# Phoenix: Rowhammer Attacks on DDR5 with Self-Correcting Synchronization

Diego Meyer† *ETH Zurich*
Patrick Jattke† *ETH Zurich*
Michele Marazzi *ETH Zurich*
Salman Qazi *Google*
Daniel Moghimi *Google*
Kaveh Razavi *ETH Zurich*

†Equal contribution joint first authors

***Abstract*—DDR5 has shown an increased resistance to Rowhammer attacks in production settings. Surprisingly, DDR5 achieves this without additional refresh management commands, pointing to the deployment of more sophisticated in-DRAM Target Row Refresh (TRR) mechanisms. This paper reverse engineers such advanced TRR schemes in DDR5 devices for the first time. Our findings show that compared to older mitigations deployed in DDR4, these new schemes have considerably fewer blind spots spread over many refresh intervals. This means that an effective DDR5 Rowhammer pattern must precisely track thousands of refresh operations, which we show is not possible with existing techniques. To address this challenge, our new DDR5 Rowhammer attack, called Phoenix, self-corrects the pattern whenever it detects a missed refresh operation during the attack. Our evaluation shows that Phoenix triggers bit flips on 15 out of 15 DDR5 devices in our test pool. Using these bit flips, we build the first Rowhammer privilege escalation exploit that obtains root on a commodity DDR5 system with default settings in as little as 109 seconds. These results provide further evidence that a principled Rowhammer mitigation, such as per-row activation counters, is mandatory for a secure operation of future devices.**

## 1. Introduction

Four years after its inception, DDR5 devices remain protected against Rowhammer attacks. Recent work shows that this protection is due to improved Target Row Refresh (TRR) mechanisms inside commodity DDR5 chips [1], [2]. What remains unclear is how these mechanisms manage to capture aggressor rows in advanced Rowhammer patterns [3], [4], [5]. Our reverse-engineering efforts show that significantly longer Rowhammer patterns are nowadays necessary to bypass these new protections. To trigger Rowhammer bit flips, such patterns need to remain in-sync with thousands of refresh commands, which is challenging. Our new Rowhammer attack, called *Phoenix*, resynchronizes these long patterns as necessary to trigger the first DDR5 bit flips in devices with such advanced TRR protections.

**Rowhammering DDR5.** In 2024, Zenhammer [6] reported the first Rowhammer bit flips on one of ten tested DDR5 DIMMs. These initial DDR5 results are surprising, given that earlier work using similar patterns could trigger bit flips on 40 out of 40 DDR4 DIMMs [3]. An improved DRAM substrate could explain the increased resilience of DDR5 devices against Rowhammer, but recent work shows that the DDR5 substrate is similarly vulnerable to Rowhammer as DDR4 devices [1]. Another possibility could be the introduction of the new Refresh Management (RFM) commands for DDR5 devices to give the device more time to perform internal mitigative refreshes, but neither CPUs from Intel nor AMD send any RFM commands under Rowhammer workloads [2]. In the absence of RFM commands and improvements to the DRAM substrate against Rowhammer, all indications point toward improved in-DRAM TRR mechanisms in DDR5 chips [1].

**TRR mechanisms in DDR5.** Our FPGA-based reverse engineering experiments on DDR5 devices from SK Hynix, currently the largest DRAM vendor [7], [8], reveal that compared to DDR4 [3], [4], [5], these new TRR mechanisms change their behavior over significantly longer periods of activity — sometimes thousands of refresh intervals. This change makes it challenging to apply the state-of-the-art reverse engineering technique that looks at individual activate commands due to this significant increase in TRR's state [5]. Instead, we devised a new reverse-engineering technique that allows studying the TRR behavior at the granularity of refresh intervals. Using this technique, we can *zoom out* to identify less frequently sampled refresh intervals among many thousands of them, and thereafter, *zoom in* to analyze the precise TRR behavior at these specific intervals with previous techniques [3], [5]. This approach enables us to study these new and more complex TRR mechanisms in DDR5 for the first time. Our results based on reverse engineering two particular TRR mechanisms show that a significant portion of activations in specific refresh intervals are not accounted for by the TRR mechanisms.

**Bypassing TRR mechanisms on DDR5.** We leverage our reverse-engineering results to build Rowhammer patterns that evade the TRR mechanisms on these DDR5 devices. These carefully crafted patterns, covering 128 and 2608 refresh intervals, hammer particular activation slots in very specific refresh intervals only. As a result, they evade the corresponding TRR mechanisms. Furthermore, these patterns need to be aligned with the right refresh command to be effective, which we optimize by hammering multiple banks simultaneously. We show that executing these new complex patterns triggers bit flips on all 10 SK Hynix DDR5 RDIMMs in our test pool. However, executing these patterns from commodity systems requires reliable synchronization with the refresh command over thousands of refresh intervals.

**Phoenix.** We find that the state-of-the-art synchronization

method [6] misses refresh commands regularly, making it unsuitable for effective Rowhammer attacks on DDR5. We explore two techniques for a more effective synchronization in the design of *Phoenix*, the first system-level Rowhammer attack that bypasses advanced TRR schemes deployed in DDR5 devices. The first technique aims to improve refresh detection by separating Rowhammer accesses and refresh synchronization accesses into different threads. Instead of improving refresh detection, the second technique aims to detect a missed refresh and resynchronize the pattern accordingly. Our results show that while the first technique improves the state of the art, it still fails to remain synchronized for a sufficiently large number of refresh intervals, preventing Phoenix from triggering bit flips. However, the second technique, which we refer to as self-correcting synchronization, can keep Phoenix synchronized with refresh commands for entire refresh windows, sufficient for our new patterns to trigger bit flips.

We evaluate Phoenix on 15 DDR5 UDIMMs produced between 2021 and 2024. Phoenix triggers bit flips on all these devices in seconds. We use Phoenix to craft the first end-to-end DDR5 page table exploit that succeeds in as little as 109 seconds. Our measurements show that increasing the refresh rate by 3× mitigates Phoenix on devices we tested while introducing an overhead of 8.4% in the SPEC2017 benchmark suite [9].

**Contributions.** We make the following contributions:

- We reverse engineer two distinct TRR implementations in DDR5 devices from SK Hynix using a series of carefully crafted experiments.
- We derive custom Rowhammer patterns that effectively bypass TRR mechanisms on these devices based on the insights from our reverse engineering.
- We present Phoenix, a new Rowhammer attack that leverages a new self-correcting refresh synchronization mechanism to execute these new Rowhammer patterns.
- We show that Phoenix triggers bit flips on all 15 DDR5 devices in our test pool. We use these bit flips to craft the first Rowhammer privilege escalation exploit on a commodity DDR5 system with default configuration.

**Responsible disclosure and open sourcing.** We started a responsible disclosure of Phoenix through the Swiss NCSC with SK Hynix, CPU vendors, and major cloud providers on June 6, 2025. The issue remained under embargo until September 15, 2025. We were informed on September 12 that there is a BIOS update for AMD client machines to address the issue, but we could not independently verify if this update adequately addresses Phoenix. Phoenix is tracked under CVE-2025-6202. More information, including the source code for all the experiments and the exploit, can be found at the following URL: https://comsec.ethz.ch/phoenix

## 2. Background

We discuss DRAM organization (§2.1) and operation (§2.2), followed by changes in DDR5 (§ 2.3). We then provide background on the Rowhammer vulnerability (§ 2.4) and the proposed mitigations (§ 2.5).

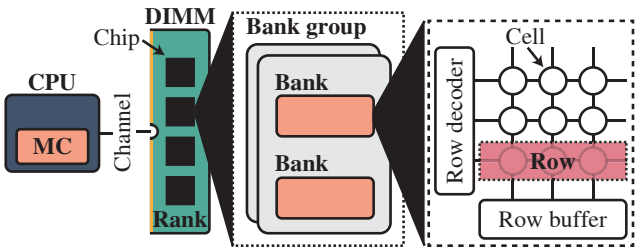


**Figure 1. DRAM organization.** The hierarchical organization of a modern DRAM system such as DDR5 DIMMs.

### 2.1. DRAM Organization

A dynamic random-access memory (DRAM) system, such as a DDR5 Dual In-line Memory Module (DIMM) or *device*, is organized hierarchically (Fig. 1). At the bottom, DRAM *cells* store single bits of information in capacitors. Matrices of cells are arranged in *rows* and *columns*, forming a *bank*. Each bank has a row decoder to select a row, transferring the entire row to the row buffer, and a column decoder to select specific columns for access. Banks are grouped into *bank groups* within a DRAM *chip*. A DDR5 DRAM module typically has two *subchannels*, each with multiple DRAM chips operating in lockstep mode and organized into *ranks*. DIMMs are connected to the CPU's memory controller through dedicated memory *channels*.

Server systems typically use Registered DIMMs (RDIMMs) instead of Unbuffered DIMMs (UDIMMs), which include an on-DIMM buffer to reduce the electrical load on the memory controller. As RDIMMs are designed for servers, most of them are equipped with rank-level Error Correction Code (ECC) in addition to On-Die Error Correction Codes (ODECC) in the DDR5 DRAM chip itself.

### 2.2. DRAM Operations

The memory controller (MC) communicates with a DDR5 DRAM device by following the DDR5 protocol [10], which specifies DRAM commands and their required timings.

**Memory Operations.** When accessing data, the MC issues an activate (`ACT`) command to the DRAM row given by the memory address. The activation loads the row bits into the row buffer, from which data can be read (`RD`) or written (`WR`) by specifying the requested column. After an access, precharge (`PRE`) prepares the bank for the next `ACT`.

**Periodic Refreshes.** The capacitors in DRAM cells leak charge over time, which would lead to retention errors. To avoid such errors, the MC issues a `REF` command every refresh interval ($t_{REFI}$, i.e., 3.9 µs on average).

### 2.3. Changes introduced in DDR5

With DDR5, several changes were introduced to the DRAM architecture and protocol, of which we briefly summarize the ones relevant to our work.

**Higher Refresh Rate.** DDR5 devices require refresh commands to be sent once every 3.9 µs on average by default, which is two times more often than on DDR4 (7.8 µs).

**Refresh Management.** The Refresh Management (RFM) command provides additional time to the DRAM device to refresh memory cells, thus protecting data integrity during periods of high memory activity (e.g., Rowhammer). This device-specific, optional feature requires MC support and may not be required by all DDR5 DIMMs. RFM is based on per-bank counters that track the number of activations, and whenever a certain threshold is reached, the MC issues an `RFM` to reduce the counter and perform a mitigative refresh. Recent work [2] showed that `RFM` is not used by the memory controllers of AMD Zen 4, Intel Alder Lake, and Intel Raptor Lake CPUs.

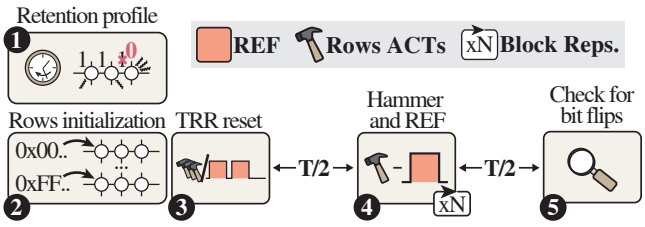


**Figure 2. U-TRR experiment.** After profiling DRAM rows, U-TRR resets the TRR and waits for half of the retention time, $T/2$. Then, it executes a hammering-refreshing payload. Finally, after the remaining retention time has passed, it checks for bit flips.

## 2.4. Rowhammer

Rowhammer is a DRAM disturbance error that was first publicly reported in 2014 on DDR3 [11]. Rowhammer allows an attacker to flip bits in DRAM *victim* rows by repeatedly activating neighboring *aggressor* rows. A multitude of attacks have demonstrated the practicality of exploitation with Rowhammer to cause denial-of-service [12], escalate privilege [13], [14], [15], [16], [17], [18], [19], leak data [20], [21], [22], [23], degrade deep learning models [24], [25], [26], and attack VMs [27], [28], browsers [29], [30], [31], [32], [33], and systems across the network [34], [35].

**Hammer Count.** The Hammer Count (HC) refers to the cumulative number of activations to aggressor rows. Accordingly, the *minimum* HC ($\mathrm{HC_{min}}$) is the lowest HC to trigger a bit flip in a DRAM row, which varies across different rows. Therefore, the $\mathrm{HC_{min}}$ characterizes a device's Rowhammer vulnerability level [36]. Generally, smaller technology nodes make DRAM cells more vulnerable to Rowhammer by reducing the $\mathrm{HC_{min}}$, for example, from 138.4 K–314 K in DDR3 to 20 K–80 K in DDR4 across different vendors [36].

DDR5 DRAM is equipped with ODECC, which increases the $\mathrm{HC_{min}}$ as its error correction capabilities need to be bypassed prior to observing any bit flips. Gloor et al. [1] were the first to measure the $\mathrm{HC_{min}}$ over 128 rows of two DDR5 RDIMMs from Samsung and Micron by suppressing `REFs` using a custom fault injector. They found that both devices have a lowest $\mathrm{HC_{min}}$ of around 32 K, which is similar to DDR4 devices. Therefore, they attributed the increased Rowhammer resistance to improved mitigations.

