# ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms

Patrick Jattke† Max Wipfli† Flavien Solt Michele Marazzi Matej Bölcskei Kaveh Razavi

*ETH Zurich*

†Equal contribution first authors

## Abstract

AMD has gained a significant market share in recent years with the introduction of the Zen microarchitecture. While there are many recent Rowhammer attacks launched from Intel CPUs, they are completely absent on these newer AMD CPUs due to three non-trivial challenges: 1) reverse engineering the unknown DRAM addressing functions, 2) synchronizing with refresh commands for evading in-DRAM mitigations, and 3) achieving a sufficient row activation throughput. We address these challenges in the design of ZenHammer, the first Rowhammer attack on recent AMD CPUs. ZenHammer reverse engineers DRAM addressing functions despite their non-linear nature, uses specially crafted access patterns for proper synchronization, and carefully schedules flush and fence instructions within a pattern to increase the activation throughput while preserving the access order necessary to bypass in-DRAM mitigations. Our evaluation with ten DDR4 devices shows that ZenHammer finds bit flips on seven and six devices on AMD *Zen 2* and *Zen 3*, respectively, enabling Rowhammer exploitation on current AMD platforms. Furthermore, ZenHammer triggers Rowhammer bit flips on a DDR5 device for the first time.

## 1 Introduction

Recent Rowhammer attacks that require the circumvention of in-DRAM mitigations have mostly been investigated on Intel platforms [6, 8–10, 13, 17, 19, 29, 30, 38, 39, 41, 44]. The success of these attacks is crucially dependent on the intimate architectural and microarchitectural details of Intel CPUs, such as how the memory controller maps physical addresses to DRAM chips, the observability of certain DRAM commands, and the behavior of memory flushing and ordering instructions. Lacking this information, Rowhammer attacks are currently absent on modern AMD CPUs based on the Zen microarchitecture. We conduct experiments to uncover this information, which we then use in the construction of ZenHammer, the first successful Rowhammer attack on Zen-based AMD platforms.

**DRAM Addressing.** Previous work that reverse engineered the DRAM addressing functions on Intel and ARM platforms assumes that these functions are constructed by XOR-ing certain physical address bits with each other [32]. We find that on AMD platforms, this assumption leads to an incomplete recovery of address functions. Our experiments show that AMD's memory controllers require offsets for certain physical address ranges *before* applying the XOR functions. Adjusting for these offsets, better handling of noisy measurements, and considering higher physical address bits lead to the recovery of correct and complete DRAM addressing functions for these CPUs. Yet, this initial version of ZenHammer only triggers bit flips on five (*Zen 2*) and none (*Zen 3*) of our ten DDR4 DRAM modules, while an Intel-based fuzzer [17] triggers bit flips on eight of them. We track down the reason to inadequate synchronization with DRAM refresh commands and the low throughput of activations sent to DRAM.

**Refresh Synchronization.** Modern DRAM devices employ in-DRAM Target Row Refresh (TRR) mitigations that detect potential victims of a Rowhammer attack and internally refresh these victims before bits can flip. These preventive refreshes happen transparently inside DRAM during the standard refresh commands issued by the memory controller. To bypass these mitigations, state-of-the-art Rowhammer patterns executed on Intel CPUs synchronize with refresh commands by repeatedly measuring the time it takes to access two rows inside DRAM [17]. Memory controllers' refresh commands delay these accesses, allowing the patterns to detect and synchronize with these commands. We find that the required flushing and ordering instructions introduce significant inaccuracies in the detection of refresh commands on AMD's Zen-based CPUs. To address this issue, we rely on synchronization using many uncached addresses that are only flushed after a refresh is detected.

**Activation Throughput.** To bypass TRR during a Rowhammer attack, maximizing the number of activations on both decoy and target DRAM locations is favorable and often necessary. We noticed that, unlike Intel CPUs, it is not trivial to saturate the activation throughput on Zen-based AMD CPUs due to the behavior of cache flushing and fencing instructions. To find the best hammering strategy, we systematically explore different memory access instructions and scheduling policies for flushing and fencing instructions. We discover that on AMD *Zen 3*, the CPU does not require a fence to order the cache flush with the memory access, allowing ZENHAMMER to achieve a higher activation throughput by omitting unnecessary fence instructions. Furthermore, we learn that on all AMD Zen CPUs, the number of necessary fence instructions to order different memory accesses can be tuned to a given DRAM vendor based on the sensitivity of its TRR mitigation to this ordering.

Equipped with better refresh synchronization and scheduling of flushing and fencing instructions, ZENHAMMER can trigger bit flips on seven of our ten sample devices (compared to eight on an Intel CPU). Our evaluation further shows that these bit flips can be used to build the page table [36], RSA public key corruption [34], and `sudo` [11] exploits on 7/6/4 of these devices, taking on average 164/267/209 seconds. To verify the attack's practicality, we implement and evaluate the page table exploit by Seaborn et al. [36] on a *Zen 3* system. We also show that ZENHAMMER can trigger bit flips on a DDR5 device for the first time.

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

- We reverse engineer the confidential DRAM addressing functions on various AMD Zen-based CPUs in different configurations, and we present a coloring technique enabling system exploitation despite higher physical address bits involved in the DRAM functions.
- We show how to synchronize effectively with refresh commands and increase the activation throughput with new fence scheduling strategies on AMD Zen-based CPUs.
- We build ZENHAMMER using the reverse engineered DRAM addressing functions, the new synchronization, and fencing strategies. Our evaluation shows that ZENHAMMER is effective on seven out of ten sample DDR4 devices, enabling Rowhammer exploitation on AMD Zen-based systems for the first time.
- Using ZENHAMMER, we show Rowhammer bit flips on a DDR5 device for the first time.

**Responsible Disclosure and Open Sourcing.** While Rowhammer is a known problem in industry, we nonetheless informed AMD about our findings and agreed to an embargo expiring on March 25, 2024. More information including the source code of ZENHAMMER can be found at: https://comsec.ethz.ch/research/dram/zenhammer

## 2 Background

We provide a high-level overview of DRAM (Section 2.1), how physical memory is mapped to it (Section 2.2), and discuss Rowhammer exploitation (Section 2.3).

### 2.1 DRAM

In desktop and server systems, dual in-line memory modules (DIMMs) are connected to the CPU's memory *channels* to equip systems with DRAM. Each of these DIMMs consists of multiple DRAM *chips*, which operate in a lockstep mode. Each chip contains several *banks*, a grid-like structure of DRAM *cells* organized in *rows* and *columns*. Each cell stores a single-bit value and consists of a capacitor and an access transistor. Further, each DRAM bank is connected to a *row buffer* that acts as a buffer for the whole DRAM row during read and write operations.

**DDRx Protocol.** The DDRx protocol is used to communicate between the DRAM device and the memory controller. The protocol dictates timing requirements and permitted command orders that the memory controller should respect to ensure the DRAM device's proper functioning. For example, in DDR4, a refresh command (`REF`) has to be sent to the DRAM devices every 7.8 µs (*tREFI*) on average. Before any data can be read from (`RD`) or written to (`WR`) a DRAM row, it must first be opened with an activate command (`ACT`), which brings its data into the row buffer. Before any other row in the same bank can be opened, the row must be closed, i.e., precharged (`PRE`).

**Row Buffer Side Channel.** *Row buffer conflicts* have been exploited as a timing side-channel to detect same-bank rows [32, 42, 43]. For this, two randomly picked addresses are repeatedly accessed in succession and their access time is measured. If the addresses map to different banks, the rows will stay open in their respective bank's row buffer and can directly be read. This means, *row buffer hits* happen, which leads to faster (lower latency) access times. However, if the rows map to the same bank, successive accesses evict each other from the row buffer, thus requiring a `PRE` and `ACT` before data can be read or written. This *row buffer conflict* yields slower (higher latency) access times.

### 2.2 DRAM Addressing

The memory controller uses an addressing scheme to map the physical address space to DRAM locations. For this, vendors employ confidential address mappings that are optimized for performance. As correctly addressing DRAM rows is essential for Rowhammer attacks, reverse engineering these proprietary functions is often a preliminary step. Unlike Intel, AMD published address mappings for their pre-Zen CPUs in the *BIOS and Kernel Developer's Guide* [2], but stopped publishing this information for its newer CPUs since 2017.

**Linearity of Functions.** Previous work [5, 14, 15, 32, 35, 42, 43] assumed the DRAM functions are *linear*. That is, a function $f_j$ is the exclusive-OR (XOR) of a set $S_j$ of physical address bits: $f_j(a) = \bigoplus_{k \in S_j} a_k$. We will show later (Section 4.2) that this assumption does not hold on our target systems.

### 2.3 Rowhammer

The DRAM vulnerability "Rowhammer" [22] allows attackers to induce memory disturbance errors. By rapidly accessing *aggressor* rows, an attacker can leak charge from adjacent *victim* rows, eventually causing bit flips in them. The effect is caused by the weak physical isolation of memory rows, and it is expected to worsen in future devices due to the ongoing miniaturization of DRAM cells [28]. In response to the worsening Rowhammer effect, DRAM vendors have deployed in-DRAM mitigations, known as Target Row Refresh (TRR), that preemptively refresh victim rows before bits flip [10, 12].

**Hammering Patterns.** The originally proposed single-sided [22, 40] and double-sided [36] patterns were later generalized by *n*-sided patterns with *n* aggressors, which helped to bypass some of the in-DRAM TRR mitigations [10]. The state-of-the-art *non-uniform* Rowhammer patterns showed how to bypass all existing mitigations on DDR4 devices [17]. These patterns are composed of double-sided aggressor pairs that are hammered with different frequencies, phases, and amplitudes to find blind spots in the deployed mitigations more effectively. Furthermore, a new Rowhammer effect known as Half-Double [24, 26] can bypass certain in-DRAM mitigations using near and far aggressors.

**Synchronization.** A key ingredient of recent Rowhammer patterns is the synchronization with the `REF` commands that trigger TRRs [9, 17]. Earlier work [9] showed that *soft synchronization* can be achieved by introducing a carefully chosen number of `NOP`s into the pattern. In this way, the memory controller is more likely to schedule `REF`s in these gaps of less DRAM activity, thus helping to keep the pattern in sync with the `REF`. Blacksmith [17], however, uses *hard synchronization* by tailoring the pattern to the length of multiple refresh intervals and exploiting the increased access latency during `REF`s to detect them.

**Discussion.** Existing research on Rowhammer mostly focused on Intel systems [7, 10, 17], where DRAM address functions [32, 42] and the effects of the hammering instruction sequence [8, 13] are well known. However, AMD has gained a significant market share in the last years and held around 36 % of the market for x86 CPUs in 2024 [1]. Yet, it is unclear if Rowhammer is similarly exploitable on these AMD systems.

## 3 Overview

Our goal is to trigger bit flips on AMD Zen-based platforms, particularly the systems listed in Table 1. These make use of DDR4 memory technology, allowing us to compare their vulnerability with a baseline on well-studied Intel systems [17].

A requirement for most Rowhammer attacks is the knowledge of the DRAM address mapping, i.e., how physical addresses map to the DRAM locations. This allows precisely selecting the location of aggressors around a victim row, as needed by most effective Rowhammer techniques [10, 24, 26, 36]. As the memory controllers of Intel and AMD systems use different DRAM address mappings, determining them poses our first challenge:

**Table 1.** Details about the Ryzen-based test systems ($Z_+$, $Z_2$, $Z_3$) used in this work.

| **Micro-architecture** | **Release Date** | **Our Test Systems** | |
|---|---|---|---|
| | | **System** | **CPU** |
| ***Zen 3*** | 11-2020 | $Z_3$ | Ryzen 5 5600G |
| ***Zen 2*** | 07-2019 | $Z_2$ | Ryzen 5 3600X |
| ***Zen+*** | 04-2018 | $Z_+$ | Ryzen 5 2600X |

**Challenge 1.** Reverse engineering the undocumented DRAM address mappings on AMD Zen-based systems.

We show in Section 4 that the state-of-the-art *DRAMA* [32] technique fails to recover the address functions on our AMD systems. Instead, a modified timing primitive and a relaxation of the common linearity assumption are required to obtain the full physical-to-DRAM address mappings. We then use these mappings to build ZenHammer and perform Rowhammer on our AMD systems with a sample of ten DRAM devices. However, even when hammering devices known to be exploitable on Intel systems, we find very few bit flips with ZenHammer, with many devices not showing any bit flips at all. Based on these results, we conclude that additional (micro-)architectural considerations are necessary for effective Rowhammer attacks on AMD Zen-based systems.

