SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection

Ali Fakhrzadehgan
University of Texas at Austin
alifakhrzadehgan@utexas.edu

Yale N. Patt
University of Texas at Austin
patt@ece.utexas.edu

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

Moinuddin K. Qureshi
Georgia Tech
moin@gatech.edu

***Abstract*—Row-Hammer (RH) is a DRAM data-disturbance failure that occurs when a row is activated frequently, which causes bit-flips in nearby rows. Row-Hammer is a significant security threat as an attacker can exploit the bit-flips to do privilege escalation and leak confidential data. While several solutions aim to mitigate RH, such solutions depend on the RH threshold and adversarial access patterns. Solutions developed for a given threshold become ineffective for newer devices with lower thresholds, and new attack patterns continue to break existing mitigations. Currently, there is no guaranteed solution for RH, which means that the system remains vulnerable to security threats even in the presence of RH mitigation.**

In this paper, we contend that simply relying on RH mitigation is insufficient to provide security in the presence of reducing threshold and motivated attackers. We propose *SafeGuard*, which equips the system with low-cost integrity protection as a defense against potential attacks that break the RH mitigation. As SafeGuard can detect arbitrary failures, it converts the problem of RH bit-flips from a security threat (silent consumption of corrupted data) to a reliability concern (detectable uncorrectable errors caused by integrity violation). We develop SafeGuard for systems that employ ECC modules and show that SafeGuard can provide strong detection to both SECDED (46-bit MAC per cache-line) and Chipkill (32-bit MAC per cache-line) while retaining the correction capability of conventional designs. SafeGuard avoids incurring any storage overheads in DRAM by simply reorganizing the ECC code to operate at a cache-line granularity (64 bytes) instead of a word granularity (8 bytes). Our evaluations show that SafeGuard has a negligible impact on both the system performance (0.7%) and the system reliability due to naturally occurring errors while still providing a strong defense against the security risk of RH by detecting arbitrary bit-flips.

***Keywords*-Row-Hammer; Reliability; Security; Integrity;**

I. INTRODUCTION

Relentless DRAM device scaling has enabled memory modules that can store several gigabits of data on a single chip. While device scaling provides higher density, it brings cells closer, which increases inter-cell interference, and leads to new modes of failure. Row-Hammer (RH) [18], [21] is one such data-disturbance failure, which happens when a row is activated frequently. Each activation can leak away a small amount of charge from nearby rows, eventually causing bit failures. RH was publicly demonstrated in 2014, and as technology scales, the bit-flips from RH have become even more frequent for modern devices.

The bit-flips from RH are not just a reliability problem but also a security threat [2], [6], [8], [10], [11], [25], [40]. RH provides the attacker with a powerful tool to flip arbitrary bits. An attacker could flip bits in the Page-Tables to access data stored at arbitrary locations or orchestrate privilege escalation. Furthermore, the failures from RH are data-dependent, and this property can be used to infer data stored in nearby rows, thus violating confidentiality [25]. Therefore, RH continues to pose a severe security risk.

Developing techniques to mitigate RH has been an active area of research [2], [18], [21], [23]. Proposed hardware-based techniques range from increasing the refresh rate (global methods) [21] to tracking frequently accessed rows and refreshing the neighboring rows (precise method) [18], [21]. Unfortunately, increasing refresh rate globally incurs impractical overheads. Precise methods are more efficient as they track the frequently accessed rows and invoke mitigation only when a certain threshold is reached (or probabilistically where the probability is based on the threshold). The mitigation is typically done by issuing a row activation to the immediate neighbors, deeming such rows as *victim*.

The efficacy of existing RH mitigation proposals depends on two factors: (1) knowing the *Row-Hammer Threshold (RH-Threshold)*, which determines the number of activations required on the aggressor rows flip bits in the victim row, (2) knowing the *adversarial access pattern*, which captures the order in which the aggressor rows are accessed and can impact the number (and location) of the victim rows.

The tracking structures typically required for RH mitigation are sized for a particular RH-Threshold, and mitigation developed for a given RH-Threshold becomes ineffective at lower RH-Threshold. Unfortunately, RH-Threshold is not constant across vendors and technology generations. As shown in Figure 1a, RH-Threshold has reduced by almost 30x in the last seven years, from 139K (in 2014 [21]) to as small as 4.8K (in 2020 [19]). One can expect RH-Threshold to become even lower for future generations. We note that a processor developed with a given mitigation needs to work with a variety of memory modules, including that may come after the processor is released. Using a memory module with an RH-Threshold lower than the one assumed while designing the mitigation would leave the system vulnerable to RH errors and the associated security attacks.

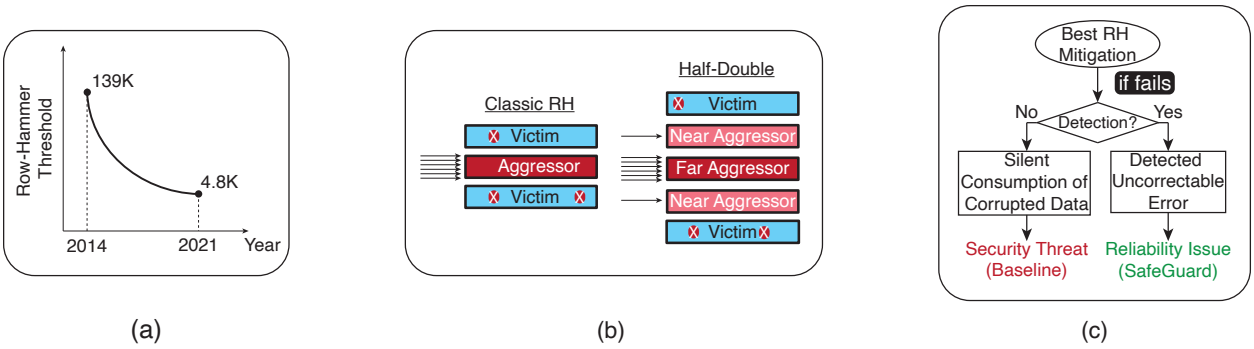


Figure 1: The moving target of Row-Hammer (a) The Row-Hammer Threshold has been reducing (b) New attack patterns break existing defenses [9] (c) Instead of allowing break-through attacks to become threats our solution detects such failures.

The second, more severe, weakness of existing mitigations is that they are implicitly designed for a particular access pattern and that the RH bit-flips are restricted to immediate neighbors. Unfortunately, newer attacks with more intelligent access patterns continue to break through existing RH mitigation, causing failures even in the presence of such mitigation. For example, the recently disclosed *Half-Double* [9] access pattern by Google, as shown in Figure 1b, causes failure at a distance of 2 from the aggressor and will defeat almost all existing precise mitigation policies. The *TRRespass* attack [8] develops an access pattern to cause failure even in the presence of *Targeted Refresh Rows (TRR)* (an in-DRAM mitigation employed in some DRAM chips). Finally, ECC chips were assumed to be resilient to RH failures, however, the *ECCploit* [6] attacks use the correction latency to develop a pattern that can cause multi-bit failures even in the presence of ECC chips. To the best of our knowledge, currently, there is no solution that is guaranteed to eliminate Row-Hammer.

We expect that developing even more effective mitigation for RH will continue to be an active area of research. However, we contend that the system must not assume that the (current and any future) mitigation will eliminate RH, and there will be no breakthrough attacks. All solutions rely on a set of assumptions, and a motivated adversary could try to break through the given mitigation by invalidating some assumptions. Therefore, to limit the security exposure from Row-Hammer even in the presence of RH mitigation, we advocate *SafeGuard*, which equips the system with low-cost integrity protection as a defense against potential attacks that break through the RH mitigation. As SafeGuard can detect arbitrary failures, it converts the severity caused by RH bit-flips from a security risk (silent consumption of corrupted data) to simply a reliability problem (detected uncorrectable errors). We would like to implement SafeGuard with negligible storage and performance overheads. Unfortunately, existing integrity protection proposals (e.g., SGX [7], [12]) store integrity protection metadata (such as *Message Authentication Code* or MAC) associated with each line in a separate area of memory and require extra memory accesses leading to significant overheads. Instead, we develop SafeGuard in the context of ECC (Error Correcting Code) memories, which contain extra chips to store the ECC code. We discuss SafeGuard for both SECDED and Chipkill.

