Panopticon: A Complete In-DRAM Rowhammer Mitigation

Tanj Bennett[§], Stefan Saroiu, Alec Wolman, and Lucian Cojocar
Microsoft, [§]Avant-Gray LLC

***Abstract*—The shortcomings of previous Rowhammer mitigations prevent their adoption in practice. Their implementations either need significant amounts of fast memory, such as CAM or SRAM, or require changes across multiple hardware and software layers. Panopticon is a complete in-DRAM Rowhammer mitigation that is both inexpensive and, for DDR4, requires no changes to any hardware components other than DRAM. Panopticon uses a novel DRAM mat design to implement counters and the DRAM's row decoding logic to access the row's corresponding counter. Finally, for DDR4, Panopticon leverages ALERT$_n$ to stop a memory controller from issuing new DRAM commands whenever it needs time to refresh potential victim rows.**

I. Introduction

Rowhammer remains a significant threat to DRAM's security and reliability. Despite memory vendors' claims that DDR4 is "Rowhammer free" and that Rowhammer attacks are "a thing of the past" [23], [10], [20], DDR4 DRAM is in fact more vulnerable than DDR3 [15], [6]. While DDR4 is resilient to older forms of attack that access one or two DRAM rows, DDR4 succumbs to multi-row-based attacks [6], [29]. Rowhammer affects all forms of DRAM and not just DDR; for example, LPDDR4 has been also shown to be vulnerable [28], [6]. Nearly a decade after the first Rowhammer publication [18], DRAM remains as vulnerable as ever.

The current situation is not due to a lack of research in addressing Rowhammer. The research community has proposed many mitigation schemes that ensure data located in victim rows is safe from Rowhammer [18], [8], [14], [1], [12], [42], [9], [31], [33], [26], [30], [21], [22], [3], [32], [19], [34], [4], [24], [36], [37], [44], [2], [40], [7], [17], [27], [45], [41], [25]. Unfortunately, none of these research proposals are widely deployed yet, for two main reasons. First, they come with significant cost and performance downsides. For example, to implement Graphene [27], a state-of-the-art tracking proposal, a DDR4 memory controller requires an additional 39.23KB of CAM memory per channel. Modern server-grade CPUs are equipped with quad-channel memory controllers [38] resulting in a total of 160KB of CAM memory, an amount that would significantly increase cost and power consumption. TWiCe [21], [22], an alternative tracking scheme, implements tracking inside the RCD chip on a DIMM. This requires a drastic redesign of the RCD chip while leaving DRAM lacking RCD (e.g., unbuffered memory) vulnerable. Similarly, memory partitioning schemes [3], [19], [34], [2], [40], [25] and new forms of DRAM [42], [30], [7] while promising also have significant shortcomings.

A second practical barrier is requiring changes at multiple hardware or software layers. For example, in addition to equipping a memory controller with significant amount of extra CAM memory, Graphene [27] requires DRAM to support a new command, called *Nearby Row Refresh*, that the memory controller uses to instruct DRAM to refresh both rows adjacent to an aggressor row. Costly changes across multiple layers are difficult to implement in practice. Incurring this cost is justified *only when* all stakeholders perform the changes necessary to enable the Rowhammer mitigation. This requires overcoming a challenging first step: hardware and software vendors must first reach consensus on the "right" Rowhammer mitigation. Unfortunately, this situation raises an instance of *Prisoner's Dilemma* [39] making it difficult for DRAM vendors, CPU vendors, and software companies to cooperate.

This paper presents Panopticon[1], a complete Rowhammer mitigation that is both inexpensive and, in the case of DDR4, requires no changes to any hardware components other than DRAM. Panopticon monitors all row activations inside the DRAM itself. Unlike previous tracking schemes that store counters in expensive forms of memory, such as CAM or SRAM, Panopticon's counters are stored in thin 16-bit wide mats co-located with the DRAM mats. Panopticon uses an *open-space, staggered* mat design for storing counters that leaves ample space for the counter increment and test logic. Unlike prior work, Panopticon does not need to implement a separate counter lookup circuit; instead, it re-uses DRAM's row decoding logic to access the counter for the activated row.

Each DRAM row is equipped with its own counter. When a counter reaches the Rowhammer threshold, a signal is sent to a *service queue* to enqueue the row address. Once enqueued, Panopticon must refresh potential victim rows in a timely manner to avoid the possibility of Rowhammer bit flips. One option is to provide extra time for mitigations during regular background refresh operations. With this design, Panopticon can service the queue when it receives a REF command (each tREFI). However, should the DRAM have no extra time or should the queue be full, the DRAM must find a way to signal the memory controller that it needs time to perform the Rowhammer remedies. Unfortunately, DRAM protocols do not specify a way for the DRAM to ask for *free time*.

Panopticon retrofits an existing signal in the DDR4 specification, called ALERT$_n$, to effectively "trick" the memory controller to pause issuing new DDR commands. DRAM uses ALERT$_n$ to signal errors to the memory controller. Upon receiving this signal, the memory controller stops issuing new DRAM commands and instead re-issues the old memory access. By making use of ALERT$_n$, Panopticon requires no modifications to any hardware other than DRAM itself.