Earlier work [9, 17] shows that synchronizing a Rowhammer pattern with DRAM refresh commands is key for bypassing TRR mitigations. State-of-the-art non-uniform patterns [17], for example, rely on a timing side channel to detect spikes in the memory access latency caused by refresh commands [6, 9, 10]. Our analysis shows that this mechanism produces inaccurate results on AMD Zen-based CPUs, even failing completely on the newer *Zen 3* platform. This leads us to our second challenge:

**Challenge 2.** Understanding and overcoming the shortcomings of timing-based refresh synchronization.

In Section 5, we design and implement modified versions of timing-based refresh synchronization. We experimentally evaluate the various implementations to achieve more reliable synchronization on AMD platforms. Additionally, we notice that the activation throughput is only about half when compared to the Intel baseline. This severely reduces the budget of activations that can be used to "trick" TRR mitigations, substantially increasing the difficulty of finding effective Rowhammer patterns. This introduces our last challenge:

**Challenge 3.** Increasing activation rate during hammering while preserving the order of memory accesses.

In Section 6, we systematically evaluate the activation throughput achieved by different memory access, flushing, and fencing instructions to find optimal access patterns tuned to the underlying DRAM device. Finally, after solving these challenges, Section 7 evaluates the effectiveness of ZenHam-

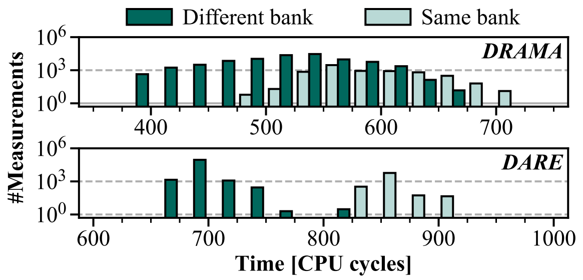

**Figure 1.** Histogram of access latencies measured on $Z_+$ when using *DRAMA* or *DARE*. Measurements are partitioned based on whether the address pair should produce a row conflict or not.

MER in triggering bit flips on the AMD *Zen 2* and *Zen 3* platforms and the ability of these bit flips for building successful Rowhammer exploits. We further show that ZENHAMMER can trigger bit flips on one of our DDR5 devices on the latest AMD *Zen 4* platform.

## 4 DRAM Addressing

*DRAMA* [32] is currently the standard approach for reverse engineering DRAM address mappings. We briefly describe the two main steps of this technique.

**Step 1: Clustering.** *DRAMA* measures the access latency of two randomly picked addresses. If the measured value exceeds the row conflict threshold, the addresses map to different rows in the same bank and otherwise to distinct banks. Using this method, *DRAMA* creates clusters of addresses in the same bank. *DRAMA* repeats this process until it finds a cluster for each bank.

**Step 2: Function Brute Forcing.** *DRAMA* then generates XOR-function candidates and tests them on the clustered addresses exhaustively. A valid function must (i) be constant for all addresses in each cluster and (ii) not produce the same result over all clusters. After removing linearly dependent functions, the resulting set of $\log_2 N$ functions, on a system with $N$ unique banks, can uniquely index every DRAM bank.

We verified that *DRAMA*, originally designed for Intel CPUs, does not produce valid results on recent AMD CPUs.[1] Either it does not find any address functions at all or functions that are incomplete. We describe how improvements to timing (Section 4.1) and taking system-specific address offsets into account (Section 4.2) enable our new DRAM reverse-engineering tool, called *DRAM Address Mapping Reverse-Engineering* (*DARE*), to successfully recover the address mappings on AMD Zen-based platforms (Section 4.3).

### 4.1 Timing Routine

The access time difference between addresses that produce row conflict and those that do not is very small. Thus, existing reverse-engineering tools use specially crafted *timing routines* to amplify the timing difference while eliminating unwanted noise (e.g., from unrelated system activity).

[1]https://github.com/IAIK/drama

In *DARE*, we perform 32 iterations of accesses to both addresses while measuring the entire loop. In contrast, *DRAMA* also measures accesses to two additional addresses that are shifted during measurement repetitions. We then perform 16 measurements and use the minimum value. However, we only do $16 \times 32 = 512$ accesses instead of $4 \times 5\,\text{K} = 20\,\text{K}$ (*DRAMA*) accesses, making our method significantly faster.

**Evaluation Setup.** We evaluate the accuracy of both timing routines, *DRAMA* and *DARE*, on $Z_+$. For each routine, we generate 100 K random address pairs and measure their access times. We expect the access latencies for bank hits and conflicts to be clearly distinguishable to allow reliable differentiation. The address pairs are then partitioned based on whether they map to the same bank or not using the ground truth obtained with an oscilloscope (see Section 4.3).

**Results.** The results, which are shown in Figure 1, clearly show that the *DRAMA* routine significantly overlaps the two cases, whereas our method shows a clearer separation. This means that it is less susceptible to noise than *DRAMA*, thus reducing the number of misclassified addresses. We further enhance *DARE* by cluster cleaning (i.e., intra-cluster pairwise testing of addresses) to ensure that we can reliably find the address functions despite system noise.

### 4.2 Address Offsets

We notice that *DARE* fails to find sufficient address function candidates (i.e., at least $\log_2 N$ for $N$ banks) but succeeds when restricted to smaller memory ranges such as 256 MiB-aligned blocks. In other words, some functions are valid over a limited area only and cease to work across larger areas.

We investigate this anomaly by applying the obtained functions on smaller windows over the entire memory range, as shown in Figure 2a. We note that the sections where the function result is 0 and where it is 1 have different sizes. However, if the address mapping was linear, the result would have to be either constant 0 or 1 (if the function is correct) or evenly split between the two (for any other linear function). We refer to Appendix A for a proof of this fact. Based on this observation, the correct functions cannot be linear, which contradicts assumptions made by previous work (see Section 2.2).

We find that removing this nonlinearity is possible by subtracting a particular constant offset to all physical addresses in the clusters before brute forcing the functions. This linearization is demonstrated in Figure 2b, where, after removing the offset, it is possible to find the correct function as shown in Figure 2c. Finally, the correctly identified function generates a constant value for all addresses in the same cluster.

**Observation 1.** DRAM address functions may be nonlinear due to physical address space remapping, in which case a constant offset needs to be subtracted from physical addresses before applying an XOR function.

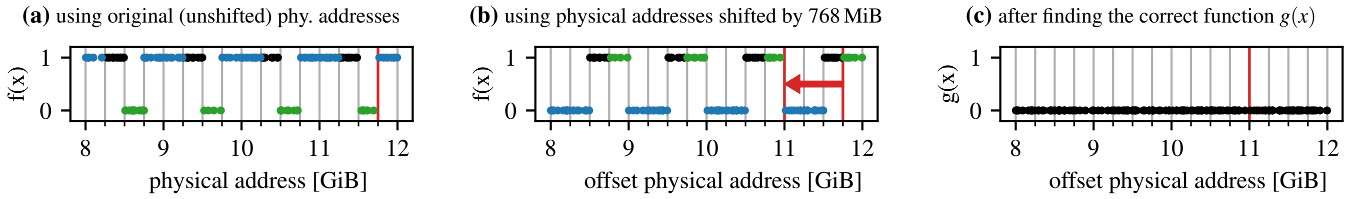

**Figure 2.** (a) Function values for $f(x)$ given by `0x64440100` for same-cluster addresses over the full address range on $Z_3$, showing a uneven distribution between "0" and "1". (b) After offsetting the physical addresses by 768 MiB before applying the function, the same function's output looks evenly distributed. (c) This allows us to find the function $g(x)$ defined by `0x44440100` that is constant for the cluster's addresses across all memory. We color the addresses whose function value changes when applying the offset in ● green ($0 \rightarrow 1$) or ● blue ($1 \rightarrow 0$).

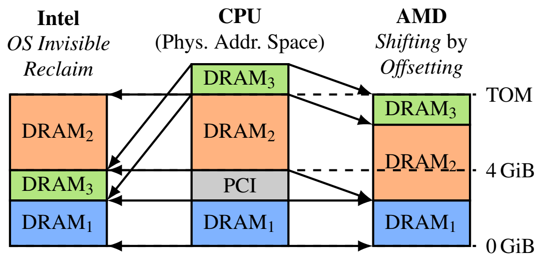

**Figure 3.** Remapping of higher address ranges to unused parts of physical memory on Intel and AMD CPUs. The Top of Memory (TOM) is the system's highest addressable memory location.

**System Address Map.** We now provide an explanation and supporting evidence for the existence of this offset.

The physical address space is divided into ranges backed by main memory (i.e., DRAM) and ranges for memory-mapped I/O (MMIO) devices. In particular, PCI(e) devices are commonly mapped just below the 4 GiB boundary to keep 32-bit compatibility, thus masking parts of main memory. Due to DRAM sizes in the order of gigabytes, CPU vendors introduced mechanisms to remap the otherwise inaccessible part of DRAM to a higher address range, as shown in Figure 3. Intel still employs this "OS Invisible Reclaim" mechanism [16, p. 19], but AMD stopped documenting "Memory Hoisting" [2, §2.9.12] with the *Zen* microarchitecture. Our findings suggest that newer AMD processors shift all physical addresses above 4 GiB by a fixed, system-specific offset. This offset depends on the system's hardware configuration, e.g., mainboard, installed PCI(e) devices.

**Automation.** To avoid having to brute force the physical address offset, we analyze the system memory map of our target systems to find the location of the primary PCI memory mapping.[2] For example, as we show in Table 2, the PCI memory range on $Z_2$ starts at 3584 MiB and ends at 3968 MiB. This allows us to precisely calculate the system's address offset by determining the difference between the PCI mapping's start address and the 4 GiB boundary, for example, $4096 - 3584 = 512$ MiB for $Z_2$. We apply the offset to our physical addresses before brute forcing the address functions to produce valid functions on all our systems.

[2]In Linux, the "PCI Bus 0000:00" in the (privileged) `/proc/iomem` file.

| System | PCI Range [MiB] | Offset [MiB] |
|---|---|---|
| $Z_+$ | 3072 – 4048 | 1024 |
| $Z_2$ | 3584 – 3968 | 512 |
| $Z_3$ | 3328 – 4076 | 768 |

**Table 2.** Primary PCI memory mappings and detected physical address offsets, i.e., difference between 4 GiB and the PCI mapping's start address.

## 4.3 Recovered Address Mappings

We run *DARE* on all our systems using single and dual-rank DIMMs. *DARE* successfully reverse engineers the address functions for all memory configurations on all three systems. For simplicity, we limit our analysis to single-channel, single-DIMM systems with default UEFI settings, as this is sufficient for performing Rowhammer.

To validate our results, we verify the functions' correctness using a high-bandwidth oscilloscope similar to previous work [32]. This also allows us to obtain the function labels (i.e., assign functions to the DRAM address components) and clarify the cases where our tool found linear combinations of the actual address functions. We note that this manual step is not required for Rowhammer attacks.

**Results.** We provide a list of all our reverse engineered and oscilloscope-validated address functions for three AMD Zen microarchitectures and different memory configurations in Table 3. Note that some physical address bits are above the 1 GiB mark, which explains why *DARE* uses as many 1 GiB superpages as possible while building same-bank address

**Observation 2.** We need access to a memory block larger than 1 GiB to entirely recover all DRAM address mappings.

**Discussion.** To the best of our knowledge, we are the first to reverse engineer and provide physically validated DRAM address mappings on recent AMD Zen-based systems with consideration of the address offsets. Further, we provide an improved reverse-engineering tool to reproduce and extend our results with more memory configurations as needed.

**Row Mapping.** *DARE*, just like *DRAMA*, does not allow the detection of physical address bits used for DRAM row and column indices. Therefore, before we can experimentally evaluate our address mappings using a Rowhammer attack, we need to extract the row mapping. Based on previous results [9, 42], we assume that the highest available address bits are used

**Table 3.** Reverse engineered address mappings and offsets for different DRAM configurations. All memory configurations are single-channel, single-DIMM, with the tuple indicating the DIMM's geometry (#ranks, #bank groups, #banks per bank group, #rows).

| Sys. | Geometry (RK, BG, BA, R) | Size [GiB] | Offt. [MiB] | DRAM Address Functions | | | Row Bits |
|---|---|---|---|---|---|---|---|
| | | | | Rank (RK) | Bank Group (BG) | Bank Address (BA) | |
| $Z_+$ | $(1, 4, 4, 2^{16})$ | 8 | 1024 | n/a | `0x088883fc0`, `0x111104000` | `0x022228000`, `0x044450000` | 32 – 17 |
| | $(2, 4, 4, 2^{16})$ | 16 | 1024 | `0x3fffe0000` | `0x111103fc0`, `0x222204000` | `0x044448000`, `0x088890000` | 33 – 18 |
| | $(2, 4, 4, 2^{17})$ | 32 | 1024 | `0x7fffe0000` | `0x111103fc0`, `0x222204000` | `0x444448000`, `0x088890000` | 34 – 18 |
| $Z_2$ | $(1, 4, 4, 2^{16})$ | 8 | 512 | n/a | `0x088883fc0`, `0x111104000` | `0x022228000`, `0x044450000` | 32 – 17 |
| | $(2, 4, 4, 2^{16})$ | 16 | 512 | `0x3fffe0000` | `0x111103fc0`, `0x222204000` | `0x044448000`, `0x088890000` | 33 – 18 |
| | $(2, 4, 4, 2^{17})$ | 32 | 512 | `0x7fffe0000` | `0x111103fc0`, `0x222204000` | `0x444448000`, `0x088890000` | 34 – 18 |
| $Z_3$ | $(1, 4, 4, 2^{16})$ | 8 | 768 | n/a | `0x022220100`, `0x044440200` | `0x088880400`, `0x111100800` | 32 – 17 |
| | $(2, 4, 4, 2^{16})$ | 16 | 768 | `0x3fffe0000` | `0x044440100`, `0x088880200` | `0x111100400`, `0x222200800` | 33 – 18 |
| | $(2, 4, 4, 2^{17})$ | 32 | 768 | `0x7fffe0000` | `0x444440100`, `0x088880200` | `0x111100400`, `0x222200800` | 34 – 18 |

for row indexing, which we verified with our oscilloscope. For example, a 16 GiB device (with $2^{16}$ rows) consists of $2^{34}$ individually addressable bytes, and a row index is described by the bits $(a_{33}, a_{32}, \ldots, a_{18})$.