**Patterns.** Various access patterns have been shown to trigger Rowhammer [4], [16], [36], [37], [38], [39]; however, the most effective one remains the *double-sided* pattern, which activates two aggressor rows on either side of the victim row [28]. To bypass in-DRAM Rowhammer mitigations, state-of-the-art *non-uniform* patterns hammer different aggressors of a pattern more or less often [3], [30] while using *decoy* rows [40] to trick the mitigation from sampling those rows instead of the aggressors. For the first time, *Zenhammer* [6] reported bit flips on DDR5 DIMMs using non-uniform patterns, but only on one of ten tested DIMMs.

**Refresh Synchronization.** SMASH [29] showed that carefully placing `NOPs` in between hammering sequences affects `REF` scheduling, which can help in bypassing TRR mitigations more effectively. Later, Blacksmith [3] adopted this idea by explicitly detecting `REFs` using the bank-conflict side channel before executing each round of its non-uniform patterns, greatly increasing the number of effective patterns and bit flips. More recently, Zenhammer [6] proposed a new *continuous* and *non-repeating* refresh synchronization method that works more reliably on AMD systems and avoids the need for flushing rows during synchronization.

## 2.5. Rowhammer Mitigations

There have been many proposals for mitigating Rowhammer in software [14], [18], [34], [41], [42], [43], [44], [45], [46], [47], inside the MC [11], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], and more recently, inside DRAM itself [40], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68]. Despite these proposals, DRAM vendors rely on proprietary TRR mitigations, whose mechanism are undisclosed. At the very high level, TRR tries to detect aggressor rows in its sampler, and proactively refresh the victim rows during the available slack in the standard refresh commands. Earlier work on DDR4 [3], [4], [5] proved that these TRR mitigations are insecure and can be bypassed.

**Reverse Engineering TRR.** Previous work reverse-engineered aspects of DDR4 TRR implementations by using the bit flips themselves as a side channel [3], [4]. For example, researchers determined how frequently `REFs` perform TRR by (i) disabling refreshes, (ii) hammering half of $\mathrm{HC_{min}}$, (iii) issuing a single `REF`, and (iv) hammering for the remaining half of $\mathrm{HC_{min}}$. If no bit flips occurred, the issued `REF` must have been a TRR [4]. As this method involves hammering for $\mathrm{HC_{min}}$ times, it may affect the TRR mechanism and conceal observing the actual behavior.

Alternatively, U-TRR [5] relies on retention errors as a side channel for detecting TRRs activity, as shown in Fig. 2. To run an experiment, U-TRR first identifies adjacent rows with similar retention times (❶). It then initializes victim and aggressor rows obtained from the profiling with a data pattern (❷) and resets the TRR-internal state by hammering dummy rows (❸). After waiting for half of the retention time, U-TRR hammers aggressor and dummy rows for multiple rounds, while issuing `REFs` to give TRR opportunities to refresh the victims (❹). Finally, after the remaining retention time of the profiled rows has passed, U-TRR checks the victims for bit flips (❺).

Although recent work shows that RFM is not deployed on recent CPUs with DDR5 [2] and the hammer count is similar compared to DDR4 despite ODECC, there have

**Table 1. DDR5 RDIMMs.** For each DIMM of the major DRAM manufacturer SK Hynix [8], we report its manufacturing date (Mf. Date); size; data transfer rate (Speed); device width (Wd.); and DRAM geometry as number of ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R).

| ID | Mf. Date [Yr-Mth] | Size [GiB] | Speed [MT/s] | Wd. [b] | Geometry #(RK,BG,BA,R) |
|---|---|---|---|---|---|
| $H_0$ | 2023-04 | 64 | 4800 | x4 | 2, 8, 4, $2^{16}$ |
| $H_1$ | 2023-04 | 64 | 4800 | x4 | 2, 8, 4, $2^{16}$ |
| $H_2$ | 2023-04 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ |
| $H_3$ | 2023-04 | 64 | 4800 | x4 | 2, 8, 4, $2^{16}$ |
| $H_4$ | 2023-07 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ |
| $H_5$ | 2023-10 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ |
| $H_6$ | 2024-02 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ |
| $H_7$ | 2024-02 | 32 | 4800 | x4 | 1, 8, 4, $2^{16}$ |
| $H_8$ | 2024-05 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ |
| $H_9$ | 2024-08 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ |

been no reports of DDR5 bit flips on more recent DDR5 devices. This suggests that the TRR mechanisms must have improved substantially. This raises the fundamental question: *are today's DDR5 DRAM devices still vulnerable to Rowhammer?*

## 3. Overview of Challenges

We aim to understand how the newly deployed TRR mechanisms on DDR5 devices operate. Rigorously analyzing these mechanisms allows us to assess their ability to prevent Rowhammer attacks. This is our first **challenge** (🧩):

🧩 **(C1.)** Understand how in-DRAM TRR mechanisms on current DDR5 DIMMs operate.

In § 4, we present the first analysis of TRR mechanisms on DDR5 devices. Due to the complexity of current TRR implementations, we devise a reverse engineering methodology to look at the TRR behavior across tREFIs rather than at individual ACTs. Our analysis covers two DIMMs from SK Hynix ($H_2$, $H_6$ in Tbl. 1), currently the largest DRAM vendor [7], [8]. The results show that the state of their TRR sampling mechanisms is large: the sampling occurs irregularly and repeats after hundreds ($H_2$) or even thousands ($H_6$) of tREFIs. Nevertheless, our experiments expose less frequently sampled tREFIs on both DIMMs, revealing opportunities to hide activations from TRR.

Given these exploitable tREFIs, our next goal is to build Rowhammer patterns that can bypass TRR. These patterns also allow us to easily cross-test other devices from the same vendor without having to reverse engineer their TRR mechanisms first. This motivates our second challenge:

🧩 **(C2.)** Leverage the insights from reverse engineered TRR mechanisms to build effective Rowhammer patterns.

In § 5, we present our new TRR-avoiding Rowhammer patterns that are fully compliant with the JEDEC DDR5 timings [10]. Our patterns are carefully designed to only hammer in selected tREFIs in patterns that consist of 128 ($H_2$) and 2608 ($H_6$) tREFIs. Our FPGA evaluation on 10 RDIMMs shows that they are *all* vulnerable to these patterns and that these patterns are highly effective in triggering bit flips.

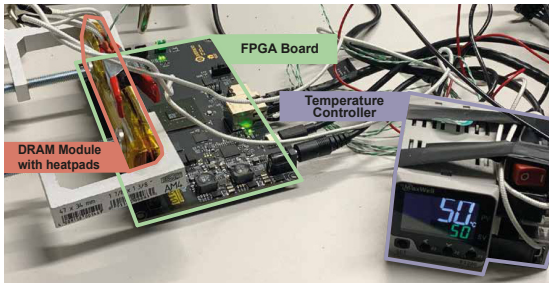


**Figure 3. FPGA testing setup with heating infrastructure.** The Antmicro DDR5 Tester board [69] with the MaxWell FT200 [70] PID controller and custom-size DRAM heater pads.

However, requiring precise synchronization for thousands of refresh intervals immediately raises concerns regarding their feasibility on commodity systems. Furthermore, it is unclear if the same TRR mitigations deployed on RDIMMs are also present on UDIMMs. This leads to our last challenge:

🧩 **(C3.)** Ensure the new Rowhammer patterns remain synchronized over thousands of refresh commands to trigger bit flips in a commodity system.

In § 6, we show how our FPGA patterns can be ported to commodity systems. Our analysis of the state-of-the-art refresh synchronization [6] method shows that it is unable to remain synchronized with a large number of refresh commands. To address this challenge, our new DDR5 Rowhammer attack, called Phoenix, relies on *self-correcting* refresh synchronization that realigns the pattern execution if it detects missed REFs. Our evaluation in § 7 using a commodity system with a default configuration shows that self-correcting refresh synchronization enables Phoenix to successfully trigger bit flips on all 15 SK Hynix UDIMMs from our test pool. We also demonstrate the practical impact of Phoenix by crafting the first end-to-end DDR5 page table exploit that succeeds in as little as 109 seconds.

## 4. Reverse Engineering TRR

In 2024, Zenhammer [6] reported the first Rowhammer bit flips on only one of ten tested DDR5 DIMMs, whereas earlier work with similar patterns triggered bit flips on all 40 tested DDR4 DIMMs [3]. These state-of-the-art fuzzers generate access patterns that span at most 16 tREFIs [3], [6]. This choice is motivated by the assumption that TRR's internal sampling behavior repeats every 16 tREFIs, which we refer to as *sampling period*. Given that the susceptibility to Rowhammer ($HC_{min}$) is similar between DDR4 and DDR5 DRAM [1], we hypothesize that TRR in recent DDR5 devices may have increased the sampling period to make it harder to bypass it. Determining this period is the first step in understanding how to craft a successful hammering pattern. This poses our first **question** (🔍):

🔍 **(Q1.)** After how many tREFIs does the sampling behavior of TRR repeat?

To answer this question, we first *zoom out* in § 4.1: instead of tracking individual ACT commands, we profile TRR activity at refresh-interval granularity to identify the

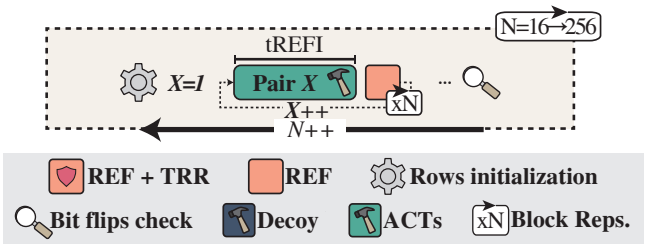


**Figure 4. Zooming-out experiment on $H_2$.**

sampling period. Once this period is known, we examine whether certain tREFI intervals inside the period are sampled less often than others. We call these sparsely monitored regions *light intervals*. This motivates our second question:

**(Q2.)** Within one sampling period, which tREFI intervals are sampled least frequently by TRR?

Having pinpointed the light intervals within a sampling period, in § 4.2 we *zoom in*: we switch our focus to per-ACT granularity, issuing controlled activation sequences exclusively inside a chosen light interval. By observing which activations TRR samples within such a tREFI and which ones it overlooks, we identify which aggressor ACTs can be consistently placed in blind spots that TRR *never* considers. This brings us to our last question:

**(Q3.)** Inside a light interval, can activations be crafted in a way that evade TRR's per-ACT sampling?

**Platform.** For all our RDIMMs experiments, we use Antmicro's RDIMM DDR5 Tester [71]. This FPGA-based memory controller allows us to issue DRAM commands and disable automatic refreshes. The platform keeps a *refresh counter* of the REFs issued since boot, which we use in our experiments. Although we initially faced reliability issues, we resolved them by investing significant debugging effort—including using a high-speed oscilloscope to find bugs in the FPGA's design—so the platform now operates stably for our studies. In Appx. A, we briefly summarize the interesting bugs that we found and are now fixed in the platform's official repository. We also attach a thermocouple sensor with two heater pads around the DDR5 RDIMM, and we use a MaxWell FT200 [70] PID controller to maintain a DRAM temperature of 50 °C during our experiments.

**Methodology.** Similar to the methodology used by previous work on DDR4 [5], we use retention errors as a side channel to infer if rows have been refreshed by TRR. This side-channel allows us to study the TRR behavior for a profiled set of rows after executing a given experiment-specific *DRAM payload*. We implemented the same approach as summarized in § 2.5 on top of the Antmicro platform.

## 4.1. Zooming Out On $H_2$

We address Q1 in this zooming out stage of our reverse-engineering methodology. We start with a candidate length of 16 tREFI intervals—matching the period assumed by state-of-the-art fuzzers—and incrementally increase this length until the sampling period becomes evident. For each tested

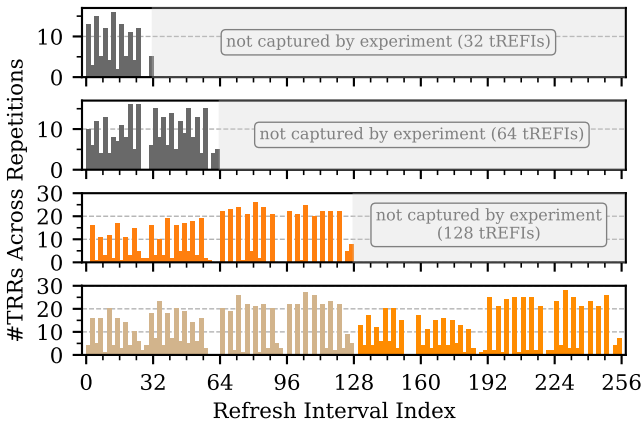


**Figure 5. Zooming out.** The result of brute-forcing the TRR sampling period over 32, 64, 128, and 256 tREFIs, presented using a bin size of 2. We show how often every refresh interval (x-axis) is targeted by TRR over 25 experiment repetitions (y-axis). We can see a repeating sampling behavior in the 256 tREFIs plot.

length $N$, we carefully craft a DRAM payload spanning exactly $N$ consecutive tREFI intervals as described next.