Conventional ECC schemes are designed to tolerate naturally occurring errors and are typically focused on the correction strength, with detection as a byproduct. For example, ECC DIMMs that provide Single-Error-Correcting-Double-Error-Detecting (SECDED) operate at 8-byte granularity (the data-bus width), with 64-bit data and 8-bit of ECC code in each bus transfer, where the 7-bit is for SEC, and 1-bit is for DED. To develop SafeGuard at a low cost, we observe that processors interact with memories at a cache-line granularity (64 bytes), so we can form the ECC code also at 64-byte granularity. SafeGuard divides the 64-bits of ECC metadata for the 64-byte line into 10-bits for ECC-1, 8-bits for protecting against column failure, and 46-bit for integrity protection (MAC). Thus, SafeGuard provides strong detection within the same space of ECC while still retaining similar correction capability as conventional codes.[1]

Server memories are typically protected with Chipkill [29], which is a stronger form of ECC capable of tolerating the failure of an entire chip. Chipkill is a symbol-based code typically implemented with x4 memory chips, where the DIMM contains 18 memory chips, 16 for data and 2 for additional metadata. To implement SafeGuard with Chipkill, we redesign Chipkill to use the two metadata chips to store 32-bit MAC and 32-bit chip-wise parity for each line, respectively. The MAC is used to detect errors, and parity is used to correct chip failures. We provide extensions that ensure high performance and resilience even in the presence of a chip failure. With our extensions, SafeGuard with Chipkill not only provides the correction for single-chip failures but also the added benefit of strong detection of arbitrary failures, which are useful to detect RH errors due to break-through attacks.

[1]We observe that increasing the ECC granularity provides similar reliability as the chance of two independent single-bit errors affecting different words of a line is negligibly small (the main obstacle with larger granularity is handling of column failures). We discuss this in more detail in Section IV-D.

Overall, our paper makes the following contributions:

1) We observe that even in the presence of RH mitigation, there is potential for attacks that breakthrough causing security vulnerabilities, and the system must be designed to handle such attacks to ensure security.
2) We propose *SafeGuard* that equips each line with a low-cost integrity protection capable of detecting arbitrary data failures (with high probability). SafeGuard converts the RH failures from a security threat to a reliability issue.
3) We propose an efficient implementation of SafeGuard with SECDED ECC modules that provides strong detection while retaining single-bit correction.
4) We propose an efficient implementation of SafeGuard for Chipkill, which provides strong detection while retaining the ability to tolerate failure of a single chip.

SafeGuard incurs no additional DRAM area overheads. Furthermore, SafeGuard retains similar performance and reliability as conventional ECC while providing the benefit of detecting arbitrary failures. We compare SafeGuard with existing designs for integrity protection, such as SGX [12] and Synergy [39], and show that SafeGuard incurs significantly lower performance and storage overheads.

II. Background & Motivation

A. Threat Model

We assume that the attacker is remote and does not have physical access to the system under attack. We assume a traditional system in which the Operating System (OS) provides isolation between different processes using virtual memory and the page tables. The system uses DRAM main memory that is vulnerable to Row-Hammer (RH). The attacker process(es) runs under a *user* privilege. The attacker uses the RH vulnerability to either escalate the user privilege or manipulate data. We assume that the system already employs RH mitigation techniques. The attacker develops complex access patterns to break through any potential RH mitigation. Our aim is to avoid the *consumption* of the corrupted (manipulated) data via such breakthrough attacks.

B. DRAM Organization

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

C. The Row-Hammer Phenomenon

As DRAM is scaled to smaller feature sizes, the cells are placed closer to each other, making them vulnerable to inter-cell interference. Row-Hammer (RH) is one such interference fault that happens when a row of cells is accessed frequently, causing bit-flips in nearby rows. The hammered row is often referred to as the *aggressor* row, and the neighboring rows are referred to as the *victim* rows. Figure 2 depicts the RH attack in which an aggressor row causes bit-flips in its adjacent victim rows over time.

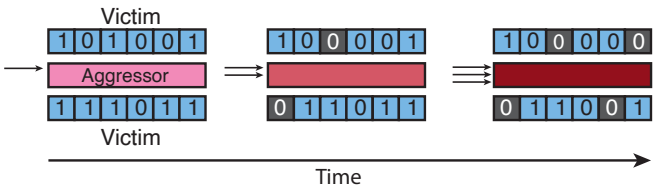


Figure 2: Example of a Row-Hammer Attack.

RH is caused by the interference between the aggressor and the victim rows due to the coupling effect [18], [19], [21]. This interference increases the amount of leakage current in the DRAM cells within the victim row(s), which causes them to lose their charge at a faster rate, before the refresh operation can reinforce the charge in the cell capacitors. Thus, for the attack to be effective, it is essential that the victim row is not accessed (activated) while the aggressor is being hammered, otherwise, the access will replenish the charge in the cells of the victim row.

Row-Hammer Threshold (RH-Threshold) denotes the number of activations required on the aggressor row(s) to cause bit-flips in the victim rows. Unfortunately, the RH-Threshold has been reducing steadily with each technology generation. When the RH phenomenon was first characterized in 2014, the RH-Threshold was 139K, whereas it has reduced by more than an order of magnitude to just 4.8K [19]-9K [9] by 2020. Table I shows the RH-Threshold for different DRAM generations over the last 7 years. As a given standard (DDR3, DDR4, LPDDR4) can span devices made at different technology nodes, we use *old* and *new* to distinguish different versions. The decreasing RH-Threshold worsens the problem of RH.

Table I: Row-Hammer Threshold Over Time

DRAM Generation	RH-Threshold
DDR3 (old)	139K [21]
DDR3 (new)	22.4K [19]
DDR4 (old)	17.5K [19]
DDR4 (new)	10K [19]
LPDDR4 (old)	16.8K [19]
LPDDR4 (new)	4.8K [19] - 9K [9]

We note that RH bit-flips is not just a reliability problem but a severe security problem. RH gives the attacker a powerful weapon to potentially flip any arbitrary bit in the

memory system, and the attacker can use it to flip bits in page tables and cause privilege escalation or use the data-dependent nature of RH failures to read confidential data.

D. Proposals for Mitigating Row-Hammer

Mitigating Row-Hammer is an active area of research (we discuss prior work in more detail in Section VIII). The RH defenses can broadly be classified into four categories.

First, *global mitigation*, whereby the refresh rate of the entire memory is increased. Unfortunately, this is not a viable method for tolerating RH at thresholds below 32K, as we would need to refresh the memory in less than 2ms (whereas it takes 2-3ms to refresh the entire memory even if the memory spends 100% of the time only doing refresh).

Second, *precise mitigation*, which consists of two parts: *when* to apply the mitigation and *where* to apply the mitigation. The *when* is based on the RH-Threshold, and there are various probabilistic or tracking schemes to identify which rows must be deemed as aggressor rows. For *where*, the mitigation is typically done by refreshing the *immediate neighbors* of the aggressor row based on the assumption that the adversary is unable to cause bit-flips beyond the immediate neighbors.

Third, *isolation-based mitigation*, whereby a *guard* row is kept between the row with sensitive data (e.g., page tables) and untrusted data. This mitigation relies on knowing the distance at which RH bit-flips can occur. For example, keeping only a single guard row is ineffective if the attacker is somehow able to flip bits beyond the immediate neighbor.

Fourth, *ECC-based mitigation*, whereby the memory contains error-correction code, which can be used to correct RH bit-flips. This scheme is effective only at low rates of bit-flips where the number of errors is below the strength of the error-correction code employed in the memory module.

We note that RH mitigations are designed to target a particular RH-Threshold. For example, for the probabilistic methods, the probability of identifying the aggressor row must be tailored carefully for a given RH-Threshold. Similarly, the hardware structures typically employed for tracking-based solutions are sized for a particular RH-Threshold, and mitigation developed for a given RH-Threshold can become ineffective at lower RH-Thresholds. A processor designed with a particular mitigation must be able to work with memory modules from different vendors and different technology nodes of a generation (e.g., DDR4-old and DDR4-new) that may arrive after the processor is manufactured. Using a memory module with an RH-Threshold lower than the one assumed while designing the mitigation would leave the system vulnerable to RH errors and the associated attacks. In addition to the reliance on RH-Threshold, there is an even more severe vulnerability for RH mitigation techniques, which is the reliance on knowing the adversarial access patterns.