## 4.4 Enabling Exploitation

On our Intel *Coffee Lake* system the bank, bank group, and rank bits all fall within the lower 21 bits, i.e., within a transparent huge page (THP). However, we noticed that the address functions on AMD *Zen 2* and *Zen 3* systems can cover up to bit 34 (see Table 3). This makes exploitation without knowing these bits challenging. Previous methods assume DRAM functions with all addressing bits falling in the lower 21 bits [24], do not take advantage of THPs [25], or color THPs for other purposes such as cache eviction [9]. We now describe how the bank conflict side channel and the reverse-engineered DRAM mappings can be combined to detect consecutive same-bank rows, which is crucial for Rowhammer attacks.

**Coloring THPs.** We allocate 256 MiB of 2 MiB-aligned memory and turn it into 2 MiB THPs by using `madvise`. We then iterate in steps of 2 MiB over the allocated memory such that the 21 lower bits are always the same. As the upper *physical* address bits are unknown, we cannot directly apply our recovered address functions. Instead, we use the bank conflict side channel to measure if the current THP conflicts with any other THP we found before. If two THPs conflict, we assign them the same color; otherwise, we assign a new color to the current THP. This approach allows us to assign a color to each THP based on the unknown upper physical address bits.

**Detecting same-bank rows.** Given that THPs are 2 MiB contiguous memory regions, we know that the lower 21 physical and virtual address bits are the same. Thus, we can group the THPs of the same color and use our recovered address functions on the lower bits to address consecutive same-bank rows. For that, we iterate over the row index bits that fall into the lower 21 bits. As they may overlap with bank address bits, it may require flipping lower (non-overlapping) bits to stay within the same bank. As the values of the DRAM functions for all THPs with the same color are identical, we can use the same THP row offsets for all THPs of the same color. Finally, we validate the row addresses using our bank conflict side channel and discard all THPs where any two rows do not cause bank conflicts.

**Results.** We measured how long the coloring and detecting same-bank rows take on our *Zen 3* system with a dual-rank DIMM ($\mathcal{S}_2$ in Table 4). The THP coloring took on overage 39.23 s and must be repeated for each attack as the THP allocation in physical memory changes. Detecting same-bank rows for each THP color is a one-time cost that can be precomputed for each system memory configuration and took on average 18 ms.

## 4.5 Evaluation

In addition to the physical validation of our mappings, we use Rowhammer on our AMD systems with non-uniform hammering patterns [17] to see if we can trigger bit flips, as this requires precise DRAM addressing. Further, we evaluate the recent Half-Double patterns [24].

**Threat Model.** In our evaluation, we assume that the CPU model of the target machine is known to the attacker and that they have obtained the correct DRAM address mappings, for example, using *DARE*. We further assume that an unprivileged attacker can execute programs on the victim's machine but does not know anything more specific about the DRAM devices (e.g., DRAM chip manufacturer).

**Setup.** We modify the reference implementation of *Blacksmith* [17].[3] Our changes include adding the address mappings we found previously and other necessary platform changes, such as timing thresholds, as the fuzzer was originally designed for an Intel *Coffee Lake* system. However, we do not apply any microarchitecture-specific optimizations. For the evaluation, we do six-hour fuzzing runs on both $Z_2$ and $Z_3$ with the ten DDR4 DIMMs listed in Table 4 that we ordered randomly from an online retailer. These DIMMs cover

[3] https://github.com/comsec-group/blacksmith

**Table 4.** DDR4 UDIMMs used in the evaluation of our AMD *Zen*-optimized Rowhammer fuzzer. We abbreviate the DRAM vendors Samsung ($\mathcal{S}$), SK Hynix ($\mathcal{H}$), and Micron ($\mathcal{M}$). For each device, we report the number of ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R).

| **ID** | **Production Date** | **Freq.** [MHz] | **Size** [GiB] | **DIMM Geometry** (RK,BG,BA,R) |
|---|---|---|---|---|
| $\mathcal{S}_0$ | Q3-2020† | 2132 | 8 | $(1, 4, 4, 2^{16})$ |
| $\mathcal{S}_1$ | Q3-2020† | 2132 | 16 | $(2, 4, 4, 2^{16})$ |
| $\mathcal{S}_2$ | Q2-2020 | 2666 | 32 | $(2, 4, 4, 2^{17})$ |
| $\mathcal{S}_3$ | Q4-2017 | 2400 | 8 | $(1, 4, 4, 2^{16})$ |
| $\mathcal{S}_4$ | Q3-2020† | 2666 | 8 | $(1, 4, 4, 2^{16})$ |
| $\mathcal{S}_5$ | Q2-2020 | 2666 | 16 | $(2, 4, 4, 2^{16})$ |
| $\mathcal{H}_0$ | Q3-2020† | 2132 | 16 | $(2, 4, 4, 2^{16})$ |
| $\mathcal{H}_1$ | Q4-2020 | 2400 | 8 | $(1, 4, 4, 2^{16})$ |
| $\mathcal{M}_0$ | Q1-2020 | 2666 | 8 | $(1, 4, 4, 2^{16})$ |
| $\mathcal{M}_1$ | Q1-2020 | 2400 | 8 | $(1, 4, 4, 2^{16})$ |

† Purchase date used as production date unavailable.

**Table 5.** Result of running *Blacksmith* with our address mappings and platform fixes (e.g., thresholds) on AMD *Zen 2* and *Zen 3* systems, compared to our Intel *Coffee Lake* baseline. We report for each device the number of patterns found ($|\mathbb{P}^+|$) and the number of bit flips over all patterns ($|\mathbb{F}_{\text{fuzz}}|$). We omit devices without any bit flips.

| **ID** | ***Zen 2*** | | ***Zen 3*** | | ***Coffee Lake*** | |
|---|---|---|---|---|---|---|
| | $|\mathbb{P}^+|$ | $|\mathbb{F}_{\text{fuzz}}|$ | $|\mathbb{P}^+|$ | $|\mathbb{F}_{\text{fuzz}}|$ | $|\mathbb{P}^+|$ | $|\mathbb{F}_{\text{fuzz}}|$ |
| $\mathcal{S}_0$ | 14 | 19 | 0 | 0 | 122 | 3,502 |
| $\mathcal{S}_1$ | 4 | 4 | 0 | 0 | 102 | 1,374 |
| $\mathcal{S}_2$ | 14 | 28 | 0 | 0 | 782 | 22,339 |
| $\mathcal{S}_3$ | 0 | 0 | 0 | 0 | 3 | 3 |
| $\mathcal{S}_4$ | 4 | 5 | 0 | 0 | 47 | 654 |
| $\mathcal{S}_5$ | 6 | 7 | 0 | 0 | 155 | 4,131 |
| $\mathcal{H}_1$ | 0 | 0 | 0 | 0 | 24 | 35 |
| $\mathcal{M}_1$ | 0 | 0 | 0 | 0 | 16 | 23 |

the three major DRAM manufacturers. To allow comparison with Intel, we further run the same code on the same DIMMs on a *Coffee Lake* (Core i7-8700K) machine.

**Results.** The result of our evaluation is presented in Table 5. It shows that with our minimal changes, we can trigger bit flips on our *Zen 2* system; however, only on 5 of 10 modules. We could not find any patterns on *Zen 3*. This is much lower than compared to 8 of 10 modules on the Intel *Coffee Lake* platform. We further note that the number of patterns found in the worst case ($\mathcal{S}_2$) is roughly 50x smaller on *Zen 2* (14 patterns) than on *Coffee Lake* (782 patterns).

We also tested Half-Double [24] patterns on all DDR4 devices with our address mappings and the reference implementation.[4] As we did not find any bit flips on our devices using these patterns, and Half-Double has not been shown to be exploitable on x86-64 machines, we disregard these patterns in the remainder of this work, and base ZenHammer on non-uniform Rowhammer patterns.

[4] https://github.com/IAIK/halfdouble

**Listing 1.** Refresh synchronization routine as used by *Blacksmith*.

```
void ref_sync_original(volatile char* rows[2]) {
  while (true) {
    uint64_t start = rdtscp();  /* START TIMER */
    lfence();
    *rows[0]; *rows[1];
    clflushopt(rows[0]); clflushopt(rows[1]);
    uint64_t stop = rdtscp();   /* STOP TIMER */
    lfence();
    if ((stop - start) > THRESHOLD) break;
  }
}
```

Based on our results, we conclude that the common hammering instruction sequence as used by *Blacksmith* [17] and others [10] encodes implicit assumptions about the underlying Intel microarchitecture. Our results show that this significantly affects Rowhammer's effectiveness on other platforms, such as the AMD systems targeted in this work. Motivated by this, we investigate the two crucial aspects of hammering, namely, refresh synchronization (Section 5) and the activation rate (Section 6) on AMD systems, and show how ZenHammer can improve them.

## 5 Refresh Synchronization

As shown by previous work [9, 10, 12, 17], it is essential to synchronize Rowhammer patterns with refresh commands. This is necessary as in-DRAM mitigations (i.e., TRR) have been shown to act during `REF`s. Synchronization is commonly done by detecting spikes in memory access latency, which correspond to when DRAM is briefly unavailable during refreshes [9, 10]. In this section, we investigate whether the refresh synchronization mechanism used by *Blacksmith* is effective on AMD Zen-based systems.

### 5.1 Blacksmith Synchronization

In Listing 1, we present *Blacksmith*'s synchronization routine, which uses two same-bank rows. This method relies on `RDTSCP` to capture timestamps, `LFENCE` to serialize the execution stream, and `CLFLUSHOPT` to immediately flush accessed rows. It assumes a `REF` has been detected whenever the timing measurements exceed a predefined threshold.

**Evaluation.** To detect whether synchronization works properly, we evaluate the time between detected refreshes, both on $Z_+$ and $Z_3$. When refresh commands are correctly detected, we expect the time between them to be around 7.8 µs, i.e., *tREFI* as specified by the DDR4 standard [18].

**Results.** The experiment results, each with 10 K iterations, are presented in Figure 4. The median latencies are 7.62 µs for $Z_+$ and 5.37 µs for $Z_3$.[5] While the data for the *Zen+* system

[5] For a fair comparison with *Blacksmith*, which uses *AsmJit* [23] to just-in-time (JIT) compile hammering patterns and their synchronization from x86-

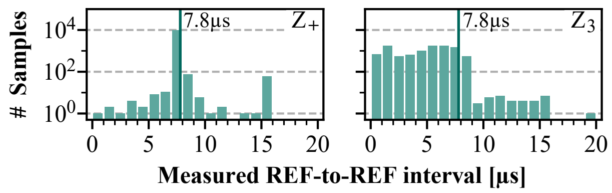

**Figure 4.** Measured time between successive REFs using the refresh synchronization routine `ref_sync_original()`, for both $Z_+$ and $Z_3$. The number of samples ($y$-axis) are logarithmically scaled.

suggests that this method works quite reliably, REFs are often detected too early on *Zen 3*. This could be because of two reasons: either the refresh detection fails most of the time, or the memory controller schedules REFs opportunistically. The latter is possible because the DDR4 standard [18] only specifies the *average* time between refresh commands and allows for some flexibility. In the following section, we will show that it is possible to detect the majority of refreshes reliably, as the original refresh synchronization method is inadequate on our AMD platforms.

## 5.2 Precise and Reliable Synchronization

We analyzed *Blacksmith*'s refresh synchronization routine as used by ZENHAMMER to identify possible measurement errors. By looking at the source code (Listing 1), we identified a brief time window, where fencing (`lfence`) happens, that is not measured between the `stop` timestamp and the next iteration's `start` timestamp. As the memory controller has some flexibility for scheduling refresh commands, it can happen that a REF sometimes remains undetected if it falls into this untimed gap. Furthermore, the memory controller may schedule the REF commands opportunistically during flush instructions, reducing the accuracy of detecting the REF commands.