**DRAM payload.** Each of the $N$ intervals in our payload is monitored by a distinct *canary* row, i.e., we need $N$ canaries in total. We perform *double-sided hammering* in each interval: we repeatedly activate the two canary-adjacent rows above and below and then issue one REF before moving to the next interval. All canaries are retention-profiled rows with similar retention times as required by the retention-error side channel. The resulting access pattern, depicted in Fig. 4, is standard-compliant, following the JEDEC DDR5 timing parameter [10]. Once the payload has been executed, we check every canary for retention errors. The experiment is designed such that if a canary survives its hammered tREFI without any retention errors, we can assume that it must have been refreshed by TRR. As the probability that our canary is target of its periodic refresh is negligibly small (once per tREFW, i.e., $1/8192$ REFs), we can safely assume that survived canaries were always refreshed by TRR.

**Analyzing REF counters.** For each experiment, we record the REF counter immediately after the interval. A single run therefore yields a set of REF counter values identifying the TRR-sampled tREFIs. For each candidate period $N$, we repeat the experiment for 25 times. After every run, we take each recorded REF counter modulo $N$ and map all observations into an $N$-slot histogram. If $N$ matches TRR's true sampling period, the histogram should reveal a stable pattern: certain tREFI intervals collect many samples (i.e., are frequently monitored) while others are sampled less often (*light* intervals). We note that if our candidate period $M$ is larger than the true sampling period $N$, we expect to see a repeating pattern.

**Results.** We perform the experiment for every candidate $N$ from 16 to 256 and present the cases $N \in \{32, 64, 128, 256\}$ in this order in Fig. 5. The first two cases, already covering up to four times more than the 16-tREFI window assumed by current fuzzers [3], [6], show no visible regularity. A clear structure appears only at $N = 128$: where a clear

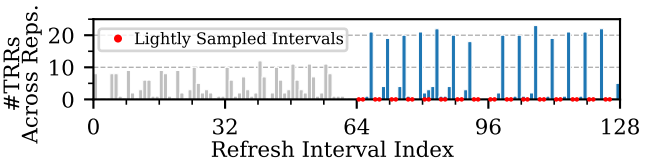


**Figure 6. Zooming in: lightly sampled intervals.** We focus on the last 64 tREFIs (blue) of the 128-tREFI period determined in Fig. 5. This region contains intervals where fewer TRRs happen. We study these lightly sampled intervals (red dots) to identify how often TRR samples them.

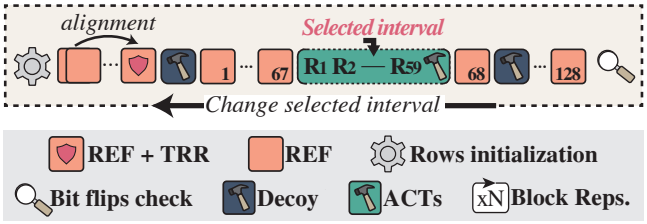


**Figure 7. Zooming-in experiment on $H_2$.**

recurring pattern of four blocks becomes visible. Doubling the candidate length $N$ to 256, lets the same pattern reappear in the upper 128-tREFI slice (128–255), confirming that it repeats every 128 tREFI intervals. This answers Q1 and is our first **observation** (👁):

👁 **(O1.)** TRR on $H_2$ operates with a sampling period of 128 tREFI intervals.

## 4.2. Zooming In On $H_2$

To gain a deeper understanding of TRR's sampling strategy, we must study how sampling works within the 128-tREFI sampling period (Fig. 6). In the first 64 tREFIs, no consistent sampling pattern is observable. The last 64 tREFIs, however, exhibit a pronounced regularity: in every group of four consecutive refresh intervals, TRR samples most in the fourth interval, less in the third, and almost never in the first two intervals. We focus on these first two *lightly sampled* tREFI*s* (see Fig. 6), of which there are 32 in total. Their low sampling probability makes them an attractive target for hammering *aggressor* rows. However, before hammering, we need to understand sampling in these intervals better as TRR still samples these intervals occasionally. We now *zoom in* on each lightly sampled tREFI and profile its behavior at ACT granularity. The goal of this fine-grained analysis is to pinpoint the exact activation slots that TRR samples in this specific interval. Investigating this, will answer Q3.

**Profiling ACTs.** To measure TRR's per-activation sampling across *all* lightly sampled intervals, we craft a DRAM payload that spans over a whole 128-tREFI period. This payload is illustrated in Fig. 7. Prior to the experiment execution, we issue REF commands until the internal refresh counter is aligned with the start of the sampling period, so the 32 lightly sampled tREFIs appear at identical offsets in every run. We issue 59 ACTs in each light interval to the rows adjacent to a fixed set of 59 canaries that we have profiled before. Thus, every light interval activates the same

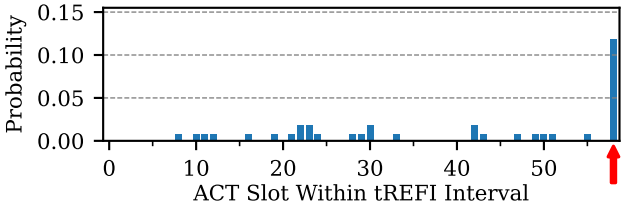


**Figure 8. Zooming in: ACT-based analysis.** We show for each possible activation slot (x-axis) inside the lightly sampled tREFI intervals (y-axis), the probability that TRR samples it. We can see that the last slot (58) is sampled significantly more often than the others, which is why we avoid hammering it.

aggressors in the same order. The remaining 96 intervals are filled with continuous hammering of a double-sided, fixed decoy aggressor pair. By monitoring which canaries TRR refreshes, we reveal the exact activation slots inside each light interval that are sampled by TRR.

**Results.** We execute the 128-tREFI payload 100 times and, for every activation slot (0–58), record how often the corresponding canary is refreshed. Dividing these counts by 100 yields the empirical refresh probability per slot. Each run also issues 128 REF commands from the *regular refresh schedule*, and—because TRR samples the lightly-sampled intervals only infrequently—refreshes from this schedule introduce a non-negligible bias. With a refresh window of 8192 REF commands, the chance that a slot is refreshed by the regular schedule in a single run is $1/8192 \approx 0.012\,\%$. Over 100 runs this baseline accumulates to $1-(1-1/8192)^{100} \approx 1.2\,\%$.

We subtract this $1.2\,\%$ baseline from the measured probabilities to isolate TRR's contribution. The data reveal a single, pronounced spike at slot 58 (Fig. 8), suggesting that TRR almost exclusively samples the *last* ACT issued before the next REF while ignoring earlier activations within a lightly sampled interval.

👁 **(O2.)** In lightly sampled intervals, TRR samples the last issued ACT before the next REF.

We now know which tREFIs in the 128-tREFI sampling period are rarely sampled by TRR. We also know which activation slots to avoid—or fill with decoys—should those intervals be sampled. With this information, we can build effective TRR-bypassing patterns for $H_2$, which we present in § 5.

## 4.3. Generalization & Summary

Our results raise the question if our observations also apply to other DIMMs (i.e., models) from the same vendor. Answering this can help to assess if deployed TRR implementations are the same across different DIMMs, which would mean that they are also vulnerable to the same Rowhammer patterns. To this end, we applied the same methodology on $H_6$ from SK Hynix (Tbl. 1) and found that the TRR implementation is different. We refer to Appx. B for the details of our findings, which we summarize in Tbl. 2. As we will show later (§ 5.3), the TRR implementations of these two DIMMs already cover all our ten DDR5 RDIMMs.

**Table 2. Summary: reverse engineering.** We summarize the key findings from our DDR5 reverse-engineering experiments. We report the sampling period, the $HC_{min}$ over all tested rows, and the measured refresh window size.

| | SK Hynix $H_2$ | SK Hynix $H_6$ |
|---|---|---|
| **Sampling Period** [tREFI] | 128 | 2608 |
| **tREFW Size** [tREFIs] | 11054 | 16582 |
| **HCmin** [min/max/avg] | (56.6 / 150.4 / 105.1) K | (29.3 / 70.3 / 51.4) K |

In the next section (§ 5), we will use these findings to build custom Rowhammer patterns that bypass the TRR mechanism. However, to better understand the time constraints and device vulnerability when hammering these patterns, we need to first determine the refresh window (tREFW) size and the minimum hammer count of our two DDR5 DIMMs.

**tREFW Size.** Previous work [5, §6.1.3] showed that the tREFW size of one of the devices is smaller than half of the typically assumed size (i.e., 8192 `REFs`). We replicate this experiment on our DIMMs and find that the tREFW size is 11 054 and 16 582 `REFs` for DIMMs $H_2$ and $H_6$, respectively. This is 1.3× and 2.0× more than commonly assumed. This is to the benefit of the attacker as it allows hammering for longer before the targeted row is refreshed.

👁 **(O3.)** The periodic row refresh of $H_2$ and $H_6$ is 1.3× and 2.0× larger than 8192 `REFs` (tREFW), respectively.

In § 4.2, we assumed tREFW = 8 192 when calculating the baseline `REF` probability. For window size $W$, the per run probability that the regular schedule refreshes a slot is $1/W$. Hence, over 100 runs the baseline refresh probability is $1-(1-1/W)^{100}$. With $W = 11\,054$ this is $\approx 0.90\,\%$ (vs. $1.2\,\%$ for $W = 8\,192$). Considering this adjustment results in a similar figure to Fig. 8: a single spike at slot 58 and near zero elsewhere.

**$HC_{min}$.** We determine the minimum hammer count ($HC_{min}$) of our two DIMMs over four data patterns (`0x0`, `0xff`, `0x33`, `0xcc`) following the existing methodology [36]. We report in Tbl. 2 the minimum, maximum, and average combined over all four patterns. The minimum is about in the same range as the minimum numbers reported for different DDR4 DIMMs, between 20 K and 80 K [36].

👁 **(O4.)** The $HC_{min}$ of our DIMMs, $H_2$ and $H_6$, is in the same range as for modern DDR4 devices.

## 5. Bypassing TRR from the FPGA

Based on the insights from §4, we now build memory access patterns that defeat TRR. §5.1 introduces two patterns, $\mathcal{P}_{128}$ and $\mathcal{P}_{2608}$, based on the reverse-engineered DIMMs, $H_2$ and $H_6$. §5.2 characterizes these patterns, laying the groundwork for running them on commodity systems. §5.3 evaluates the patterns across all our SK Hynix RDIMMs, demonstrating that every tested device is vulnerable.

**Experimental Setup.** During all experiments and while hammering, the memory controller issues one `REF` command every tREFI (3.9 µs). All access patterns respect the JEDEC DDR5 timing constraints [10]. Before each run, aggressor rows are initialized with random data and their corresponding victim rows with the bitwise inverse.

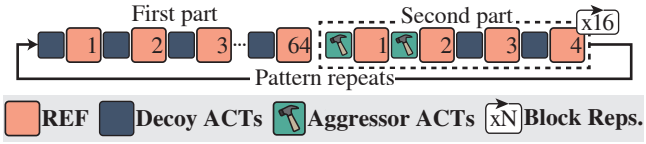


**Figure 9. Hammering pattern for DIMM $H_2$.**

**Algorithm 1:** SK Hynix pattern ($H_2$), 128 tREFIs.

```
1 for i = 1...16 do                                  ▷ 64 tREFIs
2     HAMMER_REF(decoys, 4 tREFI)
3 for i = 1...16 do                                  ▷ 64 tREFIs
4     HAMMER_REF(aggressors, 2 tREFI)
5     HAMMER_REF(decoys, 2 tREFI)
```

### 5.1. Rowhammer Patterns

Both patterns combine two interval types revealed by the reverse-engineering analysis in § 4. In a *light* interval, we hammer the aggressor pair for nearly the entire tREFI. Immediately before the next `REF`, we issue a decoy access as TRR appears to focus its sampling on the last activation (§4.2). In all other intervals, we hammer a fixed double-sided decoy row pair.

**Pattern $\mathcal{P}_{128}$ ($H_2$).** We showed that every 128 tREFIs contain a 64-tREFI period with 32 lightly sampled intervals (§ 4.1). The pattern $\mathcal{P}_{128}$ aligns with this sequence: the four-interval block is executed 16 times to cover the 64-tREFI window. The remaining 64 tREFIs are sampled more frequently, and therefore, we access only a fixed decoy pair. The pattern is visualized in Fig. 9 and described in Alg. 1.

**Pattern $\mathcal{P}_{2608}$ ($H_6$).** The pattern $\mathcal{P}_{2608}$ takes the same approach as $\mathcal{P}_{128}$ to hammer in lightly sampled intervals only. However, because of its longer pattern length—2608 tREFIs —the pattern's structure is more complex. The pattern is described in detail as pseudocode in Alg. 2.

