BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows

A. Giray Yağlıkçı[1] Minesh Patel[1] Jeremie S. Kim[1] Roknoddin Azizi[1] Ataberk Olgun[1] Lois Orosa[1]
Hasan Hassan[1] Jisung Park[1] Konstantinos Kanellopoulos[1] Taha Shahroodi[1] Saugata Ghose[2] Onur Mutlu[1]
[1]*ETH Zürich* [2]*University of Illinois at Urbana–Champaign*

Aggressive memory density scaling causes modern DRAM devices to suffer from RowHammer, a phenomenon where rapidly activating (i.e., hammering) a DRAM row can cause bit-flips in physically-nearby rows. Recent studies demonstrate that modern DDR4/LPDDR4 DRAM chips, including chips previously marketed as RowHammer-safe, are even more vulnerable to RowHammer than older DDR3 DRAM chips. Many works show that attackers can exploit RowHammer bit-flips to reliably mount system-level attacks to escalate privilege and leak private data. Therefore, it is critical to ensure RowHammer-safe operation on all DRAM-based systems as they become increasingly more vulnerable to RowHammer. Unfortunately, state-of-the-art RowHammer mitigation mechanisms face two major challenges. First, they incur increasingly higher performance and/or area overheads when applied to more vulnerable DRAM chips. Second, they require either closely-guarded proprietary information about the DRAM chips' physical circuit layouts or modifications to the DRAM chip design.

In this paper, we show that it is possible to efficiently and scalably prevent RowHammer bit-flips without knowledge of or modification to DRAM internals. To this end, we introduce BlockHammer, a low-cost, effective, and easy-to-adopt RowHammer mitigation mechanism that prevents all RowHammer bit-flips while overcoming the two key challenges. BlockHammer selectively throttles memory accesses that could otherwise potentially cause RowHammer bit-flips. The key idea of BlockHammer is to (1) track row activation rates using area-efficient Bloom filters, and (2) use the tracking data to ensure that no row is ever activated rapidly enough to induce RowHammer bit-flips. By guaranteeing that no DRAM row ever experiences a RowHammer-unsafe activation rate, BlockHammer (1) makes it impossible for a RowHammer bit-flip to occur and (2) greatly reduces a RowHammer attack's impact on the performance of co-running benign applications. Our evaluations across a comprehensive range of 280 workloads show that, compared to the best of six state-of-the-art RowHammer mitigation mechanisms (all of which require knowledge of or modification to DRAM internals), BlockHammer provides (1) competitive performance and energy when the system is not under a RowHammer attack and (2) significantly better performance and energy when the system is under a RowHammer attack.

1. Introduction

Improvements to manufacturing process technology have increased DRAM storage density by reducing DRAM cell size and cell-to-cell spacing for decades. Although such optimizations improve a DRAM chip's cost-per-bit, they negatively impact DRAM reliability [93, 100]. Kim et al. [73] show that modern DRAM chips are susceptible to the RowHammer phenomenon, where opening and closing (i.e., *activating* and *precharging*) a DRAM row (i.e., *aggressor row*) at a *high enough* rate (i.e., *hammering*) can cause bit-flips in physically-nearby rows (i.e., *victim rows*) [101, 104, 121, 159]. Many works demonstrate various system-level attacks using RowHammer to escalate privilege or leak private data (e.g., [1, 10, 13, 24, 25, 34, 35, 41, 42, 47, 50, 56, 79, 87, 101, 104, 117, 118, 120, 126, 127, 144, 147, 148, 151, 156, 160, 163]). Recent findings indicate that RowHammer is a more serious problem than ever and that it is expected to worsen for future DRAM chips [72, 101, 104]. Therefore, comprehensively protecting DRAM against all types of RowHammer attacks is essential for the security and reliability of current and future DRAM-based computing systems.

Although DRAM vendors currently implement in-DRAM RowHammer mitigation mechanisms, e.g., target row refresh [35, 53–55, 85, 95], recent works report that commodity DDR3 [112], DDR4 [1, 24, 35, 72, 117], and LPDDR4 [72] chips remain vulnerable to RowHammer. In particular, TRResspass [35] shows that an attacker can still reliably induce RowHammer bit-flips in commodity (LP)DDRx DRAM chips by circumventing the in-DRAM mitigation mechanisms. Kim et al. [72] show that from 2014 to 2020, DRAM chips have become significantly more vulnerable to RowHammer bit-flips, with over an order of magnitude reduction in the required number of row activations to induce a bit-flip (from 139.2k to 9.6k).

Given the severity of RowHammer, various mitigation methods have been proposed, which we classify into four high-level approaches: (*i*) *increased refresh rate*, which refreshes *all* rows more frequently to reduce the probability of a successful bit-flip [2, 73]; (*ii*) *physical isolation*, which physically separates sensitive data from any potential attacker's memory space (e.g., by adding buffer rows between sensitive data regions and other data) [14, 78, 148]; (*iii*) *reactive refresh*, which observes row activations and refreshes the potential victim rows as a reaction to rapid row activations [5, 73, 84, 113, 132, 137, 161]; and (*iv*) *proactive throttling*, which limits row activation rates [40, 73, 102] to RowHammer-safe levels. Unfortunately, each of these four approaches faces at least one of two major challenges towards effectively mitigating RowHammer.

Challenge 1: Efficient Scaling as RowHammer Worsens. As DRAM chips become more vulnerable to RowHammer (i.e., RowHammer bit-flips can occur at significantly lower row activation counts than before), mitigation mechanisms need to act more aggressively. A *scalable* mechanism should exhibit acceptable performance, energy, and area overheads as its design is reconfigured for more vulnerable DRAM chips. Unfortunately, as chips become more vulnerable to RowHammer, most state-of-the-art mechanisms of all four approaches either cannot easily adapt because they are based on fixed design points, or their performance, energy, and/or area overheads become increasingly significant. (*i*) Increasing the refresh rate further in order to prevent all RowHammer bit-flips is prohibitively expensive, even for existing DRAM chips [72], due to the large number of rows that must be refreshed within a refresh window. (*ii*) Physical isolation mechanisms must provide greater isolation (i.e., increase the physical distance) between sensitive data and a potential attacker's memory space as DRAM chips become denser and more vulnerable to RowHammer. This is because denser chip designs bring circuit elements closer together, which increases the number of rows across which the hammering of an aggressor row can induce RowHammer bit-flips [72, 73, 101, 159]. Providing greater isolation (e.g., increasing the number of buffer rows between sensitive data and an attacker's memory space) both wastes increasing amounts of memory capacity and reduces the fraction of physical memory that can be protected from RowHammer attacks. (*iii*) Reactive refresh mechanisms need to increase the rate at which they refresh potential victim rows. Prior work [72] shows that state-of-the-art reactive refresh RowHammer mitigation mechanisms lead to prohibitively large performance overheads with increasing RowHammer vulnerability. (*iv*) Existing proactive throttling approaches must throttle activations at a more aggressive rate to counteract the increased RowHammer vulnerability. This

requires either throttling row activations of benign applications as well or tracking per-row activation rates for the entire refresh window, incurring prohibitively-expensive performance or area overheads even for existing DRAM chips [73, 102].

Challenge 2: Compatibility with Commodity DRAM Chips. Both (*ii*) physical isolation and (*iii*) reactive refresh mechanisms require the ability to either (1) identify *all potential victim rows* that can be affected by hammering a given row or (2) modify the DRAM chip such that either the potential victim rows are internally isolated within the DRAM chip or the RowHammer mitigation mechanism can accurately issue reactive refreshes to all potential victim rows. Identifying all potential victim rows requires knowing the mapping schemes that the DRAM chip uses to internally translate memory-controller-visible row addresses to physical row addresses [9, 24, 48, 49, 62, 65, 67, 73, 81, 88, 114, 130, 135, 144]. Unfortunately, DRAM vendors consider their in-DRAM row address mapping schemes to be highly *proprietary* and do not reveal any details in publicly-available documentation, as these details contain insights into the chip design and manufacturing quality [48, 49, 62, 81, 114, 135] (discussed in Section 2.3). As a result, both physical isolation and reactive refresh are limited to systems that can (1) obtain such proprietary information on in-DRAM row address mapping or (2) modify DRAM chips internally.

Our goal in this paper is to design a low-cost, effective, and easy-to-adopt RowHammer mitigation mechanism that (1) scales efficiently with worsening RowHammer vulnerability to prevent RowHammer bit-flips in current and future DRAM chips, and (2) is seamlessly compatible with *commodity* DRAM chips, without requiring proprietary information about or modifications to DRAM chips. To this end, we propose BlockHammer, a new proactive throttling-based RowHammer mitigation mechanism. BlockHammer's key idea is to track row activation rates using area-efficient Bloom filters and use the tracking data to ensure that no row is ever activated rapidly enough to induce RowHammer bit-flips. Because BlockHammer requires no proprietary information about or modifications to DRAM chips, it can be implemented completely within the memory controller. Compared to prior works that require proprietary information or DRAM chip modifications, BlockHammer provides (1) competitive performance and energy when the system is not under a RowHammer attack and (2) significantly better performance and energy (average/maximum of 45.0%/61.9% and 28.9%/33.8%, respectively) when the system *is* under a RowHammer attack. To our knowledge, this is the first work that prevents RowHammer bit-flips efficiently and scalably without knowledge of or modification to DRAM internals.

Key Mechanism. BlockHammer consists of two components: *RowBlocker* and *AttackThrottler*. *RowBlocker* tracks and limits the activation rates of DRAM rows to a rate lower than at which RowHammer bit-flips begin to occur, i.e., the *RowHammer threshold* (N_{RH}). To track activation rates in an area-efficient manner, RowBlocker employs a false-negative-free variant of counting Bloom filters [33, 86] that eliminates the need for per-row counters. When RowBlocker observes that a row's activation count within a given time interval exceeds a predefined threshold (which we set to be smaller than N_{RH}), RowBlocker *blacklists* the row, i.e., flags the row as a potential aggressor row and limits further activations to the row until the end of the time interval, ensuring that the row's overall activation rate never reaches a RowHammer-unsafe level. As a result, RowBlocker ensures that a successful RowHammer attack is impossible.

AttackThrottler alleviates the performance degradation a RowHammer attack imposes on benign applications. To do so, AttackThrottler reduces the memory bandwidth usage of an attacker thread by applying a quota to the thread's total number of in-flight memory requests for a determined time period. AttackThrottler sets the quota for each thread inversely proportional to the rate at which the thread activates a blacklisted row. As a result, AttackThrottler reduces the memory bandwidth consumed by an attacker, thereby allowing concurrently-running benign applications to have higher performance when accessing memory. To further mitigate the performance impact of RowHammer attacks, AttackThrottler can optionally expose the rate at which each thread activates a blacklisted row to the operating system (OS). This information can be used as a dependable indicator of a thread's likelihood of performing a RowHammer attack, enabling the OS to employ more sophisticated thread scheduling and quality-of-service support.

We evaluate BlockHammer's (1) security guarantees via a mathematical proof in Section 5; (2) area, static power, access energy, and latency overheads for storing and accessing metadata by using circuit models [99, 143] in Section 6.1; and (3) performance and DRAM energy overheads using cycle-level simulations [18, 77, 125] in Section 8. Our evaluations for a realistic RowHammer threshold (32K activations within a 64 ms refresh window [72]) show that BlockHammer guarantees RowHammer-safe operation with only 0.06% area, 0.7% performance, and 0.6% DRAM energy overheads for benign (i.e., non-attacking) workloads, compared to a baseline system with no RowHammer mitigation. When a RowHammer attack exists within a multiprogrammed workload, BlockHammer successfully identifies and throttles the attacker's row activations with 99.98% accuracy, resulting in a 45.0% average improvement in the performance of concurrently-running benign applications. We show that BlockHammer more efficiently scales with increasing RowHammer vulnerability than six state-of-the-art RowHammer mitigation mechanisms, without requiring knowledge of or modification to the internals of DRAM chips.

Building on analyses done by prior work on RowHammer mitigation [41, 72, 73, 101, 102, 104], we describe in Section 9 that a low-cost, effective, and easy-to-adopt RowHammer mitigation mechanism must: (1) address a *comprehensive threat model*, (2) be seamlessly compatible with *commodity* DRAM chips (i.e., require no knowledge of or modifications to DRAM chip internals), (3) *scale* efficiently with increasing RowHammer vulnerability, and (4) *deterministically* prevent all RowHammer attacks. We find that, among all 14 RowHammer mitigation mechanisms that we examine, BlockHammer is the *only* one that satisfies all four key properties.

We make the following contributions in this work:

- We introduce the first mechanism that efficiently and scalably prevents RowHammer bit-flips *without* knowledge of or modification to DRAM internals. Our mechanism, BlockHammer, provides competitive performance and energy with existing RowHammer mitigation mechanisms when the system is *not* under a RowHammer attack, and *significantly* better performance and energy than existing mechanisms when the system *is* under a RowHammer attack.
- We show that a proactive throttling approach to prevent RowHammer bit-flips can be implemented efficiently using Bloom filters. We employ a variant of counting Bloom filters that (1) avoids the area and energy overheads of per-row counters used by prior proactive throttling mechanisms, and (2) never fails to detect a RowHammer attack.
- We show that we can greatly reduce the performance degradation and energy wastage a RowHammer attack inflicts on benign threads and the system by accurately identifying the RowHammer attack thread and reducing its memory bandwidth usage. We introduce a new metric called the *RowHammer likelihood index*, which enables the memory controller to distinguish a RowHammer attack from a benign thread.

2. Background

This section provides a concise overview of (1) DRAM organization and operation, (2) the RowHammer phenomenon, and (3) in-DRAM row address mapping. For more detail, we refer the reader to prior works on DRAM and RowHammer [19–22, 35, 37, 44–46, 62, 64–66, 69–74, 76, 81–83, 88–90, 98, 105, 106, 114–116, 119, 128–131, 139, 150].