E. Break Row-Hammer Mitigation with Complex Patterns

RH mitigation techniques are implicitly designed with particular assumptions around the attack patterns and *Blast Radius* (the distance of the victim rows from the aggressor) [27]. A solution developed for a particular attack pattern may become vulnerable when the attacker employs a more complex attack pattern that can defeat the hardware structures and/or increase the blast radius. This mode of vulnerability has been used to break several prior RH mitigation techniques and poses a continuing risk for current and future mitigations. We discuss three cases studies where RH solutions were recently broken by employing intelligent attack patterns.

Case-1: Breaking Precise Mitigation with Half-Double Precise mitigation techniques refresh the immediate neighbors, implicitly assuming that RH can flip bits only at a *distance-of-one* and the rows beyond that are safe. A recent work from Google, called Half-Double [9], shows that one can develop attack patterns that target victim rows which are at a *distance-of-two* away from the aggressor rows. Such an attack pattern, shown in Figure 1b, can cause bit-flips even in the presence of precise mitigation techniques. For example, Half-Double is able to cause more than a hundred bit-flips in LPDDR4 modules at a distance of 2 away from the aggressor rows. The key insight in Half-Double was to use the existing RH mitigation itself as an attack where the immediate neighbor gets accessed enough times to cause flips in their immediate neighboring row. As DRAM cells scale, it is reasonable to assume that an adversary can target victim rows even farther away from aggressor rows.

Case-2: Breaking Target-Row Refresh with TRRespass To mitigate RH transparently within the DRAM module, the DRAM industry developed *Target Row Refresh (TRR)*, whereby the memory module tracks a small number of frequently accessed rows within each DRAM chip. When the DRAM chip receives a refresh command, in addition to doing the regular refresh operations for a subset of the memory, the chip also performs RH mitigation by refreshing the immediate neighbors of the rows in the TRR tracking structures. DRAM-industry claimed that the RH problem could be solved with TRR and started selling *TRR-enabled* memory modules [8], [17] offering protection against RH.

Unfortunately, TRR was broken within four years of being incorporated in DDR4 and DDR5 [30]. A recent attack, called TRRespass [8], exploited the fact that TRR can keep track of only a small number of aggressor rows. Based on this observation, TRRespass hammers a large number of dummy rows alongside the intended aggressor rows. This causes capacity-based evictions in the TRR tables and makes the mechanism evict the intended aggressor row. In general, any tracking-based mitigation scheme can be vulnerable to such eviction-based attack patterns.

Case-3: Breaking Through ECC Memory with ECCploit The system can use Error Correcting Codes (ECC) to rectify RH-induced bit-flips, and this solution was deemed as robust against RH attacks [21], assuming low rates of RH bit-flips. Once the number of bit-flips is beyond the capability of the ECC code, these bit-flips can become silent errors (escaped from the ECC code) or cause miscorrection (increased errors from ECC correction). However, at least for low rates of bit-flips, the ECC-based scheme was deemed safe.

An overlooked aspect of ECC-based correction is that there is a latency difference between fault-free cache-lines and faulty cache-lines. A recent attack, called ECCploit [6], exploits this timing-channel information to learn which patterns cause RH bit-flips and use it to progressively increase the faulty bits within a line, even if each individual bit-flip was corrected by the ECC code. Thus, even in the presence of ECC chips, an attacker can induce RH faults that can bypass the detection and correction capability of the employed ECC code.

Key Takeaway: These case studies show that existing RH solutions continue to be vulnerable to breakthrough attack patterns that cause bit-flips even in the presence of mitigation. Currently, there is no guaranteed solution for RH. Thus, even with RH mitigation, the system remains vulnerable to RH security attacks.

F. Goal: Reducing Row-Hammer Security Risk at Low-Cost

We expect that future research to develop even more effective RH mitigation. However, security will remain a concern because RH-based bit-flips could still be possible due to breakthrough attacks even in the presence of mitigation. If the system could detect the bit-flips that occur due to such breakthrough attacks, then we could prevent the program from consuming corrupted values, and thus, avoid security threats that cause privilege escalation. Such detection would convert the RH bit-flips from breakthrough attacks into a reliability problem rather than a security risk. The goal of our paper is to develop low-cost integrity protection to reduce the security risk from breakthrough RH attacks. We discuss our experimental methodology before presenting our solution.

III. Experimental Methodology

Our proposal repurposes the ECC bits to incorporate both the correction metadata and the metadata for integrity protection. We perform evaluation for both performance and overall system reliability (against naturally occurring errors) to show that our proposal has low overhead and retains similar correction capability as conventional ECC designs.

A. Performance Evaluation Framework

We use an execution-driven cycle-accurate simulator [31], with the parameters described in Table II. The simulator uses Intel Pin [28] as the functional model. Main memory is modeled using Ramulator [22]. The virtual page size is 4KB. We assume that the baseline is already with the best available RH mitigation, and our goal is to detect the bit-flips that occur due to break-through attacks. We assume that the memory is made of ECC DIMMs (configured to be used as either SECDED code or Chipkill). To obtain a fast MAC, we can concurrently encrypt each of the eight 64-bit words of a line with a low-latency encryption circuit, such as QARMA [24] (latency of 2.2ns), and perform an XOR of the eight cipher-texts to obtain the 64-bit MAC. For shorter MAC, the least-significant bits of MAC-64 are used.

Table II: Configuration Parameters

Core	6-wide fetch/retire OoO, 224 entry ROB, 97 entry RS, TAGE-SC-L branch predictor, 3.2GHz, 4 cores
L1 Cache	Private 32KB d-cache, 2 cycles latency. Private 32KB i-cache. 64-Byte line, 4-way
Last Level Cache	Shared 4MB, 64-Byte line, 16-way, 18 cycles latency, write-back, inclusive
Prefetcher	Stream prefetcher
Main Memory	16GB DRAM DDR4-3200 at 1600MHz, 1 channel, 2 ranks of 16 banks, 8KB row buffer. 64 Read- and 64 Write-entry memory queues
MAC latency	8 processor cycles (4 memory controller cycles)

We use a 500 million SimPoint [41] region of the SPEC-2017 [1] rate benchmarks. We simulate a 4-core system, and each workload is replicated four times. We continue execution until all cores execute at least 500 million instructions.

B. Reliability Evaluation Framework

We use FaultSim [34] to evaluate systems reliability. To evaluate SECDED, we model single-channel 16GB memory modules with x8 devices. For Chipkill, we model single-channel 16GB memory modules with x4 devices. We model 10 million devices using Monte-Carlo simulation for a 7-year period. We consider a module as *failed* when it observes an uncorrectable or an undetectable error. We report *Probability of System Failure* as the fraction of failed modules. We use the real-world failure rates (FIT) reported in the prior work [43], shown in Table III.

Table III: Failures per Billion Hours (FIT) per device [43].

DRAM Chip Failure Mode	Failure Rate (FIT)		
	Transient	Permanent	Total
Single bit	14.2	18.6	32.8
Single column	1.4	5.6	7.0
Single word	1.4	0.3	1.7
Single row	0.2	8.2	8.4
Single bank	0.8	10	10.8
Multi-bank	0.3	1.4	1.7
Multi-rank	0.9	2.8	3.7

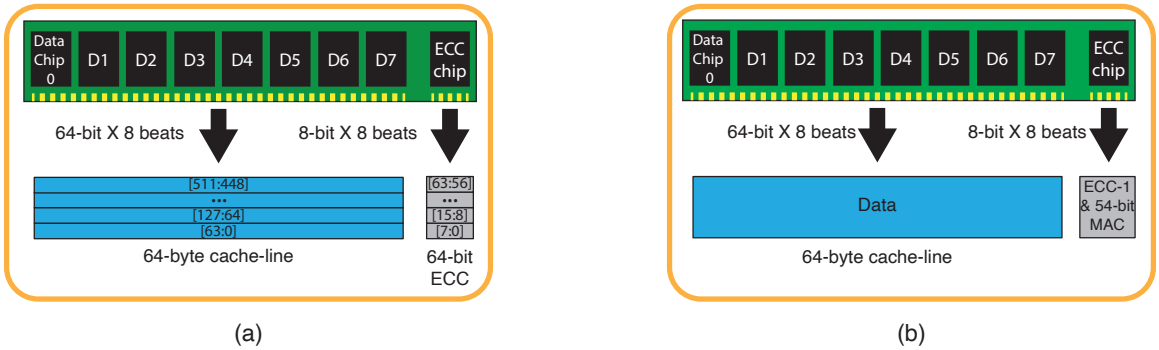


Figure 3: SafeGuard with SECDED. (a) Conventional SECDED memory operates at 8-byte granularity. (b) SafeGuard design operates at 64-byte granularity and uses the 64-bits from the ECC chips to get ECC-1 and 54-bit MAC.