### 5.2. Pattern Characterization

As shown in §4.1, TRR does not sample every tREFI interval in the same way. Our Rowhammer patterns therefore are effective only at specific refresh-counter offsets. On an FPGA, we can issue extra `REF` commands to align the counter with the offset before hammering the pattern. A commodity CPU lacks this capability, so we enumerate all vulnerable offsets to estimate the probability that a run starting at an arbitrary `REF` lands at a vulnerable offset.

**Vulnerable offsets.** To identify the refresh-alignment offsets at which a pattern triggers bit flips, we scan the entire refresh counter range modulo the pattern length. At each offset, we hammer up to 1024 rows in the current bank and stop as soon as a bit flip is observed; if any bit flips, we record the offset as vulnerable and advance to the next one. This procedure is repeated for every bank to check whether the set of vulnerable offsets is bank-specific or shared.

**Algorithm 2:** SK Hynix pattern ($H_6$), 2608 tREFIs.

```
for k = 1...8 do                                          ▷ 1288 tREFIs
    for j = 1...5 do
        HAMMER_REF(decoys, 5 tREFI)
        HAMMER_REF(aggressors, 3 tREFI)                   ▷ light interval
        HAMMER_REF(decoys, 24 tREFI)
    HAMMER_REF(decoys, 1 tREFI)                           ▷ Shift after 160 tREFIs
HAMMER_REF(decoys, 16 tREFI)
for k = 1...8 do                                          ▷ 1288 tREFIs
    for j = 1...5 do
        HAMMER_REF(decoys, 21 tREFI)
        HAMMER_REF(aggressors, 3 tREFI)                   ▷ light interval
        HAMMER_REF(decoys, 8 tREFI)
    HAMMER_REF(decoys, 1 tREFI)                           ▷ Shift after 160 tREFIs
HAMMER_REF(decoys, 16 tREFI)
```

**Table 3. Pattern characteristic: refresh alignment.** For each pattern, we report the number of vulnerable refresh alignment offsets; and the probability to start hammering at one of them both without and with our pattern optimization.

| | $\mathcal{P}_{128}$ ($H_2$) | $\mathcal{P}_{2608}$ ($H_6$) |
|---|---|---|
| **#Vulnerable Refresh Offsets** | 2 / 128 | 92 / 2608 |
| **Hit Probability** [%] | 1.56 | 3.53 |
| **Hit Probability Optim.** [%] | 24.96 | 56.48 |

**Results.** Sweeping all 128 REF offsets on $H_2$ with $\mathcal{P}_{128}$ and all 2608 offsets on $H_6$ with $\mathcal{P}_{2608}$ yields two insights. First, vulnerable alignments are sparse: 2 of 128 for $H_2$ and 92 of 2608 for $H_6$. Second, the same offsets recur in every bank. Tbl. 3 summarizes the counts and the probability to hit a vulnerable refresh alignment.

👁 **(O5.)** $\mathcal{P}_{128}$ has 2 / 128 vulnerable REF offsets (i.e., 1.56 %), while $\mathcal{P}_{2608}$ has 92 / 2608 (i.e., 3.53 %); in both cases, the same offsets recur in every bank.

**Pattern optimizations.** As our patterns need to be aligned to certain vulnerable REF offsets, we optimize the pattern execution to increase the chance of hitting a vulnerable offset. In essence, we run four shifted instances of the pattern on *each* of four banks in parallel, thus increasing the success chance by a factor of 16. This is illustrated in Fig. 10 and explained in more detail in Appx. C.

👁 **(O6.)** We can increase the chance of hitting a vulnerable REF offset by running four shifted instances of the pattern in parallel on each of the four banks.

**Activation rate vs. hammer duration.** Besides the refresh alignment, the effectiveness of our hammering patterns is limited by the number of activation commands we can issue to aggressors in a tREFI. Commodity CPUs may achieve a lower activation rate than FPGAs because of the cache-flushing overhead and memory-controller scheduling [6]. Therefore, we want to *estimate* how much slack our patterns permit by reducing the number of aggressor ACTs per tREFI.

We sweep over 1024 victim rows in a randomly selected bank at a refresh alignment previously identified as vulnerable. Within each hammered tREFI, we issue 30, 40, or 50 ACTs to the aggressor rows; and fill the remaining slots of the 59 ACTs/tREFI with decoy accesses, whereas non-hammered tREFIs contain only decoys. The pattern is

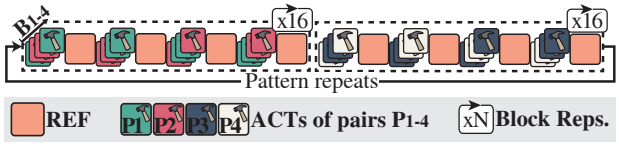


**Figure 10. Optimized $\mathcal{P}_{128}$ pattern.** We hammer four shifted pattern instances on each of four banks in parallel to increase the chance of hitting a vulnerable REF offset by a factor of 16.

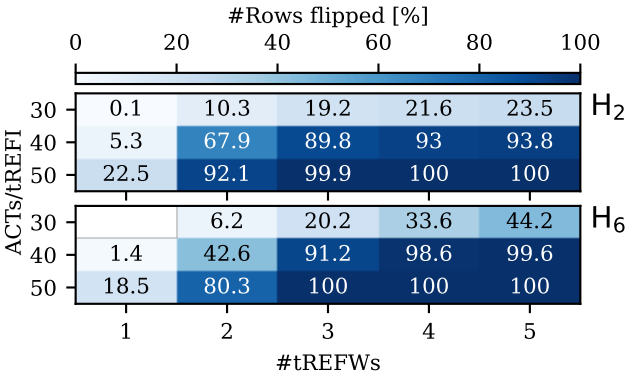


**Figure 11. Activation rate vs. hammering duration.** We show for different activation budgets (ACTs/tREFI) and hammering durations (#tREFWs), the share of rows (%) with at least one bit flip.

applied for one to five consecutive tREFWs since §4.3 shows that rows are refreshed less often than once per tREFW.

**Results.** Fig. 11 shows the share of rows that incur at least one bit flip when we hammer $\mathcal{P}_{128}$ on $H_2$ and $\mathcal{P}_{2608}$ on $H_6$, sweeping the activation budget (30–50 ACTs/tREFI) and the hammering duration (1–5 tREFWs) over 1024 victim rows in a randomly selected bank.

Even with a slack of almost 50 % in the activation budget (30 ACTs/tREFI), both patterns still flip rows: after five tREFWs, 23.5 % of the rows flip on $H_2$ and 44.2 % on $H_6$. Raising the budget to 40 ACTs/tREFI markedly increases the success rate: three tREFWs already flip 89.8 % ($H_2$) and 91.2 % ($H_6$) of the rows, and four tREFWs exceed 93 % for both devices. At the highest budget, 50 ACTs/tREFI, nearly all rows flip within two tREFWs and full (100 %) row coverage is achieved after three tREFWs. Given these results, we will focus in future experiments on the more efficient range, i.e., between 40 and 50 ACTs/tREFI.

👁 **(O7.)** Even with a reduced activation budget of 30–40 ACTs/tREFI, both $\mathcal{P}_{128}$ on $H_2$ and $\mathcal{P}_{2608}$ on $H_6$ induce bit flips when the pattern is synchronized to a vulnerable REF offset and executed for several consecutive tREFWs.

**Discussion.** We attribute the continuing rise of flipped rows across successive tREFWs to two effects. First, a row is most vulnerable when hammering starts immediately after its periodic refresh, providing the longest time to hammer. Extending the hammering duration makes it increasingly likely that every row eventually encounters this worst-case point in the refresh cycle. Second, ODECC does not write back corrected data to DRAM when reading from memory. Hence, a bit that flips in one tREFW will persist in the next one. If a second bit flips in the same ECC word, the

**Table 4. DDR5 RDIMMs: pattern coverage and effectiveness**. For each DIMM, we report the average $HC_{min}$ and the percentage of rows with at least one bit flip when hammering the pattern for one up to five refresh windows (tREFWs).

| ID | $HC_{min}$ [x1000] | Patterns $\mathcal{P}_{128}$ | Patterns $\mathcal{P}_{2608}$ | tREFWs / Flipped Rows [%] 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| $H_0$ | 108.4 | ✓ | ✗ | 6.4 | 63.7 | 86.2 | 95.1 | 96.6 |
| $H_1$ | 99.2 | ✓ | ✗ | 13.6 | 86.6 | 98.9 | 99.8 | 100.0 |
| $H_2$ | 105.1 | ✓ | ✗ | 22.5 | 92.1 | 99.9 | 100.0 | 100.0 |
| $H_3$ | 98.3 | ✓ | ✗ | 19.2 | 93.3 | 99.9 | 100.0 | 100.0 |
| $H_4$ | 114.0 | ✓ | ✗ | 4.9 | 53.6 | 77.1 | 88.6 | 93 |
| $H_5$ | 51.6 | ✗ | ✓ | 13.3 | 82.4 | 99.3 | 99.9 | 100.0 |
| $H_6$ | 51.4 | ✗ | ✓ | 18.5 | 80.3 | 100.0 | 100.0 | 100.0 |
| $H_7$ | 93.6 | ✓ | ✗ | 21 | 91.4 | 99.4 | 99.7 | 100.0 |
| $H_8$ | 104.4 | ✓ | ✗ | 14.4 | 86.3 | 98.2 | 99.6 | 99.8 |
| $H_9$ | 52.7 | ✗ | ✓ | 3.6 | 63.9 | 97.3 | 99.9 | 100.0 |

Pattern $\mathcal{P}_{128}$ originates from DIMM $H_2$ and $\mathcal{P}_{2608}$ from DIMM $H_6$.

scheme's single-bit error-correction capability is exhausted and the error becomes visible at the memory controller.

👁 **(O8.)** Bit flips can accumulate over hammering multiple tREFWs before they become visible.

### 5.3. Cross-Device Evaluation

We apply our two custom Rowhammer patterns to the SK Hynix DDR5 RDIMMs listed in Tbl. 1. For every module, we rerun the vulnerable offset experiment for each pattern to determine whether the module is susceptible to one or both patterns. Any DIMM that exhibits at least one vulnerable offset is then tested for its effectiveness: starting from an identified offset, we sweep 1024 randomly chosen rows and record how many experience at least one bit flip. We hammer the aggressor rows for 50 ACTs/tREFI, while varying the hammering duration for one to five tREFWs.

**Results.** Tbl. 4 summarizes the cross-device evaluation. Every DIMM responds to exactly one of the two patterns: $\mathcal{P}_{128}$ triggers bit flips on seven modules, whereas $\mathcal{P}_{2608}$ is effective on the remaining three. When the selected pattern is aligned with a vulnerable REF offset and executed at 50 ACTs/tREFI, it reaches near-complete coverage in just a few refresh windows: after three tREFWs, eight of the ten modules already show bit flips in at least 97.3 % of the 1024 probed rows. Extending the hammering to five tREFWs raises the minimum coverage to 93 % across all modules, with seven modules achieving 100 %.

These results reveal that SK Hynix deploys, at most, two functionally distinct TRR implementations in the DDR5 generation from 2021 to 2024 (see Tbl. 1). Thus, reverse engineering only a handful of modules is sufficient to craft patterns that generalize to the entire product line.

👁 **(O9.)** All our 10 SK Hynix RDIMMs share the same TRR mitigation as $H_2$ (7×) and $H_6$ (3×).

**Relation to $HC_{min}$.** We determine the $HC_{min}$ of our RDIMMs, in the same way as we already did in § 4.3 for DIMMs $H_2$ and $H_6$. The result in Tbl. 4 reveals that there is a connection between the pattern's length (i.e., TRR implementation) and the $HC_{min}$ of the DIMMs. Devices with a hammer count lower than 51.4 K (i.e., $H_5$, $H_6$, $H_9$) trigger bit flips only with the longer $\mathcal{P}_{2608}$ pattern, whereas those with a hammer count of at least 93.6 K are vulnerable to the shorter $\mathcal{P}_{128}$ pattern.

👁 **(O10.)** DIMMs with a lower avg. $HC_{min}$ require longer patterns to bypass TRR, whereas DIMMs with a higher avg. $HC_{min}$ can be attacked with shorter patterns.

## 6. Bypassing TRR in Commodity Systems

In our FPGA-based evaluation, we had full control over the DRAM commands. A key novelty of our patterns is that they are longer and need to be repeated over many thousands of refresh intervals. This is crucial to bypass TRR as missing a REF will lead to hammering our aggressors *too late* and erroneously detecting a REF will lead to hammering them *too early*. Unlike on the FPGA, where we can control the REF commands to synchronize, on commodity systems we have no control over when REFs are issued and need to detect REFs to synchronize our pattern execution with them.

⚠ **Assumption.** Having full control over the REF commands to synchronize our pattern execution.