2.1. DRAM Organization and Operation

Figure 1 shows the high-level structure of a typical DRAM-based system. At the lowest level of the hierarchy, DRAM stores data within *cells* that each consist of a single capaci-

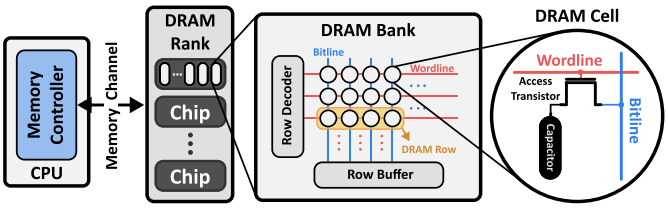

Figure 1: Structure of a typical DRAM-based system.

tor and an access transistor. Each cell encodes a single bit of data using the "high" and "low" voltage states of the capacitor. Because a DRAM cell leaks charge over time, each cell's charge is periodically restored (i.e., refreshed) (e.g., every 32 or 64 ms [53, 55, 88, 89]) to prevent data loss. Cells are arranged in two-dimensional arrays to form DRAM *banks*.

DRAM cells in a bank are addressed using *rows* and *columns*. A *wordline* drives all DRAM cells in a *row*, and a *bitline* connects all DRAM cells in a *column*. All rows within a bank share the peripheral circuitry, so only one row may be accessed per bank at any given time. Each row begins in a *closed* (i.e., *precharged*) state and needs to be *opened* (i.e., *activated*) before any READ or WRITE operations can be performed on it. Activating a row fetches the row's contents into the *row buffer*. The row buffer serves all read and write requests after fetching the data. The row must be *closed* before further accesses can be made to other rows of the same bank.

A DRAM chip contains multiple banks that can be accessed in parallel. Multiple chips form a DRAM *rank*. At the highest level of the hierarchy, the *memory controller* in the CPU die interfaces with a DRAM rank through a *memory channel*. The memory controller serves memory access requests from various system components by issuing DRAM bus commands (e.g., activate, precharge, read, write, and refresh). The memory controller must schedule commands according to standardized *timing parameters*, which are defined in DRAM datasheets to ensure that each operation has enough time to complete before starting the next [51, 53–55, 95]. The overall strategy that the memory controller uses to schedule commands is known as a *scheduling policy*. Typical policies seek to optimize performance, fairness, quality of service (QoS), and energy across applications running on a system [3, 31, 74, 75, 106, 122, 139, 140, 146]. Therefore, the scheduling policy effectively controls all accesses to all DRAM channels, banks, rows, and columns.

2.2. The RowHammer Phenomenon

RowHammer is a DRAM failure mode in which repeated activations to a single row (i.e., *aggressor* row) cause disturbance capable of inducing bit-flips in *physically-nearby* rows (i.e., *victim* rows) that are not being accessed [73]. These bit-flips manifest after a row's activation count reaches a certain threshold value within a refresh window, which we call *RowHammer threshold* (N_{RH}) (also denoted as *MAC* [55] and HC_{first} [72]). Prior works study the error characteristics of RowHammer bit-flips and show that N_{RH} varies across DRAM vendors, device models, generations, and chips [24, 35, 72, 73, 112]. Yang et al. [159] explain this N_{RH} variation based on changing physical distances between adjacent wordlines (i.e., physical DRAM rows). Since DRAM chip density increases at smaller feature sizes, both Yang et al.'s observation and recent experimental studies [35, 72, 73] clearly demonstrate that RowHammer worsens with continued technology scaling [101, 104]. In addition, recent studies show that emerging memory technologies also exhibit RowHammer vulnerability [63, 101, 104].

2.3. In-DRAM Row Address Mapping

DRAM vendors often use DRAM-internal mapping schemes to internally translate memory-controller-visible row addresses to physical row addresses [9, 24, 48, 49, 62, 65, 67, 73, 81, 88, 114, 130, 135, 144] for two reasons: (1) to optimize their chip design for density, performance, and power constraints; and (2) to improve factory yield by mapping the addresses of faulty rows to more reliable spare rows (i.e., post-manufacturing row repair). Therefore, row mapping schemes can vary with (1) chip design variation across different vendors, DRAM models, and generations and (2) manufacturing process variation across different chips of the same design. State-of-the-art RowHammer mitigation mechanisms must account for both sources of variation in order to be able to accurately identify all potential victim rows that are physically nearby an aggressor row. Unfortunately, DRAM vendors consider their in-DRAM row address mapping schemes to be highly proprietary and ensure not to reveal mapping details in any public documentation because exposing the row address mapping scheme can reveal insights into the chip design and factory yield [48, 49, 62, 81, 114, 135].

3. BlockHammer

BlockHammer is designed to (1) scale efficiently as DRAM chips become increasingly vulnerable to RowHammer and (2) be compatible with commodity DRAM chips. BlockHammer consists of two components. The first component, RowBlocker (Section 3.1), prevents any possibility of a RowHammer bit-flip by making it impossible to access a DRAM row at a high enough rate to induce RowHammer bit-flips. RowBlocker achieves this by efficiently tracking row activation rates using Bloom filters and throttling the row activations that target rows with high activation rates. We implement RowBlocker entirely within the memory controller, ensuring RowHammer-safe operation without any proprietary information about or modifications to the DRAM chip. Therefore, RowBlocker is compatible with all commodity DRAM chips. The second component, AttackThrottler (Section 3.2), alleviates the performance degradation a RowHammer attack can impose upon benign applications by selectively reducing the memory bandwidth usage of *only* threads that AttackThrottler identifies as likely RowHammer attacks (i.e., *attacker threads*). By doing so, AttackThrottler provides a larger memory bandwidth to benign applications compared to a baseline system that does not throttle attacker threads. As DRAM chips become more vulnerable to RowHammer, AttackThrottler throttles attacker threads more aggressively, freeing even more memory bandwidth for benign applications to use. By combining RowBlocker and AttackThrottler, BlockHammer achieves both of its design goals.

3.1. RowBlocker

RowBlocker's goal is to proactively throttle row activations in an efficient manner to avoid any possibility of a RowHammer attack. RowBlocker achieves this by overcoming two challenges regarding performance and area overheads.

First, achieving low performance overhead is a key challenge for a throttling mechanism because many benign applications tend to repeatedly activate a DRAM row that they have recently activated [44, 45, 57, 76]. This can potentially cause a throttling mechanism to mistakenly throttle benign applications, thereby degrading system performance. To ensure throttling *only* applications that might cause RowHammer bit-flips, RowBlocker throttles the row activations targeting *only* rows whose activation rates are above a given threshold. To this end, RowBlocker implements two components as shown in Figure 2: (1) a per-bank blacklisting mechanism, RowBlocker-BL, which blacklists all rows with an activation rate greater than a predefined threshold called the *blacklisting threshold* (N_{BL}); and (2) a per-rank activation history buffer, RowBlocker-HB, which tracks the most recently activated rows. RowBlocker enforces a time delay between two consecutive activations targeting a row *only if* the row is *blacklisted*. By doing so, RowBlocker is less likely to throttle a benign application's row activations.

Second, achieving low area overhead is a key challenge for a throttling mechanism because throttling requires tracking all row activations throughout an entire refresh window *without* losing information of any row activation. RowBlocker implements its blacklisting mechanism, RowBlocker-BL, by using area-efficient *counting Bloom filters* [11, 33] to track row activation rates. RowBlocker-BL maintains two counting Bloom filters in a time-interleaved manner to track row activation rates

for large time windows without missing any row that should be blacklisted. We explain how counting Bloom filters work and how RowBlocker-BL employs them in Section 3.1.1.

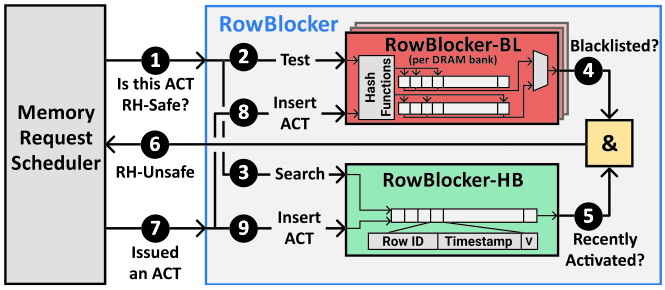

Figure 2: High-level overview of RowBlocker (per DRAM rank). An ACT is accompanied by its row address.

High-Level Overview of RowBlocker. RowBlocker modifies the memory request scheduler to temporarily block (i.e., delay) an activation that targets a *blacklisted* and *recently-activated* row until the activation can be safely performed. By blocking such row activations, RowBlocker ensures that no row can be activated at a high enough rate to induce RowHammer bit-flips. When the memory request scheduler attempts to schedule a row activation command to a bank, it queries RowBlocker (❶) to check if the row activation is RowHammer-safe. This simultaneously triggers two lookup operations. First, RowBlocker checks the RowBlocker-BL to see if the row to be activated is blacklisted (❷). A row is blacklisted if its activation rate exceeds a given threshold. We discuss how RowBlocker-BL estimates the activation rate of a row in Section 3.1.1. Second, RowBlocker checks RowBlocker-HB to see if the row has been recently activated (❸). If a row is both blacklisted (❹) *and* recently activated (❺), RowBlocker responds to the memory request scheduler with a *RowHammer-unsafe* signal (❻), consequently blocking the row activation. Blocking such a row activation is essential because allowing further activations to a blacklisted and recently-activated row could increase the row's overall activation rate and thus result in RowHammer bit-flips. The memory request scheduler does *not* issue a row activation if RowBlocker returns *unsafe*. However, it keeps issuing the *RowHammer-safe* requests. This scheduling decision effectively prioritizes RowHammer-safe memory accesses over unsafe ones. An unsafe row activation becomes safe again as soon as a certain amount of time (t_{Delay}) passes after its latest activation, effectively limiting the row's average activation rate to a RowHammer-safe value. After t_{Delay} is satisfied, RowBlocker-HB no longer reports that the row has been recently activated (❺), thereby allowing the memory request scheduler to issue the row activation (❻). When the memory request scheduler issues a row activation (❼), it simultaneously updates both RowBlocker-BL (❽) and RowBlocker-HB (❾). We explain how RowBlocker-BL and RowBlocker-HB work in Section 3.1.1 and 3.1.2, respectively.

3.1.1. RowBlocker-BL Mechanism. RowBlocker-BL uses two counting Bloom filters (CBF) in a time-interleaved fashion to decide whether a row should be blacklisted. Each CBF takes turns to make the blacklisting decision. A row is blacklisted when its activation rate exceeds a configurable threshold, which we call the *blacklisting threshold* (N_{BL}). When a CBF blacklists a row, any further activations targeting the row are throttled until the end of the CBF's turn. In this subsection, we describe how a CBF works, how we use two CBFs to avoid stale blacklists, and how the two CBFs never fail to blacklist an aggressor row.

Bloom Filter. A Bloom filter [11] is a space-efficient probabilistic data structure that is used for testing whether a set contains a particular element. A Bloom filter consists of a set of hash functions and a bit array on which it performs three operations: *clear*, *insert*, and *test*. Clearing a Bloom filter zeroes its bit array. To insert/test an element, each hash function evaluates an index into the bit array for the element, using an identifier for the element. Inserting an element sets the bits that the hash functions point to. Testing for an element checks whether all these bits are set. Since a hash function can yield the same set of indices for different elements (i.e., aliasing), testing a Bloom filter can return true for an element that was never inserted (i.e., false positive). However, the *test* operation never returns false for an inserted element (i.e., no false negatives). A Bloom filter eventually saturates (i.e., always returns true when tested for any element) if elements are continually inserted, which requires periodically clearing the filter and losing all inserted elements.

Unified Bloom Filter (UBF). UBF [86] is a Bloom filter variant that allows a system to continuously track a set of elements that are inserted into a Bloom filter within the most recent time window of a fixed length (i.e., a *rolling time window*). Using a conventional Bloom filter to track a rolling time window could result in data loss whenever the Bloom filter is cleared, as the clearing eliminates the elements that still fall within the rolling time window. Instead, UBF continuously tracks insertions in a rolling time window by maintaining *two* Bloom filters and using them in a time-interleaved manner. UBF inserts every element into both filters, while the filters take turns in responding to *test* queries across consecutive limited time windows (i.e., *epochs*). UBF clears the filter which responds to *test* queries at the end of an epoch and redirects the *test* queries to the other filter for the next epoch. Therefore, each filter is cleared every other epoch (i.e., the filter's lifetime is two epochs). By doing so, UBF ensures no false negatives for the elements that are inserted in a rolling time window of up to two epochs.

Counting Bloom Filter (CBF). To track *the number of times* an element is inserted into the filter, another Bloom filter variant, called *counting* Bloom filters (CBF) [33], replaces the bit array with a *counter* array. Inserting an element in a CBF *increments* all of its corresponding counters. Testing an element returns the *minimum* value among all of the element's corresponding counters, which represents an *upper bound* on the number of times an element was inserted into the filter. Due to aliasing, the test result can be *larger* than the true insertion count, but it *cannot* be smaller than that because counters are *never decremented* (i.e., false positives are possible, but false negatives are not).

Combining UBF and CBF for Blacklisting. To estimate row activation rates with low area cost, RowBlocker-BL combines the ideas of UBF and CBF to form our *dual* counting Bloom filter (D-CBF). D-CBF maintains *two* CBFs in the time-interleaved manner of UBF. On every row activation, RowBlocker-BL inserts the activated row's address into both CBFs. RowBlocker-BL considers a row to be *blacklisted* when the row's activation count exceeds the blacklisting threshold (N_{BL}) in a rolling time window.

