip. Similarly, an attacker can use these extra refreshes as an indirect attack mechanism [7], as we have discussed in Section 1. Most existing Rowhammer solutions fail to protect against these new attacks. One way to address this problem is to refresh both rows $r + 1$ and $r + 2$ any time an attack is suspected; this method is used in HYDRA. This is wasteful as we will demonstrate.

To prevent half-double and refresh-assisted Rowhammer attacks (in addition to the other attacks) [2][7][11][19][28], we utilize a hybrid solution that uses both deterministic and probabilistic refreshes. Anytime a Rowhammer attack is detected/suspected on row r, we use a deterministic refresh on rows $r \pm 1$. These deterministic refreshes prevent the simpler forms of Rowhammer attacks, e.g., single-sided, double-sided, TRRespass, Blacksmith, and feinting [2][7][11][19]. Apart from these deterministic refreshes, we also utilize a probabilistic

refresh on farther rows (i.e., $r \pm 2$). These probabilistic refreshes are considered every time a deterministic refresh occurs. The probabilistic refreshes prevent the half-double and refresh-assisted attacks. We choose this hybrid method as it outperforms HYDRA (shown later) that always refreshes both rows $r \pm 1$ and $r \pm 2$. Since not all attacks are half-double or refresh-assisted variants, refreshing both rows every time is unnecessary in most situations.

Since we are less likely to see bit flips at rows far away (e.g., rows $r \pm 2$) than directly adjacent victim rows (rows $r \pm 1$) [2], we do not need to refresh them as frequently. For probabilistic refresh, we determine a suitable probability that leads to an extremely low chance of success for Rowhammer. Our target for a successful Rowhammer attack is the bit error rate (BER) of the underlying DRAM, e.g., 10^{-15} as reported in [29]. In addition, to prevent refresh-assisted Rowhammer and achieve the acceptable BER, we must also refresh with a higher radius (e.g., $r \pm 3$, etc.). As mentioned earlier, existing solutions limit their mitigations to rows $r \pm 1$ and $r \pm 2$. However, these additional refreshes to rows $r \pm 1$ and $r \pm 2$ can lead to bit flips on rows $r \pm 3, r \pm 4$, and so on. This problem must be addressed. Hence, we must determine the refresh failure probability so that the attack success rate for any row is lower than 10^{-15} [29].

Overall, STAR counts only up to HC_{thr}, which leads to smaller counters and hence reduces storage overhead. In addition, once a row is accessed HC_{thr} number of times, its direct adjacent neighbors are refreshed deterministically, and further neighbors ($r \geq 2$) are refreshed probabilistically. Once mitigation occurs, the entry is no longer tracked, and the corresponding counters are then reset and used to track other (or the same) potential aggressors.

D. Security Assessment

Here, we present the proof of correctness for our mitigation approach. Theorem 1 shows our method can prevent single-sided, double-sided, and feinting Rowhammer attacks. In addition, Theorem 1 explains when STAR deterministic mitigation must be triggered. Theorem 2 addresses half-double and refresh-assisted Rowhammer attacks. It also explains when STAR probabilistic mitigation must be triggered.

***Theorem 1*: Given a value of HC_{thr}, STAR can detect and mitigate single-sided, double-sided, and feinting Rowhammer attacks.**

Proof: We use Figure 2 for the proof. STAR's counting table is reset every 64 ms. This is done to keep the counter size small. In addition, DRAM rows are automatically refreshed every 64 ms. However, the table reset, and the automatic DRAM refresh commands are not synchronized. As shown in Figure 1, a row r can be refreshed Δ units of time after the table are reset ($0 \leq \Delta <$ 64 ms). This represents a generic scenario since STAR table resets are not synchronized to automatic refreshes. X_a represents the number of times aggressor row r is accessed between the time it is refreshed ($t_o + \Delta$ in Figure 2) and when the table is reset ($t_0 + 64$ ms). Similarly, X_b represents the number of times aggressor row r is accessed between the time the table is reset ($t_0 + 64$ ms in Figure 2) and when row r is refreshed next ($t_0 + \Delta + 64ms$). A successful Rowhammer attack must meet the following condition (assuming double-sided attack, as it requires fewer accesses to cause Rowhammer):

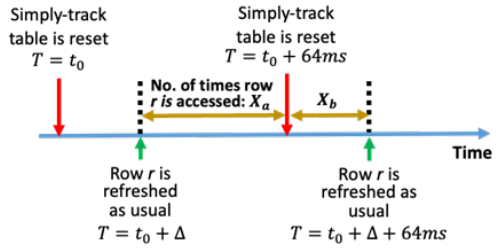


Figure 2: Rowhammer attack pattern analysis used in Theorem 1.