To eliminate this assumption, we first define the requirements of the refresh synchronization method (§ 6.1) and evaluate the state-of-the-art method by Zenhammer (§ 6.2). As this method cannot satisfy our requirements, we propose two novel methods to improve the reliability of detecting REFs (§ 6.3) and to self-correct the pattern's execution whenever we lose synchronization (§ 6.4).

**Platform.** All our system-level experiments are run on an AMD Zen 4 (R7 7700X) system running on Ubuntu 20.04 with GNU/Linux kernel 5.15.0.

**Oscilloscope.** We analyze our experiments using McSee [2], an oscilloscope platform capable of capturing and decoding the DDR5 protocol. This setup is based on a DDR5 interposer sitting in between the memory controller and the DIMM, and a software pipeline for decoding the captured traces. Equipped with this, we can precisely study when REFs happen to assess the quality of our REF synchronization method and correctness of our hammering pattern.

### 6.1. Requirements Analysis

Our patterns rely on precise synchronization with REF commands to ensure hammering specific tREFIs only. This requirement is not new as previous work [3], [6], [29], [30] already employed synchronization. However, they synchronized the pattern once per pattern hammering repetition only [3], which is insufficient for the significantly longer patterns that can trigger bit flips on DDR5 devices. Given the intricate nature of such patterns that hammer at very specific refresh intervals, we *must* synchronize with *every* tREFI. Furthermore, since these patterns hammer on comparably fewer refresh intervals inside the patterns, they need to remain synchronized with REFs for longer durations to reach a desired HC. Finally, due to tREFI being halved on DDR5 (compared to DDR4), these patterns need to

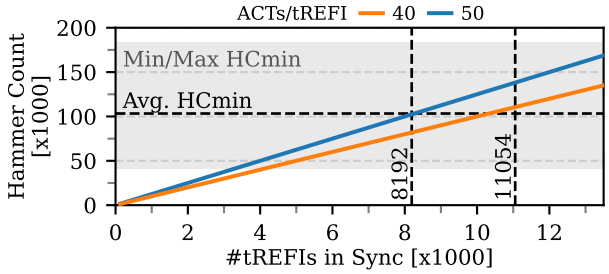


**Figure 12. Required #tREFIs in sync for pattern $\mathcal{P}_{128}$.** We show the hammer count (y-axis) achievable with a number of tREFIs in sync (x-axis). Each line represents a different activation rate (ACTs per tREFI). The $HC_{min}$ values are based on the three DIMMs vulnerable to $\mathcal{P}_{128}$, see § 5.3. We show the tREFW according to the standard (8192) and as measured (11054) in § 4.3.

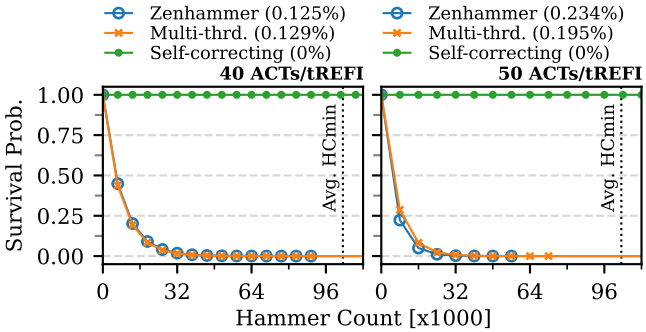


**Figure 13. Comparison of refresh synchronization methods.** We compare Zenhammer's synchronization against a multi-threaded variant of it, and our self-correcting synchronization. We show for two activation rates, the probability of reaching a hammer count (x-axis) given the synchronization's methods error rate (legend). We omit line markers where the probability is below 0.001 %.

synchronize with REF commands twice as often, Therefore, we need a synchronization method that detects REFs with high reliability. Before evaluating the existing method, we analyze for how long our pattern needs to stay in sync.

We aim to answer the following question: *how many* tREFI*s do we need to stay in sync when hammering our pattern to trigger any bit flips?* For the calculation, we consider the $\mathcal{P}_{128}$ pattern and the average $HC_{min}$ of 103.3 K, based on the three DIMMs vulnerable to this pattern (§ 5.3). As we visualize in Fig. 12, we need an activation rate of at least 50 ACTs/tREFI to reach the average $HC_{min}$ within a tREFW (i.e., 8192 REFs). The $HC_{min}$ is based on our devices vulnerable to $\mathcal{P}_{128}$ (see § 5.3). To the attacker's benefit, we determined earlier (§ 4.3) that the tREFW is actually 11054 REFs. This means, we could hammer for longer with a lower rate ($\approx$ 38 ACTs/REF) to reach the same average $HC_{min}$. This trade-off provides us with some flexibility: we can hammer with a lower activation rate in exchange for a more reliable refresh synchronization method. We will explore next if the state-of-the-art refresh synchronization method meets our minimum requirements.

## 6.2. Zenhammer Refresh Synchronization

Our goal is to evaluate how well the state-of-the-art Zenhammer [6] refresh synchronization works and if it allows us to stay in sync sufficiently long to trigger bit flips. Zenhammer's *continuous* and *non-repeating* refresh synchronization method employs precise timing measurements with multiple rows to eliminate "blind spots" and avoid cache flushes.

**Experiment.** We integrate Zenhammer's synchronization method into the code of our $H_2$ hammering pattern and use it to synchronize with *every* REF. To facilitate the analysis, we access two specific *marker rows* whenever we detect a REF. We run the hammering code on our test platform and capture 50 traces of each 2 ms (i.e., around 25 K tREFIs) with McSee. As we found that we need an activation rate between 40 and 50 to reach the average $HC_{min}$ before a row gets periodically refreshed, we repeat the experiment with both activation rates.

**Results.** In Fig. 13, we show the probability of staying in sync (y-axis) over time, expressed as hammer count (x-axis), for both activation rates in different subplots. We can see that, with neither 40 nor 50 activations per tREFI, the Zenhammer refresh synchronization is reliable enough to stay in sync for the necessary time to reach the average $HC_{min}$ of our DIMMs vulnerable to $\mathcal{P}_{128}$. In the best case (40 ACTs/tREFI), where the error probability is the lowest, we can reach a HC of 35 840, 55 040, and 72 960, before the probability of staying in sync drops below 1 %, 0.01 %, and 0.001 %, respectively. Given the survival probability $p$ of 0.000169 % to reach the average $HC_{min}$ of 103.3 K, we would need to repeat hammering the pattern for around 591.7 K times to succeed once, which makes the attack infeasible in practic in practicee.

> 👁 **(O11.)** The state-of-the-art refresh synchronization method is insufficient to keep in sync with refreshes for long enough to reach the $HC_{min}$ of vulnerable rows.

**Directions.** Our analysis of many DRAM traces with Zenhammer's refresh synchronization revealed that the most common reason for losing synchronization is because of missing just a *single* REF. If this happens, we hammer the pattern's interval $i$, although we should have skipped one interval, i.e., hammer the interval $i+1$ to not get sampled by TRR. Therefore, our goal is to optimize the synchronization method for this single-missed REF case. To tackle this, we came up with two possible directions: (i) improving the reliability of detecting REFs and (ii) realigning the pattern's execution whenever we detect a missed REF. We consider both directions: we propose a new multi-threaded variant of Zenhammer's refresh synchronization (§ 6.3) and a new self-correcting synchronization method (§ 6.4).

## 6.3. Multi-Threaded Refresh Synchronization

We present a multi-threaded variant of Zenhammer's refresh synchronization method [6], aiming to improve the reliability of detecting REFs. This is motivated by the assumption that single-threaded refresh synchronization will inevitably miss a REF at some point. Missing a REF can be caused by, for example, system events such as receiving an interrupt or being preempted by the kernel. The synchronization detection itself can also fail, for example, if another process

accesses the same bank as the one used for synchronization. We propose to separate activations used for hammering from those for synchronizing to increase the reliability of detecting REFs. Therefore, our multi-threaded refresh synchronization uses several synchronization threads that work in parallel on different banks and different cores.

**Approach.** We use three threads, each pinned to a different core and solely responsible for detecting refreshes. A fourth, *main* thread hammers in the pattern's respective intervals and thereafter, checks if the synchronization threads detected a REF. If two out of three threads detected a REF within 0.12 µs (3 % of tREFI) of each other, we consider the REF as genuine and continue with the pattern's execution.

**Results.** Fig. 13 shows the result of our multi-threaded refresh synchronization method. We can see that for 40 ACTs/tREFI, the error probability is similar to Zenhammer (0.129 % vs 0.125 %). For the case of 50 ACTs/tREFI, the error probability is by 0.039 percentage points lower but still higher than the best case with the Zenhammer method. Given this, we conclude that the refresh synchronization method is not practical to reach the average $HC_{min}$.

👁 **(O12.)** Multi-threaded refresh synchronization only slightly improves the reliability compared to the state-of-the-art method and only at a higher activation rate.

### 6.4. Self-Correcting Refresh Synchronization

Orthogonal to making the synchronization more reliable, we came up with a completely different approach: self-correcting refresh synchronization. The idea is that instead of trying to avoid missing REFs, which seems to be unrealistic to achieve in practice, we accept that we will miss some REFs and instead adjust the pattern execution whenever a REF is missed. The implementation is simple as we already retrieve a timestamp during synchronization that we can use for detecting one or multiple missed REFs.

**Approach.** Our approach for self-correcting refresh synchronization is illustrated in Fig. 14: we show how a perfect refresh synchronization would execute the pattern **(a)**, compared to the naive refresh synchronization **(b)**, and our novel self-correcting refresh synchronization **(c)**.

After detecting a REF, we compare the rdtscp timestamp of the last detected REF with the current timestamp. Based on the difference, we check if we missed any REFs, and if so, how many refresh intervals of the pattern we need to skip to realign the execution. To optimize for the most common case of missing a single REF only, we choose the threshold in a way that we avoid false positives despite REFs sometimes being slightly delayed. For example, given a threshold of 5.5 µs (i.e., 1.4× tREFI), a smaller timestamp difference will be considered as no missed REF and a larger one as *one* missed REF.

**Evaluation.** We repeat the same experiment (§6.2) using our self-correcting refresh synchronization method. We take 50 traces (each 2 ms) for each activation rate using our scope. The results in Fig. 13 show that the self-correcting refresh synchronization method makes the overall pattern execution significantly more reliable. From the 9 and 58 missed REFs

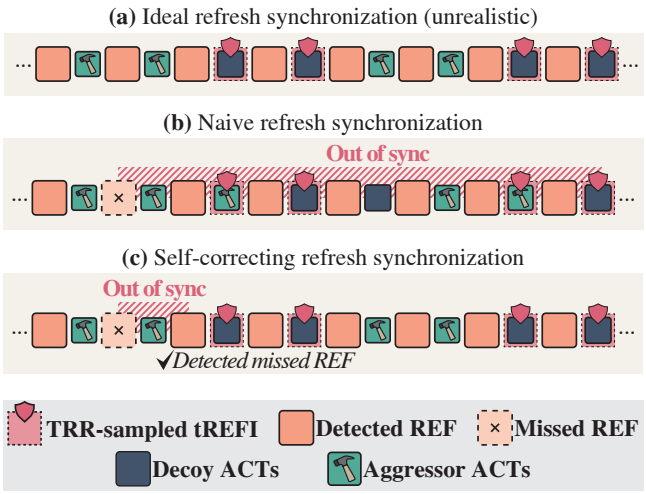


**Figure 14. Self-correcting refresh synchronization.** The ideal pattern execution where each REF is correctly detected **(a)** is compared to the execution if one REF is missed **(b-c)**. In the naive synchronization case **(b)**, the execution is continued and remains out of sync, causing the aggressors to be sampled by TRR. With our self-correcting approach **(c)**, we detect the missed REF and skip one or multiple tREFIs to restore the synchronization.

for 40 and 50 ACTs/tREFI, respectively, all of them were detected correctly, and the pattern execution was realigned. This allows us to stay in sync for many thousands of tREFIs, thus easily reaching the average $HC_{min}$ of our DIMMs. As we show later in § 7, our novel self-correcting refresh synchronization method enables us for the first time to trigger bit flips on modern DDR5 DIMMs from commodity systems.

👁 **(O13.)** Self-correcting refresh synchronization allows us to stay in sync for many thousands of tREFIs, sufficient to reach the average $HC_{min}$ of vulnerable rows.

## 7. Evaluation

In this section, we evaluate Phoenix by answering the following three questions:

1) How widely applicable are our patterns across different DDR5 UDIMMs from SK Hynix (§ 7.1)?
2) Can our patterns trigger the particular bit flips on DDR5 DIMMs that enable existing attacks (§ 7.2)?
3) Is it possible to build an end-to-end Rowhammer exploit on DDR5 DRAM with our patterns (§ 7.3)?

**Experimental Setup** We evaluate our patterns and the end-to-end attack on an AMD Zen 4 (R7 7700X) system with Ubuntu (GNU/Linux 5.15.0) and default BIOS settings. We use the UDIMMs listed in Tbl. 5. Testing these DIMMs with the state-of-the-art Rowhammer fuzzer, Zenhammer [6], did not show any bit flips.