Figure 3 illustrates how RowBlocker-BL uses a D-CBF over time. RowBlocker-BL designates one of the CBFs as *active* and the other as *passive*. At any given time, only the *active* CBF responds to *test* queries. When a *clear* signal is received, D-CBF (1) clears only the active filter (e.g., CBF_A at ❸) and (2) swaps the active and passive filters (e.g., CBF_A becomes passive and CBF_B becomes active at ❸). RowBlocker-BL blacklists a row if the row's activation count in the active CBF exceeds the blacklisting threshold (N_{BL}).

D-CBF Operation Walk-Through. We walk through D-CBF operation in Figure 3 from the perspective of a DRAM row. The counters that correspond to the row in both filters (CBF_A and CBF_B) are initially zero (❶). CBF_A is the *active* filter, while CBF_B is the *passive* filter. As the row's activation count accumulates and reaches N_{BL} (❷), both CBF_A and CBF_B decide to blacklist the row. RowBlocker applies the active filter's decision (CBF_A) and blacklists the row. As the counter values do not decrease, the row remains blacklisted until the end of Epoch 1. Therefore, a minimum delay is enforced between

consecutive activations of this row between ❷ and ❸. At the end of Epoch 1 (❸), CBF_A is cleared, and CBF_B becomes the active filter. Note that CBF_B immediately blacklists the row, as the counter values corresponding to the row in CBF_B are still larger than N_{BL}. Meanwhile, assuming that the row continues to be activated, the counters in CBF_A again reach N_{BL} (❹). At the end of Epoch 2 (❺), CBF_A becomes the active filter again and immediately blacklists the row. By following this scheme, D-CBF blacklists the row as long as the row's activation count exceeds N_{BL} in an epoch. Assuming that the row's activation count does not exceed N_{BL} within Epoch 3, starting from ❻, the row is no longer blacklisted. Time-interleaving across the two CBFs ensures that BlockHammer maintains a *fresh* blacklist that never incorrectly excludes a DRAM row that needs to be blacklisted. Section 5 provides a generalized analytical proof of BlockHammer's security guarantees that comprehensively studies all possible row activation patterns across all epochs.

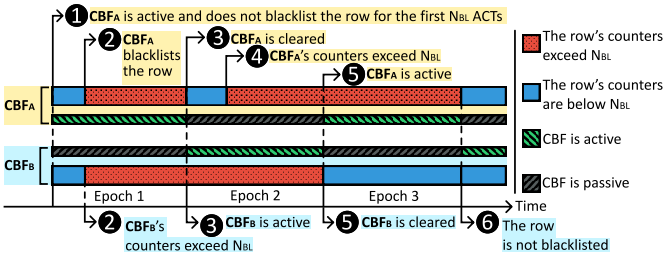

Figure 3: D-CBF operation from a DRAM row's perspective.

To prevent any specific row from being repeatedly blacklisted due to its CBF counters aliasing with those of an aggressor row (i.e., due to a false positive), RowBlocker-BL alters the hash functions that each CBF uses whenever the CBF is cleared. To achieve this, RowBlocker-BL replaces the hash function's seed value with a new randomly-generated value, as we explain next. Consequently, an aggressor row aliases with a different set of rows after every *clear* operation.

Implementing Counting Bloom Filters. To periodically send a *clear* signal to D-CBF, RowBlocker-BL implements a clock register that stores the timestamp of the latest *clear* operation. In our implementation, each CBF contains 1024 elements of 12-bit saturating counters to count up to the blacklisting threshold N_{BL}. We employ four area- and latency-efficient H3-class hash functions that consist of simple static bit-shift and mask operations [17]. We hardwire the static shift operation, so it does not require any logic gates. The mask operation performs a bitwise exclusive-OR on the shifted element (i.e., row address) and a seed. To alter the hash function when a CBF is cleared, RowBlocker simply replaces the hash function's seed value with a randomly-generated value.

3.1.2. RowBlocker-HB Mechanism. RowBlocker-HB's goal is to ensure that a blacklisted row cannot be activated often enough to cause a bit-flip. To ensure this, RowBlocker-HB delays a subsequent activation to a blacklisted row until the row's last activation becomes older than a certain amount of time that we call t_{Delay}. To do so, RowBlocker-HB maintains a first-in-first-out history buffer that stores a record of all row activations in the last t_{Delay} time window. When RowBlocker queries RowBlocker-HB with a row address (i.e., ❸ in Figure 2), RowBlocker-HB searches the row address in the history buffer and sets the *"Recently Activated?"* signal to true if the row address appears in the history buffer.

Implementing RowBlocker-HB. We implement a per-DRAM-rank history buffer as a circular queue using a head and a tail pointer. Each entry of this buffer stores (1) a row ID (which is unique in the rank), (2) a timestamp of when the entry was inserted into the buffer, and (3) a valid bit. The head and the tail pointers address the oldest and the youngest entries in the history buffer, respectively. When the memory request scheduler issues a row activation (❼ in Figure 2), RowBlocker-HB inserts a new entry with the activated row address, the current timestamp, and a valid bit set to logic '1' into the history buffer and updates the tail pointer. RowBlocker-HB checks the timestamp of the oldest entry, indicated by the head pointer, every cycle. When the oldest entry becomes as old as t_{Delay}, RowBlocker-HB invalidates the entry by resetting its valid bit to logic '0' and updates the head pointer. To test whether a row is recently activated (❸ in Figure 2), RowBlocker-HB looks up the tested row address in each *valid* entry (i.e., an entry with a valid bit set to one) in parallel. To search the history buffer with low latency, we keep row addresses in a content-addressable memory array. Any matching *valid* entry means that the row has been activated within the last t_{Delay} time window, so the new activation should not be issued if the row is blacklisted by RowBlocker-BL. We size the history buffer to be large enough to contain the worst-case number of row activations that need to be tested. The number of activations that can be performed in a DRAM rank is bounded by the timing parameter t_{FAW} [53–55, 95], which defines a rolling time window that can contain at most four row activations. Therefore, within a t_{Delay} time window, there can be at most $\lceil 4 \times t_{Delay}/t_{FAW} \rceil$ row activations.

Determining How Long to Delay an Unsafe Activation. To avoid RowHammer bit-flips, a row's activation count should not exceed the RowHammer threshold (N_{RH}) within a refresh window (t_{REFW}). RowBlocker satisfies this upper bound activation rate within each CBF's lifetime (t_{CBF}), which is the time window between two *clear* operations applied to a CBF (e.g., Epochs 1 and 2 for CBF_B and Epochs 2 and 3 for CBF_A in Figure 3). To ensure an upper bound activation rate of N_{RH}/t_{REFW} at all times, RowBlocker does not allow a row to be activated more than $(t_{CBF}/t_{REFW}) \times N_{RH}$ times within a t_{CBF} time window. In the worst-case access pattern within a CBF's lifetime, a row is activated N_{BL} times at the very beginning of the t_{CBF} time window as rapidly as possible, taking a total time of $N_{BL} \times t_{RC}$. In this case, RowBlocker evenly distributes the activations that it can allow (i.e., $(t_{CBF}/t_{REFW}) \times N_{RH} - N_{BL}$) throughout the rest of the window (i.e., $t_{CBF} - (N_{BL} \times t_{RC})$). Thus, we define t_{Delay} as shown in Equation 1.

$$t_{Delay} = \frac{t_{CBF} - (N_{BL} \times t_{RC})}{(t_{CBF}/t_{REFW}) \times N_{RH} - N_{BL}} \quad (1)$$

3.1.3. Configuration. RowBlocker has three tunable configuration parameters that collectively define RowBlocker's false positive rate and area characteristics: (1) the CBF size: the number of counters in a CBF; (2) t_{CBF}: the CBF lifetime; and (3) N_{BL}: the blacklisting threshold. Configuring the CBF size directly impacts the CBF's area and false positive rate (i.e., the fraction of mistakenly blacklisted row activations) because the CBF size determines both the CBF's physical storage requirements and the likelihood of unique row addresses aliasing to the same counters. Configuring N_{BL} and t_{CBF} determines the penalty of each false positive and the area cost of RowBlocker-HB's history buffer, because N_{BL} and t_{CBF} jointly determine the delay between activations required for RowHammer-safe operation (via Equation 1) and the maximum number of rows that RowBlocker must track within each epoch.

To determine suitable values for each of the three parameters, we follow a three-step methodology that minimizes the cost of false positives for a given area budget. First, we empirically choose the CBF size based on false positive rates observed in our experiments (Section 7 discusses our experimental configuration). We choose a CBF size of 1K counters because we observe that reducing the CBF size below 1K significantly increases the false positive rate due to aliasing.

Second, we configure N_{BL} based on three goals: (1) N_{BL} should be smaller than the RowHammer threshold to prevent RowHammer bit-flips; (2) N_{BL} should be significantly larger than the per-row activation counts that benign applications exhibit in order to ensure that RowBlocker does not blacklist benign applications' row activations, even when accounting

for false positives due to Bloom filter aliasing; and (3) N_{BL} should be as low as possible to minimize t_{Delay} (i.e., the time delay penalty for all activations to blacklisted rows, including those due to false positives) per Equation 1. To balance these three goals, we analyze the memory access patterns of 125 eight-core multiprogrammed workloads, each of which consists of eight randomly-chosen benign threads. We simulate these workloads using cycle-level simulation [77, 125] for 200M instructions with a warmup period of 100M instructions on a 3.2 GHz system with 16 MB of last-level cache. We measure per-row activation rates by counting the activations that each row experiences within a 64 ms time window (i.e., one refresh window) starting from the row's first activation. We observe that benign threads reach up to 78, 109, and 314 activations per row in a 64 ms time window for the 95th, 99th, and 100th percentile of the set of DRAM rows that are accessed at least once. Based on these observations, we set N_{BL} to 8K for a RowHammer threshold of 32K, providing (1) RowHammer-safe operation, (2) an ample margin for row activations from benign threads to achieve a low false positive rate (less than 0.01%, as shown in Section 8.3), and (3) a reasonable worst-case t_{Delay} penalty of 7.7 µs for activations to blacklisted rows.

Third, we use Equation 1 to choose a value for t_{CBF} such that the resulting t_{Delay} does not excessively penalize a mistakenly blacklisted row (i.e., a false positive). Increasing t_{CBF} both (1) decreases t_{Delay} (via Equation 1) and (2) extends the length of time for which a row is blacklisted. Therefore, we set t_{CBF} equal to t_{REFW}, which achieves as low a t_{Delay} as possible without blacklisting a row past the point at which its potential victim rows have already been refreshed.

We present the final values we choose for all BlockHammer parameters in conjunction with the DRAM timing parameters we use in Table 1 after explaining how BlockHammer addresses many-sided RowHammer attacks in Section 4.

Tuning for Different DRAM Standards. The values in Table 1 depend on three timing constraints defined by the memory standard: (1) the minimum delay between activations to the same bank (t_{RC}), (2) the refresh window (t_{REFW}), and (3) the four-activation window (t_{FAW}). The delay enforced by BlockHammer (t_{Delay}) scales linearly with t_{REFW}, while it is marginally affected by t_{RC} (Equation 1). t_{REFW} remains constant at 64 ms across DDRx standards from DDR [51] to DDR4 [55], while t_{RC} has marginally reduced from 55 ns to 46.25 ns [53–55, 95, 96]. Therefore, t_{Delay} increases only marginally across several DDR generations. In LPDDR4, t_{REFW} is halved, which allows a reduction in t_{Delay}, and thus the latency penalty of a blacklisted row. t_{FAW} affects only the size of the history buffer, and its value varies between 30–45 ns across modern DRAM standards [53–55, 95, 96].

3.2. AttackThrottler

AttackThrottler's goal is to mitigate the system-wide performance degradation that a RowHammer attack could inflict upon benign applications. AttackThrottler achieves this by using memory access patterns to (1) identify and (2) throttle threads that potentially induce a RowHammer attack. First, to identify potential RowHammer attack threads, AttackThrottler exploits the fact that a RowHammer attack thread inherently attempts to issue more activations to a blacklisted row than a benign application would. Thus, AttackThrottler tracks the exact number of times each thread performs a row activation to a blacklisted row in each bank. Second, AttackThrottler applies a quota to the total number of in-flight memory requests allowed for *any* thread that is identified to be a potential attacker (i.e., that frequently activates blacklisted rows). Because such a thread activates blacklisted rows more often, AttackThrottler reduces the thread's quota, reducing its memory bandwidth utilization. Doing so frees up memory resources for concurrently-running benign applications that are *not* repeatedly activating (i.e., hammering) blacklisted rows.

3.2.1. Identifying Ongoing RowHammer Attacks. AttackThrottler identifies threads that exhibit memory access patterns similar to a RowHammer attack by monitoring a new metric called the *RowHammer likelihood index* (*RHLI*), which quantifies the similarity between a given thread's memory access pattern and a real RowHammer attack. AttackThrottler calculates *RHLI* for each <thread, DRAM bank> pair. *RHLI* is defined as the number of blacklisted row activations the thread performs to the DRAM bank, normalized to the maximum number of times a blacklisted row can be activated in a BlockHammer-protected system. As we describe in Section 3.1, a row's activation count during one CBF lifetime is bounded by the RowHammer threshold, scaled to a CBF's lifetime (i.e., $N_{RH} \times (t_{CBF}/t_{REFW})$). Therefore, a blacklisted row that has already been activated N_{BL} times cannot be activated more than $N_{RH} \times (t_{CBF}/t_{REFW}) - N_{BL}$ times. Thus, AttackThrottler calculates *RHLI* as shown in Equation 2, during a CBF's lifetime.

$$RHLI = \frac{\textit{Blacklisted Row Activation Count}}{N_{RH} \times (t_{CBF}/t_{REFW}) - N_{BL}} \quad (2)$$

The *RHLI* of a <thread, bank> pair is 0 when a thread certainly does *not* perform a RowHammer attack on the bank. As a <thread, bank> pair's *RHLI* reaches 1, the thread is more likely to induce RowHammer bit-flips in the bank.

