ρHammer: Reviving RowHammer Attacks on New Architectures via Prefetching

Weijie Chen
Huazhong Univ. of Sci. and Tech.
Wuhan, China
The Hong Kong Polytechnic Univ.
Hong Kong, China
weijie_chen@hust.edu.cn

Shan Tang*
Huazhong Univ. of Sci. and Tech.
Wuhan, China
m202372164@hust.edu.cn

Yulin Tang*
Huazhong Univ. of Sci. and Tech.
Wuhan, China
m202372045@hust.edu.cn

Xiapu Luo
The Hong Kong Polytechnic Univ.
Hong Kong, China
csxluo@comp.polyu.edu.hk

Yinqian Zhang†
Southern Univ. of Sci. and Tech.
Shenzhen, China
yinqianz@acm.org

Weizhong Qiang‡
Huazhong Univ. of Sci. and Tech.
Wuhan, China
wzqiang@hust.edu.cn

Abstract

Rowhammer is a critical vulnerability in dynamic random access memory (DRAM) that continues to pose a significant threat to various systems. However, we find that conventional load-based attacks are becoming highly ineffective on the most recent architectures such as Intel Alder and Raptor Lake. In this paper, we present ρHammer, a new Rowhammer framework that systematically overcomes three core challenges impeding attacks on these new architectures. First, we design an efficient and generic DRAM address mapping reverse-engineering method that uses selective pairwise measurements and structured deduction, enabling recovery of complex mappings within seconds on the latest memory controllers. Second, to break through the activation rate bottleneck of load-based hammering, we introduce a novel prefetch-based hammering paradigm that leverages the asynchronous nature of x86 prefetch instructions and is further enhanced by multi-bank parallelism to maximize throughput. Third, recognizing that speculative execution causes more severe disorder issues for prefetching, which cannot be simply mitigated by memory barriers, we develop a counter-speculation hammering technique using control-flow obfuscation and optimized `NOP`-based pseudo-barriers to maintain prefetch order with minimal overhead. Evaluations across four latest Intel architectures demonstrate ρHammer's breakthrough effectiveness: it induces up to 200K+ additional bit flips within 2-hour attack pattern fuzzing processes and has a 112× higher flip rate than the load-based hammering baselines on Comet and Rocket Lake. Also, we are the first to revive Rowhammer attacks on the latest Raptor Lake architecture, where baselines completely fail, achieving stable flip rates of 2,291/min and fast end-to-end exploitation.

CCS Concepts

• **Security and privacy → Hardware attacks and countermeasures**; *Side-channel analysis and countermeasures.*

Keywords

Rowhammer, Prefetching, DRAM Address Mapping, Speculative Execution, Memory Barrier, Security, Reliability

1 Introduction

Rowhammer is a well-known circuit-level vulnerability within DRAM that exploits the read disturbance effect to flip the value stored in memory cells (a.k.a. activation-induced bit flips, AIB [50]) by repeatedly opening and closing their nearby DRAM rows for sufficient times. Since its first-time discovery [35], this critical issue has drawn profound attention from both industry and academia for over a decade. At its core, Rowhammer attack enables the manipulation of memory contents without directly accessing them, which has proven its potential to compromise a wide range of systems and application scenarios, such as privilege escalation and model intelligence degradation [5, 7, 11, 12, 15, 16, 22, 28, 43, 57, 60, 62–64, 68, 69, 73, 74], and resulted in significant security implications on system isolation and trusted computing [21, 23, 53, 55, 70, 71].

Currently, it has been widely acknowledged that the Rowhammer issue would keep exacerbating with the evolution of DDR standards and the ever-increasing density of DRAM cells [35]. In response, valiant efforts have been made to address this issue [1, 3, 38, 41, 54, 72, 77], with target row refresh (TRR) currently being the most widely implemented defense mechanism in off-the-shelf DDR4 devices. TRR works by periodically refreshing a subset of rows being frequently activated, thereby effectively preventing early Rowhammer strategies that relied on simple patterns like the double-sided hammering (i.e., two aggressor rows sandwiching one victim row) [42, 48, 49]. Unfortunately, recent

*Both authors contributed equally to this research.
†Yinqian Zhang is affiliated with Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering of SUSTech.
‡Weizhong Qiang is the corresponding author and affiliated with Jinyinhu Laboratory, Hubei Key Laboratory of Distributed System Security, and Hubei Engineering Research Center on Big Data Security.

MICRO 2025, Seoul, Korea

ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756042

research [14, 18, 29, 30, 37] has demonstrated that the TRR mitigation within DDR4 DRAMs can be effectively bypassed, particularly through the state-of-the-art non-uniform hammering techniques that confuse the TRR sampler in identifying the real victim rows.

Nonetheless, it seems that Rowhammer attacks have not evolved to be more potent over time as anticipated. Both this paper and recent literature [30, 33] acknowledge that such attacks are becoming increasingly difficult to realize on newer platforms. After systematically investigating the four most recent Intel architectures, we identify the following non-trivial challenges and propose a novel prefetch-based Rowhammer framework, ρHammer, which is significantly more effective than conventional load-based attacks and revives Rowhammer on the latest architectures.

Challenge I: Complex DRAM Address Mappings

A critical prerequisite for launching effective Rowhammer attacks is the ability to reverse-engineer the correct DRAM address mapping of the target system in order to precisely locate the victim rows. While numerous techniques [30, 56, 65, 68] have been proposed, they primarily rely on incomplete heuristics that become impractical as architectural complexity increases. For instance, in this paper, we identify that the latest Alder/Raptor Lake architectures adopt more intricate mapping schemes with expanded bank bit ranges and bank function combinations, which is likely in response to their supporting DDR5 with extra bank groups [24, 31]. To overcome these limitations, we present a new layout-agnostic reverse-engineering methodology that leverages selective pairwise measurements and structured deduction to accurately and generically recover mappings across architectures. Meanwhile, it also achieves high efficiency and scalability by addressing the combinatorial explosion issue of prior brute-force-based approaches. Evaluation across our hardware setups demonstrates that our method recovers complete mappings within 10 seconds while all available tools completely fail on Alder/Raptor Lake.

Challenge II: Insufficient Activation Rate

One major motivation of ours stems from observing that existing Rowhammer methods, which primarily rely on memory load instructions (e.g., x86 `MOV`) for DRAM accesses and memory barrier instructions to maintain their order [8, 9, 14, 19, 27, 29, 30, 33], are not as effective on newer architectures as originally reported, especially on the most recent Alder/Raptor Lake platforms where they consistently fail. A deeper analysis of this paper reveals that the activation rate of these methods still requires further improvement to make Rowhammer attacks (more) generic and exploitable. To address this, we introduce a novel prefetch-based hammering paradigm that leverages the asynchronous and non-blocking characteristics of x86 prefetch instructions to significantly enhance the hammering throughput. In addition, by combining prefetching with the multi-bank technique, we are able to amplify our attacks even further through bank-level parallelism.

Challenge III: Speculative Disorder Hazard of Prefetching

Modern speculative execution mechanisms, particularly out-of-order (OoO) execution and branch prediction, present a significant challenge to Rowhammer attacks by disrupting the intended instruction order and thereby reducing the overall hammering effectiveness. While both load-based and prefetch-based hammering are affected by this issue, we find that prefetch instructions exhibit even more severe disorder, particularly on the latest Alder/Raptor Lake platforms. Even worse, unlike explicit memory reads, prefetching is implicit access and cannot be reliably ordered following the traditional wisdom of using memory barrier instructions. To overcome the prefetching disorder issue, we develop a counter-speculation hammering technique that combines runtime control flow obfuscation with carefully tuned `NOP`-based pseudo-barriers. This technique mitigates the reordering effects derived from both OoO execution and branch prediction, restoring more controllable prefetching behavior that unlocks the potential of prefetch-based hammering.

Our evaluation demonstrates a substantial leap in attack effectiveness with ρHammer compared to the conventional load-based Rowhammer baselines. During our large-scale 2-hour Rowhammer fuzzing processes, we observe a maximum of over 200K additional bit flips on Comet Lake alone. Even on newer Alder/Raptor Lake platforms where the baseline fails to produce any meaningful bit flips, ρHammer consistently induces hundreds to thousands. Moreover, repeated attack simulation experiments across diverse physical locations reveal ρHammer's excellent stability and practicality: ρHammer achieves sustained flip rates of 187K/min and 47K/min on Comet/Rocket Lake, representing 112.4x and 47.1x improvements over the baseline. On Alder/Raptor Lake, it further maintains highly practical rates of 995/min and 2,291/min, despite the baseline could not reproduce any flips.