II. Prior Proposals Face Practical Hurdles

Prior Rowhammer mitigation proposals face hurdles that make them expensive and unsuitable for adoption in practice.

[1]For an open-source implementation of Panopticon see: https://github.com/microsoft/Panopticon.

Most previous approaches can be classified in four categories:

- **Tracking approaches.** These approaches track microarchitectural events associated with Rowhammer to detect an ongoing attack before it has a chance to succeed.
- **Sampling approaches.** These approaches randomly sample microarchitectural events associated with Rowhammer. They proactively perform a Rowhammer mitigation on each sample.
- **Partitioning approaches.** These approaches compartmentalize memory and isolate an adversary from other co-located potential victims. When memory is properly compartmentalized, a Rowhammer attack in one memory perimeter cannot affect co-located potential victims.
- **Clean slate approaches.** These approaches require *significant* changes to the memory hardware including DRAM fabrication technology, DRAM devices, or DIMMs. The fundamental nature of these changes eliminate the threat of Rowhammer by design.

A. Tracking Approaches

Many previous mitigations track microarchitectural events associated with Rowhammer whether DRAM row activations [11], [8], [14], [31], [33], [26], [21], [22], [32], [4], [24], [36], [37], [44], [27], [41], cache misses [1], or a combination of both [12]. Whenever such events occur at a high rate, these schemes perform a corrective action, such as refreshing victim rows or throttling aggressor rows. Since a Rowhammer attack discharges a victim row's capacitors, refreshing a victim row before any of its bits flip effectively undoes the Rowhammer leakage effects. An alternate approach is to throttle aggressor rows until the DRAM had a chance to refresh itself (known as an auto-refresh or background refresh) [41].

Tracking approaches are attractive because the mitigations are performed *only* when the DRAM is in danger of bits flipping due to Rowhammer. When the DRAM is not under attack, which is the common case, the schemes avoid incurring any mitigation overhead.

Unfortunately, prior tracking approaches have two shortcomings that prevent them from being adopted in practice.

1. Significant cost and performance overhead. Three recent schemes have shown how to track an entire DRAM bank using significantly less state than maintaining a counter per row [41], [27], [22]. Despite their impressive state reduction, the state required for all DRAM in a system is still very high due to the large degree of bank parallelism found in DDR4 and DDR5 DRAM. For example, a DDR4 channel in modern high-performance servers (e.g., cloud servers) might have to accommodate two DIMMs where each DIMM has four ranks with 16 banks/rank. Although Graphene requires an astonishingly small amount of state per bank (only 2,511 bits/bank for DDR4), this state adds up to $2,511 \times 2 \times 4 \times 16 = 321,408$ bits or 39.23 kilo-bytes for a DDR4 channel. This analysis assumes a Rowhammer threshold of 50K row activations that we regard as optimistic. Recent work has shown that newer DDR4 DRAM has even lower Rowhammer thresholds [15].

BlockHammer [41] and TWiCe [22] both require more per-bank state than Graphene: 13,312 bits/bank for BlockHammer (1024 13-bit counters [41]) and 22,200 bits/bank for TWiCe (or 2.71KB [22]). Table I summarizes these schemes' state sizes. All other tracking schemes either require more state, or, when they do not, their Rowhammer protection is incomplete.

TABLE I
STATE SIZE OF THREE STATE-OF-THE-ART ROWHAMMER MITIGATION SCHEMES. A DDR4 CHANNEL HAS UP TO 128 BANKS, AND A MODERN SERVER-GRADE CPU HAS UP TO 4 CHANNELS.

	Per-Bank (bits)	**Per-Channel (kilo-bytes)**	**Per-CPU (kilo-bytes)**
Graphene [27]	2,511	39.23	156.9
BlockHammer [41]	13,312	208.00	832.0
TWiCe [22]	22,200	346.88	1,387.5

These previous approaches use CAM or SRAM memory to store their counter tables. Panopticon sidesteps this overhead by storing each row counter in the DRAM row itself. Lookup is essentially free because it leverages DRAM's internal row decoding logic. When a row is activated, counter increments occur *in parallel* with the sensing and amplifying of the DRAM row. Section V will describe the internal DRAM layout Panopticon uses for counters.

CRA [14] is an earlier scheme that, like Panopticon, maintains a row activation counter for each row in DRAM. However, unlike Panopticon, CRA carves a separate portion of DRAM for storing all row counters. CRA effectively doubles memory access latencies; to reduce this concern, CRA equips the memory controller with a small SRAM-based cache of recently accessed counters. Another prior work equips DRAM with a detector that tracks aggressor rows [11].

2. Require changes at multiple layers. Most tracking schemes store their counter tables outside of the DRAM device, either in the memory controller [41], [27] or in the RCD [22]. Unfortunately, such approaches preclude the ability to refresh victim rows without the cooperation of the DRAM device. By tracking *aggressor* rows, these schemes cannot identify the affected *victim* rows unless they have visibility into the DRAM's internal physical row mappings.