$$X_a + X_b = HC_{first}/2 \quad (1)$$

STAR can detect an attack if ($X_a = HC_{first}/2, X_b = 0$), and vice versa. STAR counting table resets cause the proposed countermeasure to lose tracking information X_a for potential aggressor rows. Hence, the worst case occurs if the attack is distributed across table resets. When the attack is distributed across resets (the worst-case scenario), then X_a and X_b are non-zero values (or $X_a, X_b \neq 0$).To prevent a successful Rowhammer attack, we must appropriately set HC_{thr} such that STAR can detect a Rowhammer attack even in this worst case (where the table is reset, and all prior tracking information is lost). For this purpose, we choose the threshold considering the maximum between X_a, X_b.

$$X = \max\{X_a, X_b\} \quad (2)$$

Considering the range of maximum values in Equation (2) subject to the constraints in Equation (1), we find that:

$$HC_{first}/4 \leq X < HC_{first}/2 \quad (3)$$

The threshold must be set to detect Rowhammer attacks under all scenarios. From Equation (3), we can see that the worst case happens when $X = HC_{first}/4$. Hence, HC_{thr} must be set as described in Equation (4).

$$HC_{thr} = HC_{first}/4 \quad (4)$$

Therefore, for the generic case of $\Delta \geq 0$, we can mitigate single-sided, double-sided, and feinting Rowhammer attacks, as they rely on Equation (1), when HC_{thr} is set to $HC_{first}/4$. This proves that STAR can detect these Rowhammer attacks when the threshold is set appropriately. □

***Theorem 2*: STAR's probabilistic refreshes can mitigate half-double and refresh-assisted Rowhammer attacks with an acceptable failure probability (less than or equivalent to BER).**

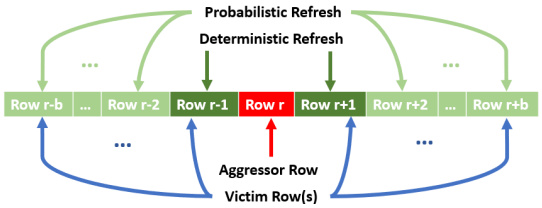


Figure 3: An example of the STAR probabilistic mitigation scheme.

Proof: Even when Rowhammer is suspected, we do not know what kind of attack might have been employed. The immediate neighboring rows are refreshed deterministically. However, as mentioned earlier, this is not sufficient for half-double and refresh-assisted attacks. To prevent these attacks, we use a probabilistic refresh for rows that are at higher radiuses (Figure 3). In this work, we aim to achieve a failure probability comparable to the DRAM BER of 10^{-15} [29]. To accomplish this, we must pick suitable refresh probability values (referred to as p) and an appropriate refresh radius value (referred to as b). The values of p and b can vary with the type of attack. Here, we use p_{HD} and p_{RA} to represent the probability of refresh for the half-double and refresh-assisted attacks, respectively. We consider HC_a as the aggressor row hammer count with the largest attack radius. We select HC_a based the half double high-frequency aggressor [28], and aggressor access count required for the refresh-assisted attack as described earlier. We consider HC_a values from both attack scenarios as they have the same attack radius (2) and have different associated HC_a values. Like HC_{first} values, HC_a is a hardware metric that system integrators can compute through Rowhammer frameworks [9][11][19].

Note that our probabilistic refresh is considered when deterministic refresh occurs. Deterministic refreshes are based on how many triggers are issued across HC_a aggressor accesses. Hence, there are HC_a/HC_{thr} opportunities for the probabilistic refresh to occur and extend the refresh to $r \pm 2$, etc. The failure probability for the refresh to not extend is $1-p$. Thus, the failure probability for the probabilistic refresh extension is equivalent to $(1-p)^{HC_a/HC_{thr}}$.

$$(1-p)^{HC_a/HC_{thr}} \leq BER, \text{ where } 0 < p \leq 1 \qquad (5)$$

Equation (5) shows the general form for the valid range of p. However, we do not want to probabilistically mitigate needlessly. Thus, to ensure minimal performance overhead, we want a small refresh probability. To meet this objective, we utilize a closed form solution as shown below.

$$p = 1 - BER^{HC_{thr}/HC_a} \qquad (6)$$

By solving Equation (6), we ensure our chosen probability value p provides sufficient failure probability. For our study, both attack scenarios (half-double and refresh-assisted) have equidistant attack radius of 2. Thus, both the probabilities and related HC_a values are considered. We first solve for p for each attack scenario using Equation (6). Higher probabilities will also cover the cases with lower probabilities. The probability p in this study is chosen as the maximum of the probabilities.

$$p = \max\{p_{HD}, p_{RA}\} \qquad (7)$$

Note that the probability value p from Equation (7) is chosen for the first probabilistic refresh extension ($r \pm 2$). We consider further probabilistic refresh extensions (e.g., $r \pm 3$, $r \pm 4$, etc.) specific to the refresh-assisted Rowhammer attack. The precomputed p_{RA} value is used for these further probabilistic refresh extensions. We utilize both p and p_{RA} to consider Rowhammer attacks with the largest attack radius and mitigate the potential refresh-assisted Rowhammer, respectively. Thus, the combination of p and p_{RA} for each probabilistic refresh extension provides sufficient failure probability. However, to provide sufficient failure probability for all extensions, a b value must be selected such that the failure probability for all probabilistic refresh extensions is also below the BER.