Summary of Contributions. The main contributions of this paper are as follows:

- We propose a generic and efficient DRAM address mapping reverse-engineering method that avoids brute-forcing using structured pairwise measurements without any prior knowledge and assumptions about the mapping pattern, achieving full mapping recovery within 10 seconds across all tested recent platforms.
- We introduce a novel prefetch-based Rowhammer attack paradigm that significantly boosts activation rate by leveraging the asynchronous nature of prefetch instructions, further enhanced by combining it with multi-bank parallelism.
- We develop a counter-speculation hammering technique that mitigates the intensified disorder effects of OoO execution and branch prediction, enabling more effective prefetch-based Rowhammer on the latest architectures.
- We open-source our implementation on Github at https://github.com/rhohammer/rhohammer and evaluate our attack on four recent Intel architectures and show that it achieves orders-of-magnitude improvements in bit flip counts and flip rates compared to load-based baselines, including Alder/Raptor Lake platforms where baselines consistently fail.

Responsible Disclosure. We have reported potential security impact of ρHammer to Intel and are awaiting their response.

2 Background

2.1 Dynamic Random Access Memory

DRAM is a volatile memory technology widely equipped as the main memory in modern computers, which is usually available on the market in the form of dual inline memory modules (DIMM). Figure 1 shows an overview of the DRAM organization. A DIMM communicates with the memory controller (MC) through one of the channels and has chips on one or both sides (i.e., single/dual-rank).

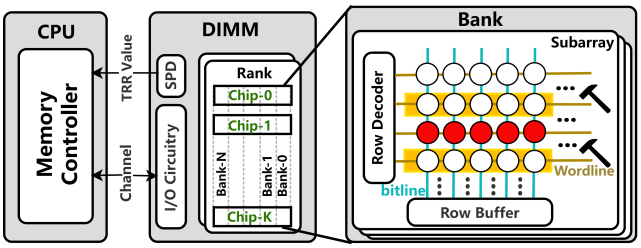

Figure 1: DRAM hierarchical organization.

Each channel operates independently, while each rank processes commands from the MC in lockstep. Orthogonally, each chip on a rank is divided into multiple banks (typically 16 banks per rank on DDR4), where each bank contains subarrays that share the same global row buffer and each subarray is a grid of memory cells. Before each DRAM access, one wordline is activated and all cells connected to it are opened, allowing the stored 0/1 value to be loaded to the row buffer, which later serves all subsequent read/write operations. As capacitors in DRAMs leak charge over time, periodic refreshing is needed to preserve data, where each refresh command refreshes a subset of rows and is issued every $t_{REFI} \approx 7.8\,\mu s$ on average.

SBDR Side Channel. When two subsequent DRAM accesses targeting the same bank but different rows (SBDR), the timing latency will be significantly higher than in other cases, namely same-row (SR) and different-bank (DB), forming the SBDR side channel (a.k.a row conflict side channel). The fundamental reason lies in the fact that only one row can be loaded into the global row buffer per bank at any given time, which accelerates subsequent accesses hitting the row buffer. However, to access a second row within the same bank, a precharge (`PRE`) command must be issued to close the currently open row, introducing an extra latency.

DRAM Address Mapping. In modern computer systems, there exists a well-defined mapping between physical addresses and geographical DRAM addresses, which is essential for the memory controller to efficiently fetch data from the correct DRAM cell. Specifically, physical addresses are translated into the bank, row, and column addresses, the process of which is governed by the memory controller and thus is CPU-specific. Accurate knowledge of the DRAM addressing function is essential for effective Rowhammer exploit, especially the row bits and bank functions (typically linear xors of bank bits), but this information is usually kept proprietary in most commercial CPUs [14, 18]. Previous studies have leveraged the SBDR side channel to reverse-engineer such mapping information [20, 30, 46, 56, 65, 68, 76].

2.2 Rowhammer Attacks

Rowhammer is a critical DRAM vulnerability that exploits the read disturbance effect, where repeated accesses to a specific memory row (aggressor row) induce bit flips in adjacent rows (victim rows). This phenomenon occurs because each activation of a row draws a tiny amount of charge from nearby rows through capacitive coupling or electron migration [50]. Over time, this cumulative charge leakage leads to the state of a memory cell unintentionally flips from 0 to 1 or vice versa. In response, the industry has widely integrated the TRR mitigation on modern DDR4 devices, which works by detecting potential victim rows during regular refresh operations and proactively refreshing them to prevent bit flips. While TRR has been effective in mitigating traditional Rowhammer attacks, recent research has successfully bypassed these defenses [14, 29, 30]. Notably, recent non-uniform hammering [29, 30] represents the most effective state-of-the-art approach that uses carefully crafted patterns to make the TRR mechanism more difficult to determine the correct victim row.

Arch. Name	CPU (Intel Core)	Max Mem Freq.
Comet Lake	i7-10700K	2933
Rocket Lake	i7-11700	2933
Alder Lake	i9-12900	3200
Raptor Lake	i7-14700K	3200

Table 1: Desktop machine setups.

ID	Production Date	Freq. (MHz)	Size (GiB)	Geometry (RK, BK, R)
$\mathcal{S}_1$	W35-2023	3200	16	(2, 16, 2^{16})
$\mathcal{S}_2$	W33-2021	3200	8	(1, 16, 2^{16})
$\mathcal{S}_3$	W30-2020	2933	16	(2, 16, 2^{16})
$\mathcal{S}_4$	W49-2018	2666	16	(2, 16, 2^{16})
$\mathcal{S}_5$	W22-2017	2400	16	(2, 16, 2^{16})
$\mathcal{H}_1$	W13-2020	2666	16	(2, 16, 2^{16})
$\mathcal{M}_1$	W01-2024	3200	32	(2, 16, 2^{17})

Table 2: DDR4 UDIMMs used in this paper. The DRAM vendors are denoted as $\mathcal{S}/\mathcal{H}/\mathcal{M}$. RK, BK, and R represent the number of ranks, banks, and rows.

2.3 x86 Prefetch Instructions

These are a set of specified instructions designed to improve memory access performance by fetching data into the cache before it is actually needed. In contrast with hardware prefetching widely integrated in modern caches that is completely transparent to users, prefetch instructions (i.e., software prefetching) provide an interface for programmers and compilers to explicitly hint the CPU about future accesses. One notable performance aspect is that these instructions are asynchronous and do not stall the CPU while the data is being fetched, which allows the CPU to continue executing other tasks concurrently. The x86 ISA supports several `PREFETCHh` instructions, namely `PREFETCHT0` (fetch to all cache levels), `PREFETCHT1` (fetch to the L2 and LLC), `PREFETCHT2` (fetch to the LLC), and `PREFETCHNTA` (non-temporal), in decreasing order of data temporal locality [25]. We exclude the prefetch write instruction `PREFETCHw` in this paper as it further changes the cache coherence state that may cause extra overhead than `PREFETCHh` instructions [17].

3 Reverse-Engineering Complex DRAM Address Mappings

3.1 Settings

As listed in Table 1, this paper covers consecutive generations of desktop Intel Core processors that represent the most recent commercial-off-the-shelf architectures, spanning across Comet Lake (10th-gen), Tiger Lake (11th-gen), Alder Lake (12th-gen), and Raptor Lake (14th-gen). The 13th-gen is skipped as it also primarily uses the Raptor Lake architecture. All machines use a Linux kernel

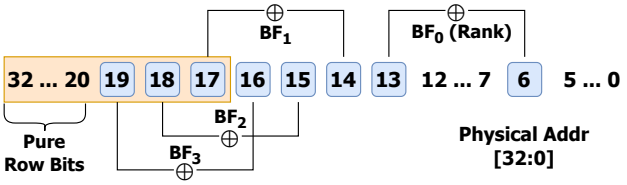

Figure 2: A traditional DRAM address mapping example.

version of 6.8.0-generic with BIOS settings such as device timings, memory frequencies, and hardware prefetchers at their default state. Although we do not discuss hardware prefetching in this paper as is much less controllable, it is important to note that software prefetch instructions are considered as hints to such speculative memory fetching behavior of the processors. For DRAM chips, as listed in Table 2, we use all brand-new DIMMs recently bought from the market covering the three major DRAM manufacturers. In order to better characterize each DIMM separately, we use single-channel configurations throughout this paper.