### 7.1. Pattern Coverage & Effectiveness

We evaluate on how many of our DDR5 UDIMMs our crafted patterns are effective and how many bit flips they can

**Table 5. Evaluation of DDR5 UDIMMs.** For each DIMM, we report its manufacturing date (Mf. Date); size; data transfer rate (Speed); device width (Wd.); DRAM geometry (number of ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R)); and the Rowhammer pattern it is vulnerable to (§ 5). For each DIMM, we sweep its pattern over 256 MB of memory and report the number of one-to-zero (1→0) and zero-to-one (0→1) bit flips. We also report the average time to find the first exploitable bit flip in mm:ss format for three Rowhammer attacks: the page table entry (PTE) attack [14], the RSA key attack [27], and the sudo binary attack [15].

| ID | Mf. Date [yyyy-mm] | Size [GiB] | Speed [MT/s] | Wd. | Geometry #(RK,BG,BA,R) | Pattern $\mathcal{P}_{128}$ | Pattern $\mathcal{P}_{2608}$ | Sweep [#bfs.] 1→0 | Sweep [#bfs.] 0→1 | Attacks [avg. time] PTE | Attacks [avg. time] RSA | Attacks [avg. time] sudo |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $H_0$ | 2021-12 | 8 | 4800 | x16 | 1, 4, 4, $2^{16}$ | ✓ | ✗ | 3 329 | 5 460 | 10 s | 4 m 57 s | 80 m 45 s |
| $H_1$ | 2022-07 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ | ✓ | ✗ | 2 917 | 4 582 | 14 s | 7 m 6 s | 49 m 37 s |
| $H_2$ | 2022-08 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ | ✓ | ✗ | 2 629 | 4 082 | 15 s | 6 m 38 s | – |
| $H_3$ | 2022-12 | 8 | 4800 | x16 | 1, 4, 4, $2^{16}$ | ✓ | ✗ | 2 837 | 4 526 | 11 s | 6 m 36 s | – |
| $H_4$ | 2022-12 | 32 | 5200 | x8 | 2, 8, 4, $2^{16}$ | ✗ | ✓ | 207 | 318 | 2 m 55 s | – | – |
| $H_5$ | 2022-12 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ | ✓ | ✗ | 3 922 | 5 821 | 11 s | 4 m 17 s | – |
| $H_6$ | 2023-01 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ | ✓ | ✗ | 6 592 | 9 672 | 7 s | 2 m 29 s | 20 m 19 s |
| $H_7$ | 2023-01 | 32 | 4800 | x8 | 2, 8, 4, $2^{16}$ | ✗ | ✓ | 566 | 860 | 39 s | – | – |
| $H_8$ | 2023-01 | 32 | 5600 | x8 | 2, 8, 4, $2^{16}$ | ✗ | ✓ | 156 | 240 | 4 m 6 s | – | – |
| $H_9$ | 2023-01 | 32 | 6000 | x8 | 2, 8, 4, $2^{16}$ | ✗ | ✓ | 314 | 486 | 2 m 17 s | 9 m 25 s | – |
| $H_{10}$ | 2023-02 | 32 | 5600 | x8 | 2, 8, 4, $2^{16}$ | ✗ | ✓ | 304 | 461 | 7 m 27 s | 23 m 57 s | – |
| $H_{11}$ | 2024-01 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ | ✗ | ✓ | 12 523 | 18 446 | 6 s | 1 m 19 s | 12 m 41 s |
| $H_{12}$ | 2024-01 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ | ✓ | ✗ | 10 833 | 15 917 | 5 s | 1 m 27 s | 21 m 17 s |
| $H_{13}$ | 2024-04 | 16 | 5600 | x8 | 1, 8, 4, $2^{16}$ | ✓ | ✗ | 8 520 | 12 761 | 5 s | 1 m 38 s | – |
| $H_{14}$ | 2024-12 | 16 | 4800 | x8 | 1, 8, 4, $2^{16}$ | ✗ | ✓ | 24 | 19 | 20 m 15 s | – | – |

trigger. This can show us if the TRR mitigations deployed on RDIMMs and UDIMMs are the same. As our UDIMM test pool covers devices with manufacturing dates between end of 2021 and end of 2024, it will also show us if the mitigation changed over time.

**Methodology.** For each DIMM, we first determine which of the two patterns, $\mathcal{P}_{128}$ or $\mathcal{P}_{2608}$, are effective in bypassing TRR. We then follow the methodology from previous work [3], [4] and *sweep* the effective pattern over the row-equivalent of 256 MB of physically contiguous memory and record the number of bit flips. While sweeping, we hammer each row for one second, i.e., for more than 15 tREFWs.

**Results.** The results of our pattern sweep is presented in Tbl. 5. For each DIMM, we report the effective pattern, and for this pattern, the number of one-to-zero (1→0) and zero-to-one (0→1) bit flips. Similar as in our FPGA-based evaluation (§ 5.3), we find that every DIMM is vulnerable to exactly one of the two patterns only: $\mathcal{P}_{128}$ (8×) or $\mathcal{P}_{2608}$ (7×). Pattern $\mathcal{P}_{128}$ with 13 050 bit flips on average, is around 2.62× more effective than the $\mathcal{P}_{2608}$ pattern with 4 989 bit flips on average. We will see next how these bits flips affect the attack time of existing Rowhammer end-to-end attacks.

## 7.2. Exploitation Analysis

We now show that the bit flips produced by our patterns enable existing Rowhammer attacks on DDR5 DIMMs by analyzing their exploitability.

**Attacks.** For this exploitability analysis, we consider three existing Rowhammer attacks targeting (i) **PTEs** (Page Table Entries) to craft a memory read/write primitive [14]; (ii) **RSA**-2048 keys of a co-located VM to break SSH authentication [3]; and (iii) the **sudo** binary to escalate local privileges to the root user [3]. We use the Rowhammer attack simulation framework *Hammertime* [72] with our results from § 7.1 to simulate the attacks based on the required attack-specific bit flip offsets.

**Results.** We present the results in Tbl. 5. For each of the three attacks, we report the average time to trigger the first exploitable bit flip. We can see that all 15 devices are vulnerable to the PTE attack and the average time to trigger the first exploitable bit flip is 2 m 36 s. For the RSA-2048 attack, 11 of 15 devices (73 %) are vulnerable, with an average time of 6 m 20 s. Although requiring very specific bit flips offsets, the sudo binary attack still works on 5 of 15 devices (33 %) with an average time of 36 m 55 s. We note that in § 7.1, we limited hammering to 256 MB but it is likely to find the necessary bit flips for the RSA-2048 and sudo binary attack when hammering for longer.

We conclude this section by showing that the exploitable bit flips that we determined can be used to escalate privileges in a real end-to-end attack. As DDR5 introduces ODECC, which could make memory templating more difficult if the attack's target data is not known in advance, such as in the PTE attack, we chose to replicate this specific attack.

## 7.3. End-to-End Exploit

We integrate Phoenix's bit flip primitive into the privilege escalation exploit *Rubicon* [14], which equips the attacker with an arbitrary memory read/write primitive. Flipping a single bit in the Page Frame Number (PFN) field of a PTE, lets the attacker forge PTEs to arbitrarily read/write into memory. Exploitation requires allocating a large buffer of contiguous physical pages, locating a row with an exploitable bit flip, massaging memory so the target PTE resides in that row, and retriggering the flip to complete the escalation.

**Threat model.** We assume a vanilla x86-64 Linux system where the attacker can run code as an unprivileged user. The system uses a DDR5 UDIMM produced by SK Hynix

employing ODECC and TRR with all BIOS defaults and vendor defenses remain enabled.

**Physically contiguous memory.** To obtain 4 MB contiguous memory blocks, we exhaust smaller page block orders to force the buddy allocator to return 4 MB blocks at the tail of the allocation. We then release everything except the 8 MB at the tail of the allocation, creating a search window of at least one physically contiguous 4 MB block. We slide a 4 MB probe across this window and use the DRAM addressing functions [6] with bank conflicts to find the block alignment. Accessing each address pair while recording the minimum latency reveals the alignment: when the probe covers the target block, every pair will trigger a row buffer conflict, marking the first page of the contiguous block.

**ECC-aware templating.** Templating typically fills aggressor and victim rows with complementary byte patterns, but ODECC makes bit flips data-dependent [73]. Therefore, we prime aggressor rows with `0xAAAAAAAAAAAAAAAA` and victim rows with a fake PTE value `0x8000000555555027`. This surrogate PTE approximates the layout of a Linux 64-bit page table entry, increasing the chance that a flip found during templating can be retriggered later. We repeatedly hammer and read back the victim rows until the first exploitable bit flip emerges.

**Memory massaging and exploitation.** After identifying a vulnerable cell, our aim is to install a PTE whose PFN differs from the page table's PFN by exactly one flippable bit. Thanks to the 4 MB contiguous aligned blocks, we can choose a sibling frame whose PFN is one bit apart from the target PFN. We free this frame, then map a file to back the file with that frame. Using Rubicon's allocator massage primitive [14], we steer the next page-table allocation onto the target frame. Mapping the same file again at a crafted virtual address writes the sibling's PFN into the PTE located at the vulnerable 8-byte slot. Immediately before hammering, we clear that slot in the sibling page to `0x0` and read it back afterward. If it now contains a valid PTE, we have control over a page table page; otherwise we repeat the entire sequence.

**Evaluation.** We test our attack on DIMM $\mathrm{H}_{14}$. Across ten successful runs, the time-to-exploitation is between $1\,m$ 49 s and $17\,m$ 6 s with an average of $5\,m$ 19 s.

## 8. Discussion

We discuss extending Phoenix (§ 8.1) and possible mitigation strategies (§ 8.2).

### 8.1. Extending Phoenix

Phoenix was a high effort project: debugging the Antmicro FPGA took us roughly 1 person-year, reverse engineering $\mathsf{H}_2$ and $\mathsf{H}_6$ took 0.5 person-year, and developing self-correcting refresh synchronization took another 0.5 person-year, totaling 2 person-years. However, we think more work is needed in this important area and discuss some interesting future directions with an estimation of effort.

**Additional DIMMs.** $\mathsf{H}_2$ and $\mathsf{H}_6$ provided a full coverage of DDR5 SK Hynix RDIMMs and UDIMMs in our pool. Future devices from this vendor, however, could introduce new TRR mechanisms. In our experience, reverse engineering a new TRR mechanism from the same vendor typically takes a shorter time given similarities in the implementation.

**Additional DRAM vendors.** We expect reverse engineering devices from other vendors, such as Micron or Samsung, to take roughly the same amount of time as it took us for SK Hynix. It remains to be seen whether additional system-level techniques, beyond self-synchronization, will be necessary to trigger bit flips on a commodity system with DIMMs from these other vendors.

**Additional CPU vendors.** Phoenix is designed to work on AMD CPUs. Self-synchronization will likely require additional tuning to work on Intel CPUs. Furthermore, triggering bit flips on Intel CPUs requires bypassing the in-CPU pTRR mitigations as shown by McSee [2]. We are not aware of any work that bypasses both in-DRAM and in-CPU mitigations simultaneously.

**Better fuzzers.** Existing Rowhammer fuzzers [3], [4], [6] fail to find the patterns we crafted in this paper. We attribute this to the TRR sampling behaviors that span over much more refresh intervals in DDR5 which makes the search space significantly larger to brute force. Future fuzzers can explore this space more effectively, for example, by focusing on finding blind spots at the granularity of refresh intervals instead of activations. Such fuzzers could benefit from Phoenix's more accurate refresh synchronization capability.

### 8.2. Mitigations

Mitigating Phoenix requires different strategies when considering existing and future DRAM devices, which we discuss here.

**Increasing refresh rate.** Unlike CPU vulnerabilities that can be mitigated via modifications to the microcode, DRAM features a much more rigid execution pipeline that is not amenable to changes after production. Hence, the only possibility to fully address Phoenix in existing systems is via increasing the refresh rate. Our measurements show that increasing the refresh rate by $3\times$ mitigates Phoenix on the most vulnerable device at the system level. SPEC2017 benchmark suite [9] shows an overhead of 8.4% due to this increase in the refresh rate in our test system. We would like to point out that there could be more effective patterns than the ones we constructed where increasing the refresh rate by $3\times$ may not be sufficient.

**Fine-granularity refresh.** As a response to our responsible disclosure, we were told that a BIOS update for AMD client CPUs has been issued. This BIOS update switches the memory controller's refresh mode to Fine-Granularity Refresh (FGR). FGR increases the refresh rate, but reduces the time allotted to each refresh command [2]. While it remains to be seen how this change affects Phoenix, we do not think that it will provide a strong protection.

**Rank-level ECC.** Rank-level ECC is known to make Rowhammer attacks more difficult in practice. Previous work, however, has shown that it does not stop Rowhammer attacks in both DDR3 [74] and DDR4 [75] server systems.