$$(p)(1-p_{RA})(p_{RA})^{b-2} \leq BER, \text{where } b > 2 \qquad (8)$$

Equation (8) shows the general form for the valid range of b. However, we only want to support a finite number of refresh extensions as to not needlessly extend refreshes. To ensure minimum performance overhead, we want a small refresh radius b. We utilize the following closed-form solution for the refresh radius b.

$$b = \log_{p_{RA}}(BER/(p(1-p_{RA}))) + 2 \qquad (9)$$

Equation (9) is derived based on p and p_{RA} that satisfy Equation (6). By solving Equation (6) and Equation (9), we ensure that STAR's probabilistic refreshes have acceptable failure probability. □

E. Hardware Implementation

Figure 4 shows the block-level implementation of STAR. STAR uses associative storage (content-addressable memory or CAM), digital comparator, and a STAR FSM that implements the tracking algorithm (Figure 1). The digital comparator is a combinational digital circuit that searches for the greatest data

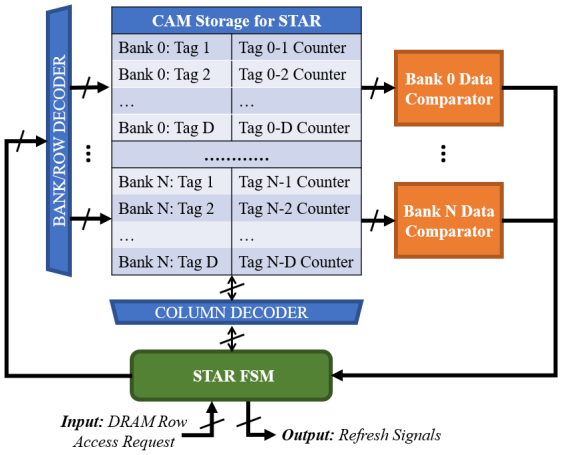
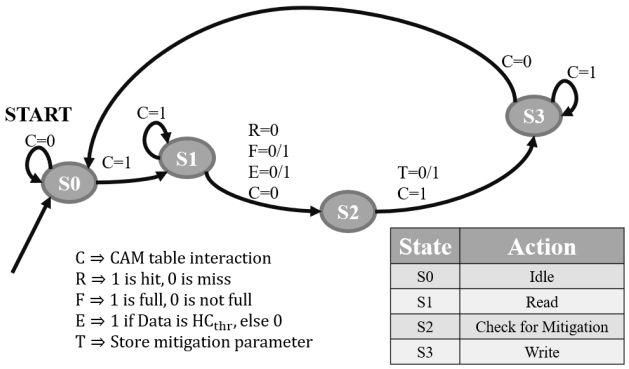


Figure 4: Block-level hardware architecture of STAR.



State	Action
S0	Idle
S1	Read
S2	Check for Mitigation
S3	Write

Figure 5: STAR finite state machine.

value in a per-bank table through parallel comparison. The digital comparator utilizes XOR gates to note the difference in bits for comparison and is used by STAR FSM. The STAR FSM consists of four states (Figure 5). The STAR remains idle until a memory request occurs (S0). When a memory request occurs, the DRAM memory address will be attempted to be read (S1). From the result of read, we check per each mitigation scenario in S2. The result of the mitigation checks leads to a write (S3). The entry written back to the table is either an update to the tracked entry, entry eviction, or entry replacement. From this write (S3) and the result of the state prior (S2), the output of the hardware is set to queue refresh operations to mitigate the neighbors of the potential victim rows. We synthesized the STAR FSM using Synopsys Design Compiler (45 nm). The area overhead associated with the STAR is negligible, only 550 gates (less than $0.0006\ \mathrm{mm}^2$ for the 45 nm Nangate library).

We utilize a CAM table to support associative operations, each CAM table entry includes a valid bit, tag with built in comparator, and data [23]. In practice, fully associative storage architectures lead to more area and static power. We show later that the overhead is small as the built-in comparators enable a parallel search across the tags, which reduce lookup latency times compared to lower associativity structures. To minimize area overhead, we place all per-bank tables in one associative storage array to reduce peripheral circuits.

IV. Experimental Evaluations

We utilize the full-system simulator Gem5 to get the memory access traces for a variety of applications [25]. We simulate a x86 4-core system running at 2 GHz in Gem5 with a Linux kernel. Each core has a 64 KB private L1 cache. The L2 cache size is 1 MB and it is the last-level cache (shared by all cores). The DRAM follows DDR4 timing specifications and has 16 banks (~65K rows) per rank and operates at 2.4 GHz. We utilize a wide range of applications from five different benchmark suites to analyze our proposed solution. These applications belong to PARSEC[1], PAMPAR[2], SPLASH-2[3], SPEC2006[4], and SPEC2017[5] [12][13][14][15][16]. In addition, we include multi-program settings, where a combination of workloads across all benchmarks are run simultaneously. We choose five multiprogram settings (referred collectively as MULTIPROG[6]) that lead to very high memory access. From a trace analysis of the Gem5 workloads, the maximal unique row accesses per bank was 1086 for single-program workloads and 1922 for multi-program workloads (with 64ms refresh window granularity). However, most workloads only had ~600 unique row access per bank. Hence, an efficient Rowhammer tracking table can be implemented with up to 600 counters per bank. We did a grid search of our sizing parameters and found that a table depth of 400 led to the most suitable performance-power trade-off results. Hence, we use 400 table entries for our solution.