System Requirements. Following prior assumptions [2, 13, 15, 40, 57, 60, 61, 63, 64], our reverse-engineering process is offline that requires root privileges. While the x86 cache flush instruction for touching the DRAM is all-user available, we need the root privilege for virtual-to-physical translation information via the `/proc/pid/pagemap` Linux interface. Also, we utilize the high-resolution timer (i.e., x86 `RDTSCP`) for all timing measurements throughout this paper. Note that while the bank functions can be empirically classified into channel, rank, bank group, and intra-group bank functions, we will not differentiate them in the remainder of this paper, as they basically serve the same role in Rowhammer attacks that change the geographical bank location. Furthermore, as Rowhammer only relies on row-granularity operations, we hereinafter exclude the discussion of column bits (i.e., the Rowpress [45] effect is out of scope).

3.2 Preliminaries

Figure 2 illustrates a traditional and typical mapping example on the Comet Lake machine, in which the entire physical address space contains row bits (orange), bank bits (blue), and column bits. In fact, bank bits may share a part with row and column bits, where we refer *pure row bits* to those contribute to row addressing alone. Such patterns are designed to maximize the intra-row and inter-bank utilization, and thus benefit the overall memory throughput. However, existing reverse-engineering methods rely heavily on heuristics based on established memory mappings, which assume simpler and more predictable patterns of row and bank bits. Generally, we identify two fundamental sources of limitation in prior works when extending them towards more recent architectures.

Complicated Bank Addressing. Despite different implementations, the majority of existing approaches [30, 56, 65] essentially utilize a brute-force-based method that basically assigns each address to a bank (i.e., coloring) and exhausts all possible bank functions, as initially proposed by DRAMA [56]. These methods were effective on older architectures, where the bank function was relatively simple and could be quickly determined. However, as more complex modern DRAM mappings incorporate more per-function and total bank bits, brute-forcing the bank functions becomes increasingly non-scalable. Empirically, the total number of candidate bank bits can increase from 19 to 32 and the maximum number of per-function bank bits can expand from 2 to 7, which would result in an 8,192x increase in the total search space and a staggering 23,800x increase in function search overhead.

Algorithm 1 Reverse-Engineering Process of ρHammer

Input: M: allocated large physical address pool
Output: $bank\_funcs$: recovered set of bank functions;
B_{row}: recovered set of row bits

```
thres = get_thres_prob_dist(M)
bank_funcs, B_row = {}
B_pure_row, B_non_pure_row = exclude_pure_row_bits(M)
for b_x, b_y ∈ B_non_pure_row do
    if avg(T_SBDR(M, {b_x, b_y})) > thres then
        bank_funcs ← (b_x, b_y) // Duet
    end if
end for
B_row = collect_higher(bank_funcs) ∪ B_pure_row
B_non_row = {}
select ∀ (b_BF, b'_BF) ∈ bank_funcs
for b_x ∈ B_non_pure_row ∧ b_x ∉ B_row do
    if avg(T_SBDR(M, {b_BF, b'_BF, b_x})) < thres then
        B_non_row ← b_x // Trios
    end if
end for
for b_x, b_y ∈ B_non_row do
    if avg(T_SBDR(M, {b_BF, b'_BF, b_x, b_y})) > thres then
        bank_funcs ← (b_x, b_y) // Quartet
    end if
end for
merge(bank_funcs)
return bank_funcs, row_bits
```

Expired Layout Assumptions. In addition, some studies also attempt to enhance the reverse-engineering process by exploiting existing domain knowledge on bit positions, such as excluding pure row bits (see Figure 2) in the first place to narrow the brute-force search space [65], as well as implicitly presume that all bank functions must include row bits [68]. Nevertheless, as we have observed from newer mappings (to be demonstrated), the former strategy is rendered ineffective as pure row bits no longer exist and the latter would overlook low-order functions that cannot be directly derived through the SBDR timings.

To overcome the foregoing limitations, we devise a novel and generic reverse-engineering method that remains both efficient and scalable across architectures. Different from previous efforts, our technique achieves fast DRAM address mapping recovery with *no* a-priori assumptions such as 1) the total number of bank bits, 2) the size of individual bank functions, or 3) the overlapping relationship between bank and row bits. Instead, it treats the entire physical address vector as an unknown search space and incrementally infers the mapping in a structured deductive manner, which also effectively reduces the complexity from the exponential brute-forcing to polynomial time.

3.3 Design

Algorithm 1 shows an overview of our reverse-engineering process. Throughout the entire process (line 4/12/17), we use a pairwise timing primitive defined as $T_{SBDR}(M, B_{diff})$: the DRAM access timing of an arbitrary address pair selected from the allocated memory pool M, where the pair addresses differ only in the bits specified by the set $B_{diff} = \{b_i\}$, while other bits remain identical. Each primitive is derived from the average of 16 random address pairs, each accessed 50 times, and then compared with a threshold to determine whether it indicates a slower SBDR timing or not.

Without loss of generality, we break down the mapping recovery into three sub-objectives: 1) **O1**: identify bank functions that include row bits (i.e., row-inclusive functions); 2) **O2**: determine the range of row bits; 3) **O3**: identify bank functions that exclude row bits (i.e., non-row functions). Then we elaborate on each step of the algorithm and how it addresses the above objectives.

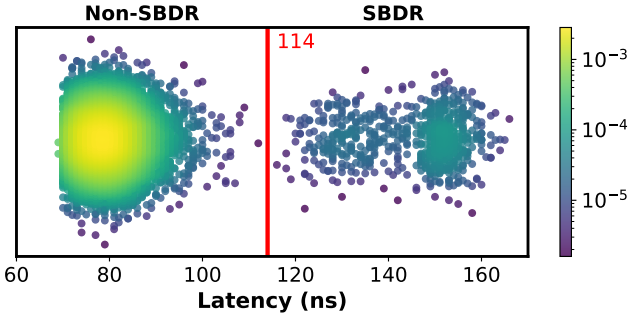

Figure 3: Top-down density distribution map of access latencies with an example threshold; the color scale indicates the proportion of address pairs.

Step 0: Finding the SBDR Threshold. The process begins by allocating 4KB pages that occupy more than half (we set this proportion to 70%) of the system memory to ensure that we can cover the entire physical address space without missing any potential bank bit. Then, we utilize Linux's pagemap interface to retrieve the page frame numbers of all allocated addresses and maintain their virtual-to-physical mappings, which allows us to quickly select virtual address pairs for T_{SBDR} measurements. Essential to the reliability of the entire process, a reasonable threshold is needed to distinguish slower SBDR timings from normal ones (line 1). Therefore, we employ a probability distribution-based method that randomly selects address pairs from (a subset of) our previously allocated memory region. Figure 3 shows an example of the distribution of two assembly areas representing SBDR and non-SBDR pairs, whose ratio is approximately $\frac{1}{\#Banks-1}$. Notably, this ratio is the most accurate if the memory region is aligned to a power-of-two and is large enough to cover at least one bit (i.e., the lowest one) of each bank function. In this way, its total amount of addresses will be uniformly distributed across all banks, because each bank function will evenly cut the region into 0/1-value halves.

Step 1: Duet. To identify all *row bits and their related bank functions*, we set $B = \{b_x, b_y\}$ and exhaust all combinations of T_{SBDR}. Figure 4 presents the heatmap results illustrating the significant mapping difference across CPU generations, as discussed in Section 3.2. The highlighted blocks indicate slower SBDR timings when

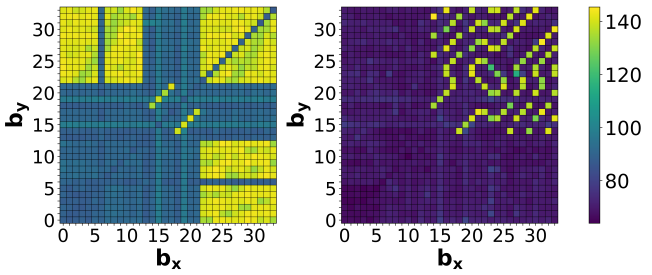

Figure 4: Heatmap of $T_{SBDR}(M, \{b_x, b_y\})$ in *ns* on Comet Lake (left) and Raptor Lake (right), representing traditional and recent mappings, respectively.

accessing pairs where both b_x and b_y differ. In the left heatmap, it is noticeable that large chunks of highlighted areas are prevalent, which is due to the presence of high-order pure row bits. When b_x and b_y are both pure row bits (top right area), or one of them is a pure row bit while another is a pure column bit (top left and bottom right areas), there will be SBDR timings as the row is changed but the bank is not. Conversely, the missing of such large SBDR chunks in the right heatmap indicates the absence of pure row bits, as their prevalence is now masked by the faster DB timings.