Investigating an ECC variant of Phoenix would be an interesting direction for future research.

**Future devices.** We strongly recommend against deploying yet another obscure mitigation without a rigorous security analysis in future devices. Instead, the DRAM vendors must deploy TRR mitigations with principled guarantees [40], [60]. The new Per Row Activation Counting specification [76], [77] provides a possibility for implementing such principled mitigations inside DRAM.

## 9. Related Work

We discuss reverse engineering of TRR implementations via FPGAs and system-level attacks that bypass TRR.

**FPGA studies.** TRRespass [4] and U-TRR [5] used an FPGA to study TRR mitigations in commodity DDR4 devices. Similar to U-TRR, we relied on cell retention time as a side channel for detecting TRRs. Our methodology, however, scales to a larger number of refresh intervals, necessary for bypassing TRR on commodity DDR5 devices. Blacksmith [3] tested LPDDR4X devices via a development board to assess the efficacy of non-uniform patterns at bypassing TRR. Half-double [37] studied the effect of far aggressors to victims and their interactions with TRR. To the best of our knowledge, there is no existing study of DDR5 devices based on an FPGA.

**System-level attacks.** Existing literature has shown Rowhammer bit flips on TRR-enabled (LP)DDR4 devices on a wide variety of systems, including on Intel [3], [4], [29], [78], AMD [6], RISC-V [79], and ARM devices [3], [4], [13], [37]. TRRespass [4] was the first study able to systematically bypass TRR on DDR4 devices at a system level. Blacksmith [3] demonstrated that all major DRAM vendors were vulnerable to the novel non-uniform aggressor patterns. Zenhammer [6] reported Rowhammer bit flips on a single DDR5 device with production year 2021 out of ten tested DDR5 devices. Furthermore, Zenhammer does not study the deployed DDR5 TRR mitigation.

**Rowhammer Fuzzers.** Zenhammer [6] is the only fuzzer that claimed to have found bit flips on DDR5 DIMMs; however, only on one out of ten tested DIMMs. We believe that current fuzzers cannot find our patterns due to three fundamental differences in our patterns: (i) significantly longer patterns (128/2608 tREFIs) [3], [6], (ii) an irregular TRR distance [3], (iii) and hammering the same aggressors for certain tREFIs almost entirely. Besides this, we require starting to hammer with the right refresh alignment and maintaining synchronization for thousands of tREFIs, which previous synchronization methods could not achieve (§ 6).

## 10. Conclusion

We presented Phoenix, the first Rowhammer attack that could trigger bit flips on DDR5 devices with modern TRR mechanisms. To achieve this, we had to reverse engineer TRR behaviors over a large number of refresh intervals to find blind spots. We then built custom patterns that hammer these blind spots over thousands of refresh intervals. To remain synchronized with these many refresh intervals, Phoenix relies on a novel self-correcting refresh synchronization method that detects missed refresh commands and readjusts the Rowhammer patterns automatically. Our evaluation shows that Phoenix triggers bit flips on all 15 SK-Hynix DDR5 UDIMMs running on a commodity system with default settings. We also built a privilege escalation exploit using these Rowhammer bit flips on a DDR5 system that succeeds in as little as 109 seconds. Our measurements show that increasing the refresh rate by $3\times$ mitigates Phoenix on the most vulnerable device we teseted while introducing an overhead of 8.4%.

## Acknowledgements

We would like to thank the anonymous reviewers for their feedback, and the Vulnerability Management team from the Swiss NCSC who provided us with significant coordination support for disclosing this issue to the affected parties. This work was supported in part by a generous gift from Google.

## References

[1] S. Gloor, P. Jattke, and K. Razavi, "REFault: A Fault Injection Platform for Rowhammer Research on Ddr5 Memory," in *$\mu$ASC '25*, Bochum, Germany, Feb. 2025.

[2] P. Jattke, F. Solt, M. Marazzi, M. Wipfli, S. Gloor, and K. Razavi, "McSee: Evaluating Advanced Rowhammer Attacks and Defenses via Automated DRAM Traffic Analysis," in *USENIX Security '25*, 2025.

[3] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "BLACKSMITH: Scalable Rowhammering in the Frequency Domain," in *IEEE S&P '22*, May 2022. [Online]. Available: https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf

[4] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE S&P '20*, 2020, pp. 747–762. [Online]. Available: https://download.vusec.net/papers/trrespass_sp20.pdf

[5] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications," in *MICRO '21*. Virtual Event Greece: ACM, Oct. 2021, pp. 1198–1213. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480110

[6] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms," in *USENIX Security '24*, Aug. 2024. [Online]. Available: https://www.usenix.org/system/files/sec24fall-prepub-1050-jattke.pdf

[7] "SK Hynix Takes Top Spot For First Time," accessed: 2025-05-16. [Online]. Available: https://www.counterpointresearch.com/insight/post-insight-research-notes-blogs-sk-hynix-takes-top-spot-for-first-time-on-continued-hbm-demand

[8] The Chosun Ilbo. (2025, Apr.) SK Hynix surpasses Samsung in global DRAM market share for first time. Accessed: 2025-05-16. [Online]. Available: https://www.chosun.com/english/industry-en/2025/04/10/KUEGT5WP7VGITHMS5CNV6LSPRM/

[9] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *ICPE*, 2018, pp. 41–42.

[10] JEDEC Solid State Technology Association, "DDR5 SDRAM," Jul. 2020. [Online]. Available: https://www.jedec.org/sites/default/files/docs/JESD79-5.pdf

[11] 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 '14*. Minneapolis, MN, USA: IEEE, Jun. 2014, pp. 361–372. [Online]. Available: http://ieeexplore.ieee.org/document/6853210/

[12] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. Shanghai China: ACM, Oct. 2017, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3152701.3152709

[13] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.

[14] M. Bölcskei, P. Jattke, and K. Razavi, "Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging," in *EuroS&P '25*. IEEE, 2015. [Online]. Available: https://comsec.ethz.ch/wp-content/files/rubicon_eurosp25.pdf

[15] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *IEEE S&P '18*, May 2018, pp. 245–261. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8418607

[16] M. Seaborn and T. Dullien, "Project Zero: Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," Mar. 2015. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[17] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses," in *MICRO '20*, Oct. 2020, pp. 28–41. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9251982

[18] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM," in *DIMVA '18*, Jun. 2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-93411-2_5

[19] S. Baek, M. Wi, S. Park, H. Nam, M. J. Kim, N. S. Kim, and J. H. Ahn, "Marionette: A RowHammer Attack via Row Coupling," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. Rotterdam Netherlands: ACM, Mar. 2025, pp. 637–652. [Online]. Available: https://dl.acm.org/doi/10.1145/3669940.3707242

[20] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 695–711. [Online]. Available: https://ieeexplore.ieee.org/document/9152687/

[21] C. Tomita, M. Takita, K. Fukushima, Y. Nakano, Y. Shiraishi, and M. Morii, "Extracting the Secrets of OpenSSL with RAMBleed," *Sensors*, vol. 22, no. 9, p. 3586, Jan. 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/9/3586

[22] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES '16*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds. Berlin, Heidelberg: Springer, 2016, pp. 602–624. [Online]. Available: https://eprint.iacr.org/2016/618.pdf

[23] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories," in *IEEE S&P '22*, May 2022, pp. 1157–1174. [Online]. Available: https://ieeexplore.ieee.org/document/9833743/

[24] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 497–514. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/hong

[25] F. Yao, A. S. Rakin, and D. Fan, "DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1463–1480. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yao

[26] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 7718–7727.

[27] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_razavi.pdf

[28] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security '16*, Austin, TX, Aug. 2016, pp. 19–35. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao

[29] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript," in *USENIX Security '21*, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ridder

[30] F. Ridder, P. Jattke, and K. Razavi, "Posthammer: Pervasive Browser-Based Rowhammer Attacks with Postponed Refresh Commands," in *USENIX Security '25*. Seattle, WA, USA: USENIX Association, Aug. 2025. [Online]. Available: https://comsec.ethz.ch/research/dram/posthammer/

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 300–321. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_15

[32] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, May 2016, pwnie Award for Most Innovative Research. [Online]. Available: Paper=https://download.vusec.net/papers/dedup-est-machina_sp16.pdfWeb=https://www.vusec.net/projects/dedup-est-machinaPress=https://goo.gl/ogBXTm

[33] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, May 2018, pwnie Award Nomination for Most Innovative Research, DCSR Paper Award Runner-up. [Online]. Available: Paper=https://download.vusec.net/papers/glitch_sp18.pdfWeb=https://www.vusec.net/projects/glitchPress=https://goo.gl/SkD9er

[34] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC '18*, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-tatar.pdf

[35] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing Rowhammer Faults Through Network Requests," in *EuroS&PW*, 2020, pp. 710–719.

[36] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020, pp. 638–651. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9138944

[37] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering From the Next Row Over," p. 18. [Online]. Available: https://www.usenix.org/system/files/sec22-kogler-half-double.pdf

[38] J. Juffinger, S. R. Neela, M. Heckel, L. Schwarz, F. Adamsky, and D. Gruss, "Presshammer: Rowhammer and Rowpress Without Physical Address Information," in *DIMVA '24*, F. Maggi, M. Egele, M. Payer, and M. Carminati, Eds. Cham: Springer Nature Switzerland, 2024, pp. 460–479.

[39] Z. Lang, P. Jattke, M. Marazzi, and K. Razavi, "Blaster: Characterizing the blast radius of rowhammer," in *3rd Workshop on DRAM Security (DRAMSec) co-located with ISCA 2023*. ETH Zurich, 2023.

[40] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "ProTRR: Principled yet Optimal In-DRAM Target Row Refresh," in *IEEE S&P '22*. San Francisco, CA, USA: IEEE, May 2022, pp. 735–753. [Online]. Available: https://ieeexplore.ieee.org/document/9833664/

[41] 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 *USENIX OSDI '18*, 2018, p. 15. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/konoth

[42] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't Touch This: Software-Only Mitigation against Rowhammer Attacks targeting Kernel Memory," in *USENIX Security*, 2017. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-brasser.pdf

[43] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 531–545. [Online]. Available: https://doi.org/10.1145/3132747.3132781

[44] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *ASPLOS*, 2016. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2954679.2872390

[45] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wang, S. Nepal, Y. Gao, K. Li, Z. Wang, and C. Wu, "{SoftTRR}: Protect page tables against rowhammer attacks using software-only target row refresh," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 399–414.

[46] A. D. Dio, K. Koning, H. Bos, and C. Giuffrida, "Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks," in *Proceedings 2023 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2023. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f337_paper.pdf

[47] 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*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 645–657. [Online]. Available: https://doi.org/10.1145/3297858.3304039

[48] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: Enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York New York: ACM, Jun. 2022, pp. 699–710. [Online]. Available: https://dl.acm.org/doi/10.1145/3470496.3527421

[49] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized Row-Swap: Mitigating Row Hammer by Breaking Spatial Correlation between Aggressor and Victim Rows," p. 14, 2022. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3503222.3507716

[50] A. G. Yağlikçi, 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 '21*, Feb. 2021, pp. 345–358. [Online]. Available: https://ieeexplore.ieee.org/document/9407238

[51] O. Canpolat, A. G. Yağlıkçı, A. Olgun, I. E. Yuksel, Y. C. Tuğrul, K. Kanellopoulos, O. Ergin, and O. Mutlu, "BreakHammer: Enhancing RowHammer Mitigations by Carefully Throttling Suspect Threads," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Nov. 2024, pp. 915–934. [Online]. Available: https://ieeexplore.ieee.org/document/10764696/?arnumber=10764696

[52] A. Olgun, Y. C. Tugrul, N. Bostanci, I. E. Yuksel, H. Luo, S. Rhyner, A. G. Yaglikci, G. F. Oliveira, and O. Mutlu, "ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation," Oct. 2023. [Online]. Available: http://arxiv.org/abs/2310.09977

[53] A. Saxena and M. Qureshi, "START: Scalable Tracking for any Rowhammer Threshold," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Edinburgh, United Kingdom: IEEE, Mar. 2024, pp. 578–592. [Online]. Available: https://ieeexplore.ieee.org/document/10476473/

[54] A. Saxena, S. Mathur, and M. Qureshi, "Rubix: Reducing the Overhead of Secure Rowhammer Mitigations via Randomized Line-to-Row Mapping," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. La Jolla CA USA: ACM, Apr. 2024, pp. 1014–1028. [Online]. Available: https://dl.acm.org/doi/10.1145/3620665.3640404

[55] S. C. Woo, W. Elsasser, M. Hamburg, E. Linstadt, M. R. Miller, T. Song, and J. Tringali, "RAMPART: RowHammer Mitigation and Repair for Server Memory Systems," in *Proceedings of the International Symposium on Memory Systems*. Alexandria VA USA: ACM, Oct. 2023, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3631882.3631886

[56] Y. Park, S. N. University, W. Kwon, S. N. University, E. Lee, S. N. University, T. J. Ham, S. N. University, J. H. Ahn, S. N. University, J. W. Lee, and S. N. University, "Graphene: Strong yet Lightweight Row Hammer Protection," in *MICRO '20*, 2020, p. 13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9251863

[57] E. Lee, I. Kang, S. Lee, G. E. Suh, and J. H. Ahn, "TWiCe: Preventing row-hammering by exploiting time window counters," in *ISCA '19*. Phoenix Arizona: ACM, Jun. 2019, pp. 385–396. [Online]. Available: https://dl.acm.org/doi/10.1145/3307650.3322232

[58] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Counter-Based Tree Structure for Row Hammering Mitigation in DRAM," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 18–21, Jan. 2017. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7579600

[59] M. Marazzi, F. Solt, P. Jattke, K. Takashi, and K. Razavi, "REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations," in *IEEE S&P '23*. San Francisco, CA, USA: IEEE, May 2023, pp. 1684–1701. [Online]. Available: https://ieeexplore.ieee.org/document/10179327/

[60] M. J. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. J. Ham, J. W. Lee, and J. H. Ahn, "Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh," *arXiv:2108.06703 [cs]*, Aug. 2021. [Online]. Available: http://arxiv.org/abs/2108.06703

[61] M. Qureshi, S. Qazi, and A. Jaleel, "MINT: Securely Mitigating Rowhammer with a Minimalist In-DRAM Tracker," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2407.16038

[62] M. Qureshi and S. Qazi, "MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2407.09995

[63] F. N. Bostanci, I. E. Yüksel, A. Olgun, K. Kanellopoulos, Y. C. Tuğrul, A. G. Yağliçi, M. Sadrosadati, and O. Mutlu, "CoMeT: Count-min-sketch-based row tracking to mitigate RowHammer at low cost," in *HPCA '24*. IEEE, 2024, pp. 593–612.