We use DRAMSIM3 for cycle-accurate simulations of the proposed solution [24][25]. We modify DRAMSIM3 to implement our solution and other existing solutions. In this work, we choose two state-of-the-art solutions, namely HYDRA and AQUA, for comparison as they provide deterministic guarantees [3][6]. For fair comparison, we implement HYDRA and AQUA with their recommend parameters with per-bank granularity as to not hinder performance across [3][6]. For HYDRA, we implement the refresh radius 2 as described [3]. For AQUA, we implement row migration [6]. Other recent solutions such as a Machine Learning and ProTRR are not chosen for comparison as they do not offer full security guarantees against Rowhammer [7][18]. We test STAR (along with HYDRA and AQUA) with real Rowhammer attacks, such as the Google's Rowhammer framework (referred as g-hammer hereafter), BLACKSMITH, and TRRespass [9][11][19]. To estimate area and power consumption, we use a popular memory simulator, CACTI [26]. CACTI report timing results, leakage, power consumption, and area for various parameters (e.g., node size, associativity, etc.). We use the 45 nm process node to estimate the area and power overhead in CACTI.

V. Results, and Analysis

A. Performance Overhead

STAR provides two performance improvements: Rohwammer false positive overhead (established as false positive rate) and execution time overhead. We utilize both false positive rate (FPR) and execution time to showcase these performance improvements of STAR. For conciseness and to show ultra-low threshold tracking effectiveness, only HC_{first} of 500 is shown to represent FPR at ultra-low Rowhammer thresholds. At ultra-low thresholds, HC_{first} of 500, we consider these benign applications will have true positives. We implement an ideal-counting solution, a counter per DRAM

[1]Blackscholes (BS), Bodytrack (BT), Canneal (CA), Dedup (DD), Fluidanimate (FA), Freqmine (FM), Streamcluster (SC), Swaptions (SW), x264 (X2)
[2]Discrete Fourier transform (DFT), Dijkstra's shortest path (DJ), Dot-product (DO), Gram-Schmidt (GS), Harmonic sum (HA), Jacobi method (JA), matrix multiply (MM), Odd-even sort (OE), Histogram similarity (HS), Turing Ring (TR)
[3]Fast Fourier transform (FFT), LU Factorization (LU), Ocean (OC), Radix (RA), Raytrace (RT), Volrend (VO), Water (WA)
[4]Bwaves (BW), Bzip-2-compress (BC), Bzip2-decompress (BD), Hmmer (HM), Leslie (LL)
[5]Deepsjeng (DJ), Exchange2 (E2), LBM (LB), Mcf (MC), Namd (ND), Povray (PR), Specrand (SR)
[6]Deepsjeng, Blackscholes, and Fast Fourier transform (M1), LBM and Swaptions (M2), Namd, LU Factorization, and Blackscholes (M3), Fluidanimate, Fast Fourier transform, and Specrand (M4), Exchange2, Radix, and Freqmine (M5)

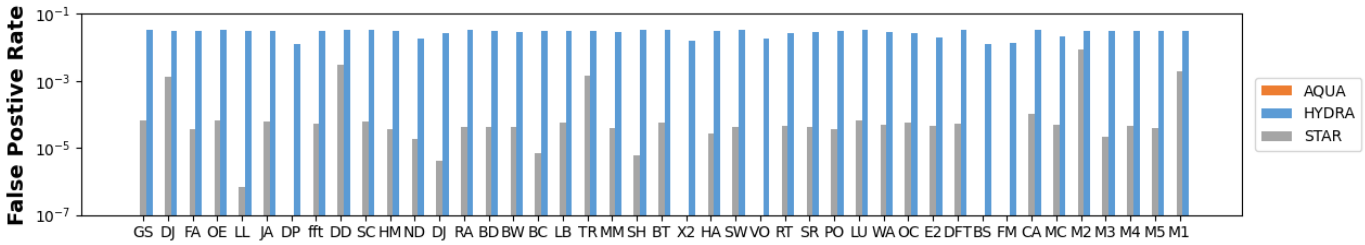


Figure 6: False Positive Rate of chosen benign benchmarks (HC_{first} is 500).

row, to determine true positives. We utilize the ideal-counting solution to determine false positives of all other solutions (AQUA, HYDRA, and STAR). Please note that the ideal counting solution is unrealistic as it is too expensive to implement a counter per row. The ideal counting solution is not chosen for future metric(s) comparison.