RHLI never exceeds 1 in a BlockHammer-protected system because AttackThrottler completely blocks a thread's memory accesses to a bank (i.e., applies a quota of zero to them) when the <thread, bank> pair's *RHLI* reaches 1, as we describe in Section 3.2.2. *RHLI* can be used independently from BlockHammer as a metric quantifying a thread's potential to be a RowHammer attack, as we discuss in Section 3.2.3.

To demonstrate example *RHLI* values, we conduct cycle-level simulations on a set of 125 multiprogrammed workloads, each of which consists of one RowHammer attack thread and seven benign threads randomly-selected from the set of workloads we describe in Section 7. We measure the *RHLI* values of benign threads and RowHammer attacks for BlockHammer's two modes: (1) *observe-only* and (2) *full-functional*. In *observe-only* mode, BlockHammer computes *RHLI* but does not interfere with memory requests. In this mode, only RowBlocker's blacklisting logic (RowBlocker-BL) and AttackThrottler's counters are functional, allowing BlockHammer to blacklist row addresses and measure *RHLI* per thread without blocking any row activations. In *full-functional* mode, BlockHammer operates normally, i.e., it detects the threads performing RowHammer attacks, throttles their requests, and ensures that no row's activation rate exceeds the RowHammer threshold. We set the blacklisting threshold to 512 activations in a 16 ms time window. We make two observations from these experiments. First, benign applications exhibit zero *RHLI* because their row activation counts never exceed the blacklisting threshold. On the other hand, RowHammer attacks reach an average (maximum, minimum) *RHLI* value of 10.9 (15.5, 6.9) in observe-only mode, showing that an *RHLI* greater than 1 reliably distinguishes a RowHammer attack thread. Second, when in full-functional mode, BlockHammer reduces an attack's *RHLI* by 54x on average, effectively reducing the *RHLI* of all RowHammer attacks to below 1. BlockHammer does not affect benign applications' *RHLI* values, which stay at zero.

AttackThrottler calculates *RHLI* separately for each <thread, bank> pair. To do so, AttackThrottler maintains two counters per <thread, bank> pair, using the same time-interleaving mechanism as the dual counting Bloom filters (D-CBFs) in RowBlocker (see Section 3.1.1). At any given time, one of the counters is designated as the active counter, while the other is designated as the passive counter. Both counters are incremented when the thread activates a blacklisted row in the bank. Only the active counter is used to calculate *RHLI* at any point in time. When RowBlocker clears its active filter for a given bank, AttackThrottler clears each thread's active counter corresponding to the bank and swaps the active and passive counters.

We implement AttackThrottler's counters as saturating counters because *RHLI* never exceeds 1 in a BlockHammer-protected system. Therefore, an AttackThrottler counter saturates at the RowHammer threshold normalized to a CBF's lifetime, which we calculate as $N_{RH} \times (t_{CBF}/t_{REFW})$. For the configuration we provide in Table 1, AttackThrottler's counters require only four bytes of additional storage in the memory controller for each <thread, bank> pair (e.g., 512 bytes in total for an eight-thread system with a 16-bank DRAM rank).

3.2.2. Throttling RowHammer Attack Threads. AttackThrottler throttles any thread with a non-zero *RHLI*. To do so, AttackThrottler limits the in-flight request count of each <thread, bank> pair by applying a quota inversely proportional to the <thread, bank> pair's *RHLI*. Whenever a thread reaches its quota, the thread is *not* allowed to make a new memory request to the shared caches or directly to the main memory until one of its in-flight requests is completed. If the thread continues to activate blacklisted rows in a bank, its *RHLI* increases and consequently its quota decreases. This slows down the RowHammer attack thread while freeing up additional memory bandwidth for concurrently-running benign threads that experience no throttling due to their zero *RHLI*. In this way, BlockHammer mitigates the performance overhead that a RowHammer attack could inflict upon benign applications.

3.2.3. Exposing RHLI to the System Software. Although BlockHammer operates independently from the system software, e.g., the operating system (OS), BlockHammer can optionally expose its per-DRAM-bank, per-thread *RHLI* values to the OS. The OS can then use this information to mitigate an ongoing RowHammer attack at the software level. For example, the OS might kill or deschedule an attacking thread to prevent it from negatively impacting the system's performance and energy. We leave the study of OS-level mechanisms using *RHLI* for future work.

4. Many-Sided RowHammer Attacks

Hammering an aggressor row can disturb physically nearby rows even if they are not immediately adjacent [72, 73], allowing *many-sided* attacks that hammer *multiple* DRAM rows to induce RowHammer bit-flips as a result of their cumulative disturbance [35]. Kim et al. [73] report that an aggressor row's impact decreases based on its physical distance to the victim row (e.g., by an order of magnitude per row) and disappears after a certain distance (e.g., 6 rows [35, 72, 73]).

To address many-sided RowHammer attacks, we conservatively add up the effect of each row to reduce BlockHammer's RowHammer threshold (N_{RH}), such that the cumulative effect of concurrently hammering each row $N_{RH}{}^*$ times becomes equivalent to hammering only an immediately-adjacent row N_{RH} times. We calculate $N_{RH}{}^*$ using three parameters: (1) N_{RH}: the RowHammer threshold for hammering a single row; (2) blast radius (r_{blast}): the maximum physical distance (in terms of rows) from the aggressor row at which RowHammer bit-flips can be observed; and (3) blast impact factor (c_k): the ratio between the activation counts required to induce a bit-flip in a victim row by hammering (*i*) an immediately-adjacent row and (*ii*) a row at a distance of k rows away. We calculate the disturbance that hammering a row N times causes for a victim row that is physically located k rows away as: $N \times c_k$. Equation 3 shows how we calculate $N_{RH}{}^*$ in terms of N_{RH}, c_k, and r_{blast}. We set $N_{RH}{}^*$ such that, even when all rows within the blast radius of a victim row (i.e., r_{blast} rows on both sides of the victim row) are hammered for $N_{RH}{}^*$ times, their cumulative disturbance (i.e., $2 \times (N_{RH}{}^* \times c_1 + N_{RH}{}^* \times c_2 + ... + N_{RH}{}^* \times c_{r_{blast}})$) on the victim row will not exceed the disturbance of hammering an immediately-adjacent row N_{RH} times.

$$N_{RH}{}^* = \frac{N_{RH}}{2\sum_1^{r_{blast}} c_k}, \quad \text{where} \begin{cases} c_k = 1, & \text{if } k = 1 \\ 0 < c_k < 1, & \text{if } r_{blast} \geq k > 1 \\ c_k = 0, & \text{if } k > r_{blast} \end{cases} \tag{3}$$

$r_{blast} = 6$ and $c_k = 0.5^{k-1}$ are the worst-case values observed in modern DRAM chips based on experimental results presented in prior characterization studies [72, 73], which characterize more than 1500 real DRAM chips from different vendors, standards, and generations from 2010 to 2020. To support a DRAM chip with these worst-case characteristics, we find that $N_{RH}{}^*$ should equal $0.2539 \times N_{RH}$ using Equation 3. Similarly, to configure BlockHammer for double-sided attacks (which is the attack model that state-of-the-art RowHammer mitigation mechanisms address [73, 84, 113, 132, 137, 161]), we calculate $N_{RH}{}^*$ as half of N_{RH} (i.e., $r_{blast} = c_k = 1$). Table 1 presents BlockHammer's configuration for timing specifications of a commodity DDR4 DRAM chip [55] and a realistic RowHammer threshold of 32K [72], tuned to address double-sided attacks.

Component	Parameters					
DRAM Features	N_{RH}	: 32K	Banks	: 16	t_{RC}	: 46.25 ns
	$N_{RH}{}^*$	: 16K	t_{REFW}	: 64 ms	t_{FAW}	: 35 ns
RowBlocker-BL	N_{BL}	: 8K	t_{CBF}	: 64 ms	t_{Delay}[1]	: 7.7 µs
	CBF size		: 1K counters per CBF			(per-bank)
	CBF Hashing		: 4 H3-class functions [17] per CBF			
RowBlocker-HB	Hist. buffer size		: 887 entries per rank (16 banks)			
AttackThrottler	2 counters per <thread, bank> pair					

Table 1: Example BlockHammer parameter values based on DDR4 specifications [55] and RowHammer vulnerability [72].

5. Security Analysis

We use the *proof by contradiction* method to prove that no RowHammer attack can defeat BlockHammer (i.e., activate a DRAM row more than N_{RH} times in a refresh window). To do so, we begin with the assumption that there exists an access pattern that can exceed N_{RH} by defeating BlockHammer. Then, we mathematically represent all possible distributions of row activations and define the constraints for activating a row more than N_{RH} times in a refresh window. Finally, we show that it is impossible to satisfy these constraints, and thus, no such access pattern that can defeat BlockHammer exists. Due to space constraints, we briefly summarize all steps of the proof. We provide the complete proof in an extended version [157].

Threat Model. We assume a comprehensive threat model in which the attacker can (1) fully utilize memory bandwidth, (2) precisely time each memory request, and (3) comprehensively and accurately know details of the memory controller, BlockHammer, and DRAM implementation. In addressing this threat model, we do not consider any hardware or software component to be *trusted* or *safe* except for the memory controller, the DRAM chip, and the physical interface between those two.

Crafting an Attack. We model a generalized memory access pattern that a RowHammer attack can exhibit from the perspective of an aggressor row. We represent an attack's row activation pattern in a series of epochs, each of which is bounded by RowBlocker's D-CBF *clear* commands to either CBF (i.e., half of the CBF lifetime or $t_{CBF}/2$), as shown in Figure 3. According to the time-interleaving mechanism (explained in Section 3.1.1), the active CBF blacklists a row based on the row's total activation count in the current and previous epochs to limit the number of activations to the row. To demonstrate that RowBlocker effectively limits the number of activations to a row, and therefore prevents all possible RowHammer attacks, we model all possible activation patterns targeting a DRAM row at the granularity of a single epoch. From the perspective of a CBF, each epoch can be classified based on the number of activations that the aggressor can receive in the previous (N_{ep-1}) and current (N_{ep}) epochs. We identify five possible epoch types (i.e., $T_0 - T_4$), which we list in Table 2. The table shows (1) the range of row activation counts in the previous epoch (N_{ep-1}), (2) the range of row activation counts in the current epoch (N_{ep}), and (3) the maximum possible row activation count in the current epoch (N_{epmax}).

[1] This is the theoretical maximum delay that a row activation can experience. Benign workloads actually experience smaller delays of up to 1.7 µs, 3.9 µs, and 7.6 µs for P50, P90, and P100 of the row activations (see Section 8.4).

Epoch Type	N_{ep-1}	N_{ep}	N_{epmax}
T_0		$N_{ep} < {N_{BL}}^*$	${N_{BL}}^* - 1$
T_1	$< N_{BL}$	${N_{BL}}^* \leq N_{ep} < N_{BL}$	$N_{BL} - 1$
T_2		$N_{ep} \geq N_{BL}$	$t_{ep}/t_{Delay} - (1 - t_{RC}/t_{Delay}){N_{BL}}^*$
T_3	$\geq N_{BL}$	$N_{ep} < N_{BL}$	$N_{BL} - 1$
T_4		$N_{ep} \geq N_{BL}$	t_{ep}/t_{Delay}

Table 2: Five possible epoch types that span all possible memory access patterns, defined by the number of row activations the aggressor row can receive in the previous epoch (N_{ep-1}) and in the current epoch (N_{ep}). N_{epmax} shows the maximum value of N_{ep}.

The epoch type indicates the recent activation rate of the aggressor row, and RowBlocker uses this information to determine whether or not to blacklist the aggressor row in the current and next epochs. A T_0 epoch indicates that the row was activated fewer than N_{BL} times in the previous epoch (i.e., $N_{ep-1} < N_{BL}$) and fewer than $N_{BL} - N_{ep-1}$ times (denoted as ${N_{BL}}^*$ for simplicity) in the current epoch. Since the row was activated fewer times than the blacklisting threshold, the row is not blacklisted in the current epoch. Compared to T_0, a T_1 epoch indicates that the row was activated greater than ${N_{BL}}^*$ times but fewer than N_{BL} times in the current epoch. Since the activation count exceeds the threshold ${N_{BL}}^*$ but not N_{BL}, the row is blacklisted in the current epoch. When a T_1 type epoch finishes, the row starts the next epoch as *not blacklisted* because the row's activation count is lower than N_{BL}. Compared to T_1, a T_2 epoch indicates that the row's activation count in the current epoch exceeds N_{BL}. Since the activation count exceeds the blacklisting threshold N_{BL}, the row is blacklisted in *both* the current and next epochs.

A T_3 epoch indicates that the row's activation count in the previous epoch exceeded N_{BL} and the row is activated fewer times than N_{BL} times in the current epoch. In this case, the row is blacklisted in the current epoch, but no longer blacklisted in the beginning of the next epoch. Compared to T_3, a T_4 epoch indicates that the row is activated more than N_{BL} times in the current epoch. The row is blacklisted in both current and next epochs, as its activation rate is too high and could lead to a successful RowHammer attack if not blacklisted.

We calculate the upper bound for the total activation count an attacker can reach during the current epoch (shown under $N_{ep_{max}}$ in Table 2). In the T_0, T_1, or T_3 epochs, by definition, a row's activation count cannot exceed ${N_{BL}}^* - 1$, $N_{BL} - 1$, and $N_{BL} - 1$, respectively. In a T_4 epoch, the row is already blacklisted from the beginning ($N_0 \geq N_{BL}$). Therefore, the row can be activated at most once in every t_{Delay} time window, resulting in an upper bound activation count of t_{ep}/t_{Delay}. In a T_2 epoch, a row can be activated ${N_{BL}}^*$ times at a time interval as small as t_{RC}, which takes $t_1 = {N_{BL}}^* \times t_{RC}$ time. Then, the row is blacklisted and further activations are performed with a minimum interval of t_{Delay}, which takes $t_2 = (N_{ep_{max}} - {N_{BL}}^*) \times t_{Delay}$ time. Since all of these activations need to fit into the epoch's time window, we solve the equation $t_{ep} = t_1 + t_2$ for N_{ep}, and derive $N_{ep_{max}}$ for an epoch of type T_2 as shown in Table 2.