Unfortunately, DRAM vendors regard internal row layout (and the mappings of logical to physical DRAM rows) as proprietary and confidential [5], and are unwilling to share them even when they could help with mitigating Rowhammer. Instead, prior works propose a new DRAM command, called *Nearby Row Refresh* [27] or *Adjacent Row Refresh* [22] by which the memory controller reports an aggressor row's address and instructs the DRAM to refresh all potential victims.

Requiring changes at multiple layers makes tracking schemes difficult to adopt in practice without the cooperation of all stakeholders. For example, a CPU vendor is reluctant to incur a tracking scheme's cost and performance overhead until all three major DRAM vendors implement Nearby Row Refresh. Similarly, a single DRAM vendor is also reluctant implementing Nearby Row Refresh until this command is standardized in JEDEC and supported by other DRAM vendors.

BlockHammer [41] is an earlier scheme that shares Panopticon's goal of avoiding changes at multiple hardware layers. BlockHammer can be implemented entirely in the memory controller with no changes to DRAM internals. However, BlockHammer still requires significant amounts of SRAM.

B. Sampling Approaches

An alternative to tracking is sampling: upon each row activate (or other microarchitectural event associated with Rowhammer), with a low probability p, treat the row as an aggressor row [18], [14]. The lack of storing and managing counters makes sampling have two important benefits over tracking: statelessness and simplicity.

Unfortunately, all prior sampling approaches implement their logic in the memory controller and suffer from the same drawback of requiring DRAM vendors to share their proprietary, internal row layouts. We are unaware of a prior sampling approach done in DRAM only. Such an implementation would need a way to generate random numbers because, to be secure, sampling must be done randomly. Generating random numbers in DRAM is expensive and intrusive although recent work has shown new promising avenues [16].

Another drawback of sampling is that Rowhammer mitigation is always active. Unlike tracking, sampling cannot distinguish between an adversarial and a normal system workload.

C. *Partitioning Approaches*

Partitioning approaches [3], [19], [34], [2], [40], [25] confine an attacker to a memory perimeter (or memory security domain) that cannot interfere with the perimeters of other potential victims in the DRAM. An intuitive form of isolation is the physical partition of memory together with adding *guard rows* (i.e., rows that do not store any data) between different memory perimeters to ensure that the rows in one perimeter are sufficiently far from another perimeter. This distance guarantees that memory accesses in one domain cannot affect the rows in a neighboring domain.

The main difficulty with partitioning rises from the mismatch between data addressing done by the DRAM vs. the CPU. In DRAM partitioning approaches, the data unit is a DRAM row. A row contains data from multiple CPU cachelines not guaranteed to be contiguous in the CPU physical address space. In fact, for servers equipped with many DIMMs, a single row of DRAM data holds cachelines from multiple different physical pages. In such scenarios, row protection is a poor abstraction because it means protecting an arbitrary set of cachelines found on an arbitrary set of physical pages.

D. *Clean Slate Approaches*

Researchers are investigating new DRAM that minimize Rowhammer rootcauses [35]. One approach is reducing electro-magnetical coupling between nearby rows by introducing various forms of electron energy barriers between cells [42], [7]. Another approach is minimizing the number of electron traps [30]. Both electro-magnetical coupling and electron traps have been shown to be rootcauses of Rowhammer [18], [43].

A different approach is using dummy weak cells that are quick to flip (more susceptible to leakage) in DRAM. Such cells can act as early warnings of an ongoing attack [9].

Finally, another clean-slate approach is a new RCD chip with extra pins that remaps row addresses from one DRAM device to another [17]. Unfortunately, such forms of *address scrambling* do not stop Rowhammer but only raise the difficulty of targeting an explicit victim row.

III. Design Goals

We designed Panopticon with the following goals in mind:

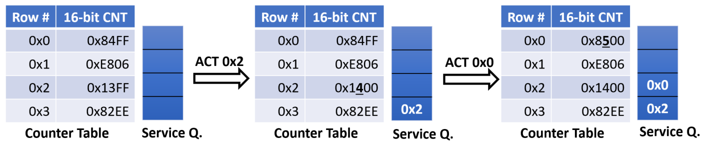


Fig. 1. Three consecutive states of the counter table and service queue. b_{10} is the threshold bit.

1. Low cost. Panopticon should store all row counters in DRAM and leverage DRAM's row decoding logic to identify the counter to increment on a row activation.

2. No changes to existing DRAM data mats and sense amplifiers. For efficiency reasons, DRAM uses tall mats with many DRAM rows. Sense amplifiers are tuned to reach the farthest row in the mat with their drive during the amplification phase. Unfortunately, the current DRAM design leaves no time for an additional operation, such as incrementing a counter.

Two possibilities are (1) shortening the data mat sizes or (2) increasing the power of sense amplifiers. Unfortunately, both options impact DRAM design significantly, and as a result have little chance to be deployed in practice. Instead, Panopticon should make no changes to existing DRAM mats and sense amplifiers.