**Continuous Measurements.** To mitigate this issue, we propose a modified refresh synchronization routine with *continuous*, *non-repeating* timing measurements: each recorded timestamp serves as both the end time of the current measurement round and the start time of the next. This ensures that all the instructions are included in the timing measurement. To ensure that the memory controller does not opportunistically schedule REF commands during the flush instructions, we avoid flushing during the synchronization phase. We solve this by designing a new method that allows a flexible number of rows and measures the latency of each memory access individually.

**Avoiding Cache Hits.** To avoid CLFLUSHOPT during synchronization, our code can only access *different* rows not to incur cache hits. To evict the cache lines for the subsequent synchronization phase, we flush the accessed rows after the REF is detected. Our *continuous*, *non-repeating* timing measurement

64 assembly, we implement all routines using *AsmJit*. We show equivalent C representations throughout this paper.

**Listing 2.** Our *continuous*, *non-repeating* refresh synchronization.

```c
void ref_sync_nonrep(volatile char* rows[64]) {
  uint64_t prev = rdtscp();
  for (size_t i = 0; i < 64; i++) {
    *rows[i];
    uint64_t curr = rdtscp();
    if ((curr - prev) > THRESHOLD) break;
    prev = curr;
  }
  // REF detected here (or ran out of rows)
  for (size_t i = 0; i < 64; i++) clflushopt(rows[i]);
}
```

| #Rows | Median [µs] | | Outliers [%] | |
|---|---|---|---|---|
| | $Z_+$ | $Z_3$ | $Z_+$ | $Z_3$ |
| 16 | 2.01 | 2.62 | 7.3 | 24.7 |
| 32 | 1.19 | 4.41 | 43.4 | 71.4 |
| 64 | 7.81 | 7.77 | 0.3 | 0.6 |
| 128 | 7.93 | 7.85 | 0.3 | 0.7 |
| 256 | 7.80 | 7.71 | 0.2 | 0.7 |
| **Orig.**[†] | 7.62 | 5.37 | 1.1 | 93.4 |

[†] The original refresh sync. routine with 2 rows (see Figure 4).

**Table 6.** REF-to-REF interval when using the *continuous*, *non-repeating* timing measurement routine (`ref_sync_nonrep`) for different numbers of rows on $Z_+$ and $Z_3$. We identify as outliers all the values that differ more than 10 % from the median.

routine is presented in Listing 2.

**Evaluation.** We evaluate our new routine using the same experiment as before. We show the obtained distribution of measured REF-to-REF intervals in Table 6. The results demonstrate that when more than 32 rows are employed in the synchronization, we correctly identify refreshes on all our systems. This means that a sufficient number of unique rows is necessary to cover an entire refresh interval (i.e., 7.8 µs) before falling through the end of the detection loop.

**Observation 3.** Continuous, non-repeating time measurements strongly improve the reliability of our refresh command detection.

## 6 Activation Rate

We noticed that the number of tested patterns on the AMD systems is significantly lower than on the Intel *Coffee Lake* baseline during fuzzing, on average by 45 % ($Z_2$) and 52 % ($Z_3$). As we fuzz for a fixed period (6 h) while hammering each pattern for 5 M activations, this suggests that each individual pattern takes significantly longer to hammer. To investigate this, we measure hammering execution times to compute the average number of activations per refresh interval (ACTs/tREFI) for each pattern. We present the comparison between $Z_+$, $Z_3$, and *Coffee Lake* in Figure 5. The data shows that the average number of ACTs/tREFI achieved on $Z_+$ (41.9) and $Z_3$ (37.2) are only about half when compared to *Coffee Lake* (76.8). The lower activation rate on the AMD systems have a direct impact on Rowhammer as discussed next.

**Hammer Count Estimation.** We now approximate the hammer count (HC) that a victim row is subjected to given these

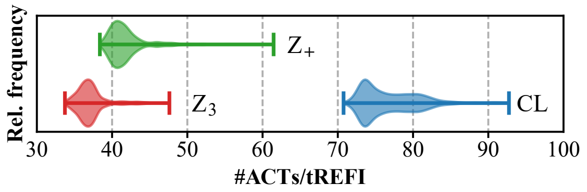

**Figure 5.** Distribution of the activation rates of non-uniform hammering patterns on $Z_+$, $Z_3$, and Intel *Coffee Lake* (CL). The whiskers indicate the minimum and maximum values.

activation rates. The estimation is made on a refresh window, as the bit flip needs to happen before the refresh of the victim row. In a refresh window, there are 8192 refresh intervals (tREFI). For an activation rate of 40 ACTs/tREFI, this results in a maximum of 328 K row activations before the victim row is refreshed. We consider a device with a Rowhammer mitigation that keeps track of 16 aggressors at a time [12]. To perform an effective double-sided Rowhammer on such a device, we need to hammer 18 rows, two aggressors and 16 dummy rows [10]. Assuming that we hammer the rows uniformly, this results in a cumulative HC of 36 K for the victim row. This is smaller than the minimum hammer count ($HC_{min}$) reported for many DDR4 devices by previous work [21, 26]. Therefore, based on these estimates, activation rates are insufficient to induce Rowhammer bit flips on many devices from AMD Zen-based CPUs. Thus, we aim to improve activation rates to enable more effective hammering. To this end, we first analyze possible hammering instruction sequences to find the optimal way to hammer.

## 6.1 Instruction Sequences

Existing studies [8, 9, 13, 33] proposed and evaluated different hammering sequences. For example, Cojocar et al. [8] showed that the sequence of machine instructions used for hammering affects the rate of activations. As they performed their experiments on Intel CPUs, it is unlikely that their results transfer to our AMD processors. Therefore, we perform our own analysis of possible instruction sequences.

We start by analyzing the standard instruction sequences used by ZenHammer. They flush the cache directly after each access ("scatter" [9]) and fence (`MFENCE`) after each flush ("fence each"). However, this instruction sequence might not be optimal on AMD systems, as our earlier results suggest. In the following (**a–e**), we present the fundamental building blocks of possible hammering instruction sequences.

**a. Cache Flushing.** Because a hammered aggressor is cached, we need to ensure that subsequent hammering accesses are fetched from DRAM again. For flushing aggressors from the cache, we can use `CLFLUSH` or the optimized `CLFLUSHOPT`. The latter avoids serialization, which improves concurrency when used back-to-back [8, 13]. Depending on the Rowhammer pattern, we might have some flexibility in deciding *when* to issue the flushing instructions: either batched together for all aggressors at the end of the pattern

**Table 7.** Heatmap of memory access rates (in ACTs/tREFI) for different instruction sequences and varying the number of accessed rows on the $Z_3$ system. We omit unsuitable sequences with a low throughput ($\leq 100$ ACTs/tREFI) and sequences indicating cache hits with a very high throughput ($\geq 1000$ ACTs/tREFI), and provide the complete table in Appendix B.

| Access Type | Flushing Strategy | Fence Type | #Rows | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 4 | 8 | 16 | 32 | 256 |
| `MOV` (load) | gather | M | 24 | 49 | 71 | 91 | 100 | 110 | 114 |
| `MOV` (load) | gather | L | 24 | 49 | 80 | 113 | 134 | 147 | 121 |
| `MOV` (load) | gather | S | 24 | 49 | 80 | 113 | 133 | 146 | 125 |
| `MOV` (load) | gather | — | 24 | 49 | 80 | 113 | 133 | 146 | 125 |
| `MOV` (load) | scatter | M | 24 | 49 | 79 | 107 | 126 | 143 | 157 |
| `MOV` (load) | scatter | L | 24 | 49 | 95 | 137 | 149 | 153 | 159 |
| `MOV` (load) | scatter | S | 24 | 48 | 97 | 154 | 159 | 159 | 159 |
| `MOV` (load) | scatter | — | 24 | 49 | 97 | 154 | 159 | 159 | 159 |
| `PREFETCHNTA` | scatter | — | 80 | 132 | 191 | 208 | 253 | 309 | 273 |
| `PREFETCHNTA` | scatter | M | 24 | 49 | 80 | 108 | 131 | 170 | 284 |
| `VGATHERDD` | scatter | — | 24 | 49 | 79 | 112 | 159 | 159 | 159 |

(i.e., "gather"), or directly after each memory accesses (i.e., "scatter") [9].

**b. Memory Barriers.** To ensure that aggressors are flushed from the cache before they are accessed again, existing approaches rely on memory barriers. For example, `MFENCE` serializes all preceding loads and stores, `LFENCE` serializes all preceding loads, and `SFENCE` serializes all preceding stores. Given our "scattered" flushing, fences can either be placed after every flush ("fence each") or only once at the end ("fence once"). Lastly, we may omit fences to sacrifice some accesses (hitting the cache) for a higher activation rate [8].

**c. Access Types.** Typically, load instructions are used to execute Rowhammer patterns on regular x86-64 machines. This is necessary because the DRAM activate command that triggers the Rowhammer effect cannot be directly issued. Instead of loads, Rowhammer is also possible using store operations, as they also induce row activations [8].

**d. Non-Temporal Instructions.** The x86 ISA specifies non-temporal instructions that bypass CPU caches entirely, thus avoiding cache flushing [33]. However, they either require non-standard write-combining (WC) memory (`MOVNTDQA`), may prefetch data from L3 cache instead of accessing DRAM (`PREFETCHNTA`), or can be cached in WC buffers (`MOVNTI`).

**e. Vector Instructions.** The *gather* family of AVX2 load instructions can be used to load data from a non-contiguous address list. As an example, `VPGATHERDD` loads up to eight 32-bit values simultaneously [3]. This method still requires cache-flush instructions and possibly memory barriers.

**Evaluation.** We implement an experiment to evaluate the performance of various instruction sequences. For this, we pick $N$ random row addresses and access them in a loop for 10 M memory accesses while recording the elapsed time. Later, we use the measured time to compute the activation rate. Note that $N$ also corresponds to the distance between consecutive accesses to the same row, which we sweep between 1 and 256 to cover the various distances in non-uniform patterns. The result from $Z_3$ is visualized in Table 7 (similar results on $Z_+$). From these, we derive the following six observations (**O1**-**O6**) and three concrete recommendations (**R1**-**R3**):

**(O1)** Non-temporal instructions hit caches: Non-temporal instructions such as `PREFETCHNTA` have access rates exceeding the available bandwidth, thus suggesting cache hits. Therefore, we disregard such instructions.

**(O2)** More rows increase the ACT rate: Using more rows almost always increases the rate of memory accesses, except for "fence each" sequences.

**(O3)** `CLFLUSHOPT` is slightly faster than `CLFLUSH`: In most cases, there is no measurable difference between them. In a few cases, `CLFLUSHOPT` produces up to 5 % higher activation rates.

**R1.** Always use `CLFLUSHOPT` over `CLFLUSH` to maximize the activation rates.

**(O4)** "scatter" is always faster than "gather": The "scatter"-style cache flushing always produces higher access rates than the equivalent "gather" sequence. Further, as our non-uniform frequency-based patterns may hammer the same aggressors multiple times consecutively, we consider only "scattered" flushes.

**R2.** Schedule cache flushes in a "scatter"-style, i.e., flush immediately after accessing an aggressor.

**(O5)** Loads are always faster than stores: All sequences using store instructions result in low memory access rates (up to 76 ACTs/tREFI), with reductions between 5 % and 56 % compared to equivalent load sequences.

**R3.** Always prefer load instructions to hammer over store instructions to optimize activation rates.

**(O6)** AVX instructions are fast but complex to implement: The AVX `VPGATHERDD` instruction produces memory access rates comparably to regular loads (i.e., `MOV`). However, it is more complex to implement than regular loads. This is especially the case for non-uniform patterns that hammer aggressors with different frequencies and with flushes in between.

Based on these results, we exclude `CLFLUSH`, "gather"-style flushing, stores, non-temporal accesses, and AVX2 vector instructions in the remaining experiments. Thus, we will focus on `CLFLUSHOPT`, "scatter"-style flushing, and the different types of fences for loads (`M/LFENCE` and "no fence").

We further run this experiment on Intel *Coffee Lake* to allow comparison with the results of our AMD systems. We provide the full results in Appendix B. The results show that the activation rates on *Coffee Lake* are generally higher for all tested configurations.

**Table 8.** Overview of our proposed fence scheduling policies. We indicate which policies are pattern-aware by taking the pattern's structure into account and which are cache-avoiding.

| **Policy** | **Fencing Frequency** Example | **Pattern-Aware** | **Cache-Avoiding** |
|---|---|---|---|
| $SP_{none}$ | no fences within pattern | ✗ | ✗ |
| $SP_{BP}$ | between base periods | ✓ | ✗ |
| $SP_{BP/2}$ | every half base period | ✓ | ✗ |
| $SP_{pair}$ | between different aggr. pairs[†] Ex.: $\mid a_1\ a_2\ a_1\ a_2 \mid a_3\ a_4 \mid$ | ✓ | ✗ |
| $SP_{rep}$ | between aggr. pair repetitions Ex.: $\mid a_1\ a_2 \mid a_1\ a_2 \mid a_3\ a_4 \mid$ | ✓ | ✓ |
| $SP_{full}$ | after every access Ex.: $\mid a_1 \mid a_2 \mid a_1 \mid a_2 \mid a_3 \mid a_4 \mid$ | ✗ | ✓ |

[†] In *Blacksmith*'s terminology [17]: rows that are 2 rows apart and have the same frequency, phase, and amplitude.