In either case, we first leverage such difference to identify all pure row bits, if any (line 3). Then, for the remaining bits, we scan over (b_x, b_y) pairs for those with SBDR timings as bank functions, which are visualized as scattered blocks across the right heatmap and in the middle of the left heatmap (line 4-8). This is because if and only if b_x and b_y reside in the same bank function (i.e., when even number of bits are different, their xor-ed value remains unchanged) and at least one of them is a row bit, we can observe an SBDR timing. In other words, the combination of all such SBDR pairs forms the complete set of row-inclusive functions (**O1** resolved). Meanwhile, after adding up all higher bits in these SBDR pairs, we can reveal all row bits related to bank functions. Therefore, combining the previously identified pure row bits, this step also recovers the range of row bits (line 9, **O2** resolved).

Step 2: Trios. Evidently, from the heatmap of the previous step, we could not observe any information from the low-order part (bottom left blue areas) of the heatmap, because bank functions there cannot change the row to form an SBDR state, such as $(6, 13)$ in Figure 2. Therefore, to identify all *remaining non-row functions*, we continue to further extend the measurements with larger $B_{diff}.size$. Specifically, in this step, we derive all non-row function bits by setting $B_{diff} = \{b_{BF}, b'_{BF}, b_x\}$, where (b_{BF}, b'_{BF}) is an arbitrarily chosen bank function pair obtained from the previous step (line 11). In other words, we have "borrowed" an initial SBDR state from a row-inclusive function. Then, when we traverse all remaining non-row b_x and measure T_{SBDR}, there will be two cases: 1) if b_x is a bank bit, there will be fast DB timings; 2) if b_x is a non-bank bit, the timing would be slow as it still reflects the initial SBDR state. Consequently, we can distinguish whether b_x is a non-row bank bit and insert it to B_{non_row} (line 12-16).

Step 3: Quartet. Finally, we need to determine all non-row functions using $B_{diff} = \{b_{BF}, b'_{BF}, b_x, b_y\}$ and traverse all possible combinations of b_x and b_y from B_{non_row}. Here, we will observe SBDR timings if and only if b_x and b_y reside in the same

bank function, because for $b_x, b_y \in BF' \wedge BF \neq BF'$, there will be even number of different bits in both bank functions (line 17-21, **O3** resolved). Lastly, we merge all pairs in $bank\_funcs$ (e.g., $(12, 19), (8, 12) \rightarrow (8, 12, 19)$) to construct all bank functions (line 22). Now, the entire reverse-engineering process completes as we have achieved all sub-objectives, and the evaluation results are later presented in Section 5.1.

With $B_{diff}.size = 4$, we find the algorithm already capable of recovering all existing DDR4 mappings with full accuracy, but further expanding the size and combinations of B_{diff} can provide extra cross-validation. Due to its layout-agnostic design with polynomial complexity, we believe the method would still preserve its compatibility and scalability towards future mappings even if they further complicate or enlarge the bank search space (e.g., supporting larger physical memory size or using more bank bits).

4 Hammering Down New Architectures via Counter-Speculation Prefetching

4.1 Overview

After retrieving correct DRAM address mappings, we design our novel prefetch-based Rowhammer paradigm extending from the state-of-the-art non-uniform hammering frameworks [29, 30] to advance such attacks on new architectures, which generally addresses two pivotal challenges. The first one lies in the fundamental limitations of conventional load-based hammering regarding the activation rate. While it has been proven partially successful on earlier platforms, we notice a considerable gap to make Rowhammer attacks effective on modern architectures. In response to this, we introduce a novel prefetch-based hammering paradigm that leverages prefetch instructions to substantially enhance hammering throughput. While prior work [30] evaluated the activation rate of using `PREFETCHNTA`, it attributed the observed improvements solely to increased cache hits and suggested not to use such instructions for hammering. Alongside, we uncover that the recent multi-bank hammering technique [33] can further boost the efficiency of prefetch-based attacks, as we have seen the same TRR-bypassing patterns yield significantly more bit flips when combined with prefetching and multi-bank techniques.

More importantly, the other challenge pertains to the impact of modern speculative execution mechanisms, which disrupt the intended access order of hammering patterns. Notably, we observe that prefetches exhibit even more pronounced disorder issues compared to loads across all platforms. Furthermore, since prefetches are not explicit load operations, traditional wisdom such as memory barriers cannot be directly applied to resolve the prefetching disorder issue. Regarding this, we also note an additional benefit of the multi-bank technique that partially alleviates the disorder effect as its bank-interleaving nature adds pipeline latency to per-bank hammering. However, as more recent architectures feature intensified speculative behavior, we further introduce a counter-speculation technique to fully unlock the potential of prefetch-based hammering and ensure effectiveness on newer platforms. By integrating runtime control flow obfuscation and intricate `NOP`-based pseudo-barriers, we strike the optimal balance between introducing extra overhead and mitigating the negative speculation impacts, thus maximizing the overall effectiveness of our approach.

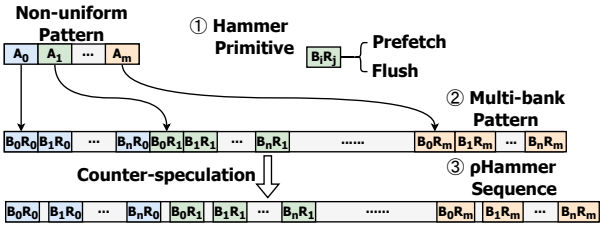

Figure 5: Overview of ρHammer's workflow. B/R refers to bank/row.

Figure 5 illustrates our general workflow. Recent Rowhammer attacks against TRR-protected DIMMs typically require executing effective non-uniform hammering patterns - a carefully crafted, ordered sequence of aggressor rows ($A_0 \ldots A_m$). As noticed by prior work [29, 58], TRR sampler may always observe a subset of activations within a t_{REFI} or not sample from every t_{REFI}. Therefore, such patterns are constructed by assigning varied access frequencies, repetitions, and offsets to a large set of aggressors, attempting to find patterns that effectively confuse the sampler by hiding the true aggressors periodically. Building upon this concept, ρHammer incorporates three key techniques, which we will elaborate in the following subsections.

Section 4.2: ρHammer adopts x86 prefetch instructions as the instrumental hammering primitive (①), exploiting their asynchronous nature to increase memory access throughput.

Section 4.3: ρHammer distributes each aggressor row across multiple DRAM banks (②), allowing the non-uniform attack process to further take advantage of the increased parallelism and alleviated disorder effects.

Section 4.4: ρHammer addresses the more pronounced disorder issues of prefetching on the latest architectures by using a counter-speculation hammering technique that intricately inserts runtime latencies to mitigate the branch prediction and OoO effects while minimizing extra overhead to fully realize the potential of prefetch-based hammering throughput (③).

Threat Model. Similar to reverse-engineering, hammering experiments presented in this section do not enforce strict end-to-end attack constraints, such as native without `sudo` and superpages to facilitate convenient and comprehensive characterization on various setups. However, as we will show in Section 5.3, our proposed attack strategy is able to greatly amplify the chances of templating useful bit flips and can be seamlessly integrated with end-to-end massaging and exploitation [4, 33, 62] techniques.

Fuzzing & Sweeping. Following original studies [29, 30], we briefly explain two key operations indicative of non-uniform hammering effectiveness, which will be used for the systematic characterizations in the subsequent sections. The *fuzzing* process leverages a fuzzer to generate pseudo-random and unique non-uniform patterns, from which *effective patterns* are identified based on the presence of any bit flips during a few execution trials at different physical locations, with the *best pattern* being the one that induces the maximum number of flips. As each pattern only encodes the relative offsets among aggressor rows, the *sweeping* operation applies an effective pattern repeatedly at a much larger range of different physical locations, which simulates the templating process of real exploits. While fuzzing primarily reflects the overall effectiveness

of a given attack strategy by revealing its difficulty in finding TRR-bypassing patterns, sweeping focuses on the reliability and practical viability of deploying such attacks under general conditions.

4.2 Potentials & Disorder Issue of Prefetching

As Rowhammer attacks are basically activating and precharging DRAM rows as fast as possible, the activation rate is one of the most critical vectors to measure the hammering effectiveness, as it indicates the highest possible amount of hammer attempts within a DRAM refresh interval. Nevertheless, a critical aspect overlooked by past hammering approaches, which only use memory load instructions (e.g., x86 `MOV`), is that Rowhammer attacks focus solely on the process of retrieving data from DRAM, rather than the final destination. In other words, the overall activation rate ought to benefit from a shorter lifecycle of the accessed DRAM data. Prefetch instructions align perfectly with this objective, as their asynchronous nature enables faster execution than normal loads and they only place data in specific caches instead of farther CPU registers, resulting in a shorter data path.

```
1     // synchronize with refresh command...
2     // flush all addr in aggr_row_addrs[]
3     // start hammer pattern:
4     for(int idx = 0; idx < num_of_act; idx++){
5         hammer(agg_row_addrs[idx]);
6         clflushopt(aggr_row_addrs[idx]);
7     }
```

Listing 1: A brief overview of the original C++ hammering privitive without any barriers. `hammer` is either load-based or prefetch-based.