Constraints of a Successful RowHammer Attack. We mathematically represent a hypothetically successful RowHammer attack as a permutation of many epochs. We denote the number of instances for an epoch type i as n_i and the maximum activation count the epoch i can reach as $N_{ep_{max}}(i)$. To be successful, the RowHammer attack must satisfy three constraints, which we present in Table 3. (1) The attacker should activate an aggressor row more than N_{RH} times within a refresh window (t_{REFW}). (2) Each epoch type can be preceded only by a subset of epoch types.[2] Therefore, an epoch type T_x cannot occur more times than the total number of instances of all epoch types that can precede epoch type T_x. (3) An epoch cannot occur for a negative number of times.

(1)	$N_{RH} \leq \sum(n_i \times N_{ep_{max}})$,	$t_{REFW} \geq t_{ep} \times \sum n_i$
(2)	$n_{0,1,2} \leq n_0 + n_1 + n_3$;	$n_{3,4} \leq n_2 + n_4$;
(3)	$\forall n_i \geq 0$	

Table 3: Necessary constraints of a successful attack.

We use an analytical solver [154] to identify a set of n_i values that meets all constraints in Table 3 for the BlockHammer configuration we provide in Table 1. We find that there exists no combination of n_i values that satisfy these constraints. Therefore, we conclude that no access pattern exists that can activate an aggressor row more than N_{RH} times within a refresh window in a BlockHammer-protected system.

6. Hardware Complexity Analysis

We evaluate BlockHammer's (1) chip area, static power, and access energy consumption using CACTI [99] and (2) circuit latency using Synopsys DC [143]. We demonstrate that BlockHammer's physical costs are competitive with state-of-the-art RowHammer mitigation mechanisms.

6.1. Area, Static Power, and Access Energy

Table 4 shows an area, static power, and access energy cost analysis of BlockHammer alongside six state-of-the-art RowHammer mitigation mechanisms [73, 84, 113, 132, 137, 161], one of which is concurrent work with BlockHammer (Graphene [113]). We perform this analysis at two RowHammer thresholds (N_{RH}): 32K and 1K.[3]

Main Components of BlockHammer. BlockHammer combines two mechanisms: RowBlocker and AttackThrottler. RowBlocker, as shown in Figure 2, consists of two components (1) RowBlocker-BL, which implements a dual counting Bloom filter for each DRAM bank, and (2) RowBlocker-HB, which implements a row activation history buffer for each DRAM rank. When configured to handle a RowHammer threshold (N_{RH}) of 32K, as shown in Table 1, each counting Bloom filter has 1024 13-bit counters, stored in an SRAM array. These counters are indexed by four H3-class hash functions [17], which introduce negligible area overhead (discussed in Section 3.1.1). RowBlocker-HB's history buffer holds 887 entries per DRAM rank. Each entry contains 32 bits for a row ID, a timestamp, and a valid bit. AttackThrottler uses two counters per thread per DRAM bank to measure the *RHLI* of each <thread, bank> pair. We estimate BlockHammer's overall area overhead as 0.14 mm^2 per DRAM rank, for a 16-bank DDR4 memory. For a high-end 28-core Intel Xeon processor system with four memory channels and single-rank DDR4 DIMMs, BlockHammer consumes approximately 0.55 mm^2, which translates to only 0.06% of the CPU die area [152]. When configured for an N_{RH} of 1K, we reduce BlockHammer's blacklisting threshold (N_{BL}) from 8K to 512, reducing the CBF counter width from 13 bits to 9 bits. To avoid false positives at the reduced blacklisting threshold, we increase the CBF size to 8K. With this modification, BlockHammer's D-CBF consumes 0.74 mm^2. Reducing N_{RH} mandates larger time delays between subsequent row activations targeting a blacklisted row, thereby increasing the history buffer's size from 887 to 27.8K entries, which translates to 0.83 mm^2 chip area. Therefore, BlockHammer's total area overhead at an N_{RH} of 1K is 1.57 mm^2 or 0.64% of the CPU die area [152].

Area Comparison. Graphene, TWiCe, and CBT need to store 5.22 kB, 37.12 kB, and 24.50 kB of metadata in the memory controller per DRAM rank, for the same 16-bank DDR4 memory, which translates to similarly low area overheads of 0.02%, 0.06%, and 0.08% of the CPU die area, respectively. Graphene's area overhead per byte of metadata is larger than other mechanisms because Graphene is fully implemented with CAM logic, as shown in Table 4. PARA, PRoHIT, and MRLoc are extremely area efficient compared to other mechanisms because they are probabilistic mechanisms [73, 137, 161], and thus do not need to store kilobytes of metadata to track row activation rates.

[2] Since we define epoch types based on activation counts in both the previous and current epochs, we note that consecutive epochs are dependent and therefore limited: an epoch of type T_0, T_1, or T_2 can be preceded only by an epoch of type T_0, T_1, or T_3, while an epoch of type T_3 or T_4 can be preceded only by an epoch of type T_2 or T_4.

[3] We configure each mechanism as we describe in Section 7.

Mitigation Mechanism	N_{RH}=32K*						N_{RH}=1K					
	SRAM KB	*CAM* KB	*Area* mm²	*Area* % CPU	Access Energy (pJ)	Static Power (mW)	*SRAM* KB	*CAM* KB	*Area* mm²	*Area* % CPU	Access Energy (pJ)	Static Power (mW)
BlockHammer	**51.48**	**1.73**	**0.14**	**0.06**	**20.30**	**22.27**	**441.33**	**55.58**	**1.57**	**0.64**	**99.64**	**220.99**
Dual counting Bloom filters	48.00	-	0.11	0.04	18.11	19.81	384.00	-	0.74	0.30	86.29	158.46
H3 hash functions	-	-	< 0.01	< 0.01	-	-	-	-	< 0.01	< 0.01	-	-
Row activation history buffer	1.73	1.73	0.03	0.01	1.83	2.05	55.58	55.58	0.83	0.34	12.99	62.12
AttackThrottler counters	1.75	-	< 0.01	< 0.01	0.36	0.41	1.75	-	< 0.01	< 0.01	0.36	0.41
PARA [73]	**-**	**-**	**< 0.01**	**-**	**-**	**-**	**-**	**-**	**< 0.01**	**-**	**-**	**-**
ProHIT [137]*	**-**	**0.22**	**< 0.01**	**<0.01**	**3.67**	**0.14**	×	×	×	×	×	×
MrLoc [161]*	**-**	**0.47**	**< 0.01**	**<0.01**	**4.44**	**0.21**	×	×	×	×	×	×
CBT [132]	**16.00**	**8.50**	**0.20**	**0.08**	**9.13**	**35.55**	**512.00**	**272.00**	**3.95**	**1.60**	**127.93**	**535.50**
TWiCE [84]	**23.10**	**14.02**	**0.15**	**0.06**	**7.99**	**21.28**	**738.32**	**448.27**	**5.17**	**2.10**	**124.79**	**631.98**
Graphene [113]	**-**	**5.22**	**0.04**	**0.02**	**40.67**	**3.11**	**-**	**166.03**	**1.14**	**0.46**	**917.55**	**93.96**

* PRoHIT [137] and MRLoc [161] do *not* provide a concrete discussion on how to adjust their empirically-determined parameters for different N_{RH} values. Therefore, we (1) report their values for a fixed design point that each paper provides for N_{RH}=2K and (2) mark values we cannot estimate using an ×.

Table 4: Per-rank area, access energy, and static power of BlockHammer vs. state-of-the-art RowHammer mitigation mechanisms.

We repeat our area overhead analysis for future DRAM chips by scaling the RowHammer threshold down to 1K. While BlockHammer consumes 1.57 mm² of chip area to prevent bit-flips at this lower threshold, TWiCe's and CBT's area overhead increases to 3.3x and 2.5x of BlockHammer's. We conclude that BlockHammer scales better than both CBT and TWiCe in terms of area overhead. Graphene's area overhead does not scale as efficiently as BlockHammer with decreasing RowHammer threshold, and becomes comparable to BlockHammer when configured for a RowHammer threshold of 1K.

Static Power and Access Energy Comparison. When configured for an N_{RH} of 32K, BlockHammer consumes 20.30 pJ per access, which is half of Graphene's access energy; and 22.27 mW of static power, which is 63% of CBT's. BlockHammer's static power consumption scales more efficiently than that of CBT and TWiCe as N_{RH} decreases to 1K, whereas CBT and TWiCe consume 2.42x and 2.86x the static power of BlockHammer, respectively. Similarly, Graphene's access energy and static power drastically increase by 22.56x and 30.2x, respectively, when N_{RH} scales down to 1K. As a result, Graphene consumes 9.21× of BlockHammer's access energy.

6.2. Latency Analysis

We implement BlockHammer in Verilog HDL and synthesize our design using Synopsys DC [143] with a 65 nm process technology to evaluate the latency impact on memory accesses. According to our RTL model, which we open source [124], BlockHammer responds to an *"Is this ACT RowHammer-safe?"* query (❶ in Figure 2) in only 0.97 ns. This latency can be hidden because it is one-to-two orders of magnitude smaller than the row access latency (e.g., 45–50 ns) that DRAM standards (e.g., DDRx, LPDDRx, GDDRx) enforce [36, 53, 55].

7. Experimental Methodology

We evaluate BlockHammer's effect on a typical DDR4-based memory subsystem's performance and energy consumption as compared to six prior RowHammer mitigation mechanisms [73, 84, 113, 132, 137, 161]. We use Ramulator [77, 125] for performance evaluation and DRAMPower [18] to estimate DRAM energy consumption. Table 5 shows our system configuration.

Processor	3.2 GHz, {1,8} core, 4-wide issue, 128-entry instr. window
Last-Level Cache	64-byte cache line, 8-way set-associative, 16 MB
Memory Controller	64-entry each read and write request queues; Scheduling policy: FR-FCFS [122, 164]; Address mapping: MOP [60]
Main Memory	DDR4, 1 channel, 1 rank, 4 bank groups, 4 banks/bank group, 64K rows/bank

Table 5: Simulated system configuration.

Attack Model. We compare BlockHammer under the same RowHammer attack model (i.e., double-sided attacks [73]) as prior works use [73, 84, 113, 132, 137, 161]. To do so, we halve the RowHammer threshold that BlockHammer uses to account for the cumulative disturbance effect of both aggressor rows (i.e., $N_{RH}* = N_{RH}/2$). In Sections 8.1 and 8.2, we set $N_{RH}* = 16K$ (i.e., $N_{RH} = 32K$), which is the minimum RowHammer threshold that TWiCe [84] supports [72]. In Section 8.3, we conduct an N_{RH} scaling study for double-sided attacks, across a range of $32K > N_{RH} > 1K$, using parameters provided in Table 7.

Comparison Points. We compare BlockHammer to a baseline system with no RowHammer mitigation and to six state-of-the-art RowHammer mitigation mechanisms that provide RowHammer-safe operation: three are probabilistic mechanisms [73, 137, 161] and another three are deterministic mechanisms [84, 113, 132]. (1) PARA [73] mitigates RowHammer by injecting an adjacent row activation with a low probability whenever the memory controller closes a row following an activation. We tune PARA's probability threshold for a given RowHammer threshold to meet a desired failure probability (we use 10^{-15} as a typical consumer memory reliability target [15, 16, 52, 92, 116]) in a refresh window (64 ms). (2) PRoHIT [137] implements a history table of recent row activations to extend PARA by reducing the probability threshold for more frequently activated rows. We configure PRoHIT using the default probabilities and parameters provided in [137]. (3) MRLoc [161] extends PARA by keeping a record of recently-refreshed potential victim rows in a queue and dynamically adjusts the probability threshold, which it uses to decide whether or not to refresh the victim row, based on the row's temporal locality information. We implement MRLoc by using the empirically-determined parameters provided in [161]. (4) CBT [133] proposes a tree of counters to count the activations for non-uniformly-sized disjoint memory regions, each of which is halved in size (i.e., moved to the next level of the tree) every time its activation count reaches a predefined threshold. After being halved a predefined number of times (i.e., after becoming a leaf node in the tree), all rows in the memory region are refreshed. We implement CBT with a six-level tree that contains 125 counters, and exponentially increase the threshold values across tree levels from 1K to the RowHammer threshold (N_{RH}), as described in [132]. (5) TWiCe uses a table of counters to track the activation count of every row. Aiming for an area-efficient implementation, TWiCe periodically prunes the activation records of the rows whose activation counts cannot reach a high enough value to cause bit-flips. We implement and configure TWiCe for a RowHammer threshold of 32K using the methodology described in the original paper [84]. Unfortunately, TWiCe faces scalability challenges due to time consuming pruning operations, as described in [72]. To scale TWiCe for smaller RowHammer thresholds, we follow the same methodology as Kim et al. [72]. (6) Graphene [113] adopts Misra-Gries, a frequent-element detection algorithm [97], to detect the most frequently activated rows in a given time window. Graphene maintains a set of counters where it keeps the address and activation count of frequently activated rows. Whenever a row's counter reaches a multiple of a predefined threshold value, Graphene refreshes its adjacent rows. We configure Graphene by evaluating the equations provided in the original work [113] for a given RowHammer threshold.