3. No changes to the existing DDR4 protocol and memory controllers. An important security requirement is to refresh all potential *victim rows* in a timely manner. We believe, upon receiving a refresh command (REF), some DRAM devices do not make use of the entire allocated time (tRFC) to perform background refresh. In such a case, the DRAM could use this idle time to refresh potential victim rows if needed.

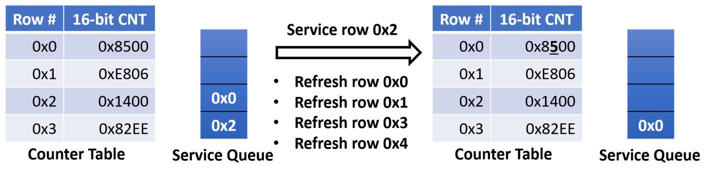
Nevertheless, we also expect well-tuned high-density DRAM to have little opportunity for additional idle time to service the queue. In such cases, DRAM needs to stop the memory controller from issuing new DRAM commands. Panopticon should perform this step without requiring changes to existing DDR4 protocol or to the memory controller.

IV. High-Level Overview

Panopticon maintains a counter table in-DRAM where each counter corresponds to a DRAM row and increments each time the corresponding row is activated. Unlike previous schemes, Panopticon does not maintain a Rowhammer threshold *value*, but a threshold *bit*. Whenever this bit is toggled during a counter increment, Panopticon enqueues the row address into a service queue.

Figure 1 illustrates three consecutive states of the counter table and service queue in a configuration where the threshold bit is b_{10} (b_0 corresponds to the least significant bit), the service queue has four entries, and the rows activated are row 0x2 and 0x0, respectively. In its initial state, b_{10} has a value of 0 for row 0x2, and 1 for row 0x0. However, in each case, an additional row activation *toggles* b_{10} causing Panopticon to enqueue each row address in the service queue, respectively.

Using a threshold bit rather than a value lets Panopticon avoid having to reset its counters. While a trivial operation when performed in SRAM, resetting a DRAM counter is expensive because it would require extra circuitry and additional latency to clear each DRAM cell. Panopticon ensures that a row is serviced every 2^i activates, where i is the threshold bit (e.g., $i = 10$ for b_{10}).



Row #	16-bit CNT
0x0	0x8500
0x1	0xE806
0x2	0x1400
0x3	0x82EE

Counter Table

Row #	16-bit CNT
0x0	0x8500
0x1	0xE806
0x2	0x1400
0x3	0x82EE

Counter Table

Fig. 2. Servicing row 0x2. DRAM row disturbance affects up to two nearby rows on each side of an aggressor. Servicing a row requires refreshing four potential victim rows.

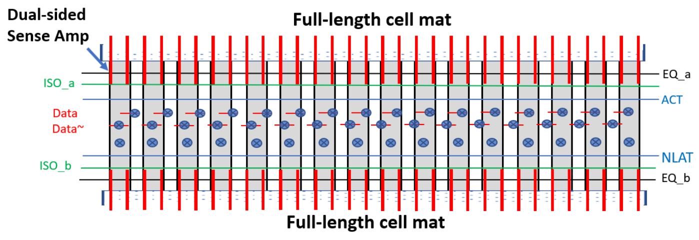


Fig. 3. Arrangement of sense amplifiers with full length mats.

Panopticon services rows from the queue when receiving a REF command (if there is time), or by signaling the memory controller to stop issuing new DRAM commands. To service a row, Panopticon refreshes all potential victim rows. DRAM vendors regard the number and identity of potential victim rows in their DRAM as highly confidential. However, with Panopticon, this information remains in the hands of the DRAM vendors; they need to fine-tune Panopticon to perform the correct remedy applicable to each DRAM device. Figure 2 shows servicing row 0x2 in a configuration where row disturbance affects up to two nearby rows on each side of the aggressor (±2).

V. In-DRAM Architecture

Panopticon comprises of a set of narrow DRAM mats (i.e, sub-arrays) to store counters, a small piece of logic for each counter mat that increments and writes back the counters, and a tiny state machine in each bank that implements a service queue and the ALERT$_n$ signaling.

A. Counter Mats

Panopticon needs to increment, update, and store row counters without slowing down normal DRAM operations. A design that lays out counter cells in a manner similar to the DRAM data cells (i.e., the counters and the DRAM cells have the same geometry) will introduce significant latency overhead due to the additional time needed to read and store back the counter value along the data lines. Instead, Panopticon uses an open-space design that places the counter mat in a staggered manner leaving space for the incrementer logic.

Sense amplifiers' arrangement with full length mats in today's DRAM. Sense amplifiers bridge cell-mats and connect to either the upper or lower data arrays depending on how the DRAM circuit is activated. Mat selection is done using the equalizer (EQ_a and EQ_b) and isolation (ISO_a and ISO_b) signals. Panopticon must keep the same spacing for the sensing and routing of the counter bits that run beside these full length mats and use the same row lines. Figure 3 illustrates the sense amplifiers layout between two cell mats above and below.