**Ordering of Loads and Cache-Flushes.** We notice that the sequences without memory barriers ("no fence") do not exceed the activation rate of sequences with fences. This suggests that memory loads are served by DRAM, and consequently, load-flush-load sequences to the same address are strongly ordered. This is surprising, as AMD documents loads only to be ordered with same-cacheline *stores* [4].

To confirm our observation that all load requests are served by DRAM, we use the CPU's performance counters to measure the number of data cache fills by DRAM. On $Z_3$, we find that the number of measured cache fills does not differ between sequences with and without memory barriers. Moreover, this value is equal to the number of loads issued while hammering. Instead, on $Z_+$ and $Z_2$, this is not the case, and we observe up to 70 % cache hits in some cases.

**Observation 4.** Memory load requests following a `CLFLUSH(OPT)` to the same cache line never incur cache hits on *Zen 3*, but do incur cache hits on *Zen+* and *Zen 2*.

As omitting all fences leads to very high activation rates without incurring cache hits (on $Z_3$), it seems like the optimal choice for efficient Rowhammering. However, omitting memory barriers allows reordering the accesses of different aggressors. The reason is that both load and flush instructions are not ordered between different cache blocks [4], and thus may be rearranged by the processor. This can hinder us in effectively bypassing some TRR mitigations, which are sensitive to the order of row activations [12]. Therefore, we need to determine the optimal balance between high activation rates and strict ordering.

## 6.2 Fence Scheduling Policies

Based on Observation 4, we hypothesize that we may omit some fences to speed up pattern execution, while keeping others to preserve sufficient ordering. To explore this trade-off between high activation rates and strict ordering of memory accesses, we propose six different fence scheduling poli-

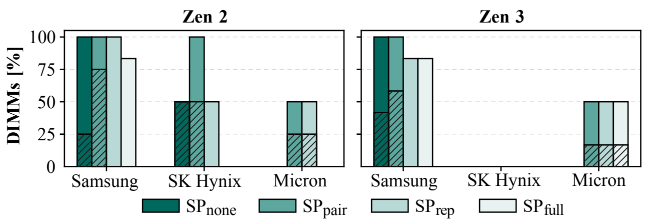

**Figure 6.** Comparison of the four effective scheduling policies ($SP_{none}$, $SP_{pair}$, $SP_{opt}$, $SP_{full}$) grouped by vendors, normalized by #devices per vendor. The dashed areas indicate how often each policy was the *best* in the no. of effective patterns. The percentages per vendor sum up to the total percentage of devices with bit flips.

cies (SPs), which are summarized in Table 8. Besides the two simple polices, no fences ($SP_{none}$) and fencing after every access ($SP_{full}$), we propose four policies that take the pattern's structure into account, fencing every ($SP_{BP}$) or every half base period ($SP_{BP/2}$), fencing between aggressor pairs ($SP_{pair}$), and fencing between repetitions of the same aggressors ($SP_{rep}$). Some scheduling policies are *cache-avoiding*, i.e., they strongly order all consecutive accesses to the same aggressor. However, we still consider all policies on all our systems, as previous work has shown that omitting fences can lead to both higher activation rates [8] and more bit flips [43] despite possibly incurring cache hits.

**Evaluation.** We evaluate the effectiveness of our fence scheduling policies in two ways. To begin with, we build a theoretical model for the amount of ordering provided by different scheduling policies, and contrast this with the hammering speeds obtained with the respective policies on our systems, as described in Appendix C. The results show that $SP_{pair}$ and $SP_{rep}$ can provide significantly higher activation rates when compared to $SP_{full}$ without allowing significant reordering. To validate our theoretical model against the real world, we perform 6 h fuzzing for each of our ten DIMMs (Table 4. We employ the two proposed policies $SP_{pair}$ and $SP_{rep}$, and for comparison $SP_{none}$ and $SP_{full}$. As the activation rate experiment (Section 6.1) was inconclusive in defining which memory barrier is optimal, we randomize the fence type between `MFENCE` and `LFENCE`.

In Figure 6, we show the results of our experiments. We present how many configurations generated at least one effective hammering pattern per vendor, normalized by the number of DIMMs from that vendor. These results describe which configuration is most widely effective for each DRAM vendor. From the data, we observe that fencing is not strictly required, as $SP_{none}$ found bit flips on all devices from Samsung on both *Zen 2* and *Zen 3*. However, $SP_{pair}$ is the most effective policy on *Zen 2* across most devices (75%). The same, but less significantly, applies to *Zen 3*.

**Observation 5.** For Samsung devices, the scheduling policy $SP_{pair}$ is the most widely applicable (across devices) and most effective (across patterns).

For SK Hynix devices, we can see that $SP_{pair}$ works on all tested devices. We have also found effective patterns with $SP_{none}$ and $SP_{rep}$ on half of all devices.

**Observation 6.** For SK Hynix devices, choosing $SP_{pair}$ works best across different devices.

Lastly, we have not found any effective hammering pattern for Micron devices using $SP_{none}$, which indicates that ordering is essential for these chips. This behavior could be explained by the type of deployed in-DRAM mitigation. Rowhammer mitigations that sample rows with non-uniform probabilities are harder to evade if the accesses are uncontrollably reordered.

**Observation 7.** Preserving ordering in hammer patterns is essential on Micron devices.

As the results show that the best scheduling policy may vary for different devices from the same vendor, we do not incorporate vendor-specific policies in ZENHAMMER.

## 7 Evaluation

In this section, we compare ZENHAMMER, especially designed for Rowhammer on Zen-based systems, to the baseline established on Intel in Section 4.5. In addition, we assess the impact of our optimizations on the effectiveness of ZENHAMMER and evaluate the exploitability of the discovered bit flips. We first describe our evaluation setup and methodology (Section 7.1) and then present and discuss the results (Section 7.2). We conclude by applying ZENHAMMER on DDR5 devices (Section 7.3).

### 7.1 Setup and Methodology

For our evaluation, we pick the same previously used DDR4 devices (Sections 4 and 6), covering DRAM chips from all three major DRAM manufacturers, Samsung ($\mathcal{S}$), SK Hynix ($\mathcal{H}$), and Micron ($\mathcal{M}$). For establishing the Intel baseline, we used an Intel Core i7-8700K. The AMD *Zen 2* and *Zen 3* machines are equipped with the CPUs listed in Table 1. All machines use default UEFI settings and device timings.

In line with previous work [10, 17], we evaluate ZENHAMMER in three stages: (i) fuzzing for 6 h using ZENHAMMER for each configuration (i.e., fence scheduling policy), (ii) determining the *best pattern* using a *minisweep* over all effective patterns by moving the pattern over a physically contiguous 4 MiB of memory, and (iii) sweeping the best pattern found over a physically contiguous 256 MiB memory range to assess the device's vulnerability level and assess the bit flips' exploitability. We note that our approach does not rely on any DRAM device-specific knowledge as we tested all fence scheduling policies and fence types on each device to determine the optimal per-device configuration (see Section 6).

| ID | *Zen 2* |  |  |  | *Zen 3* |  |  |  | *Coffee Lake* |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | $SP_{opt}$ | $\lvert\mathbb{P}^+\rvert$ | $\lvert\mathbb{F}_{fuzz}\rvert$ | $\lvert\mathbb{F}_{swp}\rvert$ | $SP_{opt}$ | $\lvert\mathbb{P}^+\rvert$ | $\lvert\mathbb{F}_{fuzz}\rvert$ | $\lvert\mathbb{F}_{swp}\rvert$ | $SP_{opt}$ | $\lvert\mathbb{P}^+\rvert$ | $\lvert\mathbb{F}_{fuzz}\rvert$ | $\lvert\mathbb{F}_{swp}\rvert$ |
| $\mathcal{S}_0$ | $SP_{rep}$ | 51 | 151 | 6,945 | $SP_{none}$ | 31 | 124 | 17,775 | $SP_{full}$ | 122 | 3,502 | 6,782 |
| $\mathcal{S}_1$ | $SP_{rep}$ | 26 | 97 | 1,758 | $SP_{pair}$ | 25 | 144 | 15,613 | $SP_{full}$ | 102 | 1,374 | 10,106 |
| $\mathcal{S}_2$ | $SP_{none}$ | 97 | 1,685 | 12,893 | $SP_{none}$ | 45 | 471 | 79,306 | $SP_{full}$ | 782 | 22,339 | 1,708 |
| $\mathcal{S}_3$ | $SP_{none}$ | 8 | 15 | 2,020 | $SP_{pair}$ | 1 | 1 | 667 | $SP_{full}$ | 3 | 3 | 0 |
| $\mathcal{S}_4$ | $SP_{none}$ | 60 | 182 | 1,183 | $SP_{pair}$ | 43 | 297 | 13 | $SP_{full}$ | 47 | 654 | 18,357 |
| $\mathcal{S}_5$ | $SP_{none}$ | 25 | 83 | 1,911 | $SP_{pair}$ | 26 | 87 | 10,741 | $SP_{full}$ | 155 | 4,131 | 5,860 |
| $\mathcal{H}_0$ | $SP_{none}$ | 6 | 13 | 182 | – | 0 | 0 | 0 | – | 0 | 0 | 0 |
| $\mathcal{H}_1$ | – | 0 | 0 | 0 | – | 0 | 0 | 0 | $SP_{full}$ | 24 | 35 | 0 |
| $\mathcal{M}_0$ | – | 0 | 0 | 0 | – | 0 | 0 | 0 | – | 0 | 0 | 0 |
| $\mathcal{M}_1$ | – | 0 | 0 | 0 | – | 0 | 0 | 0 | $SP_{full}$ | 16 | 23 | 2 |

**Table 9.** ZENHAMMER results on AMD *Zen 2* and *Zen 3* as well as Intel *Coffee Lake*. For each of our ten devices, we report the best scheduling policy ($SP_{opt}$) and the number of effective patterns ($\lvert\mathbb{P}^+\rvert$) and bit flips ($\lvert\mathbb{F}_{fuzz}\rvert$) found while fuzzing with the best policy. We also show the number of bit flips found when sweeping the best patterns over a 256 MiB range ($\lvert\mathbb{F}_{swp}\rvert$).

**Table 10.** Analysis of the bit flip exploitability found during the sweep over 256 MiB on AMD *Zen 2*, *Zen 3*, and Intel *Coffee Lake*. For each attack, we indicate the number of exploitable bit flips (#Ex.) and average time to find an exploitable bit flip (Time). We mark DIMMs with a single exploitable bit flip by (*). We omit DIMMs without any exploitable bit flips.

| DIMM | **PTE** [36] |  |  |  |  |  | **RSA-2048** [34] |  |  |  |  |  | **sudo** [11] |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | *Zen 2* |  | *Zen 3* |  | *Coffee Lake* |  | *Zen 2* |  | *Zen 3* |  | *Coffee Lake* |  | *Zen 2* |  | *Zen 3* |  | *Coffee Lake* |  |
|  | #Ex. | Time | #Ex. | Time | #Ex. | Time | #Ex. | Time | #Ex. | Time | #Ex. | Time | #Ex. | T. | #Ex. | Time | #Ex. | Time |
| $\mathcal{S}_0$ | 7 | 6m 4s | 7 | 2m 55s | 3 | 4m 15s | 17 | 2m 47s | 37 | 46s | 14 | 1m 36s | – | – | 4 | 3m 13s | 1 | *23m 49s |
| $\mathcal{S}_1$ | 90 | 9s | 1474 | 2s | 846 | 2s | 6 | 2m 2s | 27 | 30s | 21 | 26s | – | – | 1 | *6m 50s | 1 | *1m 20s |
| $\mathcal{S}_2$ | 641 | 21s | 5326 | 1s | 126 | 11s | 30 | 2m 16s | 170 | 6s | 6 | 1m 59s | – | – | 12 | 1m 17s | – | – |
| $\mathcal{S}_3$ | 142 | 9s | 61 | 32s | – | – | 7 | 2m 21s | – | – | – | – | – | – | – | – | – | – |
| $\mathcal{S}_4$ | 220 | 28s | 3 | 23m 52s | 2658 | 1s | 7 | 12m 29s | 1 | *23m 52s | 53 | 26s | – | – | – | – | 4 | 5m 16s |
| $\mathcal{S}_5$ | 102 | 6s | 625 | 2s | 330 | 4s | 6 | 1m 14s | 28 | 33s | 11 | 1m 5s | – | – | 2 | 5m 58s | 3 | 2m 34s |
| $\mathcal{H}_0$ | 11 | 53s | – | – | – | – | – | – | – | – | – | – | – | – | – | – | – | – |

## 7.2 Effectiveness and Exploitability

The results of our evaluation are presented in Table 9. We show for each tested platform (AMD *Zen 2* and *Zen 3*, Intel *Coffee Lake*) and each DDR4 device, the number of effective patterns found ($\lvert\mathbb{P}^+\rvert$) and the number of bit flips ($\lvert\mathbb{F}_{fuzz}\rvert$) found during fuzzing with the device's best fence scheduling policy ($SP_{opt}$) that we used in all three stages. For Intel *Coffee Lake*, we assumed the scheduling policy $SP_{full}$, which corresponds to the one used by the original *Blacksmith* fuzzer.