Workloads. We evaluate BlockHammer and state-of-the-art RowHammer mitigation mechanisms with 280 (30 single-core and 250 multiprogrammed) workloads. We use 22 memory-intensive benign applications from the SPEC CPU2006 benchmark suite [138], four disk I/O applications from the YCSB benchmark suite [26], two network I/O applications from a commercial network chip [108], and two synthetic microbenchmarks that mimic non-temporal data copy. We categorize these benign applications based on their row buffer conflicts per kilo instruction (*RBCPKI*) into three categories: *L* (*RBCPKI* < 1), *M* (1 < *RBCPKI* < 5), and *H* (*RBCPKI* > 5). *RBCPKI* is an indi-

cator of row activation rate, which is the key workload property that triggers RowHammer mitigation mechanisms. There are 12, 9, and 9 applications in the *L*, *M*, and *H* categories, respectively, as listed in Table 8. To mimic a double-sided RowHammer attack, we use a synthetic trace that activates two rows in each bank as frequently as possible by alternating between them at every row activation (i.e., R_A, R_B, R_A, R_B, ...).

We randomly combine these single-core workloads to create two types of multiprogrammed workloads: (1) 125 workloads with *no RowHammer attack*, each including eight benign threads; and (2) 125 workloads with a *RowHammer attack present*, each including one RowHammer attack and seven benign threads. We simulate each multiprogrammed workload until each benign thread executes at least 200 million instructions. For all configurations, we warm up the caches by fast-forwarding 100 million instructions, as done in prior work [72].

Performance and DRAM Energy Metrics. We evaluate BlockHammer's impact on *system throughput* (in terms of weighted speedup [32, 94, 136]), *job turnaround time* (in terms of harmonic speedup [32, 91]), and *fairness* (in terms of maximum slowdown [27–30, 74, 75, 105, 139–142]). Because the performance of a RowHammer attack should not be accounted for in the performance evaluation, we calculate all three metrics only for benign applications. To evaluate DRAM energy consumption, we compare the total energy consumption that DRAMPower provides in Joules. DRAM energy consumption includes both benign and RowHammer attack requests. Each data point shows the average value across all workloads, with minimum and maximum values depicted using error bars.

8. Performance and Energy Evaluation

We evaluate the performance and energy overheads of BlockHammer and six state-of-the-art RowHammer mitigation mechanisms. First, we evaluate all mechanisms with single-core applications and show that BlockHammer exhibits no performance and energy overheads, compared to a baseline system without any RowHammer mitigation. Second, we evaluate BlockHammer with multiprogrammed workloads and show that, by throttling an attack's requests, BlockHammer significantly improves the performance of benign applications by 45.4% on average (with a maximum of 61.9%), compared to both the baseline system and a system with the prior best-performing state-of-the-art RowHammer mitigation mechanism. Third, we compare BlockHammer with state-of-the-art RowHammer mitigation mechanisms when applied to future DRAM chips that are projected to be more vulnerable to RowHammer. We show that BlockHammer is competitive with state-of-the-art mechanisms at RowHammer thresholds as low as 1K when there is no attack in the system, and provides significantly higher performance and lower DRAM energy consumption than state-of-the-art mechanisms when a RowHammer attack is present. Fourth, we provide an analysis of BlockHammer's internal mechanisms.

8.1. Single-Core Applications

Figure 4 presents the execution time and energy of benign applications (grouped into three categories based on their *RBCPKI*; see Section 7) when executed on a single-core system that uses BlockHammer versus six state-of-the-art mitigation mechanisms, normalized to a baseline system that does not employ any RowHammer mitigation mechanism.

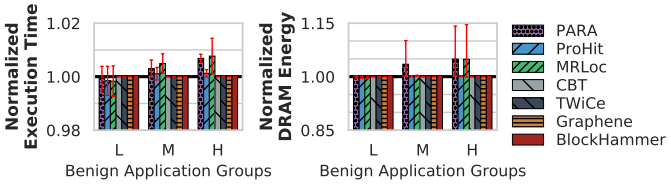

Figure 4: Execution time and DRAM energy consumption for benign single-core applications, normalized to baseline.

We observe that BlockHammer introduces no performance and DRAM energy overheads on benign applications compared to the baseline configuration. This is because benign applications' per-row activation rates never exceed BlockHammer's blacklisting threshold (N_{BL}). In contrast, PARA/MRLoc exhibit 0.7%/0.8% performance and 4.9%/4.9% energy overheads for high *RBCPKI* applications, on average. CBT, TWiCe, and Graphene do not perform any victim row refreshes in these applications because none of the applications activate a row at a high enough rate to trigger victim row refreshes. We conclude that BlockHammer does not incur performance or DRAM energy overheads for single-core benign applications.

8.2. Multiprogrammed Workloads

Figure 5 presents the performance and DRAM energy impact of BlockHammer and six state-of-the-art mechanisms[4] on an eight-core system, normalized to the baseline. We show results for two types of workloads: (1) *No RowHammer Attack*, where all eight applications in the workload are benign; and (2) *RowHammer Attack Present*, where one of the eight applications in the workload is a malicious thread performing a RowHammer attack, running alongside seven benign applications. We make four observations from the figure.

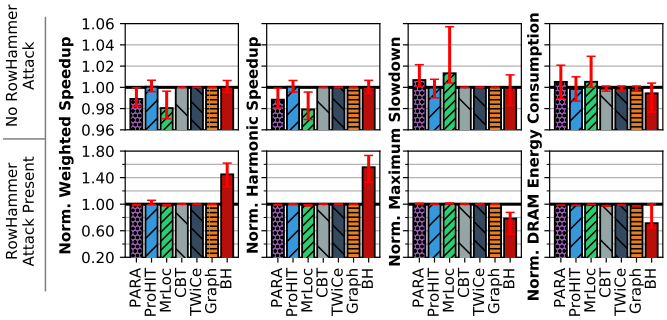

Figure 5: Performance and DRAM energy consumption for multiprogrammed workloads, normalized to baseline.

No RowHammer Attack. First, BlockHammer has a very small performance overhead for multiprogrammed workloads when there is no RowHammer attack present. BlockHammer incurs less than 0.5%, 0.6%, and 1.2% overhead in terms of weighted speedup, harmonic speedup, and maximum slowdown, respectively, compared to the baseline system with no RowHammer mitigation. In comparison, PRoHIT, CBT, TWiCe, and Graphene do not perform enough refresh operations to have an impact on system performance, while PARA and MRLoc incur 1.2% and 2.0% performance (i.e., weighted speedup) overheads on average, respectively. Second, BlockHammer *reduces* average DRAM energy consumption by 0.6%, while for the worst workload we observe, it increases energy consumption by up to 0.4%. This is because BlockHammer (1) increases the standby energy consumption by delaying requests and (2) reduces the energy consumed for row activation and precharge operations by batching delayed requests and servicing them when their target row is activated. In comparison, PRoHIT, CBT, TWiCe, and Graphene *increase* average DRAM energy consumption by less than 0.1%, while PARA and MRLoc *increase* average DRAM energy consumption by 0.5%, as a result of the unnecessary row refreshes that these mitigation mechanisms must perform.

RowHammer Attack Present. Third, unlike any other RowHammer mitigation mechanism, BlockHammer *reduces* the performance degradation inflicted on benign applications when one of the applications in the workload is a RowHammer attack. By throttling the attack, BlockHammer significantly improves the performance of benign applications, with a 45.0% (up to 61.9%) and 56.2% (up to 73.4%) increase in weighted and harmonic speedups and 22.7% (up to 45.4%) decrease in maximum slowdown on average, respectively. In contrast, PARA, PRoHIT, and MRLoc incur 1.3%, 0.1% and 1.7% performance overheads, on average, respectively, while the average performance overheads of CBT, TWiCe, and Graphene are all less than 0.1%. Fourth, BlockHammer *reduces* DRAM energy consumption by 28.9%

[4]We label Graphene as "Graph" and BlockHammer as "BH" for brevity.

on average (up to 33.8%). In contrast, all other state-of-the-art mechanisms *increase* DRAM energy consumption (by up to 0.4%). BlockHammer significantly improves performance and DRAM energy because it increases the row buffer locality that benign applications experience by throttling the attacker (the row buffer hit rate increases by 177% on average, and 23% of row buffer conflicts are converted to row buffer misses).

We conclude that BlockHammer (1) introduces very low performance and DRAM energy overheads for workloads with no RowHammer attack present and (2) significantly improves benign application performance and DRAM energy consumption when a RowHammer attack is present.

8.3. Effect of Worsening RowHammer Vulnerability

We analyze how BlockHammer's impact on performance and DRAM energy consumption scales as DRAM chips become increasingly vulnerable to RowHammer (i.e., as the RowHammer threshold, N_{RH}, decreases). We compare BlockHammer with three state-of-the-art RowHammer mitigation mechanisms, which are shown to be the most viable mechanisms when the RowHammer threshold decreases [72, 113]: PARA [73], TWiCe [84],[5] and Graphene [113]. We analyze the scalability of these mechanisms down to N_{RH}= 1024, which is approximately an order of magnitude smaller than the minimum observed N_{RH} reported in current literature (i.e., 9600) [72]. Figure 6 shows the performance and energy overheads of each mechanism for our multiprogrammed workloads as N_{RH} decreases, normalized to the baseline system with no RowHammer mitigation. We make two observations from Figure 6.

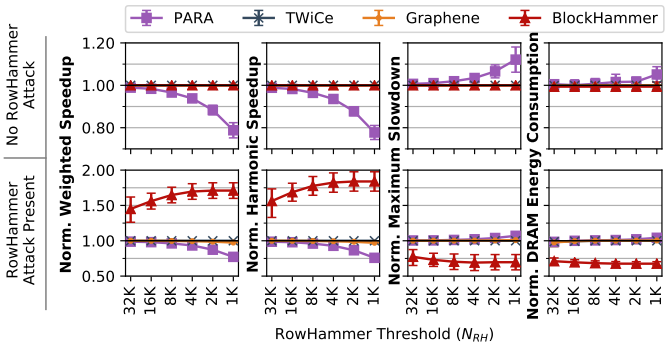

Figure 6: Performance and DRAM Energy for different N_{RH} values, normalized to baseline (N_{RH} *decreases* along the x-axis).

No RowHammer Attack. First, BlockHammer's performance and DRAM energy consumption are better than PARA and competitive with other mechanisms as N_{RH} decreases. When N_{RH}=1024, the average performance and DRAM energy overheads of BlockHammer, Graphene, and TWiCe are less than 0.6% because they do not act aggressively enough to cause significant performance or energy overheads. On the other hand, PARA performs reactive refreshes more aggressively with increasing RowHammer vulnerability, which leads to a performance overhead of 21.2% and 22.3% (weighted and harmonic speedup) and an energy overhead of 5.1% on average.

RowHammer Attack Present. Second, BlockHammer's performance and DRAM energy benefits increase as N_{RH} decreases. At N_{RH}=1024, BlockHammer more aggressively throttles a RowHammer attack and mitigates the performance degradation of benign applications. As a result, compared to the baseline, BlockHammer improves average performance by 71.0% and 83.9% (weighted and harmonic speedups) while reducing the maximum slowdown and DRAM energy consumption by 30.4% and 32.4%, respectively. In contrast, the additional refresh operations that Graphene and TWiCe perform cause 2.9% and 0.9% average performance degradation and 0.4% and 0.2% DRAM energy increase for benign applications, respectively. BlockHammer is the only RowHammer mitigation mechanism that improves performance and energy when a RowHammer attack is present in the system.

We conclude that (1) BlockHammer's performance and energy overheads remain negligible at reduced RowHammer thresholds as low as N_{RH}=1K when there is no RowHammer attack, and (2) BlockHammer scalably provides much higher performance and lower energy consumption than all state-of-the-art mechanisms when a RowHammer attack is present.

[5] As described in Section 7, TWiCe faces latency issues, preventing it from scaling when $N_{RH} < 32K$ [72]. Our scalability analysis assumes a TWiCe variation that solves this issue, the same as TWiCe-Ideal in [72].

8.4. Analysis of BlockHammer Internal Mechanisms

BlockHammer's impact on performance and DRAM energy depends on (1) the false positive rate of the blacklisting mechanism and (2) the false positive penalty resulting from delaying row activations. We calculate (1) the false positive rate as the number of row activations that are mistakenly delayed by BlockHammer's Bloom filters (i.e., activations to rows that would not have been blacklisted if the filters had no aliasing) as a fraction of all activations, and (2) the false positive penalty as the additional time delay a mistakenly-delayed row activation suffers from. We find that for a configuration where N_{RH}=32K, BlockHammer's false positive rate is 0.010%, and it increases to only 0.012% when N_{RH} is scaled down to 1K. Therefore, BlockHammer successfully avoids delaying more than 99.98% of benign row activations. Even though we set t_{Delay} to 7.7 µs, we observe 1.7 µs, 3.9 µs, and 7.6 µs of delay for the 50th, 90th, and 100th percentile of mistakenly-delayed activations (which are only 0.012% of all activations).

Note that the worst-case latency we observe is at least two orders of magnitude smaller than typical quality-of-service targets, which are on the order of milliseconds [61]. Therefore, we believe that BlockHammer is unlikely to introduce quality-of-service violations with its low worst-case latency (on the order of µs) and very low false positive rate (0.012%).

9. Comparison of Mitigation Mechanisms

We qualitatively compare BlockHammer and a number of published RowHammer mitigation mechanisms, which we classify into four high-level approaches, as defined in Section 1: (*i*) *increased refresh rate*, (*ii*) *physical isolation*, (*iii*) *reactive refresh*, and (*iv*) *proactive throttling*. We evaluate RowHammer mitigation mechanisms across four dimensions: *comprehensive protection*, *compatibility with commodity DRAM chips*, *scaling with RowHammer vulnerability*, and *deterministic protection*. Table 6 summarizes our comprehensive qualitative evaluation.