Counter mats design and layout. The counter mats in Panopticon are half-length and support only half the number of rows. Both a mat's top and bottom have one-sided sense amps. Halving the counter mats' size has two important benefits. First, the half-length data lines offer lower latency for counter increments within the time needed to read the data cells. Second, it creates a large open-space adjacent to the sense amplifiers to place logic for an incrementer (and also for testing). The location is ideal because it has access to all process layers including interconnect to create the logic needed for controlling and operating the incrementer.

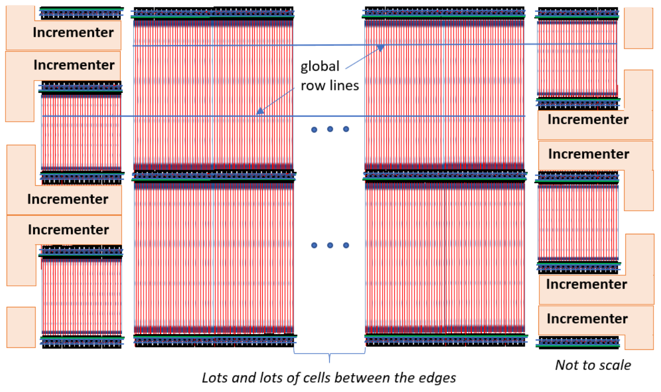


Fig. 4. Panopticon's open-space design and staggered layout of counter mats. The mats are half length leaving space for incrementer and testing logic.

The open-space design comes with a cost: a single counter mat covers half the data rows only. To offer a complete set of counters, Panopticon alternates (staggers) half the counter mats left and the remaining half right such that each row of a standard mat continues into a counter half-mat.

The open-space staggered layout leaves plenty of space for incrementers. The incrementer logic is placed in the open space next to both the upper and lower sense amps for short distances and latencies. The row lines are not extended into the open spaces that can be further extended to the outside if more room for logic is needed. Figure 4 shows the staggered layout of half length counter mats.

B. Incrementer

The incrementer makes use of the read and writeback cycle inherent in DRAM row activation to perform its logic. A 16-bit counter value is incremented each time a row is activated. The incrementer is a chain of 16 flip-flops that latch the data acquired when the row is activated. A single clock then toggles the flip-flops, cascading down the chain in a classic sequential adder. The resulting value is then coupled back from the flip-flops into the columns, where it updates the counter cells following the classing read, modify, write timing comprising the row activate command.

We also designed testing logic for the incrementer. With this logic, the DRAM vendor can inject a value into a counter, generate an alarm when the Rowhammer threshold bit has toggled, and read back the counter values. Figure 5 illustrates the incrementer and its testing logic.

C. Service Queue

When a high-order bit of a counter toggles, Panopticon sends a signal to the refresh logic to enqueue the row address in a service queue. For example, an implementation of Panopticon that sends the signal whenever the 11th bit of a

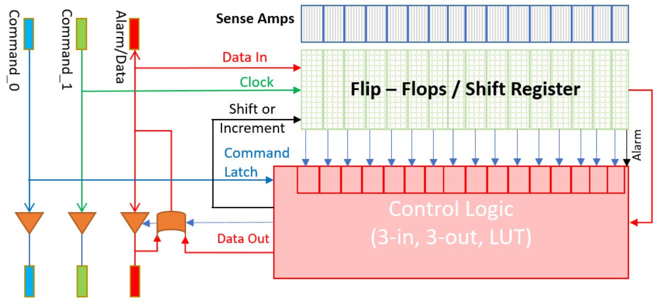


Fig. 5. Incrementer. Control logic is used for DRAM counter testing.

counter (b_{10}) toggles would place the row address into the service queue every 1,024 row activations. The signal uses the Alarm/Data line shown in Figure 5.

Panopticon uses SRAM to implement the service queue. However, this queue is small; in our performance evaluation of Panopticon, we used a queue of 8 entries per bank. A row address is at most 18 bits long in DDR4; in this case, the amount of SRAM required by Panopticon is 144 bits of SRAM per bank (in contrast, Graphene [27] requires 2,511 bits/bank).

D. Row Refresh & ALERT$_n$ Signaling

Once enqueued, a row must be serviced in a timely manner. One option is to service the row each REF command, at an average rate of 7.8 µs in DDR4. Upon receiving a REF command, the DRAM can *re-purpose* some of its time to service an enqueued row by refreshing its neighbors.

Servicing a row likely requires refreshing multiple rows. DRAM disturbance often affects more than just the two rows adjacent to an aggressor row although these effects decay with distance in a super-linear manner [18], [15], [5]. During DRAM testing and qualifying, vendors can tune Panopticon to refresh all rows affected by DRAM disturbance to eliminate any possibility of Rowhammer.