Figure 6 shows the FPR of benign applications across all simulated solutions. As seen in Figure 6, AQUA's has very few to almost no false positives (tested FPR is 0.0). This is because AQUA's tracking scheme utilizes the Misra-Gries algorithm [6]. Misra-Gries is a very efficient counting algorithm that leads to low or almost no false positives [4][34]. However, AQUA's usage of Misra-Gries is very expensive in terms of area and execution time (shown later). From Figure 6, we establish that STAR has lower false positives than HYDRA across all benign applications. This indicates that STAR has a more effective tracking scheme than HYDRA. HYDRA utilizes one counter for multiple rows; therefore, the actual count of a row access is often overestimated. Hence, we see more false positives and by extension, higher execution time overheads at ultra-low HC_{first} values. The impact of STAR's lower FPR can be seen in our execution time results shown later.

Figure 7 presents STAR's overall execution time overhead normalized to a system with no Rowhammer mitigation in-place. For conciseness, only HC_{first} of 500 and 16K is shown to represent our minimum and maximum of our considered Rowhammer thresholds. As seen in Figure 7, the normalized execution time overheads show little variance compared to the non-secure benchmarks. The worst-case scenario occurs for the single-program workload dedup (DD) with 5.4% execution time overhead. This happens because many unique rows are accessed (900+) per bank, each of these rows are accessed multiple times (470+) in DD; these repeated accesses to multiple rows can cause bit flips when HC_{first} is very low. Such behavior is not observed for other applications. Next, we compare the average execution time overhead across all benchmarks for all the implemented solutions. In Figure 8, the average execution time overheads for all benchmarks are shown for each HC_{first} value. Across HC_{first} values, the average execution time overhead for HYDRA and AQUA increase relative to lower HC_{first} values.

STAR's average execution time overhead considering all benchmarks is low (0.064%) for HC_{first} of 16K. AQUA's average execution time overhead is 1.4 × higher for HC_{first} of 16K and increases to 31.7 × higher for HC_{first} of 500. AQUA's average execution time overhead is higher is due to its mitigation scheme based on row migration. Row migration requires copying (moving) data, which is more costly than a simple refresh-based mitigation. Hence, row migration leads to higher execution time overheads across all HC_{first} values. HYDRA's average execution time overhead at HC_{first} of 500 is 5.1% while STAR's average execution time is only 0.53% at such ultra-low HC_{first}. It is notable that STAR's average execution time overhead is lower for ultra-low HC_{first} values. STAR is more efficient because HYDRA leads to more false positives at ultra-low HC_{first} values. We present more information on false positive rate in the next section (Table 1).

We see similar results when we compare the maximum execution time overheads considering all the benchmarks. At HC_{first} of 16K, STAR's maximum execution time overhead is 1.3%, which is slightly higher than HYDRA's and AQUA's maximum execution time overhead. However, at HC_{first} of

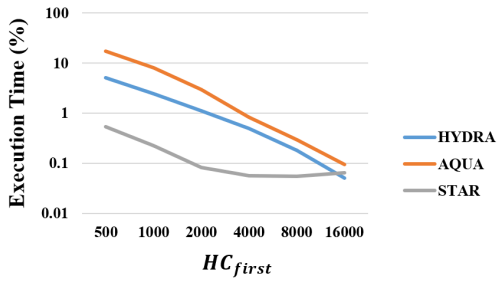


Figure 8: Average execution time overhead.

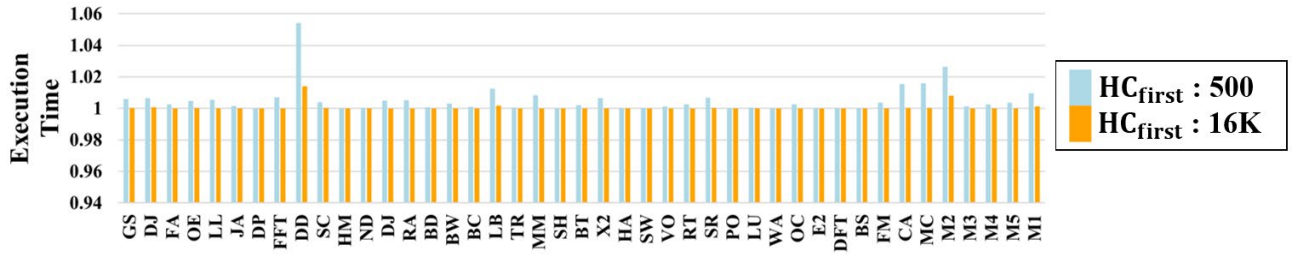


Figure 7: STAR normalized execution time overhead across chosen benign benchmarks.

500, STAR's maximum execution time overhead is 5.4% while HYDRA's and AQUA's maximum execution time overhead is 8.0 × and 27.0 × greater than STAR, respectively. This is consequential as HC_{first} is projected to decrease with technology scaling. A suitable solution should therefore have lower execution time overheads at lower HC_{first}. Overall, at lower HC_{first} values, STAR has lower maximum and average execution time overhead than two of the other leading Rowhammer mitigation methods.