Approach	Mechanism	Comprehensive Protection	Compatible w/ Commodity DRAM Chips	Scaling with RowHammer Vulnerability	Deterministic Protection
Increased Refresh Rate [2, 73]		✓	✓	✗	✓
Physical Isolation	CATT [14]	✗	✗	✗	✓
	GuardION [148]	✗	✗	✗	✓
	ZebRAM [78]	✗	✗	✗	✓
Reactive Refresh	ANVIL [5]	✗	✗	✗	✓
	PARA [73]	✓	✗	✗	✗
	PRoHIT [137]	✓	✗	✗	✗
	MRLoc [161]	✓	✗	✗	✗
	CBT [132]	✓	✗	✗	✓
	TWiCe [84]	✓	✗	✗	✓
	Graphene [113]	✓	✗	✓	✓
Proactive Throttling	Naive Thrott. [102]	✓	✓	✗	✓
	Thrott. Supp. [40]	✓	✗	✗	✓
	BlockHammer	✓	✓	✓	✓

Table 6: Comparison of RowHammer mitigation mechanisms.

1. Comprehensive Protection. A RowHammer mitigation mechanism should comprehensively prevent *all* potential RowHammer bit-flips regardless of the methods that an attacker may use to hammer a DRAM row. Unfortunately, four key RowHammer mitigation mechanisms [5, 14, 78, 148] are effective only

against a limited threat model and have already been defeated by recent attacks [25, 41, 42, 79, 118, 162] because they (1) trust system components (e.g., hypervisor) that can be used to perform a RowHammer attack [78, 148]; (2) disregard practical methods (e.g., flipping opcode bits within the attacker's memory space [14]) that can be used to gain root privileges; or (3) detect RowHammer attacks by relying on hardware performance counters (e.g., LLC miss rate [5]), which can be oblivious to several attack models [41, 118, 145, 147]. In contrast, BlockHammer comprehensively prevents RowHammer bit-flips by monitoring all memory accesses from within the memory controller, even if the entire software stack is compromised and the attacker possesses knowledge about all hardware/software implementation details (e.g., the DRAM chip's RowHammer vulnerability characteristics, BlockHammer's configuration parameters).

2. Compatibility with Commodity DRAM Chips. Especially given that recent works [24, 35, 72] experimentally observe RowHammer bit-flips on cutting-edge commodity DRAM chips, including ones that are marketed as RowHammer-free [24, 35, 72], it is important for a RowHammer mitigation mechanism to be compatible with *all* commodity DRAM chips, current and future. To achieve this, a RowHammer mitigation mechanism should *not* (1) rely on any proprietary information that DRAM vendors do not share, and (2) require any modifications to DRAM chip design. Unfortunately, both physical isolation and reactive refresh mechanisms need to be fully aware of the internal physical layout of DRAM rows or require modifications to DRAM chip design either (1) to ensure that isolated memory regions are not physically close to each other [14, 78, 148] or (2) to identify victim rows that need to be refreshed [5–8, 40, 59, 68, 73, 84, 113, 132, 133, 137, 161]. In contrast, designing BlockHammer requires knowledge of only six readily-available DRAM parameters: (1) t_{REFW}: the refresh window, (2) t_{RC}: the ACT-to-ACT latency, (3) t_{FAW}: the four-activation window, (4) N_{RH}: the RowHammer threshold, (5) the blast radius, and (6) the blast impact factor. Among these parameters, t_{REFW}, t_{RC}, and t_{FAW} are publicly available in datasheets [53–55, 95]. N_{RH}, the blast radius, and the blast impact factor can be obtained from prior characterization works [35, 72, 73]. Therefore, BlockHammer is compatible with all commodity DRAM chips because it does not need any proprietary information about or any modifications to commodity DRAM chips.

3. Scaling with Increasing RowHammer Vulnerability. Since main memory is a growing system performance and energy bottleneck [12, 39, 58, 100, 103, 107, 111, 134, 149, 153, 155], a RowHammer mitigation mechanism should exhibit acceptable performance and energy overheads at low area cost when configured for more vulnerable DRAM chips.

Increasing the refresh rate [2, 73] is *already* a prohibitively expensive solution for modern DRAM chips with a RowHammer threshold of 32K. This is because the latency of refreshing rows at a high enough rate to prevent bit-flips overwhelms DRAM's availability, increasing its average performance overhead to 78%, as shown in [72].

Physical isolation [14, 78, 148] requires reserving as many rows as twice the *blast radius* (up to 12 in modern DRAM chips [72]) to isolate sensitive data from a potential attacker's memory space. This is expensive for most modern systems where memory capacity is critical. As the blast radius has increased by 33% from 2014 [73] to 2020 [72], physical isolation mechanisms can require reserving even more rows when configured for future DRAM chips, further reducing the total amount of secure memory available to the system.

Reactive refresh mechanisms [5–8, 40, 59, 68, 73, 84, 113, 132, 133, 137, 161] generally incur increasing performance, energy, and/or area overheads at lower RowHammer thresholds when configured for more vulnerable DRAM chips. ANVIL samples hardware performance counters on the order of ms for a RowHammer threshold (N_{RH}) of 110K [5]. However, a RowHammer attack can successfully induce bit-flips in less than 50 µs when N_{RH} is reduced to 1K, which significantly increases ANVIL's sampling rate, and thus, its performance and energy overheads. PRoHIT and MRLoc [137, 161] do not provide a concrete discussion on how to adjust their empirically-determined parameters, so we cannot demonstrate how their overheads scale as DRAM chips become more vulnerable to RowHammer. TWiCe [84] faces design challenges to protect DRAM chips when reducing N_{RH} below 32K, as described in Section 7. Assuming that TWiCe overcomes its design challenges (as also assumed by prior work [72]), we scale TWiCe down to N_{RH}= 1K along with three other state-of-the-art mechanisms [73, 113, 132]. Table 4 shows that the CPU die area, access energy, and static power consumption of TWiCe [84]/CBT [132] drastically increase by 35x/20x, 15.6x/14.0x, and 29.7x/15.1x, respectively, when N_{RH} is reduced from 32K to 1K. In contrast, BlockHammer consumes only 30%/40%, 79.8%/77.8%, 35%/41.3% of TWiCe/CBT's CPU die area, access energy, and static power, respectively, when configured for N_{RH}= 1K. Section 8.3 shows that PARA's average performance and DRAM energy overheads reach 21.2% and 22.3%, respectively, when configured for N_{RH}= 1K. We observe that Graphene and BlockHammer are the two most scalable mechanisms with worsening RowHammer vulnerability. When configured for N_{RH}=1K, BlockHammer (1) consumes only 11% of Graphene's access energy (see Table 4) and (2) improves benign applications' performance by 71.0% and reduces DRAM energy consumption by 32.4% on average, while Graphene incurs 2.9% performance and 0.4% DRAM energy overheads, as shown in Section 8.3.

Naïve *proactive throttling* [40, 73, 102] either (1) blocks all activations targeting a row until the end of the refresh window once the row's activation count reaches the RowHammer threshold, or (2) statically extends each row's activation interval so that no row's activation count can ever exceed the RowHammer threshold. The first method has a high area overhead because it requires implementing a counter for each DRAM row [73, 102], while the second method prohibitively increases t_{RC} [51, 53–55] (e.g., 42.2x/1350.4x for a DRAM chip with N_{RH}=32K/1K) [73, 102]. BlockHammer is the first efficient and scalable proactive throttling-based RowHammer prevention technique.

4. Deterministic Prevention. To effectively prevent all RowHammer bit-flips, a RowHammer mitigation mechanism should be deterministic, meaning that it should ensure RowHammer-safe operation at all times because it is important to guarantee zero chance of a security failure for a critical system whose failure or malfunction may result in severe consequences (e.g., related to loss of lives, environmental damage, or economic loss) [4]. PARA [73], ProHIT [137], and MRLoc [161] are probabilistic by design, and therefore cannot reduce the probability of a successful RowHammer attack to zero like CBT [132], TWiCe [84], and Graphene [113] potentially can. BlockHammer has the capability to provide zero probability for a successful RowHammer attack by guaranteeing that no row can be activated at a RowHammer-unsafe rate.

10. Related Work

To our knowledge, BlockHammer is the first work that (1) prevents RowHammer bit-flips efficiently and scalably without requiring any proprietary knowledge of or modification to DRAM internals, (2) satisfies all four of the desired characteristics for a RowHammer mitigation mechanism (as we describe in Section 9), and (3) improves benign application performance and system energy when the system is under a RowHammer attack. Sections 6.1, 8, and 9 already qualitatively and quantitatively compare BlockHammer to the most relevant prior mechanisms, demonstrating BlockHammer's benefits. This section discusses RowHammer mitigation and memory access throttling works that are loosely related to BlockHammer.

In-DRAM Reactive Refresh. A subset of DRAM standards [53, 55] support a mode called *target row refresh* (TRR), which refreshes rows that are physically nearby an aggressor row without exposing any information about the in-DRAM row address mapping outside of DRAM chips. TRRespass [35]

demonstrates that existing proprietary implementations of TRR are not sufficient to mitigate RowHammer bit-flips: many-sided RowHammer attacks reliably induce and exploit bit-flips in state-of-the-art DRAM chips that already implement TRR.

Making Better DRAM Chips. A different approach to mitigating RowHammer is to implement architecture- and device-level techniques that make DRAM chips stronger against RowHammer. CROW [44] maps potential victim rows into dedicated *copy rows* and mitigates RowHammer bit-flips by serving requests from copy rows. Gomez et al. [38] place *dummy cells* in DRAM rows that are engineered to be more susceptible to RowHammer than regular cells, and monitor dummy cell charge levels to detect a RowHammer attack. Three other works [43, 123, 158] propose manufacturing process enhancements or implantation of additional dopants in transistors to reduce wordline crosstalk. Although these methods mitigate the RowHammer vulnerability of DRAM chips, they (1) cannot be applied to already-deployed commodity DRAM chips and (2) can be high cost due to the required extensive chip modifications.

Other Uses of Throttling. Prior works on quality-of-service- and fairness-oriented architectures propose selectively throttling main memory accesses to provide latency guarantees and/or improve fairness across applications (e.g., [3, 23, 29–31, 74, 75, 80, 98, 105, 106, 109, 110, 122, 139, 140, 146]). These mechanisms are *not* designed to prevent RowHammer attacks and thus do not interfere with a RowHammer attack when there is no contention between memory accesses. In contrast, BlockHammer's primary goal is to prevent RowHammer attacks from inducing bit-flips. As such, BlockHammer is complementary to these mechanisms, and can work together with them.

11. Conclusion

We introduce BlockHammer, a new RowHammer detection and prevention mechanism that uses area-efficient Bloom filters to track and proactively throttle memory accesses that can potentially induce RowHammer bit-flips. BlockHammer operates entirely from within the memory controller, comprehensively protecting a system from all RowHammer bit-flips at low area, energy, and performance cost. Compared to existing RowHammer mitigation mechanisms, BlockHammer is the first one that (1) prevents RowHammer bit-flips efficiently and scalably without knowledge of or modification to DRAM internals, (2) provides all four desired characteristics of a RowHammer mitigation mechanism (as we describe in Section 9), and (3) improves the performance and energy consumption of a system that is under attack. We believe that BlockHammer provides a new direction in RowHammer prevention and hope that it enables researchers and engineers to develop low-cost RowHammer-free systems going forward.

Acknowledgments

We thank the anonymous reviewers of HPCA 2020, ISCA 2020, MICRO 2020, and HPCA 2021 for feedback. We thank the SAFARI Research Group members for valuable feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: Google, Huawei, Intel, Microsoft, and VMware.