IV. SafeGuard: Low-Cost Integrity Protection

We can limit the security exposure from RH (that may happen even in the presence of RH mitigation) if we equip each line with a strong detection code (such as Message Authentication Code, or MAC) that can detect arbitrary failures. Secure computing designs, such as SGX [12], already provision per-line MAC to do integrity protection against data tampering. These designs store the MACs in a separate area of memory, and on a memory access, they access the MAC concurrently with the data line. Unfortunately, such secure memory designs incur significant performance overheads and area overheads (12.5%, so 2GB for 16GB memory), which makes such solutions unsuitable for widespread adoption. To enable integrity protection at negligible cost, we propose *SafeGuard*, which uses ECC-based memory modules to provide strong detection of arbitrary failures while still ensuring that the correction capability remains similar to conventional ECC designs. In this section, we discuss how to design SafeGuard with SECDED memories.

A. SafeGuard with SECDED

ECC DIMMs are equipped with an additional chip to store the metadata for the ECC code, as shown in Figure 3a. The DIMM is designed with a 72-bit data bus, from which 64 bits carry the data, and 8 bits carry the ECC metadata. In conventional ECC memory, each 64-bit word is protected by an 8-bit SECDED (Single Error Correction Double Error Detection) code. Thus, each transfer on the bus undergoes an independent ECC check. This design has been used from days where the processors had a small line size and memory had the capability to transfer just one word. However, modern processors interact with the memory at the 64-byte cache-line granularity. Furthermore, to support memory timings, the current memory protocols dictate a minimum burst length of 8 [32], which means that the memory will transfer at least 8 bursts of contiguous data for each access. Thus, from the viewpoint of both the processor and the memory, the granularity of interaction is 64-bytes. If we can reorganize the ECC to operate at the granularity of 64-bytes instead of the conventional 8-bytes, then we can do single-error correction (ECC-1) for the 64-byte line with only 10 bits and use the remaining bits to do integrity protection. This rethinking of granularity allows implementation of SafeGuard at a low cost.

Figure 3b shows the design of SafeGuard with SECDED. SafeGuard reorganizes the module-level ECC to operate at 64-byte, so it views the ECC bits as a collection of 64-bits rather than 8 independent ECC packs of 8 bits each. SafeGuard uses 10-bits to implement ECC-1 for the 64-byte line and 54-bit to implement a strong detection code. We considered using error detection codes such as CRC (Cyclic Redundancy Code), however, such codes can be reverse-engineered by an adversary, as they have a predictable parity-based pattern. For strong detection, SafeGuard uses a MAC (although any other cryptographic signature can be used). Each memory controller contains a 16-byte key initialized randomly at boot time. We concatenate the line address with the key to use as the effective key.

While writing a line to the memory, the memory controller writes the data, ECC-1, and the 54-bit MAC. ECC-1 is computed on the 512-bit data and its 54-bit MAC. When the processor reads a line, the memory controller first performs ECC-1 correction (if any) and then computes the MAC of the 512-bit data and compares it with the retrieved MAC. A mismatch signals *Detected Unrecoverable Error (DUE)*. Note that SafeGuard performs the MAC verification regardless of whether the ECC-1 performs any correction.

SafeGuard provides a much stronger detection capability than SECDED, and we show that it still retains similar correction effectiveness (and overall system reliability against natural faults) as conventional SECDED.

B. Reliability Implications of Increased Granularity

As SafeGuard reorganizes the ECC to operate at a 64-byte granularity instead of an 8-byte granularity, one may think this has an 8x higher failure rate, as we are covering

8x more area. However, the only case in which SECDED at *word* granularity is stronger than ECC-1 at *line* granularity (i.e., SafeGuard) is when the line simultaneously contains multiple words, each with a single-bit fault. However, the likelihood of such cases is extremely low. The number of faults that the memory accumulates over time and the probability of these faults landing on the same cache-line follow the *birthday collision* probability model. That means, given that the memory has N lines, after f faults, the probability of the next fault landing on an already faulty line would be $\frac{f}{N}$. Thus, to observe a line that has at least two faults, it requires about $\sqrt{N}$ faults. Assuming a 64GB memory (i.e., 2^{30} lines), it requires about $32K$ faults to eventually observe two bit-flips in a line. That means the probability of a fault landing on an already faulty line is about $\frac{1}{32K}$. From that, roughly $\frac{1}{8}$th land on the same word and is not correctable by either of the schemes. Thus, the probability of the case that SECDED is superior to SafeGuard for single-bit errors is $\frac{7}{8} * \frac{1}{32K} = 3.51 \times 10^{-5}$.

This is a negligibly small probability. For example, even if naturally occurring single-bit faults happened at 100x higher FIT Rate, the single-bit error rate for 64GB memory would be once every 6 months, and it will take approximately 2,500 years to encounter a line with two independent single-bit errors mapping to its two different words.

Note that for the previous calculations, we assumed that the failures only occur due to single-bit faults and these faults are completely independent. However, an internal device or circuit failure could potentially increase the likelihood of multi-bit failures, which can cause spatially correlated bits to fail. Table IV summarizes the correction and detection capability of SECDED and SafeGuard against different fault patterns. Both SECDED and SafeGuard can correct single-bit errors. In addition, SafeGuard can provide strong detection of arbitrary bit failures, including the ones caused by multi-bit faults. We note that multi-bit failures from column failures can be corrected by SECDED but not by SafeGuard. We discuss this in more detail next.

Table IV: Resiliency of SECDED vs. SafeGuard.

DRAM Chip Failure Mode		SECDED		SafeGuard	
		Detect	Correct	Detect	Correct
Single bit		✓	✓	✓	✓
Multi-bit	Single column	✓	✓	✓	✖
	Single word	✖	✖	✓	✖
	Single row	✖	✖	✓	✖
	Single bank	✖	✖	✓	✖
	Multi-bank	✖	✖	✓	✖
	Multi-rank	✖	✖	✓	✖

C. Tolerating Column Faults via Column Parity

Column faults typically occur because of a faulty pin or because of the failure of the bit-line circuitry used to read or write [20], [43]. Figure 4 shows how this fault pattern can cause multiple bit-flips in the same position of several words in a cache-line. In this case, the fault pattern in a line is *vertical* with only one fault in each ECC code-word. SECDED can correct these faults, while SafeGuard cannot.

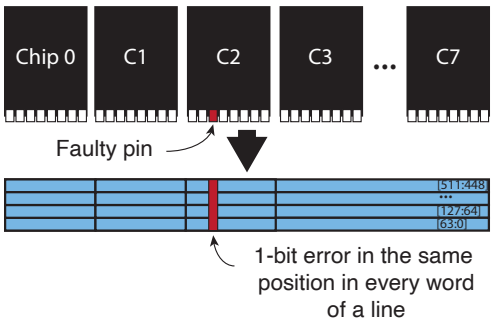


Figure 4: Cache-line fault-pattern due to a pin failure.

To mitigate column failure, we extend SafeGuard to keep metadata that can be used to recover the data lost due to the column failure. The failure of a single column would corrupt 8-bits of data (coming from that column) in a 64-byte line. Therefore, to recover from column failure, our design stores 8-bits of column parity [20]. Consider that the 8-bit data provided by a pin as a symbol, then the column-parity is simply the XOR of the 64 symbols (one for each pin). We call this design *SafeGuard with Column Parity*, and it is shown in Figure 5.