[64] M. Qureshi, A. Saxena, and A. Jaleel, "ImPress: Securing DRAM Against Data-Disturbance Errors via Implicit Row-Press Mitigation," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2407.16006

[65] J. Woo, S. C. Lin, P. J. Nair, A. Jaleel, and G. Saileshwar, "Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 1021–1037.

[66] O. Canpolat, A. G. Yağlıkçı, G. F. Oliveira, A. Olgun, N. Bostancı, I. E. Yuksel, H. Luo, O. Ergin, and O. Mutlu, "Chronus: Understanding and securing the cutting-edge industry solutions to dram read disturbance," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 887–905.

[67] M. Son, H. Park, J. Ahn, and S. Yoo, "Making DRAM Stronger Against Row Hammering," in *DAC '17*. Austin TX USA: ACM, Jun. 2017, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3061639.3062281

[68] S. Hong, D. Kim, J. Lee, R. Oh, C. Yoo, S. Hwang, and J. Lee, "DSAC: Low-Cost Rowhammer Mitigation Using In-DRAM Stochastic and Approximate Counting Algorithm," Feb. 2023. [Online]. Available: http://arxiv.org/abs/2302.03591

[69] Antmicro AB, "Data Center RDIMM DDR5 Tester · antmicro/ddr5-tester@rev.1.0.1-production," Jul. 2022. [Online]. Available: https://github.com/antmicro/rdimm-ddr5-tester/commits/rev.1.0.1-production

[70] Maxwell Technologies, "FT20X Supercapacitor Datasheet," https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf, Sept. 2020, technical Datasheet. [Online]. Available: https://www.maxwell-fa.com/upload/files/base/8/m/311.pdf

[71] "LiteX Rowhammer Tester," Antmicro, Oct. 2021. [Online]. Available: https://github.com/antmicro/litex-rowhammer-tester

[72] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID '18*, 2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-00470-5_3

[73] G. LLC, "Half-Double: Next-Row-Over Assisted Rowhammer," Google LLC, Tech. Rep., May 2021. [Online]. Available: https://security.googleblog.com/2021/05/introducing-half-double-new-hammering.html

[74] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, 2019.

[75] N. Kamadan, W. Wang, S. van Schaik, C. Garman, D. Genkin, and Y. Yarom, "Ecc.fail: Mounting rowhammer attacks on ddr4 servers with ecc memory," in *USENIX Security*, 2025.

[76] "Jesd79-5c: Double data rate 5 (ddr5) sdram," 2024, jEDEC Solid State Technology Association.

[77] T. Bennett, S. Saroiu, A. Wolman, L. Cojocar, and A.-G. Llc, "Panopticon: A Complete In-DRAM Rowhammer Mitigation," in *DRAMSec*, 2021, p. 7. [Online]. Available: https://dramsec.ethz.ch/papers/panopticon.pdf

[78] I. Kang, W. Wang, J. Kim, S. van Schaik, Y. Tobah, D. Genkin, A. Kwong, and Y. Yarom, "{SledgeHammer}: Amplifying rowhammer via bank-level parallelism," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1597–1614.

[79] M. Marazzi and K. Razavi, "Risc-h: Rowhammer attacks on risc-v," in *4th Workshop on DRAM Security (DRAMSec) co-located with ISCA 2024*, 2024.

# Appendices

## Appendix A. Platform

We summarize some of the issues that we had with the DDR5 Rowhammer RDIMM Tester platform by Antmicro [69]. All reported bugs have meanwhile been fixed in the official Rowhammer Tester repository [71].

**Refresh counter bug**[1]**.** We found a bug where activating certain rows would increment the refresh counter as a side effect, even though no refresh command was issued. Their logic assumed that all commands were single-cycle command, which is not the case for DDR5. Since our experiments depend on the refresh counter, we worked with Antmicro to resolve it.

**Payload executor bugs.** We found several bugs in the payload executor, which is the component that generates DRAM payloads based on a simple assembly-like language. For example, upon our error report, Antmicro found a bug in the activation condition of the DDR4-to-DDR5 adapter unit which caused the payload executor to not work correctly. Another payload executor bug[2] included a wrong encoding of the `LOOP` instruction caused by a silent truncation.

**Incorrect addresses.** Using McSee [2], we found that hammered addresses defined in our FPGA code do not correspond to the ones we can see on the bus. More precisely, we hammered double-sided in the code but saw that two far-apart rows were activated on the bus. After reporting this issue, Antmicro found that this was caused by a bug that creates a mismatch between DRAM address component bits in the FPGA design and the platform's software.

**RDIMM support.** When we first started working with the platform, only a handful of DDR5 RDIMMs successfully passed the memory training. Among others, we had issues with dual-rank DIMMs. We worked with Antmicro to improve the supported list of memory modules, which they now also document on their website[3].

## Appendix B. Generalization to Another DIMM

To reverse engineer TRR on $\mathsf{H}_6$, we apply the same method as in § 4. We first *zoom out* to capture the TRR behavior at refresh granularity. We measure the sampling period and identify the tREFIs that TRR samples far less frequently than in others. We then *zoom in* to `ACT` granularity, analyzing each lightly sampled interval to discover which activation slots TRR actually samples. This fine-grained view lets us craft a hammering pattern in which aggressor rows are activated throughout the interval, while decoy rows are carefully placed into activation slots known most likely to sample. Any TRR refresh therefore targets only the decoys and leaves the aggressors unsampled.

### B.1. Zooming Out On $\mathsf{H}_6$

We brute force the sampling period by incrementing $N$ and rerunning the *zoom out* experiment of § 4.1. Starting at $N{=}16$ and increasing it step-wise to $N{=}256$, shows no signs of a repeating pattern, hinting at a much longer sampling period. However, testing larger $N$, requires adding more regular refreshes to the DRAM payload, which inevitably leads to noise in the results. Hence, we need a new way to scale our zooming out even further.

We came up with a new technique to scale our methodology to sampling periods covering thousands of tREFIs. The key idea is that if we found the correct sampling period, then repeating the same experiment for refresh counters $N, 2N, 3N, \ldots$ should give us a clear pattern in the TRRs-per-tREFI histogram (e.g., Fig. 5) already by looking only at the first few hundred tREFIs.

We test this approach by examining only the first 256 tREFIs of the sampling window for each candidate $N$: we align the refresh counter $c$ to $c \equiv 0 \pmod{N}$ before every run and look for a stable pattern. We repeat this experiment for many repetitions. This refined method uncovers a clear, repeatable pattern at $N{=}2608$, showing that $\mathsf{H}_6$ operates on a 2608-tREFI sampling period.

1. https://github.com/antmicro/rowhammer-tester/issues/199

2. https://github.com/antmicro/rowhammer-tester/issues/191

3. https://antmicro.github.io/rowhammer-tester/memory_testing.html

👁(**O14.**) $H_6$ employs a 2608-tREFI sampling period.

With the 2608-tREFI sampling period known, we scan the entire period in smaller, 256-tREFI chunks. Before each run, the refresh counter is advanced by 256 `REFs`, so every experiment covers a fresh 256-tREFI slice. Ten such shifts cover the first 2560 intervals; the eleventh iteration captures the remaining 48 intervals and wraps around, completing the sweep. The results reveal that roughly three tREFIs in every 32-tREFI block are sampled significantly less often.

👁(**O15.**) Across the full 2608-tREFI period, approximately three tREFIs per 32-tREFI block are sampled significantly less often.

### B.2. Zooming In On $H_6$

After isolating the lightly sampled tREFIs in the 2608-tREFI period, we *zoom in* to `ACT` granularity. Repeating the per-activation test from § 4.2 for all the candidate tREFIs, we have the same result as for $H_2$: TRR is most likely to sample the *last* `ACT` issued before the next `REF`. This means we know both *which* tREFIs to hammer and *how* to schedule activations inside each interval.

### B.3. Pattern $\mathcal{P}_{2608}$

Alg. 2 on page 8 shows the pseudocode for $\mathcal{P}_{2608}$. The complete sequence spans 2608 tREFI intervals and is divided into two identical 1288-tREFI segments (lines 1–8). Within each segment the aggressor pair is hammered for $8 \times 5 \times 3 = 120$ tREFIs, while decoy activations occupy the remaining 1168 tREFIs. The 120 hammered tREFIs (lines 4, 11) correspond exactly to the lightly sampled intervals identified earlier.

## Appendix C. Pattern Optimizations

We show how brute forcing the refresh alignment can be made 16× more efficient by leveraging the pattern's structure. Our reverse engineering (§ 4) revealed that the TRR sampling behavior shows a repeating pattern recurring after 128 ($H_2$) and 2608 ($H_6$) refreshes. To bypass TRR, we need to hammer in specific refresh intervals within these repeating patterns. Therefore, we need to align to certain refresh commands before executing our hammering pattern. For example, for DIMM $H_2$, there are two `REFs` within each sequence of 128 `REFs` where we can start hammering to trigger bit flips (§ 5.3). This means, our chance to hit the right refresh alignment is $2/128 = 1.56\,\%$. Next, we show how we increase the pattern's success rate by a factor of 16× by implementing two optimizations.

**OPT. 1: Pattern instances.** We hammer only selected tREFIs in our pattern, for example, 32 tREFIs in our $H_2$ pattern. Consequently, there are 92 unused tREFIs in which we only access decoys to stay in sync. However, instead of decoys, we could make use of these tREFIs to hammer three more instances of the pattern (i.e., in $3 \times 32$ tREFIs) with different refresh alignments and targeting other rows. This reduces the number of decoy rows to the ones used for synchronization as we are now hammering a double-sided aggressor pair in every tREFI. This optimization increases the chance to hit the right refresh alignment from $1/64 = 1.56\,\%$ to $4/64 = 6.25\,\%$.

**OPT. 2: Bank-level parallelism.** Earlier work [78] showed that bank-level parallelism can be exploited to increase the number of bit flips by hammering multiple rows on different banks in parallel. In our case, however, we argue that the probability of hitting a vulnerable refresh offset ($6.25\,\%$) is significantly lower than not reaching the $\mathrm{HC}_{\min}$ of any of the four targeted rows. For example, if we hammer our pattern $\mathcal{P}_{128}$ for 8192 tREFIs with 50 `ACTs`/tREFI, we can accumulate a HC of approx. 102.4 K, which is roughly the average $\mathrm{HC}_{\min}$ (103.3 K) of the DIMMs vulnerable to this pattern. Therefore, we use bank-level parallelism to further increase the number of different refresh alignments that we try in one pattern execution (instead of testing different rows with the same refresh alignment). We can use different banks for this as we showed in § 5.2 that the vulnerable refresh offsets are identical across banks. By hammering four banks in parallel with shifted pattern instances, we can try $128/32 = 4$ alignments at once. This optimization combined with OPT. 1 increases our success rate from originally $1/64 = 1.56\,\%$ to $(4 \times 4)/64 = 25\,\%$.

Similar to $H_2$, the pattern for $H_6$ allows us to increase the success probability by trying different refresh alignments on different banks in parallel. This increases the success rate from $92/2608 = 3.53\,\%$ to $(4 \times 4) * 92/2608 = 56.48\,\%$.