References

[1] M. T. Aga *et al.*, "When Good Protections Go Bad: Exploiting Anti-DoS Measures to Accelerate Rowhammer Attacks," in *HOST*, 2017.
[2] Apple Inc., "About the Security Content of Mac EFI Security Update 2015-001," https://support.apple.com/en-us/HT204934, June 2015.
[3] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[4] T. Aven, "Identification of Safety and Security Critical Systems and Activities," *Reliability Engineering & System Safety*, 2009.
[5] Z. B. Aweke *et al.*, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016.
[6] K. Bains *et al.*, "Row Hammer Refresh Command," U.S. Patent 9,117,544. 2015.
[7] K. S. Bains and J. B. Halbert, "Distributed Row Hammer Tracking," U.S. Patent 9,299,400. 2016.
[8] K. S. Bains and J. B. Halbert, "Row Hammer Monitoring Based on Stored Row Hammer Threshold Value," U.S. Patent 9,384,821. 2016.
[9] A. Barenghi *et al.*, "Software-Only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks," in *IVSW*, 2018.
[10] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, 2016.
[11] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *CACM*, 1970.
[12] A. Boroumand *et al.*, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[13] E. Bosman *et al.*, "Dedup Est Machina: Memory Deduplication as An Advanced Exploitation Vector," in *S&P*, 2016.
[14] F. Brasser *et al.*, "Can't Touch This: Software-Only Mitigation Against Rowhammer Attacks Targeting Kernel Memory," in *USENIX Security*, 2017.
[15] Y. Cai *et al.*, "Error Characterization, Mitigation, and Recovery in Flash Memory Based Solid-State Drives," *Proc. IEEE*, 2017.
[16] Y. Cai *et al.*, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.
[17] J. Carter and M. Wegman, "Universal Classes of Hash Functions," *JCSS*, 1979.
[18] K. Chandrasekar *et al.*, "DRAMPower: Open-Source DRAM Power & Energy Estimation Tool," http://www.drampower.info/.
[19] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[20] K. K. Chang *et al.*, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[21] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[22] K. K. Chang *et al.*, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
[23] K. K. Chang *et al.*, "HAT: Heterogeneous Adaptive Throttling for On-Chip Networks," in *SBAC-PAD*, 2012.
[24] L. Cojocar *et al.*, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *S&P*, 2020.
[25] L. Cojocar *et al.*, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, 2019.
[26] B. Cooper *et al.*, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.
[27] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
[28] R. Das *et al.*, "Application-Aware Prioritization Mechanisms for On-Chip Networks," in *MICRO*, 2009.
[29] E. Ebrahimi *et al.*, "Fairness via Source Throttling: A Configurable and High Performance Fairness Substrate for Multi Core Memory Systems," in *ASPLOS*, 2010.
[30] E. Ebrahimi *et al.*, "Prefetch-Aware Shared Resource Management for Multi-Core Systems," in *ISCA*, 2011.
[31] E. Ebrahimi *et al.*, "Parallel Application Memory Scheduling," in *MICRO*, 2011.
[32] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, 2008.
[33] L. Fan *et al.*, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *TON*, 2000.
[34] P. Frigo *et al.*, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.
[35] P. Frigo *et al.*, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.
[36] S. Ghose *et al.*, "Demystifying Complex Workload–DRAM Interactions: An Experimental Study," in *SIGMETRICS*, 2019.
[37] S. Ghose *et al.*, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.
[38] H. Gomez *et al.*, "DRAM Row-Hammer Attack Reduction Using Dummy Cells," in *NORCAS*, 2016.
[39] J. Gómez-Luna *et al.*, "Benchmarking a New Paradigm: Understanding a Modern Processing-in-Memory Architecture," in *SIGMETRICS*, 2021.
[40] Z. Greenfield and T. Levy, "Throttling Support for Row-Hammer Counters," U.S. Patent 9,251,885. 2016.
[41] D. Gruss *et al.*, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.
[42] D. Gruss *et al.*, "Rowhammer.js: A Remote Software-Induced Fault Attack in Javascript," arXiv:1507.06955 [cs.CR], 2016.
[43] J. Han *et al.*, "Surround Gate Transistor With Epitaxially Grown Si Pillar and Simulation Study on Soft Error and Rowhammer Tolerance for DRAM," *TED*, 2021.
[44] H. Hassan *et al.*, "CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability," in *ISCA*, 2019.
[45] H. Hassan *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[46] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[47] S. Hong *et al.*, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks," in *USENIX Security*, 2019.
[48] M. Horiguchi, "Redundancy Techniques for High-Density DRAMs," in *ISIS*, 1997.
[49] K. Itoh, *VLSI Memory Chip Design*. Springer, 2001.
[50] Y. Jang *et al.*, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *SOSP*, 2017.
[51] JEDEC, *JESD79F: Double Data Rate (DDR) SDRAM Standard*, 2008.
[52] JEDEC, *JEP122G: Failure Mechanisms and Models for Semiconductor Devices*, 2012.
[53] JEDEC, *JESD209-4B: Low Power Double Data Rate 4 (LPDDR4) Standard*, 2017.
[54] JEDEC, *JESD235C: High Bandwidth Memory (HBM) DRAM*, 2020.
[55] JEDEC, *JESD79-4C: DDR4 SDRAM Standard*, 2020.
[56] S. Ji *et al.*, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ASIACCS*, 2019.
[57] M. Kandemir *et al.*, "Memory Row Reuse Distance and Its Role in Optimizing Application Performance," in *SIGMETRICS*, 2015.
[58] S. Kanev *et al.*, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.
[59] I. Kang *et al.*, "CAT-TWO: Counter-Based Adaptive Tree, Time Window Optimized for DRAM Row-Hammer Prevention," *IEEE Access*, 2020.
[60] D. Kaseridis *et al.*, "Minimalist Open-Page: A DRAM Page-Mode Scheduling Policy for the Many-Core Era," in *MICRO*, 2011.
[61] H. Kasture and D. Sanchez, "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," in *IISWC*, 2016.
[62] B. Keeth and R. Baker, *DRAM Circuit Design: A Tutorial*. Wiley, 2001.
[63] M. N. I. Khan and S. Ghosh, "Analysis of Row Hammer Attack on STTRAM," in *ICCD*, 2018.
[64] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.

[65] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[66] S. Khan *et al.*, "A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM," *CAL*, 2016.
[67] S. Khan *et al.*, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[68] D.-H. Kim *et al.*, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2015.
[69] J. S. Kim *et al.*, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[70] J. S. Kim *et al.*, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency–Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.
[71] J. S. Kim *et al.*, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.
[72] J. S. Kim *et al.*, "Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques," in *ISCA*, 2020.
[73] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[74] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[75] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[76] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[77] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," *CAL*, 2016.
[78] R. K. Konoth *et al.*, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, 2018.
[79] A. Kwong *et al.*, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *S&P*, 2020.
[80] C. J. Lee *et al.*, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.
[81] D. Lee *et al.*, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.
[82] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[83] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[84] E. Lee *et al.*, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *ISCA*, 2019.
[85] J. Lee, "Green Memory Solution," Investor's Forum, Samsung Electronics, 2014.
[86] Z. Li *et al.*, "Compression of Pending Interest Table with Bloom Filter in Content Centric Network," in *CFI*, 2012.
[87] M. Lipp *et al.*, "Nethammer: Inducing Rowhammer Faults Through Network Requests," arXiv:1805.04956 [cs.CR], 2018.
[88] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices," in *ISCA*, 2013.
[89] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[90] H. Luo *et al.*, "CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off," in *ISCA*, 2020.
[91] K. Luo *et al.*, "Balancing Thoughput and Fairness in SMT Processors," in *ISPASS*, 2001.
[92] Y. Luo *et al.*, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," in *JSAC*, 2016.
[93] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[94] P. Michaud, "Demystifying Multicore Throughput Metrics," *CAL*, 2012.
[95] Micron Technology, "SDRAM, 4Gb: x4, x8, x16 DDR4 SDRAM Features," 2014.
[96] Micron Technology, "TN-40-03: DDR4 Networking Design Guide," 2014.
[97] J. Misra and D. Gries, "Finding Repeated Elements," *Science of Computer Programming*, 1982.
[98] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.
[99] N. Muralimanohar *et al.*, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.
[100] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[101] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.
[102] O. Mutlu, "RowHammer," https://people.inf.ethz.ch/omutlu/pub/onur-RowHammer-TopPicksinHardwareEmbeddedSecurity-November-8-2018.pdf, Top Picks in Hardware and Embedded Security. 2018.
[103] O. Mutlu *et al.*, "A Modern Primer on Processing in Memory," in *arXiv*, 2020.
[104] O. Mutlu and J. S. Kim, "RowHammer: A Retrospective," *TCAD*, 2019.
[105] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.
[106] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.
[107] O. Mutlu and L. Subramanian, "Research Problems and Opportunities in Memory Systems," *SUPERFRI*, 2014.
[108] NXP Semiconductors, "QorIQ Processing Platforms: 64-Bit Multicore SoCs," https://www.nxp.com/products/processors-and-microcontrollers/applications-processors/qoriq-platforms:QORIQ_HOME.
[109] G. Nychis *et al.*, "Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?" in *HOTNETS*, 2010.
[110] G. Nychis *et al.*, "On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects," in *SIGCOMM*, 2012.
[111] G. F. Oliveira *et al.*, "A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study," in *SIGMETRICS*, 2021.
[112] K. Park *et al.*, "Statistical Distributions of Row-Hammering Induced Failures in DDR3 Components," *Microelectronics Reliability*, 2016.
[113] Y. Park *et al.*, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.
[114] M. Patel *et al.*, "Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics," in *MICRO*, 2020.
[115] M. Patel *et al.*, "Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices," in *DSN*, 2019.
[116] M. Patel *et al.*, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.
[117] P. Pessl *et al.*, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.
[118] R. Qiao and M. Seaborn, "A New Approach for RowHammer Attacks," in *HOST*, 2016.
[119] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[120] K. Razavi *et al.*, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016.
[121] M. Redeker *et al.*, "An Investigation into Crosstalk Noise in DRAM Structures," in *MTDT*, 2002.
[122] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.
[123] S.-W. Ryu *et al.*, "Overcoming the Reliability Limitation in the Ultimately Scaled DRAM using Silicon Migration Technique by Hydrogen Annealing," in *IEDM*, 2017.
[124] SAFARI Research Group, "BlockHammer — GitHub Repository," https://github.com/CMU-SAFARI/blockhammer.
[125] SAFARI Research Group, "Ramulator — GitHub Repository," https://github.com/CMU-SAFARI/ramulator.
[126] SAFARI Research Group, "RowHammer — GitHub Repository," https://github.com/CMU-SAFARI/rowhammer.
[127] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," *Black Hat*, 2015.
[128] V. Seshadri *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[129] V. Seshadri *et al.*, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[130] V. Seshadri *et al.*, "Gather-Scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-Unit Strided Accesses," in *MICRO*, 2015.
[131] V. Seshadri and O. Mutlu, "In-DRAM Bulk Bitwise Execution Engine," *arXiv:1905.09822*, 2019.
[132] S. M. Seyedzadeh *et al.*, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *ISCA*, 2018.
[133] S. M. Seyedzadeh *et al.*, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," *CAL*, 2017.
[134] R. Sites, "It's the Memory, Stupid," *Microprocessor Report*, 1996.
[135] R. T. Smith *et al.*, "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM," *JSSC*, 1981.
[136] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for A Simultaneous Multithreaded Processor," in *ASPLOS*, 2000.
[137] M. Son *et al.*, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.
[138] Standard Performance Evaluation Corp., "SPEC CPU 2006," http://www.spec.org/cpu2006/.
[139] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.
[140] L. Subramanian *et al.*, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," *TPDS*, 2016.
[141] L. Subramanian *et al.*, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.
[142] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.
[143] Synopsys, Inc., "Synopsys Design Compiler," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.
[144] A. Tatar *et al.*, "Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer," in *RAID*, 2018.
[145] A. Tatar *et al.*, "Throwhammer: Rowhammer Attacks Over the Network and Defenses," in *USENIX ATC*, 2018.
[146] H. Usui *et al.*, "DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *TACO*, 2016.
[147] V. van der Veen *et al.*, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, 2016.
[148] V. van der Veen *et al.*, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.
[149] L. Wang *et al.*, "Bigdatabench: A Big Data Benchmark Suite from Internet Services," in *HPCA*, 2014.
[150] Y. Wang *et al.*, "FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching," in *MICRO*, 2020.
[151] Z. Weissman *et al.*, "JackHammer: Efficient Rowhammer on Heterogeneous FPGA–CPU Platforms," arXiv:1912.11523 [cs.CR], 2020.
[152] WikiChip, "Cascade Lake SP - Intel," https://en.wikichip.org/wiki/intel/cores/cascade_lake_sp.
[153] M. V. Wilkes, "The Memory Gap and the Future of High Performance Memories," *CAN*, 2001.
[154] Wolfram Research, Inc., "WolframAlpha," http://www.wolframalpha.com/.
[155] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *CAN*, 1995.
[156] Y. Xiao *et al.*, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security*, 2016.
[157] A. G. Yağlıkçı *et al.*, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," *arXiv*, 2021.
[158] C. Yang *et al.*, "Suppression of RowHammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology," *TDMR*, 2016.
[159] T. Yang and X.-W. Lin, "Trap-Assisted DRAM Row Hammer Effect," *EDL*, 2019.
[160] F. Yao *et al.*, "Deephammer: Depleting the Intelligence of Deep Neural Networks Through Targeted Chain of Bit Flips," in *USENIX Security*, 2020.
[161] J. M. You and J.-S. Yang, "MRLoc: Mitigating Row-Hammering Based on Memory Locality," in *DAC*, 2019.
[162] Z. Zhang *et al.*, "TeleHammer: A Stealthy Cross-Boundary Rowhammer Technique," arXiv:1912.03076 [cs.CR], 2019.
[163] Z. Zhang *et al.*, "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses," in *MICRO*, 2020.
[164] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," U.S. Patent 5,630,096. 1997.

A. Appendix Tables

Table 7 shows BlockHammer's configuration parameters used for each RowHammer threshold (N_{RH}) in Sections 6.1 and 8.3.

N_{RH}	$N_{RH}{}^*$	CBF Size	N_{BL}	t_{CBF}
32K	16K	1K	8K	64 ms
16K	8K	1K	4K	64 ms
8K	4K	1K	2K	64 ms
4K	2K	2K	1K	64 ms
2K	1K	4K	512	64 ms
1K	512	8K	256	64 ms

Table 7: BlockHammer's configuration parameters used for different N_{RH} values.

Table 8 lists the 30 benign applications we use for cycle-level simulations. We report last-level cache misses (*MPKI*) and row buffer conflicts (*RBCPKI*) per kilo instructions for each application. Non-temporal data copy, YCSB Disk I/O, and network accelerator applications do not have an *MPKI* value because they directly access main memory.

Category	Benchmark Suite	Application	MPKI	RBCPKI
L	SPEC2006	444.namd	0.1	0.0
		481.wrf	0.1	0.0
		435.gromacs	0.2	0.0
		456.hmmer	0.1	0.0
		464.h264ref	0.1	0.0
		447.dealII	0.1	0.0
		403.gcc	0.2	0.1
		401.bzip2	0.3	0.1
		445.gobmk	0.4	0.1
		458.sjeng	0.3	0.2
	Non-Temp. Data Copy	movnti.rowmaj	-	0.2
	YCSB Disk I/O	ycsb.A	-	0.4
		ycsb.F	-	1.0
		ycsb.C	-	1.0
		ycsb.B	-	1.1
M	SPEC2006	471.omnetpp	1.3	1.2
		483.xalancbmk	8.5	2.4
		482.sphinx3	9.6	3.7
		436.cactusADM	16.5	3.7
		437.leslie3d	9.9	4.6
		473.astar	5.6	4.8
H		450.soplex	10.2	7.1
		462.libquantum	26.9	7.7
		433.milc	13.6	10.9
		459.GemsFDTD	20.6	15.3
		470.lbm	36.5	24.7
		429.mcf	201.7	62.3
	Non-Temp. Data Copy	movnti.colmaj	-	30.9
	Network accelerator	freescale1	-	336.8
		freescale2	-	370.4

Table 8: Benign applications used in cycle-level simulations.