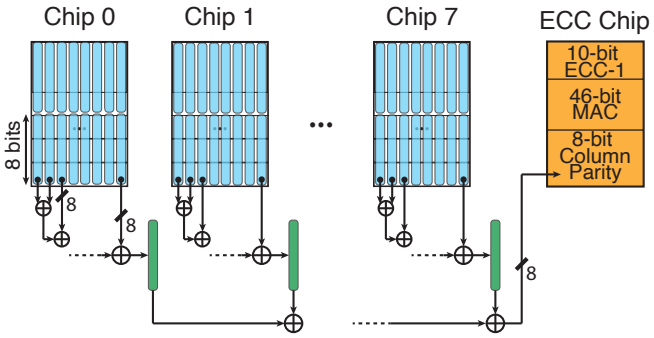


Figure 5: Extending SafeGuard to tolerate column-failure.

The 64-bits from the ECC chip are divided into: 10 bits for ECC-1, 8-bits for column-parity, and 46-bit MAC. When a line is read from memory, we first check the MAC. In case of a mismatch, we try ECC-1 correction and then recheck the MAC. If that fails, we invoke the mitigation for column failure. As we do not know the location of the column failure, our design employs an iterative correction mechanism, whereby in each iteration, we try recovery (by reconstructing data for that column using the column-parity) for one column and use the MAC check to verify if that recovery is correct. In case of a MAC mismatch, we try

the next location. In case of a MAC match, we provide the repaired data to the processor and remember the location of the failed column (to speed up future recovery). If there has been no MAC match after we tried all 64 possibilities, we have encountered a *Detected Unrecoverable Error (DUE)*, which can happen multi-bit failure modality or due to RH. SafeGuard signals the system that it has encountered a DUE.

Iterative correction can be slow, as it requires up to 64 rounds of MAC verification. Fortunately, iterative correction is invoked only on column failures, which occur at a low rate (on average, once every 100+ years for a 16GB DIMM). If the column failure is transient, we incur the latency overhead (less than one microsecond) only on rare occasions. If the fault is permanent, we can remember the last column that resulted in repaired data that matched with the MAC. The correction begins first with that location, which will avoid the penalty of iterative correction. Furthermore, in such cases, after a few rounds of correction, we skip the first MAC check (as it would consistently fail), and directly recover the data, and only then do the MAC check to avoid the latency penalty of two MAC checks. Thus, during fault-free operation and for permanent column-failures, the latency overhead remains approximately one MAC check (parity-based reconstruction is fast and can be done in one cycle).

D. Reliability Evaluation

Figure 6 compares the reliability of SafeGuard (with and without column-parity) with SECDED over a period of seven years. SafeGuard without column parity has a 1.25x higher failure rate than SECDED, however, SafeGuard with column-parity has virtually identical reliability as SECDED. Thus, SafeGuard provides strong detection while retaining the correction capability of the conventional ECC design.

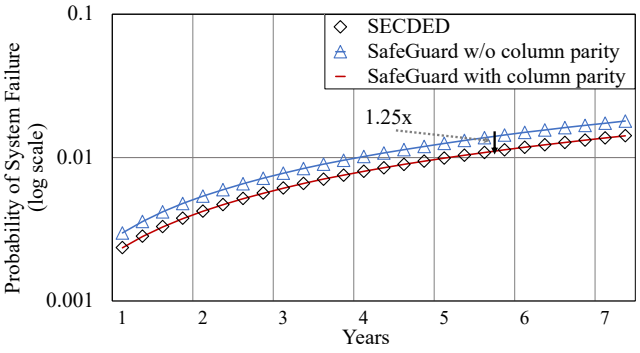


Figure 6: SafeGuard vs. SECDED reliability. SafeGuard provides virtually identical correction capability as SECDED.

E. Performance Evaluation

The correction latency of SafeGuard is incurred only in case of failure (which is a rare event). Therefore, we focus on performance during fault-free operation. SafeGuard incurs the latency overhead of a MAC check, which is in the critical path of the read operations and is the primary source of performance overhead. Figure 7 shows the performance of SafeGuard compared to the baseline. SafeGuard incurs an average slowdown of only 0.7%. The largest slowdown (3.6%) is incurred for Omnetpp, which is a latency-critical workload, and the added latency impacts performance.

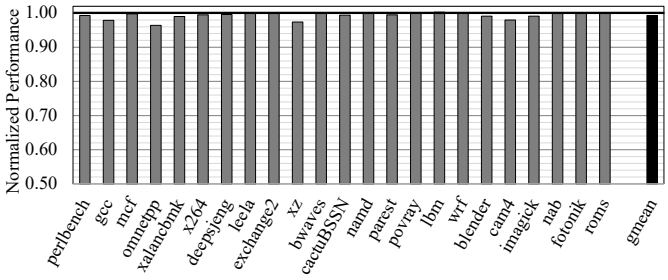


Figure 7: Overall Performance of SafeGuard vs. SECDED.

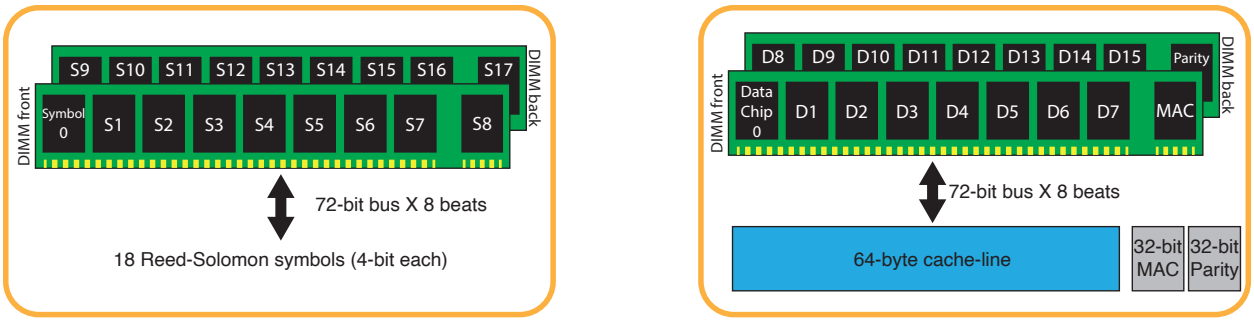
F. Storage and Logic Overheads

SafeGuard reuses the ECC bits and does not require any additional DRAM storage. SafeGuard requires minor changes to the memory controller. First, the ECC logic (3K XOR-gates) has to operate at the line granularity. Second, the memory controller needs a MAC computation unit and storage of a 16-byte key. Finally, simple parity units for vertical parities and logic to test different positions. The total SRAM overhead is less than 32 bytes.

V. SafeGuard with Chipkill

Memory modules are susceptible to large granularity failures, such as word failures, row failures, and bank failures [43]. Unfortunately, SECDED is not effective against these failures. Systems that require a high level of reliability often use a stronger form of ECC called Chipkill, which can tolerate the failure of an entire chip. Chipkill-equipped memories are typically created using x4 devices, as shown in Figure 8a. This module has 18 memory chips, 16 of which provide the 64-bit data word (4 bits per device) and two additional chips to store the Chipkill redundancy information. Chipkill uses a symbol-based code and provides *Single-Symbol-Correction and Double-Symbol-Detection (SSCDSD)*, in which a symbol is the 4-bit data provided by each device. Thus, Chipkill can correct up to 4-bit errors in each bus transfer, given that all errors are confined to a single symbol.

In addition to correcting one chip failure, Chipkill can also detect failure of two chips. However, if the fault spans more than two chips, Chipkill either cannot detect or may miscorrect, resulting in silent consumption of corrupted data. Prior work [6] discusses how RH attacks can exploit this limitation and corrupt multiple symbols simultaneously to bypass Chipkill's error detection. We show SafeGuard can be designed with Chipkill to provide integrity protection.



(a) (b)

Figure 8: SafeGuard with Chipkill. (a) Conventional Chipkill memory using symbol-based code operating on 18 x4-chips (S0-S17), located on front and back side of the DIMM. (b) SafeGuard with Chipkill provides single-chip-correction and integrity protection by using MAC for error detection and chip-wise parity for correction. Data is stored in plain form.

A. *Organization of SafeGuard with Chipkill*

Figure 8b shows SafeGuard with Chipkill organization. Instead of using symbol-based code, SafeGuard stores data in plain form and uses MAC for error detection and parity for correcting failure. We use one of the chips to store a 32-bit MAC that is generated on a line. The other chip stores the chip-wise parity across all 17 chips (32-bits per 64-byte).