Whenever, Panopticon requires additional time to service its queue entries, Panopticon asserts ALERT$_n$ in a manner similar to a command and address parity error. In this case, a DDR4 memory controller must wait and re-try the failing command [13]. The wait duration is specified using a register whose value is read when DDR first initializes. With this mechanism, Panopticon can *effectively* pause the memory controller by making it re-issue the same DRAM command. During this time, the DRAM can service its queue and refresh all relevant potential victim rows. To resume functionality, Panopticon de-asserts ALERT$_n$.

VI. Security Analysis

In our threat model, the attacker has complete control over all DDR4 commands sent to DRAM, but cannot violate the DDR protocol including the commands' ordering and timings. We set the Rowhammer threshold value to 512 (corresponding to toggling b_9), conservatively.

Our security analysis investigates the difficulty of creating undesirable scenarios for Panopticon, such as placing rows in the service queue during several consecutive, back-to-back refresh intervals (tREFI) and filling up the service queue. We make two assumptions to simplify our analysis. Even under these simplifying assumptions, our analysis will show

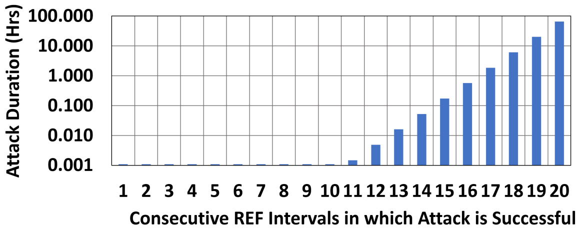


Fig. 6. Duration of an attack that aims to place a row in the service queue in consecutive refresh intervals (tREFI).

that such attacks are relatively easy to mount. Relaxing these assumptions will make the attacks even more potent.

Our first assumption is that the background refresh schedule is regular: a REF command is sent every 7.8 µs [13]. Between two consecutive REFs, an attacker has a *budget* of 156 row activations to activate any rows (or the same row). This is a simplifying assumption because the gap between two REF commands can be up to $9 \times 7.8\,\mu s$ in DDR4 (e.g., these gaps are due to postponing or pulling-in REF commands). Such gaps give the attacker an even larger budget of row activates.

Second, the attacker cannot determine any counter's value. The assumption is unrealistic because of the following side-channel: an attacker learns that the counter of the most recently activated row reached the threshold whenever the DRAM asks the memory controller for free time. However, we wanted to understand whether Panopticon can be made secure in the absence of such side channels. In this case, one could build DRAM that never needs to ask for time to perform its Rowhammer mitigations as long as the service queue is appropriately sized so it is never full.

Under these assumptions, we answer two questions:

1. Is placing a row in the service queue in consecutive refresh intervals (tREFI) difficult? Well-tuned high-performance DRAM might not need all 8192 REFs received in a refresh window [13] for performing background refresh and could re-purpose a small number (say less than 1%) for servicing the queue. Upon receiving a REF, Panopticon would check whether an entry is present in the service queue. In that case, the REF would be "hijacked" away from background refresh to perform the Rowhammer mitigation. This model works *as long as* the hijacked REFs are spread evenly throughout a refresh window. An uneven spread in which rows are placed in the queue in each several consecutive refresh intervals (tREFI) could cause undesirable interference with the background refresh schedule.

We analyze an attack that chooses a random row and uses the budget (156 ACTs) to activate the row in each tREFI. Figure 6 shows the expected number of hours until the attack is successful as a function of the number of consecutive REF intervals. By choosing a random row each refresh interval (tREFI), in less than an hour, an attacker can place a row in the service queue for 16 *consecutive* tREFIs. This result demonstrates that it is trivial for an attacker to create scenarios in which the DRAM must perform Rowhammer mitigations for *many consecutive REFs in a row*.

2. Is it realistic to use a large-sized queue to ensure it can never be full? Each tREFI, the attack picks a set of Q random rows and activates all rows in a round-robin manner (Q is the queue size). How long will such an attack take to

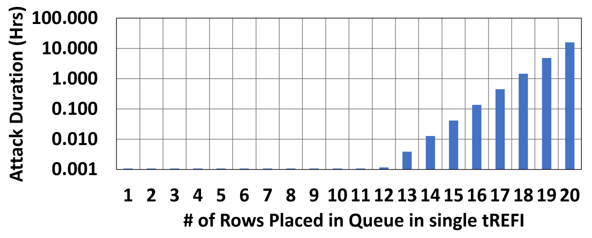


Fig. 7. Duration of an attack that aims to fill up the queue in a single refresh interval (tREFI) with 50% chance of success.

have a 50% chance to place all Q rows in the queue during the same tREFI interval?

Figure 7 shows the number of hours until the attack has a 50% chance of success. When under attack, even a 20-entry queue will be full in less than 16 hours. To be safe (i.e., the rate of success must be negligible), Panopticon's service queue size must be prohibitively large.

Summary: Both results speak convincingly that an attacker has an easy time creating scenarios in which the DRAM must mitigate victim rows for many consecutive tREFI intervals or the DRAM is overwhelmed with mitigating victim rows. To remain safe under such scenarios, the DRAM must be able to ask the memory controller for additional free time.