We also show for the best pattern, the total number of bit flips over the sweeped 256 MiB of physically contiguous memory ($\lvert\mathbb{F}_{swp}\rvert$), which we then use to assess exploitability of three known Rowhammer end-to-end attacks in Table 10.

For the exploitability analysis, we follow prior work [7, 10, 17] and use the Rowhammer attack simulation framework *Hammertime* [37] to estimate the required time for three previously proposed Rowhammer attacks targeting (i) page table entries (**PTE**) to craft an arbitrary memory read/write primitive [36], (ii) **RSA-2048** keys to break the SSH public-key authentication [34], and (iii) the **`sudo`** binary to elevate the privilege to the `root` user [11]. We use the bit flips we found during the sweep with the best pattern to perform the exploitability analysis.

**Results.** Our results in Table 9 show that our Zen-based platform optimizations have strongly improved the number of devices we can trigger bit flips on, from 5 and 0 devices before any optimizations (see Table 5) to 7 and 6 devices afterward, for *Zen 2* and *Zen 3*, respectively. The number of effective hammering patterns found further increased drastically, in the best case ($\mathcal{S}_2$) by roughly six times (from 14 to 97). Moreover, the results on *Zen 3*, where we had not found any bit flips previously, stress the need for our optimizations to trigger any bit flips on the AMD *Zen 3* platform. This shows that the hammering instruction sequence and fence scheduling policy are important when adapting Rowhammer attacks to new platforms.

Nevertheless, we note that there are still strong differences in terms of hammering effectiveness between AMD and Intel. On Intel, four of eight DIMMs have a higher bit flips count in the sweep than the same devices on *Zen 2*. Interestingly, there is one device ($\mathcal{H}_0$) where we could not find any bit flip on *Coffee Lake* while ZENHAMMER is successful on *Zen 2*. Generally, our optimizations seem to be more effective on *Zen 3*, where the number of bit flips of the best pattern during the sweep is in 5 out of 6 cases higher than on *Coffee Lake*. In the best case ($\mathcal{S}_2$), we find 46x more bit flips on *Zen 3* (79,306) than on *Coffee Lake* (1,708). These results suggest that the effectiveness of a Rowhammer attack does not entirely depend on the activation rate, which is generally higher on *Coffee Lake* than on *Zen 3*, but also on enforcing the order of aggressor accesses (i.e., the fencing policy) and CPU-specific memory controller optimizations.

Table 11. Reverse engineered address mappings and offsets for our *Zen 4* (Ryzen 7 7700X) system. All memory configurations are single-channel, single-DIMM, with the tuple indicating the DIMM's geometry (#subchannels, #ranks, #bank groups, #banks per bank group, #rows).

| **Geometry** (SC,RK,BG,BA,R) | **Size** [GiB] | **Offt.** [MiB] | **DRAM Address Functions** | | | | **Row Bits** |
|---|---|---|---|---|---|---|---|
| | | | Subchannel | Rank | Bank Group (BG) | Bank Address (BA) | |
| (2, 1, 4, 4, $2^{16}$) | 8 | 2048 | `0x1fffe0040` | n/a | `0x088880100`, `0x111100200` | `0x022220400`, `0x044440800` | 32 – 17 |
| (2, 1, 8, 4, $2^{16}$) | 16 | 2048 | `0x3fffc0040` | n/a | `0x042100100`, `0x084200200`, `0x108401000` | `0x210840400`, `0x021080800` | 33 – 18 |
| (2, 2, 8, 4, $2^{16}$) | 32 | 2048 | `0x7fff80040` | `0x000040000` | `0x084200100`, `0x108400200`, `0x210801000` | `0x421080400`, `0x042100800` | 34 – 19 |

**Exploitability Analysis.** The larger number of bit flips after our optimizations strongly facilitates exploitation, as we show in Table 10. The PTE attack by Seaborn [35] can be exploited in the best case in around one second on both *Zen 3* and *Coffee Lake*. Due to the lower number of exploitable bit flips on *Zen 2*, we need in the best case six times as long (6 s) as on the two other systems. There is one device ($\mathcal{S}_3$) where exploitation is not possible at all on *Coffee Lake* due to missing bit flips, but on *Zen 2* and *Zen 3* we can find exploitable bit flips in 9 s and 32 s, respectively. We note that even if the number of bit flips is very low (e.g., 3 bit flips on $\mathcal{S}_4$, *Zen 3*), we were still able to exploit the system in a practical time (23 m 52 s).

The RSA-2048 key attack [34] is on 4 of 5 exploitable devices on average 38 s faster on *Zen 3* than on *Coffee Lake*. Overall, the average time to find an exploitable bit flip is 3 m 52 s, 29 s, and 1 m 6 s for *Zen 2*, *Zen 3*, and *Coffee Lake*, respectively. We note that the device $\mathcal{H}_0$ with bit flips only on *Zen 2* is not exploitable. Our data shows that even if we find a very low number of patterns only (e.g., 7 pattern for $\mathcal{S}_3$), we still are likely to find an exploitable bit flip (2 m 21 s).

Lastly, the `sudo` binary exploit [11] is the hardest attack as it requires a precise set of bit flips. Given the low number of bit flips on *Zen 2*, we cannot find any exploitable bit flips for this attack. For the remaining platforms, *Zen 3* and *Coffee Lake*, we find an equal number of exploitable devices (4) and a similar average time to find an exploitable bit flip, 3 m 29 s and 3 m 55 s, respectively, when excluding devices with a single bit flip only. The exploitable devices are those that showed the highest number of bit flips while sweeping on these platforms

**End-to-End Attack's Practicality.** As our exploitability analysis is based on simulation results, we further verified the practicality of the PTE attack by Seaborn and Dullien [36]. Our attack's implementation is based on the THP coloring technique described in Section 4.4. Moreover, we modified our ZenHammer fuzzer to use THPs like it has been done before for *n*-sided patterns [9]. This means we distribute aggressors across THPs such that aggressor pairs are placed on the same THP and the pattern is spread across multiple THPs. We successfully verified the attack's feasibility on device $\mathcal{S}_2$. Over ten successful attack runs (i.e., obtaining root privileges), we report an average time of 93 seconds for the end-to-end attack once an exploitable bit flip has been found. This includes the time for THP coloring as reported in Section 4.4.

**Discussion.** These results show that using the techniques we discussed in this paper, ZenHammer enables practical Rowhammer exploits on AMD Zen-based platforms for the first time. We also believe that our insights will make it easier to port Rowhammer attacks to newer platforms in the future, such as DDR5 devices, as we will show next.

### 7.3 ZenHammer on DDR5

As part of our evaluation, we tested whether ZenHammer is effective in triggering bit flips on more recent devices (DDR5). We reverse engineered the DRAM address functions of our *Zen 4* system (Ryzen 7 7700X) and present the functions in Table 11. As for DDR4, we randomly picked ten DDR5 devices (Table 16 in Appendix D) and repeated the experiment described in Section 6.2 to find the best fence scheduling policy for each device.

We found bit flips on only 1 of 10 tested devices ($\mathsf{S}_1$), suggesting that the changes in DDR5 such as improved Rowhammer mitigations, on-die error correction code (ECC), and a higher refresh rate (32 ms) make it harder to trigger bit flips. On $\mathsf{S}_1$ with the policy $\text{SP}_{\text{none}}$, we found 109 patterns and 23,110 bit flips during fuzzing. The best pattern triggered 41,995 bit flips during the sweep over 256 MiB of memory. Given the lack of bit flips on 9 of 10 DDR5 devices, more work is needed to better understand the potentially new Rowhammer mitigations and their security guarantees.

## 8 Related Work

In this section, we discuss differences between *DARE* and existing tools for reverse engineering DRAM address functions (Section 8.1). Thereafter, we discuss similar and orthogonal approaches used to reverse engineer the DRAM address functions (Section 8.2). Lastly, we summarize previous efforts regarding Rowhammer on pre-Zen AMD systems (Section 8.3).

### 8.1 Comparison to Existing Tools

In Table 12, we compare our new reverse engineering tool *DARE* to the open-source tool *DRAMA* [32] and concurrent work *AMDRE* [14]. *DRAMA* was not able to recover the correct DRAM address mappings on our Zen-based systems,

**Table 12.** Comparison of *DARE* with *AMDRE* and *DRAMA*. The table shows features and changes made for correctness (Corr.), noise handling (Noise), and performance improvement (Perf.).

| | Tool | | | Goal | | |
|---|---|---|---|---|---|---|
| | ***DARE*** | ***AMDRE*** | ***DRAMA*** | **Corr.** | **Noise** | **Perf.** |
| **Thresh. Detection** | | | | | | |
| – Autom. Detection | ✔ | ✔ | ✔ | ● | ● | ○ |
| – Reliable Timing | ✔ | ✔ | ✘ | ● | ● | ○ |
| **Clustering** | | | | | | |
| – Superpages | ✔ | ✘ | ✘ | ● | ○ | ○ |
| – Pairwise Testing | ✔ | ✔ | ✘ | ○ | ● | ○ |
| **Brute forcing** | | | | | | |
| – Address Offsets | ✔ | ✘ | ✘ | ● | ○ | ○ |
| – Strict Validation | ✔ | ✔ | ✘ | ● | ○ | ● |

while *AMDRE* could only partially (up to bit 21) recover the *Zen 2* functions due to its limitation to 2 MiB THPs.

Our changes enabled us to recover the complete and correct DRAM address mappings in a fast and reliable way. Like *DRAMA* and *AMDRE*, our tool requires superuser privileges for the virtual-to-physical address translation. However, an attacker could recover the DRAM address mappings *offline*, i.e., on another system with the same hardware configuration. We now discuss our improvements to the existing work.

**Reliable Timing.** The timing routine used in *DRAMA* does not reliably work on AMD Zen-based systems, leading to many outliers. In *AMDRE*, the timing routine works mostly reliably, except for the few occasions where the automatic threshold detection fails. We designed an optimized and more reliable timing routine in Section 4.1.

**Superpages.** During reverse engineering, we use all available 1 GiB superpages as higher physical address bits (above 1 GiB) are involved in some address mappings. Both *DRAMA* and *AMDRE* can be configured to use more memory; however, only with 4 KiB pages and 2 MiB THPs, respectively.

**Pairwise Testing.** We reduce false positives by measuring pairwise latencies for cluster addresses and removing those conflicting with less than 75% of the cluster, thus creating perfect bank clusters. *AMDRE* uses a similar technique to remove false positives.

**Address Offsets.** The functions found by *DRAMA* and *AMDRE* are not valid across the whole physical address space. This is caused by the remapping of physical memory above the 4 GiB mark, which introduces a nonlinearity. *DARE* is the first tool to take this into account by applying a system-specific offset prior to brute forcing the XOR functions.

**Strict Validation.** *DRAMA* only requires that candidate functions do not produce the same result across the clusters. Our and *AMDRE*'s condition is stronger, requiring that every function returns the same result on *exactly* half of all clusters. This condition allows us to filter out many invalid address functions early on during brute forcing the functions.

## 8.2 Comparison to Other Techniques

The approaches used by existing work to reverse engineer the secret DRAM address mappings can be divided into software-based and hardware-based approaches. Software-based approaches generally require side channels, such as bank conflicts. Instead, hardware-based techniques require specialized equipment like a logic analyzer. We compare the existing approaches in Table 13, which we now explain in more detail.

Our comparison considers three categories: requirements, results, and features. For the **Requirements**, we compare the monetary costs involved (*Cst.*), if any special hardware is needed (*HW*), and if the method relies on a side channel (*SC*). In the **Results** category, we look at how generic (*Gen.*) the approach is (i.e., if it also works with different memory configurations), the result's completeness (*Cpl.*) w.r.t. the different DRAM address components, and the result's precision (i.e., how reliable results are). Lastly, the **Features** category considers whether the approach can obtain labels for the found functions (*Lbl.*) and analyze the devices' internal row remapping (*RR*).