SafeGuard provides the same correction capability as in Chipkill (i.e., SSC), while it can provide stronger error and tampering detection, including multi-symbol failures. Using MACs enables detection of arbitrary failures. Similar to conventional Chipkill, SafeGuard can correct up to one complete chip failure. For correction, SafeGuard uses the chip-wise parities. Parities are generated for each bus transfer using chip-wise XOR of data.

B. *Operation of SafeGuard with Chipkill*

On a data read, the processor uses the retrieved MAC for error detection. A MAC mismatch indicates a fault and triggers the error correction mechanism. Note that the chip-wise parities cannot directly locate the faulty chip, and instead, we need to search for the faulty chip iteratively. The memory controller performs the error correction by going through each chip, assuming the chip has been failed and uses the parity to correct its data, as shown in Figure 9a.

In each iteration, the corrected line goes under another round of MAC verification. If the verification passes, we deem that the error has been successfully corrected. Otherwise, the memory controller starts the next iteration to test a different chip. Thus, error correction can take up to 16 iterations. In the end, if no iteration results in a MAC match, then the system has encountered a *Detected Unrecoverable Error (DUE)*, either due to a larger granularity failure (e.g., rank failure) or due to multi-bit data tampering from Row-Hammer that spans multiple chips. SafeGuard informs the system about the DUE so that it can take corrective actions.

Iterative correction incurs high latency, which is in the critical path of memory access. Note that this latency overhead is incurred only in the presence of errors (which are rare events), so the overall impact on system performance is quite low during normal operations. However, in cases of permanent chip failure, the latency overhead is significantly high, and we discuss how to reduce that next.

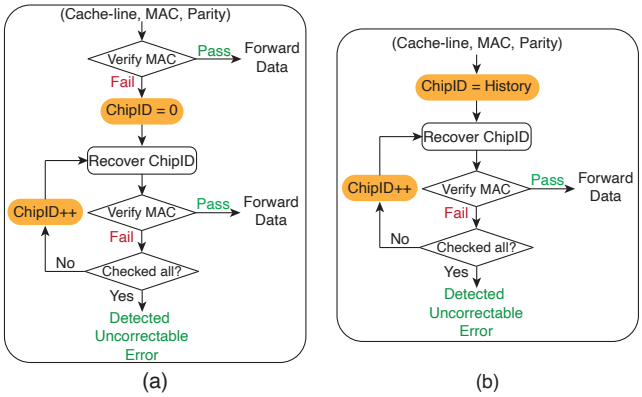


Figure 9: SafeGuard with (a) Iterative correction and (b) Eager Correction.

C. *Reliability Challenge Under Permanent Chip Failure*

The first time there is a failure, we do not know the location of the failed chip. However, when a permanent chip failure occurs, then iterative correction is not necessary, as we are likely to see the repair for the same failed chip again and again. For permanent failures, we can avoid the latency overhead of iterative correction by starting the iterative correction from the faulty chip that failed the last time. While such a history-based design will avoid the latency of iterative correction, it still suffers from two shortcomings: (1) there are two MAC checks required, one for the data-line retrieved from memory and the second after

the repair, so the latency overhead of extra MAC-check is still present (2) the faulty data retrieved from memory will eventually escape the detection of the 32-bit MAC, resulting in silent data corruption. The escape from MAC-32 happens because under chip failure, every single memory access will provide corrupted data, and it will take just 4 billion memory accesses on average (less than 1 minute) to find the corrupted data that escapes the MAC-32 check. Thus, this simple history-based design does not provide Chipkill-level reliability in the presence of permanent chip failures.

D. Ensuring Reliability with Eager Correction

The vulnerability from the MAC check accumulates when we continuously send faulty data to the MAC unit. There is a small escape probability ($1/2^n$ for n-bit MAC) with each check. This probability does not come into play when checking fault-free data. Under permanent chip failure, we expect the first MAC check to fail, and then the second MAC check (after repair of the faulty chip) pass, and this will happen repeatedly. To avoid the vulnerability from the first MAC check, we propose *Eager Correction*, which skips the first MAC check, eagerly repairs the data for the chip that failed the last time, and only then performs the MAC check on the reconstructed data. We note that if there was no faulty chip, then eagerly reconstructing the data and then checking the MAC will still pass (i.e., no impact on the overall reliability of the system). Figure 9b shows the overview of Eager Correction. If Eager Correction fails (a different chip is faulty), then SafeGuard resorts to the iterative correction.

Eager Correction effectively mitigates the MAC-32 escape vulnerability in SafeGuard in each of the following scenarios. If the previously failed-chip is indeed the only corrupted chip, then SafeGuard will perform the MAC-32 check exclusively on the corrected data, eliminating the vulnerability due to the first check on the faulty data. If more than one chip is corrupted (e.g., a multi-chip failure, or a chip failure and some transient fault), then the correction will produce a wrong value, which can be detected by the post-correction MAC-32 verification and be declared as a DUE. The only remaining case that can escape Eager Correction is when several chips are failing interchangeably. However, fault patterns across multiple chips are highly improbable to occur naturally and are not expected to be repaired by Chipkill.[2] Therefore, SafeGuard can simply declare a DUE after several rounds of ping-pong between faulty chips.

[2] In some rare cases, a memory system can have multiple lines with single-bit permanent failures. Chipkill can correct such failures. However, SafeGuard can cause an iterative correction whenever a different faulty line gets accessed. Such a scenario can be avoided by provisioning the memory controller with a few (4-5) spare lines, and on a correction of a single bit fault, simply copy the corrected line into the spare lines. Subsequent accesses to such lines are serviced by the spares in the memory controller.

E. Reliability Evaluation

To provide reliability against chip failures, we assume that SafeGuard is implemented with Eager Correction as the default. Figure 10 compares the reliability of SafeGuard with Chipkill. SafeGuard provides strong detection while retaining the correction capability of the conventional Chipkill design for naturally occurring errors.

To assess the effectiveness of SafeGuard at higher fault rates, we also evaluate a system where the FIT-Rates are increased by a factor of 10. SafeGuard continues to provide similar correction reliability as Chipkill for naturally occurring errors while providing the additional capability of detecting arbitrary bit-failures.

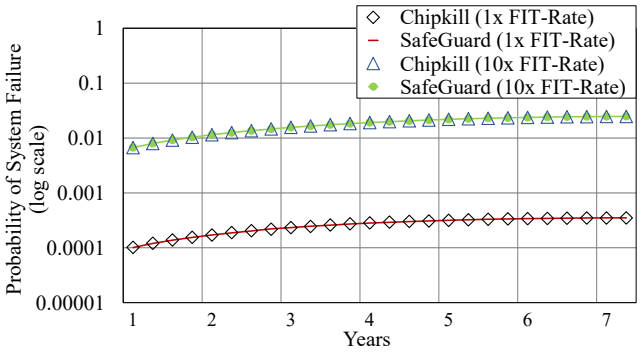


Figure 10: Reliability of SafeGuard vs. Chipkill. SafeGuard provides virtually identical correction capability as Chipkill.

F. Performance Evaluation

SafeGuard incurs the latency overhead of iterative correction only in the case of failure (which is a rare event). Therefore, we focus on performance during fault-free operation. Under normal operations, SafeGuard incurs the latency overhead of a MAC check, which is in the critical path of read operations and is the primary source of performance overhead. Figure 11 shows the performance of SafeGuard compared to the Chipkill. SafeGuard incurs an average slowdown of only 0.7% (similar to the slowdown when implemented with SECDED, as both designs have the latency of a MAC check in the critical path).

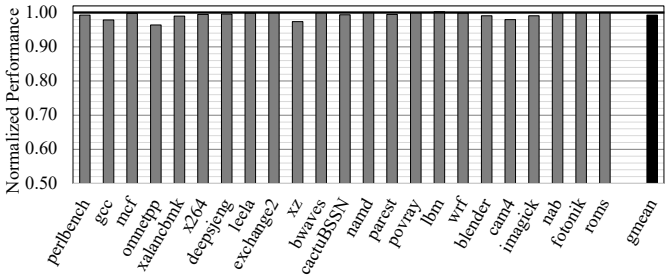


Figure 11: Overall Performance of SafeGuard vs. Chipkill.