In this section, we focus on exploring the performance potentials of prefetch-based hammering compared to conventional load-based hammering, while also revealing the hidden disorder hazard behind the hammering speed boost enabled by prefetch instructions. We begin by comparing the attack efficiency using a typical hammering primitive shown in Listing 1 with different hammer instructions. For each platform, we randomly select 80 generated patterns and iteratively execute them until each pattern hits 5 million accesses. Figure 6 illustrates the average attack time per pattern using loads and four types of prefetch instructions across the four architectures. Despite that different prefetch instructions place data in different numbers and levels of caches, their actual attack time shows little variation, yet is still substantially lower than the load-based counterparts. In the remainder of this paper, ρHammer empirically uses `PREFETCHT2` or `PREFETCHNTA` for prefetch-based hammering that yields marginally better attack performance as they only place data in one level of cache and feature the least temporal locality hint that would minimize unnecessary cache pollution.

However, aligning with prior work [30], we find that the observed advantage in attack time also implicitly includes the increased cache hits caused by the CPU's speculative behavior. Speculative execution is a critical feature of modern high-performance processors, designed to improve execution speed by issuing instructions in advance rather than strictly following the program order. Two dominant techniques of speculative execution are out-of-order (OoO) execution and branch prediction. Although they aim to resolve pipeline stalling issues regarding different aspects, they both

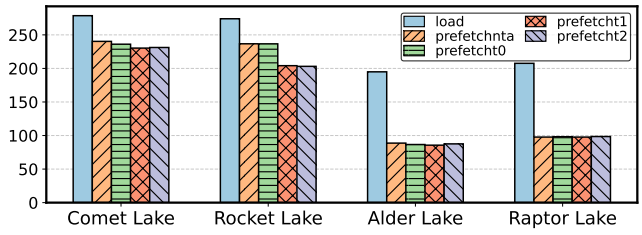

Figure 6: Average attack completion time in *ms* using load-/prefetch as hammer instructions.

rearrange the hammering instruction order that severely affect the attack effectiveness. Initially, we gain insights into such an impact by looking into the disorder difference between the C++ primitive shown in Listing 1 and its loop-unrolled assembly variant generated via AsmJit [36], without any barrier instructions. Although functionally equivalent, we notice a sharp decline of bit flips after switching from the C++ to the even faster AsmJit primitive on Comet/Rocket Lake machines. To further investigate, we count their average cache miss rate using the x86 hardware performance counters (HPC)[1] and the total execution time, where we also observe a dramatic decrease of both when using the AsmJit variant, as shown in the leftmost bars of Figure 8. By analyzing their dumped binaries, we identify the root cause that just-in-time compiling uses immediate addresses for hammer and flush instructions at each unrolled loop, which allows the CPU to execute aggressively without order. In contrast, the original C++ primitive features an `idx` counter serving as a load dependency for indirect addressing at lines 4 and 5 in Listing 1, which creates a pipeline latency that partially neutralizes the speculative effect.

More importantly, for both types of primitives, the cache miss rate of prefetch-based is much lower than that of load-based hammering. A deeper reason for such a phenomenon lies in the asynchronous nature of prefetch instructions that retire as soon as after finishing the address translation and forwarding a request to the L1-D's line fill buffer (LFB), then they do not stall the subsequent execution while waiting for the pending cache line to be fetched from DRAM and stored in the cache level specified by the hint [39]. Also, according to the Intel Software Developer's Manual [26], flush instructions and prefetching behavior are not ordered with each other. Consequently, as visualized in Figure 7 when a prefetch instruction is issued close enough before a previous flush instruction's effects targeting the same line are completed, such an upcoming prefetch hint will be ignored by the CPU if the cache line is already in the cache, thereby reducing the actual activation rate. Therefore, this highlights the importance of addressing the intrinsic disorder issue of prefetch instructions to ensure reliable and effective prefetch-based Rowhammer attacks.

4.3 Combining Bank-Level Parallelism

In order to further boost the activation rate, we introduce another critical multi-bank technique following a quite recent work [33] and investigate its interplay with prefetch-based hammering. Essentially, this technique exploits bank-level parallelism to share all

[1]Cache misses are measured via the Linux Perf library's *L1-dcache-load-misses* event during the hammer loop.

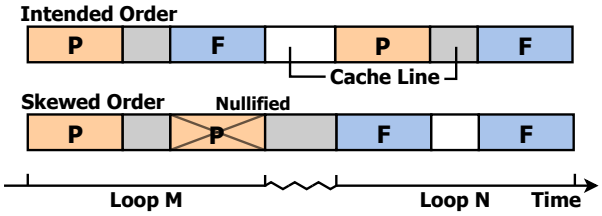

Figure 7: Visualization of prefetch (P) and flush (F) instruction interaction on the same cache line across close hammer loops, where white/gray blocks represent the line absent/present in the cache.

activations across multiple banks (i.e., sacrificing the per-bank activation rate) and thus maximize the overall hammering throughput. Using the same setup as Figure 6, we perform load/prefetch-based attacks with C++ and AsmJit primitives, ranging from 1 to 8 banks, and measure the average execution time and cache miss rate of each pattern. We show the example results on Comet Lake in Figure 8.

Reflecting the previous section, the overall trend in cache miss rate across the number of banks further justifies that prefetching exhibits more severe disorder than loads. Interestingly, besides increased parallelism, we observe another overlooked effect of multi-bank hammering that mitigates the disorder issue, as the cache miss rate increases with more number of banks. One primary cause for such an effect is that the latency between hammer attempts on each bank is extended due to the interleaving of hammer attempts on other banks, which gradually reduces the degree of disorder and leads to an increasingly controlled hammering execution. As for hammering speed, our results demonstrate that prefetching doubles the attacks speed when cache miss rates reach their peak. This suggests that contrary to the conclusions drawn by [30], even excluding the cache hit influence, prefetch instructions can still significantly enhance the hammering throughput. Additionally, our comparison between C++ and AsmJit primitives reveals that the former achieves full cache miss saturation much faster. In contrast, the latter's cache miss rate remains well below 100% even under 8 banks, while excessively increasing the bank count severely degrades per-bank activation rates. Therefore, ρHammer adopts the more preferable C++ primitive for prefetch-based hammering to ensure improved attack performance and reliability.

To further demonstrate the advantages of multi-bank prefetch-based hammering, we measure the total number of bit flips during 2-hour load/prefetch-based fuzzing operations using varying numbers of banks across all four architectures. Figure 9 evidently proves that the attack performance of prefetch-based hammering is significantly superior to load-based hammering on Comet/Rocket Lake, further emphasizing the outstanding attack capabilities derived from prefetching. Meanwhile, the results for Comet Lake align with our findings in Figure 8: the highest number of bit flips occurs when the cache miss rate is reaching its peak (i.e., #Banks = 3). After reaching the highest flip count (51,911), increasing the number of banks further would result in a gradual decline in hammering effectiveness, as expected. Although the highest flip count here in the figure is at #Banks = 3, it is important to clarify that the optimal number of banks could vary across architectures and DIMMs. Therefore, for simplicity, the multi-bank configurations mentioned

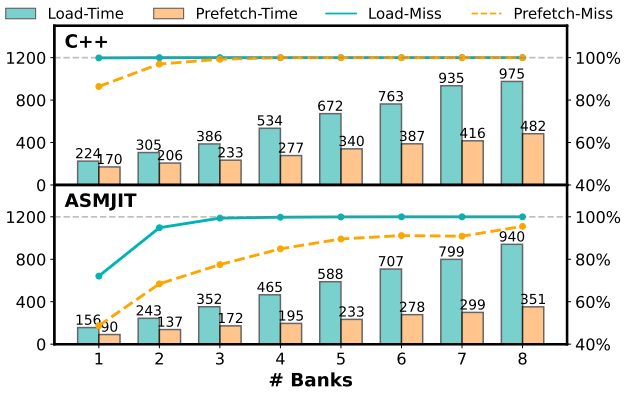

Figure 8: Average cache miss rate and attack time in *ms* using the C++/AsmJit primitives with load/prefetch-based hammering on Comet Lake.

in the rest of this paper refer to the optimal bank number identified by ρHammer through fuzzing.