VII. Discussion

A. A Rowhammer Attack Turns into a DoS Attack

Should attackers know the row counters' values, they could craft a malicious workload that *consistently* fills up the service queue with aggressor rows. In such a case, DRAM would need to spend a fraction of its time refreshing potential victim rows rather than servicing DRAM requests. This leads to a form of DoS because a portion of DRAM's bandwidth is effectively disrupted and re-purposed for performing Rowhammer mitigations.

We believe *any form* of a Rowhammer defense would need to consume resources to perform mitigations *when the system is under attack*. This will inherently open up the possibility of DoS. However, we argue this trade-off is fundamental: when under attack, the DRAM needs to slow down (or stop) performing additional row activations and refresh victim rows instead. Once the attack stops, the DRAM will resume normal operation servicing memory reads and writes.

One potential optimization is initializing Panopticon's counters with random values at boot time. This will make it difficult (*but not impossible*) for the attacker to learn the row counters' values. Even lacking this information, an attacker could overwhelm the queue, but can do so only periodically (see Section VI). It is much more difficult doing it *consistently*. However, a determined attacker could use a side-channel (such as observing when ALERT$_n$ is asserted) to slowly determine when counters reach their Rowhammer threshold. Nevertheless, we argue that initializing counter values randomly is helpful because it would significantly raise the bar to mounting a DoS attack in practice.

B. The Impact of Normal Workloads

As described, Panopticon never resets its counters. Thus, even under a normal workload, rows would need to be serviced when their counters reach the Rowhammer threshold. This leads to additional undesirable overhead *even when the system is not under attack*.

A naïve possibility is to reset a row's counter whenever all its potential victims are refreshed by the DRAM's background refresh cycle. However, we believe this is a dangerous design choice. For a given aggressor row, different victims could be refreshed at different points in time. In this case, no form of counter reset is safe: the counter is reset either too early, before some of the victim rows had a chance to be refreshed, or too late, after some of the victim rows have been affected by the aggressor row. Either choice provides a way for row activations to escape being tracked.

A different, safer possibility is to clear a counter's low order bits (say lowest 5 bits) each time its corresponding row is refreshed (i.e., once every 64ms in DDR4). This would be very effective at reducing Panopticon's overhead because most rows are not activated more than 32 times within a refresh window. To be safe, the Rowhammer threshold must be set conservatively to accommodate the possibility that up to 32 row activates are not accounted for.

An additional optimization is clearing a single low order bit (say bit 5) rather than 5 bits. This would reduce the complexity of the clearing counters circuit. With this optimization in place, each counter would be decremented by a value between 0 and 32 each refresh window.

C. The Power and Space Overhead of Counter Mats

DRAM vendors must accommodate the power and space overheads of counter mats in their DRAM devices. However, we believe this overhead to be low. The counters need not be larger than the Rowhammer threshold; a 16-bit counter can accommodate threshold values of up to 65,536 row activations, and recent work has shown that modern DDR4's Rowhammer thresholds are much lower in fact [15], [6]. Thus, 32 columns of counters (counting both the left and right mat areas) are sufficient to accommodate a full row that consists of thousands of columns.

VIII. Conclusions

This paper presents Panopticon, a complete in-DRAM Rowhammer mitigation scheme. Panopticon uses an open-space, staggered mat design to equip each DRAM row with an exclusive counter. Panopticon leverages the DRAM row decoding logic to avoid a separate implementation of a counter lookup circuit. For DDR4, Panopticon leverages the ALERT$_n$ signal to pause a DRAM controller from issuing any new, previously unseen DRAM commands. During this time, Panopticon can perform its Rowhammer mitigation and refresh potential victim rows. Our security analysis shows that Panopticon must be able to ask for time from the memory controller, or, otherwise, an attacker could trivially overwhelm the DRAM with Rowhammer mitigation work.

Acknowledgments. We would like to thank the anonymous reviewers for their insightful comments. We also received helpful feedback from Kuljit Bains, David Blankenbeckler, Tim Cowles, Brett Dodds, Todd Farrell, Terry Grunzke, Todd Merritt, Bill Nale, and Kushagra Vaid.

References

[1] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. ORren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016.

[2] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "RIP-RH: Preventing Rowhammer-Based Inter-Process Attacks," in *ASIA-CCS*, 2019.

[3] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't Touch This: Practical and Generic Software-only Defenses Against RowHammer Attacks," *USENIX Sec.*, 2017.

[4] A. Chakraborty, M. Alam, and D. Mukhopadhyay, "Deep Learning based Diagnostics for Rowhammer Protection of DRAM Chips ," in *ATS*, 2019.

[5] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers," in *IEEE S&P*, 2020.

[6] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[7] S. K. Gautam, S. K. Manhas, A. Kumar, M. Pakala, and E. Yieh, "Row Hammering Mitigation Using Metal Nanowire in Saddle Fin DRAM," *IEEE T-ED*, vol. 66, 2019.

[8] M. Ghasempour, M. Lujan, and J. Garside, "ARMOR: A Run-Time Memory Hot-Row Detector," http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/armor.html, 2015.