G. Storage and Logic Overhead

SafeGuard uses the same DIMM as Chipkill and does not need additional DRAM storage. SafeGuard requires minor changes to the memory controller. First, the logic to compute the chip-wise parity. Second, MAC unit and storage of a 16-byte key. Finally, for Eager Correction, we need a register to track the ChipID of the faulty chip and a counter to track the number of corrections. The total overhead of SafeGuard with Chipkill is less than 32 bytes.

VI. Comparison with Other MAC Organization

SafeGuard is a low-cost integrity protection scheme, which provides strong detection of arbitrary failures using per-line MAC that is kept within the ECC space. Detecting data-corruption can also be achieved by prior secure systems that provide memory integrity protection. However, such techniques typically incur significant performance and storage overhead, making them impractical for widespread adoption. In contrast, SafeGuard provides integrity protection at a low cost and without requiring any modification to the existing commodity ECC DIMMs. We compare SafeGuard with two alternative organizations to store per-line MACs.

A. Prior Memory Organizations for Integrity Protection

1. SGX-Style MAC Organization: Intel SGX [12] is a secure computing design that provides integrity protection guarantees for a small designated region of memory. Each data line is protected by a per-line MAC that is stored in a separate location in the memory and incurs 12.5% storage overhead. On every memory access, the processor needs to make an extra access for MAC, which increases the memory bandwidth pressure. To make a fair comparison, we do not consider the overheads associated with accessing any other metadata of SGX (encryption counters or integrity trees).

2. Synergy-Style MAC Organization: Synergy [39] reduces the memory bandwidth required for MAC by co-designing for security and reliability.[3] Synergy uses x8 ECC DIMMs and stores the 64-bit MAC in the ECC chip and 64-bit chip-wise parity in a different location of the memory. Synergy eliminates the MAC access overhead caused by the memory reads, however, writes still require two accesses (one for data and another for parity). Synergy continues to incur 12.5% storage overhead to store the MAC.

[3]We note that while our design of SafeGuard with Chipkill is inspired by Synergy, there are key differences. Synergy was developed for x8 DIMMs (SECDED), uses a 64-bit MAC, and stores parity in a different DIMM. In our case, we use x4 devices, which are typically used for Chipkill. For these DIMMs, each chip provides only 32-bits for each line and uses two extra chips to store ECC. Thus, we are limited to using 32-bit MAC and the remaining 32-bits for parity. As shown in Section V-C, a 32-bit MAC does not provide Chipkill-level reliability in the presence of permanent chip failures (this problem would not occur if we could use 64-bit MAC as employed in Synergy). Our proposed implementation (Eager Correction) overcomes this limitation while still retaining 32-bit MAC and avoiding the second access for parity update.

B. Performance Comparison

Figure 12 compares the performance of SafeGuard with SGX-style MAC and Synergy-style MAC. We use the same MAC latency for all designs. To make a fair performance comparison with SafeGuard, we only consider the MAC access overhead for SGX-style MAC and Synergy-style MAC (thus, there are no performance overheads for any other metadata such as encryption counters or integrity trees). SGX-style MAC and Synergy-style MAC incur 18.7% and 7.8% slowdown, respectively, whereas SafeGuard incurs only a 0.7% slowdown.

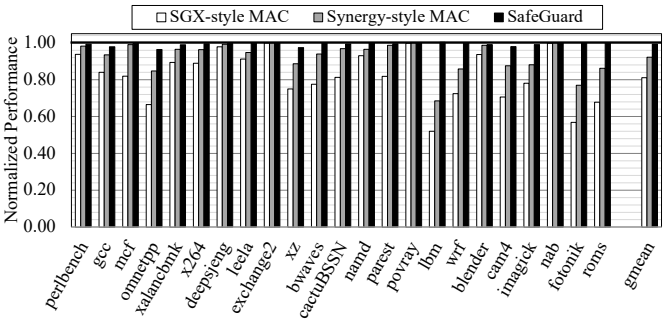


Figure 12: Performance of SafeGuard vs. alternative organization for storing the per-line MAC (SGX or Synergy style).

C. Comparison of DRAM Storage Overheads

As we want the system to correct naturally occurring errors, we assume a baseline memory system that is equipped with ECC DIMM. We compare SGX-style MAC, Synergy-style MAC, and SafeGuard in terms of DRAM overheads. Both SGX-style MAC and Synergy-style MAC require an overhead of 12.5% to store the MAC (or parity in case of Synergy) in a separate region in memory. Table V compares the storage overhead of SGX-style MAC, Synergy-style MAC, and SafeGuard as the memory system size is increased from 16GB to 64GB to 256GB. Both SGX-style MAC and Synergy-style MAC significantly reduce the usable memory space, whereas SafeGuard provides full memory as the usable memory space. This makes SafeGuard even more appealing for widespread adoption.

Table V: Comparison of DRAM Storage Overhead (baseline uses ECC DIMM. All designs need 12.5% DRAM for ECC).

Baseline Memory (ECC DIMM)	Usable Memory Capacity	
	SGX/Synergy-style MAC	**SafeGuard**
16GB	14GB (2GB loss)	**16GB**
64GB	56GB (8GB loss)	**64GB**
256GB	224GB (32GB loss)	**256GB**

D. Sensitivity to MAC Latency

SafeGuard can be used with any MAC algorithm. We use a default MAC latency of 8 processor cycles (See Table II). Figure 13 shows the performance overhead of SafeGuard, SGX-Style MAC, and Synergy-style MAC as the MAC latency is varied from 8 cycles to 80 cycles. Even with 80 cycles MAC, SafeGuard incurs 5.8% performance overhead, outperforming SGX-style and Synergy-style MAC organizations by 19.5% and 7.1%, respectively, with no additional DRAM overheads.

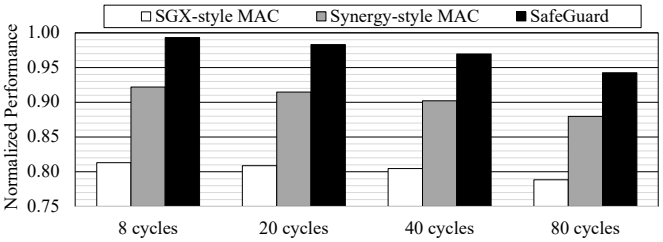


Figure 13: Performance sensitivity to MAC latency.

VII. Security Discussion

We design SafeGuard as a low-cost integrity protection scheme, motivated primarily by the problem of arbitrary bit-flips from Row-Hammer. We assume that the system employs some form of RH mitigation to correct RH faults in the common case. So, we are mainly concerned about attacks that break through the RH mitigation. We discuss the security issues for our design.

A. Actions After Detection

Similar to current systems (SGX) that provide integrity protection, SafeGuard informs the system when a DUE (Detected Unrecoverable Error) is encountered. This lets the system take preventative actions, such as restarting the process, relocating the process to a different machine (in the case of cloud systems), or rebooting the server. We observe that with RH, the attacker can theoretically flip an arbitrary number of bits in a line, and it is virtually impossible to rely on ECC (or the hardware) alone to correct these errors.

B. Vulnerability to Denial-of-Service

Similar to systems that provide integrity protection, SafeGuard relies on the software to take corrective actions in case of an uncorrectable error. An adversary who is able to persistently cause failures can use this property to orchestrate Denial-of-Service (DoS) attacks [16]. We note that in the absence of SafeGuard, the adversary would be able to silently launch much worse attacks (such as privilege escalation), so the SafeGuard approach is still preferable. Furthermore, in case of persistent failures, the system can identify potentially malicious processes and take preventative actions [10], [33].

C. Vulnerability to Replay Attacks

MAC-based checking is vulnerable to attacks that replay an older pair of data and MAC. Our threat model assumes a remote adversary without physical access to the system. To the best of our knowledge, replay attacks are impractical to launch with remotely controlled Row-Hammer (adversary would need to know the previous value and precisely flip a large number of bits in both the data and MAC). Therefore, we do not consider replay attack protection in our work.

D. Vulnerability to Timing Channels

SafeGuard performs correction of faulty data, and this correction can leak the presence of data errors due to a timing difference. Prior work [6] used this timing channel to progressively increase the RH bit-flips and escape the detection of SECDED. SafeGuard provides strong detection, so even if the attacker exploits this timing channel to increase the number of bit-flips, SafeGuard will detect it.

RAMBleed [25] uses the data-dependent nature of RH to infer the data of neighboring row. While SafeGuard can prevent data corruption (via ECC correction), the attacker could potentially exploit the timing channel of ECC correction to break confidentiality. RAMBleed can be prevented using low-cost memory encryption (e.g., Intel TME [14]).