In stark contrast, the results on Alder/Raptor Lake shown in Figure 9, reveal that combining the multi-bank technique with prefetch-based hammering is still insufficient to incur bit flips on these architectures. In fact, beyond this specific result, we have seen a sharp decline in bit flips on Alder Lake and a complete absence of flips on Raptor Lake, despite using the same flippable DIMMs, TRR-bypassing access patterns, and physical locations with identical extent of hammering tolerance. These consistent failures strongly suggest that the more aggressive speculative execution mechanisms are the primary culprit suppressing our attack towards these newer architectures and more advanced hammering techniques are required to overcome their interference.

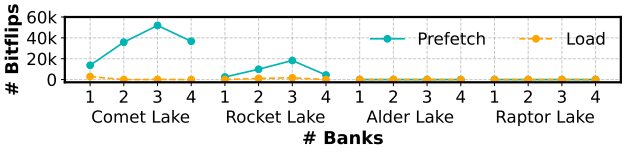

Figure 9: Overall effectiveness using load/prefetch-based hammering across 1-4 banks on all four architectures.

4.4 Counter-Speculation ρHammering

In this section, we elaborate ρHammer's counter-speculation technique that enables Rowhammer attacks on the latest architectures with intensified speculative disorder issues. Generally, the technique combines runtime control flow obfuscation and intricate `NOP`-based pseudo-barriers, carefully inserting runtime latencies to counteract the adverse effects of branch prediction and OoO execution, while minimizing performance overhead to preserve the high throughput of prefetch-based hammering.

Misleading the Branch Predictor. Firstly, we incorporate a control flow obfuscation technique to suppress the branch prediction's impact across loop iterations shown in Listing 1. Specifically, we generate randomness from the `rdrand` and `rdtscp` instructions, which dynamically derives the indexes of multiple execution paths

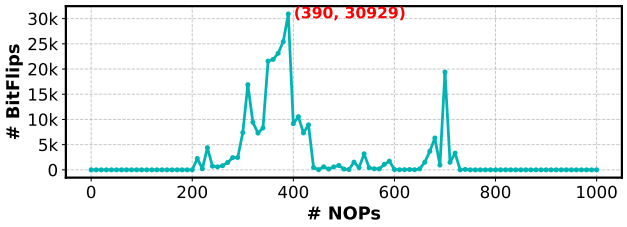

Figure 10: Number of bit flips with different nop counts. Each data point is the sum of sweeping the best pattern on Raptor Lake for 1,000 base addresses * 5 million activations.

through additional bit-wise and arithmetic manipulations. The obfuscation renders two pivotal components for branch prediction ineffective: 1) the branch target buffer (BTB), which stores the target addresses of recently encountered branches, are frequently invalidated; 2) the pattern history table (PHT), which relies on past branch outcomes to predict future branches, becomes barely reliable as it struggles to adapt to the fluctuating control flow that is only resolved at the beginning of each loop iteration. By continually confusing the branch predictor, this technique forces the processor to fall back on a more conservative, in-order execution path.

NOP-based Pseudo-barriers. We introduce another key technique of ρHammer to mitigate the OoO execution effects. One of the critical hardware structures used for OoO execution is the reorder buffer (ROB), which tracks the original program order of instructions and ensures their results are committed correctly. The ROB has a finite size, and once it is highly occupied, the processor's ability to reorder instructions is constrained. Therefore, we utilize the NOP instruction to consume reorder buffer slots while minimizing extra computational overhead. After injecting sufficient NOPs between two subsequent instructions to saturate the ROB, their execution would be fully serialized [6]. Building upon this concept, we build a pseudo-barrier strategy that inserts NOPs moderately but not excessively as we need to strike a balance between the hammering reliability and extra barrier overhead rather than ensure deterministic in-order behavior.

Optimal NOP Count. We have observed that there is substantial room for improvement by discovering an optimal NOP range that strikes the perfect of such a balance. As is illustrated in Figure 10, we show the example of sweeping one best pattern on Raptor Lake, where excessively low NOP count is insufficient to counter the OoO execution and the opposite sacrifices too much activation rate, both of which fail to yield any flip. The optimum flip count (390) lies in the intermediate positive range, implying that a real attacker can take the same approach to search the optimum before exploitation. In fact, the optimal NOP count is platform specific due to different OoO designs and ROB sizes, but one optimum is usually also effective across various patterns on the same platform (i.e., within the positive range). Therefore, ρHammer incorporates a tuning phase to identify and use the optimal number of NOP instructions inserted between prefetches. However, when we attempt to adapt our counter-speculation technique to load-based hammering, we still cannot find any bit flip after trying all possible number of NOPs within the same [0, 1000] range on Raptor Lake, which also reflects the key factor leading to such a difference lies in the activation rate.

Pitfalls of Using Barrier Instructions. Prior work [30] studied the potential hazard of speculative execution and highlighted the selective use of the x86 fence instructions (i.e., [L|S|M]FENCE) for load-based hammering. However, on the two newer architectures, we have observed zero bit flips regardless of how long we perform load-based hammering, both with and without fence instructions. As discussed earlier, prefetching is inherently more susceptible to speculative disorder, so we further investigate whether existing fence strategies are useful for prefetching and compare them with our NOP-based pseudo-barrier technique.

Arch.	None	CPUID	MFENCE	LFENCE (load)	LFENCE (prefetch)	NOP
Alder	0	0	0	0	8,113	5,250
Lake	93.7	2,104.4	1,179.2	649.6	211.7	212.6
Raptor	0	0	0	0	1,903	7,210
Lake	92.8	1,861.3	1,149.6	466.2	210.4	212.4

Table 3: Comparison of different barriers on Alder Lake and Raptor Lake. Upper digits are number of bit flips and lower ones are time in *ms*.

Specifically, we perform prefetch-based sweeping over best patterns for 200 base addresses * 10 million activations using various barrier instructions as well as our NOP-based pseudo-barriers, all with control flow obfuscation enabled. The maximal number of bit flips and the completion time are recorded in Table 3. The Intel Software Developer's Manual [26] states that PREFETCHh instructions are not ordered with respect to the fence instructions and other PREFETCHh instructions, but are ordered with respect to serializing instructions such as CPUID. However, the table indicates that using CPUID for serializing prefetches and MFENCE is too time-consuming to meet the minimum activation rate, which is even worse than the conventional load+LFENCE strategy.

From the table, it seems that LFENCE is also effective for prefetch-based hammering, exhibiting execution time comparable to that of the NOP-based pseudo-barriers, especially on Alder Lake. Although this appears to contradict the manual's statement that prefetches are not ordered by fences, we argue that the manual remains correct in this regard. To further investigate, we reuse the previous AsmJit primitive that leverages immediate rather than indirect addressing (i.e., no data dependency chain), but with LFENCE inserted. Under this modified setup, we observe a sharp decline in the bit flip count, confirming that the effectiveness of LFENCE does not result from *direct ordering* on prefetches. Instead, the effect stems from LFENCE impacting the address calculation process at line 4&5 of Listing 1, where the target address of the current prefetch is explicit memory read that must be architecturally resolved before issuing the next, thereby imposing an *indirect ordering* on prefetches.

4.5 Summary & Implications

Here we summarize our observations and elaborate on why ρHammer is fundamentally more preferable than load-based methods.

In order to achieve higher activation rates, prior efforts have demonstrated using multi-threaded hammering [28, 61], mainly on DDR3 devices. However, as reported by a quite recent work [19]: on DDR4 devices equipped with TRR, multi-threaded hammering is in fact much less effective than single-threaded and the trend could

Arch.	DRAM Geometry (Size, # Rank, # Bank)		
	(8G, 1, 16)	(16G, 2, 16)	(32G, 2, 16)
Comet Lake Rocket Lake	**Bank Func:** (16, 19), (15, 18), (14, 17), (6, 13); **Row:** 17–32	**Bank Func:** (17, 21), (16, 20), (15, 19), (14, 18), (6, 13); **Row:** 18–33	**Bank Func:** (17, 21), (16, 20), (15, 19), (14, 18), (6, 13); **Row:** 18–34
Alder Lake Raptor Lake	**Bank Func:** (14, 17, 21, 26, 29, 32), (15, 18, 20, 23, 24, 27, 30), (16, 19, 22, 25, 28, 31), (9, 11, 13); **Row:** 17–32	**Bank Func:** (14, 18, 26, 29, 32), (16, 20, 23, 24, 27, 30, 33), (17, 21, 22, 25, 28, 31), (15, 19), (9, 11, 13); **Row:** 18–33	**Bank Func:** (14, 18, 26, 29, 32), (16, 20, 23, 24, 27, 30, 33), (17, 21, 22, 25, 28, 31, 34), (15, 19), (9, 11, 13); **Row:** 18–34

Table 4: Reverse-engineered DRAM address mapping on the four most recent Intel architectures. All configurations are single-channel, single-DIMM, with different DRAM geometry.

be even worse as the growing number of threads. The root cause is that effective bypassing the TRR requires a strict access order (i.e., the non-uniform pattern used in this paper). Once multiple threads hammer concurrently, their asynchronous requests collide in the memory-controller queue, disturb the pattern, and trigger extra refreshes by the TRR sampler, while enforcing a global order with synchronization mechanisms such as locks or semaphores re-introduces serialization and further degrade the activation rate to even lower than just using a single thread [19].