B. *Defense Assessment against Rowhammer Frameworks*

We used a combination of several Rowhammer framework attacks to assess the effectiveness of STAR. We used g-hammer, TRRespass, and BLACKSMITH for this purpose [9][11][19]. Table 1 shows our results. With STAR, we can detect and mitigate all types of attacks. These tests were performed at the lowest and highest HC_{first} values considered in this work, i.e., 500 and 16K. We use the true positive rate (TPR) and FPR to characterize STAR's mitigation scheme. Across all the benchmarks, the TPR is 1.0, i.e., all Rowhammer instances are detected (verifying Theorem 1 and Theorem 2). At HC_{first} of 16K, there are no false positives. At HC_{first} of 500, STAR has a marginal increase in FPR. However, the FPR is negligible, and it does not impact performance at ultra-low HC_{first} values. In Table 2, we compare the FPR of STAR, HYDRA, and AQUA at HC_{first} of 500. HYDRA's FPR is low but significantly higher than STAR's as seen in Table 2 and previously in Figure 6. AQUA implements the Misra-Gries algorithm for its detection scheme [6][34]. The Misra-Gries algorithm is a highly efficient tracking scheme as its tested FPR is 0.0, as seen previously in Figure 6 and now in Table 2 [34]. However, AQUA accumulates higher execution time overheads due to row-migration mitigation as shown previously in Figure 8. In addition, the area cost overhead increases significantly at lower HC_{first} values to support the tracking algorithm (we discuss area overhead in the next section).

Rowhammer Attack	$HC_{first} = 500$		$HC_{first} = 16K$	
	TPR	FPR	TPR	FPR
g-hammer [9]	1.0	6.92E-5	1.0	0.0
BLACKSMITH [11]	1.0	7.46E-5	1.0	0.0
TRRespass [19]	1.0	6.85E-5	1.0	0.0

Table 1: True positive rate (TPR) and false positive rate (FPR) of STAR from Rowhammer benchmarks.

Rowhammer Attack	STAR	HYDRA	AQUA
g-hammer [9]	6.92E-5	3.27E-2	0.0
BLACKSMITH [11]	7.46E-5	3.27E-2	0.0
TRRespass [19]	6.85E-5	3.27E-2	0.0

Table 2: False positive rate of Rowhammer benchmarks (HC_{first} is 500).

C. *Area Overhead*

The area requirements for STAR and other solutions are simulated across memory densities. These memory densities are chosen as they represent specific memory generations. We accomplish this by scaling each solution based on the number of banks in each DRAM generation, e.g., DDR4 has 16 banks, DDR5 has 32 banks, and HBM has 256 banks [20][21][22]. Estimating area across DRAM memory densities gives insights into how scalable these solutions are across HC_{first}. HYDRA and AQUA require storage on both the memory controller and DRAM devices for effective mitigation. As previously stated, we utilize the optimized parameters for HYDRA and AQUA as reported in [3][6]) on a per-bank granularity as to not hinder performance metrics. For area overhead, we simulate all solutions in CACTI (45 nm). Figure 9 shows the overhead for each solution in terms of area (mm^2). STAR uses associative storage efficiently at lower bank-density memories which leads to lower area overhead. At HC_{first} of 500, the area overhead for STAR is 4.3× less than that for HYDRA. The slightly higher area for our solution for HBM3 is due to the increased amount of tracking information that is required for associative storage. Associative storage utilizes many comparators as shown in Figure 9, which leads to higher area cost; this is not the case for HYDRA and AQUA. However, STAR does not reduce the amount of DRAM memory available to the user. In contrast, HYDRA and AQUA reduce the amount of DRAM memory available to the user. HYDRA uses a row-count table in DRAM and AQUA uses a row quarantine zone in DRAM [3][6]; both these implementations require some part of the DRAM to be allocated for mitigation. In Figure 10, we show the DRAM memory space reduction for HC_{first} of 2K. While STAR introduces slightly more area for higher memory densities (HBM3), it does not reduce the available DRAM memory.

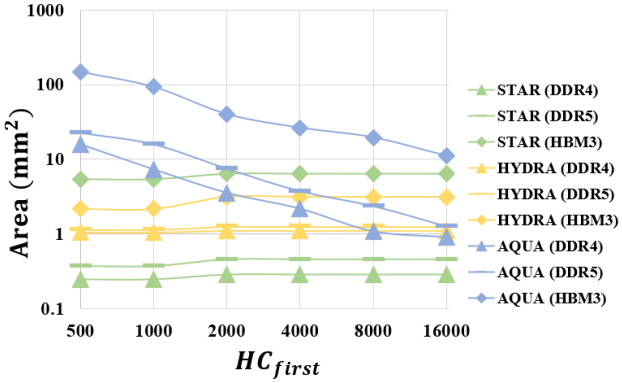


Figure 9: Comparison of area overhead results across memory density/generation.