[9] H. Gomez, A. Amaya, and E. Roa, "DRAM Row-hammer Attack Reduction using Dummy Cells," in *NORCAS*, 2016.

[10] M. Greenberg, "Row Hammering: What it is, and how hackers could use it to gain access to your system," https://blogs.synopsys.com/committedtomemory/2015/03/09/row-hammering-what-it-is-and-how-hackers-could-use-it-to-gain-access-to-your-system/, 2015.

[11] Z. Greenfield, J. B. Halbert, and K. S. Bains, "Method, apparatus and system for determining a count of accesses to a row of memory," Patent No. US 2014/0085995, 2014.

[12] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Stopping Microarchitectural Attacks Before Execution," *IACR*, 2016.

[13] JEDEC, *Double Data Rate 4 (DDR4) SDRAM Standard*, 2012.

[14] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural Support for Mitigating Row Hammering in DRAM Memories," *CAL*, 2015.

[15] J. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.

[16] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput," in *HPCA*, 2019.

[17] M. Kim, J. Choi, H. Kim, and H.-J. Lee, "An Effective DRAM Address Remapping for Mitigating Rowhammer Errors," in *TC*, 2019.

[18] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[19] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, 2018.

[20] M. Lanteigne, "How Rowhammer Could be Used to Exploit Weaknesses in Computer Hardware," http://www.thirdio.com/rowhammer.pdf, 2016.

[21] E. Lee, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Time Window Counter Based Row Refresh to Prevent Row-Hammering," *CAL*, 2018.

[22] ——, "TWiCe: Preventing Row-hammering by Exploiting Time Window Counters," in *ISCA*, 2019.

[23] J.-B. Lee, "Green Memory Solution," http://aod.teletogether.com/sec/20140519/SAMSUNG_Investors_Forum_2014_session_1.pdf, 2014.

[24] C. Li and J.-L. Gaudiot, "Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters," in *COMPSAC*, 2019.

[25] K. Loughlin, S. Saroiu, A. Wolman, and B. Kaskci, "Stop! Hammer Time: Rethinking Our Approach To Rowhammer Mitigations," in *HotOS*, 2021.

[26] O. Mutlu, "The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser," in *DATE*, 2017.

[27] Y. Park, W. Kwon, E. Lee, T. J. Han, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO*, 2020.

[28] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Sec.*, 2016.

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

[30] S. Ryu, K. Min, J. Shin, H. Kwon, D. Nam, T. Oh, T. Jang, M. Yoo, Y. Kim, and S. Hong, "Overcoming the reliability limitation in the ultimately scaled DRAM using silicon migration technique by hydrogen annealing," in *IEEE IEDM*, 2017.

[31] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-based Tree Structure for Row Hammering Mitigation in DRAM," *CAL*, 2017.

[32] ——, "Mitigating Wordline Crosstalk Using Adaptive Trees of Counters," in *ISCA*, 2018.

[33] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC*, 2017.

[34] V. Van Der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM," in *DIMVA*, 2018.

[35] A. J. Walker, S. Lee, and D. Beery, "On DRAM Rowhammer and the Physics of Insecurity," *IEEE T-ED*, vol. 68, 2021.

[36] Y. Wang, Y. Liu, P. Wu, and Z. Zhang, "Detect DRAM Disturbance Error by Using Disturbance Bin Counters," in *CAL*, 2019.

[37] ——, "Reinforce Memory Error Protection by Breaking DRAM Disturbance Correlation Within ECC Words," in *ICCD*, 2019.

[38] Wikipedia, "Xeon," https://en.wikipedia.org/wiki/Xeon.

[39] Wikipedia, "Prisoner's dilemma," https://en.wikipedia.org/wiki/Prisoner%27s_dilemma, 2020.

[40] X.-C. Wu, T. Sherwood, F. T. Chong, and Y. Li, "Protecting Page Tables from RowHammer Attacks using Monotonic Pointers in DRAM True-Cells," in *ASPLOS*, 2019.

[41] A. G. Yağlıkçı, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi, S. Ghose, and O. Mutlu, "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows," in *HPCA*, 2021.

[42] C.-M. Yang, C.-K. Wei, Y. J. Chang, T.-C. Wu, H.-P. Chen, and C.-S. Lai, "Suppression of Row Hammer Effect by Doping Profile Modification in Saddle-Fin Array Devices for Sub-30-nm DRAM Technology," *IEEE T-DMR*, vol. 16, 2016.

[43] T. Yang and X.-W. Lin, "Trap-assisted DRAM Row Hammer Effect," *EDL*, 2019.

[44] J. M. You and J.-S. Yang, "MRLoc: Mitigating Row-hammering based on memory Locality," in *DAC*, 2019.

[45] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wangk, N. Surya, Y. Gao, K. Li, Z. Wang, and C. Wu, "SoftTRR: Protect Page Tables Against RowHammer Attacks using Software-only Target Row Refresh," *arxiv.org*, 2021.