**Table 13.** Comparison of existing software-based (top) and hardware-based (bottom) techniques for recovering DRAM address mappings. Our work uses row buffer conflicts to find the functions and an oscilloscope to verify their validity.

| **Technique** | **Requirements** | | | **Results** | | | **Features** | |
|---|---|---|---|---|---|---|---|---|
| | Cst. | HW | SC | Gen. | Cpl. | Prec. | Lbl. | RR |
| ❶ **Row buffer conflict** [5, 14, 32, 40, 42, 43] | ○ | ○ | ● | ◐ | ○ | ◐ | ○ | ○ |
| ❷ Rowhammer [35] | ○ | ● | ● | ● | ○ | ◐ | ○ | ● |
| ❸ Perf. counters [15] | ○ | ● | ○ | ○ | ○ | ● | ◐ | ○ |
| ❹ **Oscilloscope** [32] | ◐ | ● | ○ | ● | ◐ | ● | ● | ○ |
| ❺ Logic analyzer [31] | ● | ● | ○ | ● | ● | ● | ● | ○ |
| ❻ Retention + Temp. [20] | ◐ | ● | ● | ○ | ◐ | ● | ● | ● |

**Requirements.** Software-based approaches ❶–❸ are cost-effective, essentially free. Oscilloscopes ❹ are affordable ◐, while logic analyzers ❺ are more expensive ●. Approach ❻ requires an FPGA and special heating equipment ◐. Using Rowhammer bit flips as side channel ❷ requires a vulnerable device ●, which might be hard to obtain. To the best of our knowledge, only server platforms provide hardware-based performance counters ❸ with DRAM-related data ●. Besides Rowhammer bit flips ❷, other side channels used are row buffer conflicts ❶ and DRAM retention time ❻.

**Results.** Oscilloscopes ❹, logic analyzers ❺, and Rowhammer ❷ are purely generic ● and support any DRAM device configuration. Exploiting row buffer conflicts ❶ may require tweaking timing thresholds in multi-DIMM/-channel setups ◐. Only logic analyzers ❺ can recover all DRAM address components ● as the limited number of channels on oscilloscopes ❹ may make data filtering for some address component hard or impossible ◐. The retention time approach ❻ cannot recover DRAM address bits requiring multiple DRAM devices ◐. The hardware-based approaches ❹–❻ and performance counters ❸ provide high precision ●, whereas row buffer conflicts ❶ require a reliable timing function ◐. Using Rowhammer itself ❷ might be imprecise as mitigations in the memory controller or the devices themselves could disturb the bit flip feedback channel ◐.

**Features.** All hardware-based approaches ❹–❻ provide information to derive labels for DRAM address mappings ●. Depending on the availability, performance counters ❸ may have separate counters per bank and/or rank, allowing to derive some labels only ◐. Rowhammer bit flips ❷ and DRAM retention ❻ are the only techniques allowing to reverse the DRAM-internal row remapping ●.

**Relation to Our Work.** Similar to previous work, we rely on the row buffer conflict side channel ❶ to reverse engineer the DRAM address mappings. However, as the first work, we take the address offset into account and collect addresses from multiple superpages, enabling us to recover the correct mappings on all Zen-based systems. Furthermore, we use an oscilloscope ❹, with the same method as in previous work [32], to physically validate our address mappings.

### 8.3 Rowhammer on AMD

Little attention has been paid to Rowhammer on AMD in the past decade. The original Rowhammer study from 2014 by Kim et al. [22] showed bit flips on Intel and AMD *Piledriver*. In these older systems, using the same hammering instructions on the two systems was still effective. We demonstrated that this is not the case anymore for modern CPUs.

Later, in 2016, a comparative analysis looked into Rowhammer on Intel (*Sandy Bridge*, *Ivy Bridge*, and *Haswell*) and AMD (*Piledriver*) platforms. They showed that not only the access rate is much lower on AMD (6.1 M/s compared to 11.6 M/s–12.3 M/s), but also the number of bit flips observed is roughly two orders of magnitude larger for Intel (16.1 k–22.9 k) than on AMD (59) [27]. Our findings show a lower number of bit flips on AMD *Zen 2* compared to Intel systems, even after our optimizations.

## 9 Conclusion

We presented ZenHammer, the first successful Rowhammer attacks launched from AMD Zen-based CPUs. To build ZenHammer, we needed to overcome a number of challenges including the reverse engineering of the DRAM addressing functions by taking physical address offsets into account, a new mechanism for synchronization with refresh commands, and careful scheduling of flushing and fencing instructions to improve the activation throughput of Rowhammer patterns. ZenHammer is capable of flipping bits on 7 and 6 out of our ten DDR4 samples on AMD *Zen 2* and *3* respectively, enabling Rowhammer exploits on recent AMD platforms for the first time. We further show Rowhammer bit flips on a DDR5 device for the first time.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This research was supported by the Swiss National Science Foundation under NCCR Automation, grant agreement 51NF40_180545, by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE), and by a Microsoft Swiss JRC grant.

## References

[1] PassMark CPU Benchmarks: AMD vs Intel Market Share. URL https://www.cpubenchmark.net/market_share.html.

[2] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, January 2013. URL https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/42301_15h_Mod_00h-0Fh_BKDG.pdf.

[3] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions, November 2021. URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/26568.pdf.

[4] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions, June 2023. URL https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf.

[5] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks. In *IVSW '18*, pages 19–24, July 2018.

[6] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D. Keromytis, Yossi Oren, and Yuval Yarom. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In *CCS '22*, pages 547–561, November 2022.

[7] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *IEEE S&P '19*, pages 55–71, May 2019.

[8] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *IEEE S&P '20*, pages 712–728, May 2020.

[9] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *USENIX Security '21*, pages 1001–1018, August 2021.

[10] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE S&P '20*, pages 747–762, May 2020.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *IEEE S&P '18*, pages 245–261, May 2018.

[12] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO '21*, pages 1198–1213, October 2021.

[13] Wei He, Zhi Zhang, Yueqiang Cheng, Wenhao Wang, Wei Song, Yansong Gao, Qifei Zhang, Kang Li, Dongxi Liu, and Surya Nepal. WhistleBlower: A System-level Empirical Study on RowHammer. *IEEE Transactions on Computers*, pages 1–15, January 2023.

[14] Martin Heckel and Florian Adamsky. Reverse-Engineering Bank Addressing Functions on AMD CPUs. In *DRAMSec '23*, pages 1–6, June 2023.

[15] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters. In *MASCOTS '20*, pages 1–8, November 2020.

[16] Intel. 12th Generation Intel Core Processors, Datasheet Volume 2 of 2, April 2022. URL https://cdrdv2.intel.com/v1/dl/getContent/655259.

[17] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *IEEE S&P '22*, pages 716–734, May 2022.

[18] JEDEC Solid State Technology Association. DDR4 SDRAM, September 2012. URL https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf.

[19] Michael Fahr Jr, Thinh Dang, Hunter Kippen, Jacob Lichtinger, Andrew Kwong, Dana Dachman-Soled, Daniel Genkin, and Alexander Nelson. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. In *CCS '22*, pages 979–993, November 2022.

[20] Matthias Jung, Carl C. Rheinländer, Christian Weis, and Norbert Wehn. Reverse Engineering of DRAMs: Row Hammer with Crosshair. In *MEMSYS '16*, pages 471–476, October 2016.

[21] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA '20*, pages 638–651, May 2020.

[22] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA '14*, pages 361–372, June 2014.

[23] Petr Kobalicek. AsmJit: Low-Latency Machine Code Generation, 2023. URL https://asmjit.com/.

[24] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering From the Next Row Over. In *USENIX Security '22*, pages 3807–3824, August 2022.

[25] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *IEEE S&P '20*, pages 695–711, May 2020.

[26] Zhenrong Lang, Patrick Jattke, Michele Marazzi, and Kaveh Razavi. Blaster: Characterizing the Blast Radius of Rowhammer. In *DRAMSec '23*, pages 1–7, June 2023.

[27] Mark Lanteigne. A Tale of Two Hammers: A Brief Rowhammer Analysis of AMD vs. Intel. Technical report, Third I/O, May 2016. URL http://www.thirdio.com/rowhammera1.pdf.

[28] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *IEEE S&P '23*, pages 1684–1701, May 2023.

[29] Koksal Mus, Yarkın Doröz, M. Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS Signing Keys via Rowhammer Faults. In *IEEE S&P '23*, pages 1719–1736, May 2023.

[30] Lois Orosa, Ulrich Rührmair, A. Giray Yaglikci, Haocong Luo, Ataberk Olgun, Patrick Jattke, Minesh Patel, Jeremie Kim, Kaveh Razavi, and Onur Mutlu. SpyHammer: Using RowHammer to Remotely Spy on Temperature, October 2022. URL https://arxiv.org/abs/2210.04084.

[31] Minesh Patel, Jeremie S. Kim, and Onur Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA '17*, pages 255–268, June 2017.

[32] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security '16*, pages 565–581, August 2016.

[33] Rui Qiao and Mark Seaborn. A New Approach For Rowhammer Attacks. In *HOST '16*, pages 161–166, May 2016.

[34] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security '16*, pages 1–18, August 2016.

[35] Mark Seaborn. How physical addresses map to rows and banks in DRAM, May 2015. URL https://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

[36] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges, March 2015. URL https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[37] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations Against Rowhammer: A Surgical Precision Hammer. In *RAID '18*, pages 48–66, September 2018.

[38] M. Caner Tol, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. Don't Knock! Rowhammer at the Backdoor of DNN Models. In *DSN '23*, pages 109–122, June 2023.

[39] Chihiro Tomita, Makoto Takita, Kazuhide Fukushima, Yuto Nakano, Yoshiaki Shiraishi, and Masakatu Morii. Extracting the Secrets of OpenSSL with RAMBleed. *Sensors*, 22(9):3586, January 2022.

[40] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS '16*, pages 1675–1689, October 2016.

[41] Hari Venugopalan, Kaustav Goswami, Zainul Abi Din, Jason Lowe-Power, Samuel T. King, and Zubair Shafiq. Centauri: Practical Rowhammer Fingerprinting, June 2023. URL https://arxiv.org/abs/2307.00143.

[42] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping. In *DAC '20*, July 2020.

[43] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security '16*, pages 19–35, August 2016.

[44] 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*, pages 28–41, October 2020.

## Appendices

## A Equally-sized Bins in XOR Partition

In Section 4.2, we assumed that the result of any XOR function on a bin of addresses returns either a constant value (i.e., 0 or 1) for all addresses or evenly splits the addresses. We prove this assumption in the following.

**Claim.** Consider an aligned power-of-two range of addresses $A = [m \cdot 2^n, (m+1) \cdot 2^n - 1]$ $(m, n \in \mathbb{N})$, a XOR function $f$ which is non-constant on $A$, and the set of addresses $B = \{a \in A \mid f(a) = 0\}$. Partitioning the addresses in $B$ using a different, non-constant XOR function $g$ results in two equally-sized bins where $g$ is constant 0 and constant 1, respectively.

**Proof.** First, we show that the claim holds for *one* function $g_1 \neq f$. We construct $g_1$ by extending $f$ to include another previously unused bit in the XOR computation.[6] We note that adding this new bit leads to a different function result for exactly half of all addresses in $B$ (namely, those where that address bit is set). As the function result was previously constant 0 for all $b \in B$, it must now be equally distributed between 0 and 1, satisfying our claim.

Second, we show that we can successively modify $g_1$ to obtain an arbitrary function $g$ without changing the size of the two bins. To do this, we successively add (or remove) a bit to (or from) the XOR computation in $g_1$ until reaching $g$. During each of these steps, the function result will flip for half of all addresses. We note that the addresses where the affected bit is set are always split evenly between the two bins. Thus, the affected addresses are split evenly between the two bins, keeping the size of the two bins equal after each step and satisfying our claim for any function $g$.

## B Heatmap of Memory Access Rates

Table 14 shows the same data as Table 7. However, we also show the instruction sequences that were previously excluded due to their throughput either being low ($\leq$ 100 ACTs/tREFI) or very high, indicating cache hits ($\geq$ 1000 ACTs/tREFI).

In Table 15, we show the results of the same experiment for the Intel *Coffee Lake* system.

[6] Alternatively, a bit could be removed from the XOR computation.