DRAM Generation	STAR	HYDRA	AQUA
DDR4	0 MB	-2 MB	-184 MB
DDR5	0 MB	-4 MB	-368 MB
HBM3	0 MB	-32 MB	-2945 MB

Figure 10: DRAM memory reduction (HC_{first} is 2K)

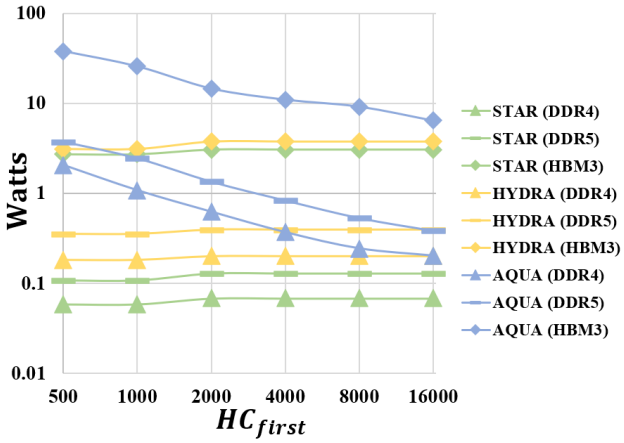


Figure 11: Total power consumption comparison across memory density/generation.

D. Power Overhead

We use CACTI to estimate both static power and dynamic power consumption. Due to the greater number of comparator circuits required, the overall static power follows a similar trend as the area results (Figure 9). However, STAR's dynamic power consumption is less than that of HYDRA and AQUA. This happens because HYDRA and AQUA rely on multiple tracking tables, e.g., these solutions require multiple reads and writes for tracking. This leads to higher dynamic power in comparison to STAR. Next, we present the total power consumption, which is the sum of static and dynamic power (both obtained using CACTI). In our power consumption estimation, we consider the dynamic power of each solution given the same scenario to ensure fair comparison, e.g., maximum access rate per bank. HYDRA and AQUA require more reads and writes to their tracking tables compared to STAR. This results in greater power consumption than STAR. In Figure 11, we show the results for total power consumption. Overall, we achieve lower power consumption for DDR4, DDR5, and HBM3. At HC_{first} of 500, STAR requires up to 3.3× less power than the next leading solution.

VI. Future Study

In this work, we showcase STAR, an efficient and scalable Rowhammer tracking and mitigation scheme. Advancing such hardware security techniques is an active area of research. However, these security solutions are dependent on HC_{first} values. The HC_{first} values in real-life may be noisy for in-field DRAM memory [36]. This noise is due to aging, layout specifications, and manufacturing process variations for DRAM [36]. Hence, in future work we will characterize Rowhammer as malicious DRAM faults [35]. To advance next-generation memory security techniques, manufacturing and in-field testing need to be considered.

VII. Conclusion

Rowhammer attacks remain an open problem and they affect DRAMs across multiple generations. Data from some of the latest DRAMs suggest that Rowhammer attacks are getting worse as technology scaling reduces HC_{first}. Existing solutions are either ineffective or have very high implementation overheads at very low HC_{first} (e.g., 3.2K). We have addressed this problem by proposing a new Rowhammer mitigation scheme, referred to as STAR, that is implemented inside only the memory controller. Based on our evaluation using multiple workloads, STAR achieves 9.5× and 8.0 × lower average and maximum execution time overhead than HYDRA at ultra-low Rowhammer thresholds, e.g., HC_{first} of 500 [3]. In addition, STAR achieves up to 4.3× and 3.3× average area and power savings at ultra-low HC_{first} (i.e., 500). Overall, STAR outperforms two state-of-the-art Rowhammer mitigation solutions in terms of power, performance, and area overheads.

References

[1] Y. Kim, et al., "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM/IEEE 41ST International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361-372, DOI: 10.1109/ISCA.2014.6853210.

[2] O. Mutlu, and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2020, DOI: 10.1109/TCAD.2019.2915318.

[3] M. Qureshi, et al., "Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," *2022 Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2022, pp. 699-710, DOI: 10.1145/3470496.3527421.

[4] Y. Park, et al., "Graphene: Strong yet lightweight row hammer protection," *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1-13, DOI: 10.1109/MICRO50266.2020.00014.

[5] A. G. Yaglikci, et al., "Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed DRAM rows," *2021 IEEE International Symposium on High-performance Computer Architecture (HPCA)*, 2021, pp. 345-358, DOI: 10.1109/HPCA51647.2021.00037.

[6] A. Saxena, et al., "Aqua: Scalable Rowhammer mitigation by quarantining aggressor rows at runtime," *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 108-123, DOI: 10.1109/MICRO56248.2022.00022.

[7] M. Marazzi, et al., "ProTRR: Principled yet optimal in-dram target row refresh," *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 735-753, DOI: 10.1109/SP46214.2022.9833664.

[8] A. Kwong, et al., "Rambleed: Reading bits in memory without accessing them," *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 695-711, DOI: 10.1109/SP40000.2020.00020.