In the case of single-threaded hammering, recall in Figure 8, under a full cache-miss situation, tight loops of prefetch-based hammering are significantly faster than load-based counterparts while targeting same aggressors. Moreover, the former achieves evidently higher overall hammering effectiveness and flip rate, which we will soon demonstrate in Section 5. These results give two important implications: 1) **Single-threaded loads cannot saturate the DRAM bandwidth**, a widely acknowledged fact that is also supported by the massive reported results through public micro-benchmarks such as Stream [47]. 2) **Prefetches better utilize the single-threaded DRAM bandwidth than loads**. The underlying μarch reason is that a regular load occupies a load-queue entry until the line returns, throttling the issue rate once these structures are filled. By contrast, as previously stated, a prefetch instruction does not stall and is immediately *marked as complete* in the ROB as soon as the address translation is done [25]. In support of this, our results in Table 3 have shown prefetch+`LFENCE` is much faster than load+`LFENCE` as `LFENCE` only orders pending load requests that are *not marked as complete*, also noted by [44] on AMD platforms.

5 Evaluation

We evaluate ρHammer by demonstrating the efficiency of recovering correct DRAM address mappings, as well as the overall hammering effectiveness and the practicality towards real exploits.

5.1 DRAM Address Mapping Recovery

Table 4 summarizes our reverse-engineering results across all tested setups. We observe that, on a given architecture, the DRAM address mappings are consistent across DIMMs with identical geometry, with Comet/Rocket Lake sharing one mapping scheme and Alder/Raptor Lake adopting another. Then, we compare our method against all publicly available existing tools [30, 56, 65] (properly configured to our setups) and record the average timing results of 50 independent runs each in Table 5. We only observe that DRAMDig and DARE are able to yield correct results on Comet/Rocket Lake. Specifically, our method outperforms DRAMDig by more than two orders of magnitude in terms of runtime speed. As for DARE, even when we allocate the maximum number of superpages for bank function coloring, the results still exhibited quite a few errors (accuracy: 34/50 on Comet Lake, 39/50 on Raptor Lake). Furthermore, none of the prior methods are adaptable to the latest Alder/Raptor Lake, where both DRAMA and DARE fail to produce any correct mappings, while DRAMDig terminates prematurely due to the requirement of pure row bits. These findings highlight the superior efficiency and scalability of our reverse-engineering technique across recent architectures.

	i7-10700K	i7-11700	i9-12900	i7-14700K
DRAMA[1]	-	-	-	-
DRAMDig[2]	867.6s	1,329.9s	-	-
DARE[3]	36.5s*	33.1s*	-	-
ρHammer	8.5s	6.1s	4.6s	4.1s

Table 5: Reverse-engineering time comparing to prior art [30, 56, 65]. (*) denotes partially non-deterministic while (-) denotes no correct result or the process aborts with failure.

5.2 Overall Hammering Effectiveness

We implement ρHammer by extending the code from the existing non-uniform hammering tools, namely Blacksmith[4] and ZenHammer[5]. In our attack evaluation, we use their original load-based non-uniform hammering as the baseline and report the performance of both ρHammer and the baseline under single-bank and optimal multi-bank configurations to establish fair comparisons.

Table 6 summarizes the bit flip counts from all effective patterns and the best pattern during our large-scale 2-hour fuzzing operations for all DIMMs and architectures. Across the board, ρHammer significantly outperforms the baseline in raw flip counts, which is most evident in the following per-architecture standout results (marked in bold): 1) Comet Lake - $\mathcal{S}_4$: ρ-M achieves 257,881 total bit flips, far surpassing the 50,700 from BL-S and making it the most flip-prone across all configurations; 2) Rocket Lake - $\mathcal{S}_3$: ρ-M produces 94,395 flips, whereas BL-S yields only 9,772; 3) Alder Lake - $\mathcal{S}_2$: Despite the baseline showing no effectiveness, ρ-M achieves 918 flips; 4) Raptor Lake - $\mathcal{S}_2$: Still, the baseline produces negligible flips, while ρ-M achieves 2,113.

[1] https://github.com/isec-tugraz/drama
[2] https://github.com/dramdig/DRAMDig
[3] https://github.com/comsec-group/zenhammer/tree/dare
[4] https://github.com/comsec-group/blacksmith
[5] https://github.com/comsec-group/zenhammer

DIMM	Comet Lake				Rocket Lake			
	BL-S	BL-M	ρ-S	ρ-M	BL-S	BL-M	ρ-S	ρ-M
$\mathcal{S}_1$	2840, 529	0, 0	9117, 1506	121431, 25361	58, 18	1088, 490	10458, 2547	24548, 5257
$\mathcal{S}_2$	12288, 2274	437, 178	55270, 9720	87824, 24177	3815, 720	7448, 1255	17200, 1254	42485, 4831
$\mathcal{S}_3$	36406, 4439	9373, 521	67552, 1566	205742, 7526	9772, 3041	10211, 923	48419, 1235	**94395, 2896**
$\mathcal{S}_4$	50700, 7839	1472, 199	76645, 5223	**257881, 13880**	6560, 2030	4323, 1103	53706, 3334	81983, 5824
$\mathcal{S}_5$	365, 28	130, 38	722, 123	3660, 446	79, 12	171, 14	301, 12	963, 80
$\mathcal{H}_1$	136, 18	33, 19	406, 30	1714, 441	0, 0	23, 4	918, 68	1498, 271
$\mathcal{M}_1$	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0

DIMM	Comet Lake				Rocket Lake			
	BL-S	BL-M	ρ-S	ρ-M	BL-S	BL-M	ρ-S	ρ-M
$\mathcal{S}_1$	0, 0	0, 0	169, 67	571, 158	0, 0	0, 0	20, 6	266, 53
$\mathcal{S}_2$	3, 1	2, 1	582, 137	**918, 233**	0, 0	1, 1	490, 137	**2113, 1564**
$\mathcal{S}_3$	2, 2	10, 7	107, 31	696, 141	8, 7	0, 0	183, 53	924, 140
$\mathcal{S}_4$	52, 48	4, 2	127, 12	413, 89	0, 0	4, 4	62, 22	536, 99
$\mathcal{S}_5$	0, 0	0, 0	7, 2	22, 7	0, 0	0, 0	18, 4	20, 6
$\mathcal{H}_1$	0, 0	0, 0	40, 176	402, 27	0, 0	0, 0	92, 6	384, 34
$\mathcal{M}_1$	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0	0, 0

Table 6: Bit flip counts (total, best pattern) during 2-hour fuzzing on all platforms, for each of which we record the result using baseline/ρHammer (BL/ρ) and single-bank/multi-bank (S/M).

Generally, the results demonstrate that ρHammer not only significantly amplifies the Rowhammer threat on architectures where baseline attacks remain limitedly effective (e.g., Comet/Rocket Lake), but more importantly, it revives Rowhammer on the most recent platforms like Alder/Raptor Lake, where baseline techniques almost completely fail to induce any meaningful flips. Moreover, ρ-M always outperforms ρ-S across all setups, which highlights the crucial role of the multi-bank technique in maximizing the full potential of prefetch-based hammering. In contrast, the baseline configurations display a number of counterintuitive cases where BL-S outperforms BL-M, especially on Comet Lake. This aligns with the fundamental limitation of loads as discussed in Section 4.5: since load-based hammering is heavily constrained by activation rate, adding more banks can exacerbate this bottleneck rather than alleviate it. Additionally, on platforms with relatively weaker speculative disorder (as primarily observed on Comet Lake; recall Figure 8), the disorder effect from single-bank load-based hammering may already be minimal, thus little benefit from alleviated disorder could be gained by distributing accesses across multiple banks.

5.3 Practical Exploitability

To more intuitively justify ρHammer's security impact, we further: 1) analyze flip rates that reflect the speed and chance of finding useful flips; 2) perform end-to-end PTE-corruption attack POCs.