E. Vulnerability to MAC Collisions

When faulty data is checked against a MAC, there is a small escape probability ($1/2^n$ for n-bit MAC) with each check. A larger MAC provides stronger protection but incurs more storage overhead. The MAC size that is sufficient depends on the frequency of checks of corrupted data and the actions on detection (restart or continue). While SGX uses 56-bit MAC, recent *Intel Trust Domain Extensions (TDX)* [15] uses 28-bit MAC to protect the virtual machines.

SafeGuard uses MAC ranging from 32-bits (Chipkill) to 46-bit (SECDED). Given that RH-based bit-flips require a long time to occur (on the granularity of refresh interval) and the system may take preventative actions, we deem these MAC sizes to be sufficient to prevent RH attacks.

For example, for SafeGuard with SECDED, if the breakthrough attack corrupts one line in memory every refresh period (64ms), then it will take the adversary a continuous attack of 1000+ years with 46-bit MAC to get one episode of MAC escape on average. For the Chipkill version (32-bit MAC), if we use the iterative correction, the same attack rate can exhaust the MAC detection within 6 months, as each fault can incur up to 18 MAC verification failures. Eager correction effectively increases the attack time by 18x to 9 years as it only requires a single MAC check. Note that these bounds are derived assuming no preventative actions are taken in any of the prior million(s) of times when the MAC correctly identified the faulty lines. In reality, the system will take some preventative actions in the interim. Thus, the MAC sizes that we use are sufficient for SafeGuard.

VIII. Other Related Work

Hardware-based RH defenses can be divided into tracking (CRA [18], ProHit [42], TWiCE [26], and Graphene [35]) and probabilistic (PRA [18], PARA [21]). Both defenses rely on knowing the RH-Threshold and are vulnerable to more complex patterns. BlockHammer [47] is a recent RH mitigation that uses Bloom-Filter to detect *hot-rows* and reduces the access rate to these rows, such that the number of activations within a refresh interval remains below the RH-Threshold. Block-Hammer can drastically increase memory latency for some rows (e.g., at RH-Threshold of 1K, a memory access can take more than 125 micro-seconds). Furthermore, Block-Hammer requires knowledge of the RH-Threshold and using a module with a lower RH-Threshold than the one assumed while designing the system can still lead to bit-flips, even in the presence of the solution.

ANVIL [2] is a software mitigation that uses CPU performance counters to detect RH attacks and issues activations to the victim rows. ANVIL relies on the RH-Threshold, and it is vulnerable to complex attack access patterns. CATT [4], GuardION [46], ZebRAM [23], and RIP-RH [3] create in-DRAM isolation by allowing a guard row between adjacent DRAM rows to protect different security domains. These techniques only consider the immediate adjacent row and may be vulnerable to more complex patterns.

VS-ECC [48] virtualizes ECC for non-ECC memories. Unfortunately, it requires extra memory accesses to fetch the ECC metadata. Bamboo ECC [20] uses *vertical* Reed-Solomon [37] codes to protect against pin failures, but it does not provide strong detection of arbitrary failures. Morphable ECC [5] re-configures the ECC bits to dynamically employ stronger (ECC-6) code at line-granularity.

Secure systems, such as SGX and Synergy, protect memory integrity, however, they incur significant slowdown. VAULT [45] proposes a variable arity integrity tree to lower the integrity tree accesses and uses compression to store data and its MAC in one line. Morphable Counters [38] uses dynamic counter formats to reduce the size of the integrity trees. Taassori et al. [44] try to reduce the parity update overhead of Synergy using a compact integrity tree format that also includes parities. IVEC [13] proposes placing parities in ECC chips using a non-bonsai MAC tree for integrity protection. This requires MACs to be stored in separate cache-line and makes caching them challenging. Compared to SGX, IVEC incurs a slowdown of 26% [39].

IX. Conclusion

Current Row-Hammer mitigations continue to be broken by the emergence of complex access patterns. In this paper, we contend that as long as the system remains vulnerable to Row-Hammer (even in the presence of mitigations), it is important to detect the bit-flips caused by RH as silent consumption of corrupted data can lead to security threats. To this end, we propose *SafeGuard* for low-cost detection of arbitrary data errors using per-line MACs. SafeGuard avoids the storage and performance overheads of the conventional integrity-protected memories. SafeGuard can effectively reduce the severity of RH attacks from a security risk to a reliability issue. We implement SafeGuard in the context of conventional ECC designs such as SECDED and Chipkill. We show that while SafeGuard provides strong detection of arbitrary errors, it has similar correction strength as conventional ECC designs against naturally occurring errors. As we step into the era of reducing memory reliability and increasingly complex security attacks, we believe solutions such as SafeGuard would become increasingly necessary to protect the memory system against both conventional fault models (well-known and characterized) and unknown fault models (emerging and adversarial).

Acknowledgment

We thank Stefan Saroiu, Kaveh Razavi, and Sudhanva Gurumurthi for their suggestions (on an earlier version of this paper [36]) that significantly helped shape this work. This work was supported in part by Intel, the Cockrell Foundation, Arm, NSF (Award #2011145), the Natural Sciences and Engineering Research Council of Canada (RGPIN-2019-05059), and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP). We thank the members of the HPS Research Group and Poulami Das for their editorial suggestions and feedback. We also thank TACC for providing computing resources.

References

[1] Spec 2017. https://www.spec.org/cpu2017.

[2] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," in *Proceedings of the Twenty-First ASPLOS'16*. Association for Computing Machinery, 2016, p. 743–755.

[3] 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.

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

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

[6] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[7] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[8] 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.

[9] Google. "half-double": Next-row-over assisted rowhammer. Available Online.

[10] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

[12] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[13] R. Huang and G. E. Suh, "Ivec: Off-chip memory integrity protection for both security and reliability," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 395–406. [Online]. Available: https://doi.org/10.1145/1815961.1816015

[14] Intel, "Intel® architecture memory encryption technologies specification," Available Online.

[15] ——, "Intel® trust domain extensions, white paper," Available Online.

[16] Y. Jang, J. Lee, S. Lee, and T. Kim, "Sgx-bomb: Locking down the processor via rowhammer attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3152701.3152709

[17] M. Kaczmarski, "Thoughts on intel xeon e5-2600 v2 product family performance optimisation–component selection guidelines," 2014.

[18] 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.

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

[20] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st HPCA*. IEEE, 2015, pp. 101–112.

[21] 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," in *Proceeding of the 41st ISCA*. IEEE Press, 2014, p. 361–372.

[22] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[23] 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.

[24] M. Kounavis, S. Deutsch, S. Ghosh, and D. Durham, "K-cipher: A low latency, bit length parameterizable cipher," in *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020, pp. 1–7.

[25] 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.

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

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

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference, PLDI*. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200.

[29] Advanced Micro Devices (AMD) Inc., "Bios and kernel developer's guide (bkdg) for amd family 15h models 00h-0fh processors," Jan 2013.

[30] AVNET. Micron ddr5 sdram. Available Online.

[31] HPS/SAFARI Research Group, "Scarab," https://github.com/hpsresearchgroup/scarab.

[32] Micron. Ddr4 sdram datasheet. Available Online.

[33] Microsoft, "Isolation in the azure public cloud," Available Online.

[34] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Faultsim: A fast, configurable memory-reliability simulator for conventional and 3d-stacked systems," *ACM TACO*, vol. 12, no. 4, pp. 1–24, 2015.

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

[36] M. Qureshi, "Rethinking ECC in the era of row-hammer," *First workshop on DRAM Security, co-located with ISCA 2021*.

[37] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[38] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM MICRO*. IEEE, 2018, pp. 416–427.

[39] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE HPCA*, 2018, pp. 454–465.

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

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th ASPLOS*. New York, NY, USA: Association for Computing Machinery, 2002, p. 45–57.

[42] 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.

[43] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[44] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, "Compact leakage-free support for integrity and reliability," in *2020 ACM/IEEE 47th ISCA*. IEEE, 2020, pp. 735–748.

[45] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *ASPLOS*, 2018, pp. 665–678.

[46] 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.

[47] 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 HPCA*, 2021, pp. 345–358.

[48] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 397–408.