**Table 14.** Heatmap of memory access rates (in ACTs/tREFI) for all tested instruction sequences and varying numbers of accessed rows on the AMD $Z_3$ system. We abbreviate scatter, fence each by "s.f.e."

| Access Type | Flushing Strategy | Fence Type | #Rows 1 | 2 | 4 | 8 | 16 | 32 | 256 |
|---|---|---|---|---|---|---|---|---|---|
| `MOV` (load) | gather | M | 24 | 49 | 71 | 91 | 100 | 110 | 114 |
| `MOV` (load) | gather | L | 24 | 49 | 80 | 113 | 134 | 147 | 121 |
| `MOV` (load) | gather | S | 24 | 49 | 80 | 113 | 133 | 146 | 125 |
| `MOV` (load) | gather | — | 24 | 49 | 80 | 113 | 133 | 146 | 125 |
| `MOV` (load) | scatter | M | 24 | 49 | 79 | 107 | 126 | 143 | 157 |
| `MOV` (load) | scatter | L | 24 | 49 | 95 | 137 | 149 | 153 | 159 |
| `MOV` (load) | scatter | S | 24 | 48 | 97 | 154 | 159 | 159 | 159 |
| `MOV` (load) | scatter | — | 24 | 49 | 97 | 154 | 159 | 159 | 159 |
| `MOV` (load) | s.f.e. | M | 24 | 33 | 33 | 33 | 33 | 34 | 34 |
| `MOV` (load) | s.f.e. | L | 24 | 49 | 65 | 70 | 69 | 71 | 70 |
| `MOV` (load) | s.f.e. | S | 24 | 41 | 70 | 72 | 72 | 73 | 74 |
| `MOV` (store) | gather | M | 24 | 32 | 50 | 67 | 71 | 72 | 72 |
| `MOV` (store) | gather | L | 24 | 32 | 49 | 66 | 72 | 71 | 71 |
| `MOV` (store) | gather | S | 24 | 32 | 49 | 67 | 67 | 73 | 72 |
| `MOV` (store) | gather | — | 24 | 32 | 49 | 67 | 70 | 73 | 72 |
| `MOV` (store) | scatter | M | 24 | 32 | 54 | 72 | 73 | 73 | 72 |
| `MOV` (store) | scatter | L | 24 | 32 | 54 | 72 | 73 | 72 | 72 |
| `MOV` (store) | scatter | S | 24 | 32 | 52 | 73 | 72 | 73 | 72 |
| `MOV` (store) | scatter | — | 24 | 32 | 54 | 73 | 73 | 73 | 72 |
| `MOV` (store) | s.f.e. | M | 24 | 24 | 28 | 28 | 28 | 28 | 28 |
| `MOV` (store) | s.f.e. | L | 24 | 32 | 53 | 72 | 74 | 75 | 72 |
| `MOV` (store) | s.f.e. | S | 24 | 24 | 48 | 49 | 49 | 50 | 50 |
| `MOVNTDQA` | none | — | 15K | 20K | 24K | 26K | 28K | 26K | 12K |
| `MOVNTI` | none | — | 15K | 24K | 29K | 8K | 618 | 367 | 131 |
| `PREFETCHNTA` | none | — | 15K | 24K | 29K | 29K | 30K | 27K | 19K |
| `PREFETCHNTA` | scatter | — | 80 | 132 | 191 | 208 | 253 | 309 | 273 |
| `PREFETCHNTA` | scatter | M | 24 | 49 | 80 | 108 | 131 | 170 | 284 |
| `VGATHERDD` | scatter | — | 24 | 49 | 79 | 112 | 159 | 159 | 159 |

**Table 15.** Heatmap of memory access rates (in ACTs/tREFI) for all tested instruction sequences and varying numbers of accessed rows on the Intel CL system. We abbreviate scatter, fence each by "s.f.e."

| Access Type | Flushing Strategy | Fence Type | #Rows 1 | 2 | 4 | 8 | 16 | 32 | 256 |
|---|---|---|---|---|---|---|---|---|---|
| `MOV` (load) | gather | M | 83 | 110 | 130 | 144 | 150 | 153 | 158 |
| `MOV` (load) | gather | L | 151 | 90 | 115 | 142 | 146 | 154 | 159 |
| `MOV` (load) | gather | S | 248 | 128 | 138 | 148 | 159 | 154 | 159 |
| `MOV` (load) | gather | — | 226 | 136 | 160 | 160 | 163 | 153 | 159 |
| `MOV` (load) | scatter | M | 83 | 110 | 130 | 144 | 152 | 156 | 160 |
| `MOV` (load) | scatter | L | 151 | 98 | 121 | 144 | 154 | 157 | 160 |
| `MOV` (load) | scatter | S | 186 | 121 | 142 | 156 | 160 | 160 | 160 |
| `MOV` (load) | scatter | — | 248 | 128 | 143 | 160 | 160 | 160 | 160 |
| `MOV` (load) | s.f.e. | M | 83 | 83 | 83 | 83 | 83 | 83 | 83 |
| `MOV` (load) | s.f.e. | L | 156 | 100 | 99 | 99 | 99 | 99 | 99 |
| `MOV` (load) | s.f.e. | S | 164 | 120 | 137 | 160 | 160 | 160 | 160 |
| `MOV` (store) | gather | M | 87 | 154 | 233 | 322 | 464 | 646 | 87 |
| `MOV` (store) | gather | L | 95 | 206 | 364 | 670 | 867 | 852 | 87 |
| `MOV` (store) | gather | S | 94 | 150 | 262 | 427 | 611 | 712 | 87 |
| `MOV` (store) | gather | — | 94 | 206 | 361 | 670 | 871 | 849 | 88 |
| `MOV` (store) | scatter | M | 89 | 93 | 82 | 101 | 103 | 98 | 86 |
| `MOV` (store) | scatter | L | 94 | 183 | 108 | 110 | 108 | 108 | 86 |
| `MOV` (store) | scatter | S | 95 | 116 | 92 | 101 | 106 | 106 | 86 |
| `MOV` (store) | scatter | — | 94 | 187 | 111 | 112 | 108 | 107 | 86 |
| `MOV` (store) | s.f.e. | M | 89 | 54 | 53 | 51 | 51 | 50 | 50 |
| `MOV` (store) | s.f.e. | L | 94 | 190 | 107 | 116 | 109 | 107 | 86 |
| `MOV` (store) | s.f.e. | S | 94 | 71 | 70 | 69 | 70 | 69 | 68 |
| `MOVNTDQA` | none | — | 12K | 17K | 14K | 25K | 17K | 7K | 6K |
| `MOVNTI` | none | — | 12K | 9K | 14K | 15K | 950 | 721 | 107 |
| `PREFETCHNTA` | none | — | 13K | 19K | 14K | 29K | 32K | 14K | 221 |
| `PREFETCHNTA` | scatter | — | 253 | 314 | 261 | 271 | 160 | 160 | 160 |
| `PREFETCHNTA` | scatter | M | 83 | 110 | 130 | 144 | 152 | 156 | 160 |
| `VGATHERDD` | scatter | — | 156 | 219 | 228 | 143 | 152 | 160 | 160 |

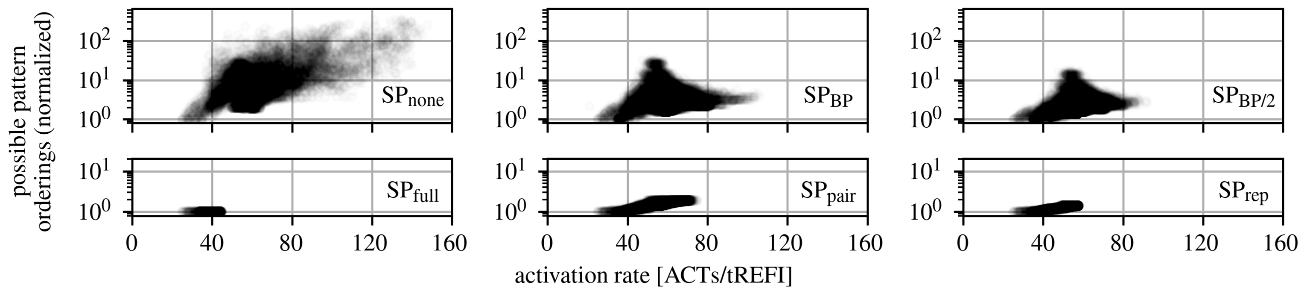

**Figure 7.** Activation rates and possible pattern orderings for non-uniform hammering patterns when using different scheduling policies. The data was collected on $Z_3$ using the `MFENCE` barrier.

## C Modelling Fence Scheduling Policies

In this appendix, we first present a theoretical model for the amount of ordering enforced by a scheduling policy (as described in Section 6.2) based on a simple CPU behavior model. We then evaluate the trade-off provided by different scheduling policies by contrasting the amount of ordering provided with the patterns' hammering speeds.

**Computing Pattern Permutations** To analyze the amount of ordering provided by a scheduling policy, we use a model for the processor's memory subsystem which assumes that (a) load requests cannot be reordered around memory barriers, as guaranteed by `M/LFENCE` [4], and (b) all load requests are served by DRAM, including consecutive ones to the same cache line with flushing in between accesses (Obs. 4). Using this model, we can compute the number of theoretically possible orderings of a hammering pattern.

We assume that patterns are always ordered at their beginning and their end, and we compute the number of permutations for each interval (delineated by memory barriers) individually. For a multiset $M$, containing $l$ different elements with multiplicities $m_1, m_2, \dots, m_l$, the number of permutations is given by the multinomial coefficient $\binom{m}{m_1, m_2, \dots, m_l} = \frac{m!}{m_1!\, m_2! \dots m_l!}$. To obtain the total number of all permutations, we multiply the numbers for the different intervals.

In practice, it is highly unlikely that memory accesses are reordered over large distances, even if theoretically possible based on ordering semantics. However, as the realistic extent of reordering is unknown, we use this simpler model.

**Example.** To illustrate, we use an example non-uniform pattern $|a_1\ a_2\ a_1\ a_2\ a_3\ a_4|$ where fences are shown using vertical bars. The number of possible orderings is computed as $\binom{6}{2,2,1,1} = \frac{6!}{2!2!1!1!} = 180$. When inserting another fence after the fourth access (corresponding to SP$_{\text{pair}}$), we get the pattern $|a_1\ a_2\ a_1\ a_2|a_3\ a_4|$ with $\binom{4}{2,2} \cdot \binom{2}{1,1} = 12$ possible orderings. By inserting a single memory barrier in the middle of the patterns, the number of possible orderings has been reduced drastically.

**Ordering vs. Hammering Speed.** To explore the trade-off provided by our scheduling policies, we contrast the provided ordering and hammering speeds of 15 K random non-uniform patterns. We implement all proposed scheduling policies (see Table 8) in our fuzzer, hammer the generated patterns using the different policies, and record their activation rates. We then compute the number of pattern permutations using the theoretical model introduced above.[7]

We plot the results for $Z_3$ in Figure 7, where we show, for each policy and each generated pattern, the hammering speed ($x$-axis) and the number of possible orderings ($y$-axis). We omit the similar results from $Z_+$, where we also ran this experiment.

[7]To account for different pattern lengths ($L$), we use the normalized ordering metric $\tilde{N} := \sqrt[L]{N}$, where $N$ is the number of possible orderings.

**Observations.** As expected, the scheduling policies differ significantly in the trade-off they provide. SP$_{\text{none}}$ provides very high activation rates, as it allows the most reordering. On the contrary, SP$_{\text{full}}$ allows zero reordering at the expense of low activation rates (of 37 ACTs/tREFI on average). The pattern-aware policies show two different types of distributions. For SP$_{\text{BP}}$ and SP$_{\text{BP/2}}$, the distributions are somewhat similar to SP$_{\text{none}}$, albeit without the very fast outliers. On the other hand, SP$_{\text{pair}}$ and SP$_{\text{rep}}$ provide ordering that is nearly as strict as SP$_{\text{full}}$, while allowing faster hammering when compared to the latter, with average activation rates increased by 51 % (SP$_{\text{pair}}$) and 39 % (SP$_{\text{full}}$) respectively.

Based on these results, we believe SP$_{\text{pair}}$ and SP$_{\text{rep}}$ could be well suited to reduce the amount of fencing without significantly impacting a pattern's ordering.

## D Analyzed DDR5 Devices

In Table 16, we present the list of ten randomly chosen DDR5 UDIMMs covering all three major manufacturers, i.e., Samsung, SK Hynix, and Micron. We report each device's production date, speed, size, and DRAM geometry.

**Table 16.** DDR5 UDIMMs used in the evaluation of our AMD *Zen*-optimized Rowhammer fuzzer. We abbreviate the DRAM vendors Samsung (S), SK Hynix (H), and Micron (M). We report for each device, the number of subchannels (SC), ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R).

| ID | Production Date | Freq. [MHz] | Size [GiB] | DIMM Geometry (SC,RK,BG,BA,R) |
|---|---|---|---|---|
| S$_0$ | Q4-2021 | 4800 | 8 | (2, 1, 4, 4, $2^{16}$) |
| S$_1$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |
| S$_2$ | Q4-2021 | 5600 | 8 | (2, 1, 4, 4, $2^{16}$) |
| S$_3$ | Q4-2021 | 4800 | 8 | (2, 1, 4, 4, $2^{16}$) |
| H$_0$ | Q4-2021 | 4800 | 8 | (2, 1, 4, 4, $2^{16}$) |
| M$_0$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |
| M$_1$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |
| M$_2$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |
| M$_3$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |
| M$_4$ | Q4-2021 | 4800 | 16 | (2, 1, 8, 4, $2^{16}$) |