[9] C. Evans, *Exploiting the DRAM rowhammer bug to gain kernel privileges*, 09-Mar-2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[10] H. Hassan, et al., "Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom Rowhammer Patterns, and Implications", *IEEE/ACM*

International Symposium on Microarchtecture (MICRO), 2021, pp. 1198-1213, DOI: 10.1145/3466752.3480110.
[11] P. Jattke, et al., "Blacksmith: Scalable Rowhammering in the frequency domain," *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 716-734, DOI: 10.1109/SP46214.2022.9833772.
[12] C. Bienia, et al., "The parsec benchmark suite: Characterization and architectural implications," *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, 2008, pp. 72-81, DOI: 10.1145/1454115.1454128.
[13] A. M. Garcia, et al., "PAMPAR: A new parallel benchmark for performance and Energy Consumption Evaluation," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, 2019, DOI: 10.1002/cpe.5504.
[14] S. C. Woo, et al., "The splash-2 programs: Characterization and methodological considerations," *Proceedings 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24-36, DOI: 10.1109/ISCA.1995.524546.
[15] Standard Performance Evaluation Corporation, *SPEC CPU 2006,* 15-Dec.-2011. [Online]. Available: https://www.spec.org/cpu2006/
[16] Standard Performance Evaluation Corporation, *SPEC CPU 2017,* 16-Mar.-2021. [Online]. Available: https://www.spec.org/cpu2017/
[17] J. S. Kim, et al., "Revisiting RowHammer: An experimental analysis of modern DRAM devices and mitigation techniques," *Proceedings of the ACM/IEEE 47TH Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 638-651, DOI: 10.1109/ISCA45697.2020.00059.
[18] B. K. Joardar, et al., "Machine learning-based Rowhammer mitigation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022, DOI: 10.1109/TCAD.2022.3206729.
[19] P. Frigo, et al., "TRRespass: Exploiting the many sides of target row refresh," *IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 742-762, DOI: 10.1109/SP40000.2020.00090.
[20] Micron, *DDR4 SDRAM,* 2021. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_ddr4_sdram.pdf
[21] JEDEC Solid State Technology Association, *JEDEC STANDARD High Bandwidth Memory DRAM (HBM3),* Jan-2023. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd238
[22] Micron, *DDR5 SDRAM,* 2022. [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr5/ddr5_sdram_core.pdf?rev=f76eb9631b674e66a2026c324a95cb67
[23] K. Pagiamtzis, et al., "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006, DOI: 10.1109/JSSC.2005.864128.
[24] S. Li, et al., "Dramsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters*, vol. 19, pp. 106–109, 2020, DOI: 10.1109/LCA.2020.2973991.
[25] J. L. Power, et. al., "The gem5 Simulator: Version 20.0+," arXiv:2007.03152, 2020, DOI: 10.48550/arxiv.2007.03152.
[26] N. Muralimanohar, et al., "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 3-14, DOI: 10.1109/MICRO.2007.33.
[27] P. Pessl, et al., "DRAMA: Exploting DRAM Addressing for Cross-CPU Attacks." *Proceedings of the 25th USENIX Conference on Security Symposium,* 2016, pp. 565-581.
[28] S. Qazi, et al., "Introducing Half-Double: New hammering technique for DRAM Rowhammer bug" *Google Online Security Blog,* 25-May-2021. [Online]. Available: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html
[29] P. Keller, "Understanding the New Bit Error Rate DRAM Timing Specifications" *JEDEC Server Memory Forum,* 1-Nov-2011. [Online]. Available: https://www.jedec.org/sites/default/files/Understanding%20BER%20based%20timing%20specifications%20Agilent%202011_1101.pdf
[30] R. Quiao, et al., "A new approach for Rowhammer attacks," *2016 IEEE International Symposium on Hardware Orientated Security and Trust (HOST),* 2016, pp. 161-166, DOI: 10.1109/HST.2016.7495576.
[31] M. Wang, et al., "DRAMDig: a knowledge-assisted tool to uncover DRAM address mapping," *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC),* 2020, pp. 1-6.
[32] Y. Jiang, et al., "TRRScope: Understanding Target Row Refresh Mechanism for Modern DDR Protection," *2021 IEEE International Symposium on Hardware Orientated Security and Trust (HOST),* 2021, pp. 239-247, DOI: 10.1109/HOST49136.2021.9702274.
[33] M. Lipp, et al., "Nethammer: Inducing Rowhammer Faults through Network Requests," *IEEE European Symposium on Security and Privacy Workshop (EuroS&PW),* 2020, pp. 710-719, DOI: 10.1109/EuroSPW51379.2020.00102.
[34] J. Misra, and D. Gries, "Finding repeated elements," *Science of Computer Programming*, vol. 2, no. 2, 1982, pp.143-152, DOI: 10.1016/0167-6423(82)90012-0.
[35] Z. Al-Ars, et al., "DRAM-Specific Space of Memory Tests," *2006 IEEE International Test Conference*, 2006, pp. 1-10, doi: 10.1109/TEST.2006.297701.
[36] Lois Orosa, et al., "A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses," *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* 2021, pp. 1182–1197, DOI: 10.1145/3466752.3480069