Flip Rate. Figure 11 illustrates the increasing trends of bit flips during 1-hour sweeping using *the best pattern* and *the better of single/multi-bank* respectively for ρHammer and baselines. Given that Rowhammer vulnerability depends on the DIMM location, we perform sweeping across four architectures using the same set of non-repeating row addresses. For the Alder/Raptor Lake where the baseline produces zero flips, we use ρHammer's best pattern for sweeping the baseline as a fallback.

The results reveal striking differences: on Comet/Rocket Lake, ρHammer achieves an average flip rate of 187K/min and 47K/min, significantly faster than the baseline by 112.4x and 47.1x. Moreover, on Alder/Raptor Lake, the baselines fail to reproduce any flips even if we switch to other DIMMs (e.g., $\mathcal{S}_4$) and patterns with occasional flips during fuzzing, but ρHammer still reaches the impressive average flip rates of 995/min and 2,291/min, which are also much higher compared with previous data (e.g., ~133/min on Alder Lake reported by [33]). Another noteworthy finding is that ρHammer's bit flips smoothly progress over time (remind that the figure is logarithmic-scaled), which indicates that desired flips could be equally found at all positions.

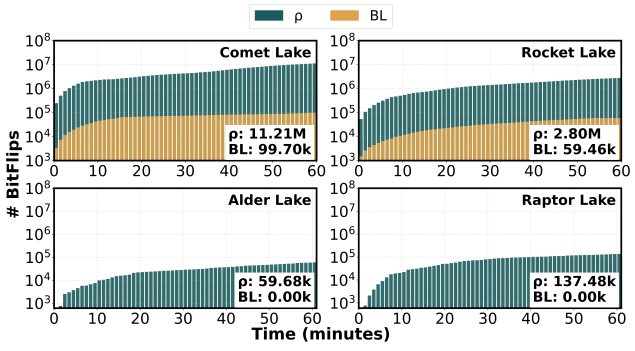

Figure 11: Accumulative average number of bit flips over iterative 1-hour sweeping on the four architectures.

End-to-End Attack. Combining the latest massaging techniques proposed by Rubicon [4], we implement and perform end-to-end PTE corruption attacks on the newer two architectures. We do not rely on superpages but sweep discrete 4MB contiguous memory regions and template all bit flip locations, which is the largest

possible size an unprivileged attacker can manage to assure contiguity by exhausting the Linux's buddy allocator. Using the best pattern, the templating phase on Alder/Raptor Lake platforms respectively identifies: 892/846 total flips within 38/35s, of which 61/43 are exploitable. Here, *exploitable* flips are those residing in a desired sub-range of PTE frame number (e.g., [12, 19]), while their physical location and flip direction allow self-reference after being massaged into a page table page. Subsequently, we successfully achieve page table read/write capability on both platforms using an average runtime of only 3min 8s (Min: 54s; Max: 6min 44s) and 51s (Min: 20s; Max: 1min 55s), each for five independent trials.

6 Discussion

Implications on Defenses. Beyond commodity TRR, another production-level defense proposed by Intel is the pseudo-TRR [32] (pTRR) that refreshes rows pre-emptively, yet it is only enabled when both the MC and the DIMM are compliant. Unfortunately, as reported in [14] and to the best of our knowledge, the consumer-grade CPUs tested in this paper do not to support this feature. Even worse, all our tested DIMMs set their Maximum Activation Count (MAC) value that indicates their claimed Rowhammer tolerance to "unlimited", usually derived from the less potent load-based hammering tests or even no thorough inspection, which is also noted by [14, 18, 33]. However, as our prefetch-based hammering paradigm further exacerbates the activation rate, such bottom-line confidence is more misplaced than ever. Recently in academia, numerous schemes have also been proposed, such as address-mapping scrambling that reorders the bank/row mapping using a boot-time key [34], and randomized row-swapping that periodically exchanges the contents of random row pairs to keep aggressor accesses from concentrating on the same victim [59, 66, 67]. These efforts are expected to mitigate our attack because they would break the TRR-bypassing pattern and disperse the activations. However, the additional remapping circuitry demands non-trivial modifications to both DRAM chips and memory controllers, which is unlikely to be widely deployed in the near term.

Towards Future Research on DDR5. Recently prevailing DDR5 devices integrate more intricate defense mechanisms than TRR, such as refresh management (RFM) [31]. Thus, we have not observed any effective pattern on our setups with DDR5 DIMMs, which is also acknowledged by a concurrent non-uniform hammering work [58]. Nevertheless, our prefetch-based hammering paradigm would inspire future research on DDR5 by 1) supporting higher activation rate against doubled refresh rate, and 2) increasing the likelihood of multi-bit errors for on-die ECC corruption [9, 31]. We have evaluated our reverse-engineering tool on Alder/Raptor Lake DDR5 systems and observed that further efforts are required to identify extra *sub-channel* functions. Another concurrent study [51] shows that the RFM refreshing behavior forms an accurate new side-channel on GDDR6 and LPDDR5 devices. Combining this new primitive with our efficient algorithm is thus an interesting future work that would further extend ρHammer towards DDR5.

7 Related Work

DRAM Address Mapping Reverse-Engineering. As discussed in Section 3.2, Pessl et al. [56] conducted the pioneering work that introduced a brute-force strategy for mapping recovery. To reduce the brute-force search space, Wang et al. [65] proposed a knowledge-assisted approach that excludes pure row bits before searching. Jattke et al. [30] extended the strategy to AMD platforms with extra consideration of a constant address offset. To the best of our knowledge, only two prior studies by Xiao et al. [68] and Marazzi et al. [46] exclude brute-forcing. However, the former is limited to identifying simple single-bit functions on RISC-V platforms, while both this work and Wang et al. [65] have found the latter fails to obtain correct mappings on more recent x86 platforms.

Recent Advances in Rowhammer. Orosa et al. [52] conducted a systematic analysis of Rowhammer vulnerability across 272 chips and revealed that the problem may vary with physical locations, which is the reason for our controlling the base physical addresses for comparisons. Frigo et al. [14] proposed the many-sided hammering strategy that bypasses TRR by concealing the accesses of real aggressor rows using dummy ones. Ridder et al. [10] further introduced synchronized many-sided hammering from JavaScript, leveraging `NOP`s to align memory accesses with DRAM refreshes and bypass TRR's sampler, whose goal fundamentally differs from our use of `NOP`s as memory barriers. Unlike previous uniform hammering approaches, Jattke et al. [29] presented the pioneering non-uniform hammering tool against TRR-protected DDR4 DIMMs, which could fuzz hammer patterns with aggressor pairs featuring various frequency-domain parameters. Kang et al. [33] first exploited bank-level parallelism to amplify Rowhammer effects. Although they have observed the technique being the most effective on 6-10th generations with a maximum 13.1x increase on i7-6700, their total flip counts also suffered much reduction when targeting the 11/12th generations. Jattke et al. [30] extended their non-uniform hammering approach to AMD Zen platforms. They compared the activation rates of different types of DRAM accessing instructions, including `PREFETCHNTA`, but simply attributed the rate improvement to cache hits and suggested not to use such instructions as opposed to our work Zhang et al. [73, 75] introduced the concept of *implicit* hammer attacks, where memory accesses are induced via system calls or page table walks to target addresses that the attacker cannot explicitly access. Although prefetching results in implicit memory accesses, our attack remains fundamentally *explicit*, as it must directly specify target addresses for prefetching.

8 Conclusion

In this paper, we revisit the viability of Rowhammer attacks on recent Intel architectures and identifies key bottlenecks rendering load-based state-of-the-arts ineffective. By introducing ρHammer, we systematically addresses challenges related to DRAM address mapping complexity, limited activation rates, and speculative hammering disorder. Our contribution involves a fast and accurate reverse-engineering method, a high-throughput prefetch-based hammering paradigm combining bank parallelism, and a counter-speculation technique tailored to maintain hammering order while maximizing the potentials of prefetching. Extensive evaluation shows that ρHammer not only amplifies attack effectiveness on known vulnerable platforms, but also enables practical exploitation on newer ones such as Intel Raptor Lake for the first time, calling for a renewed consideration of Rowhammer on recent architectures.

References

[1] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016.* https://doi.org/10.1145/2872362.2872390

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* https://doi.org/10.1109/SP.2016.63

[3] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

[4] Matej Bölcskei, Patrick Jattke, Johannes Wikner, and Kaveh Razavi. 2025. Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging. In *EuroS&P*. Paper=https://comsec.ethz.ch/wp-content/files/rubicon_eurosp25.pdf

[5] Wei Chen, Zhi Zhang, Xin Zhang, Qingni Shen, Yuval Yarom, Daniel Genkin, Chen Yan, and Zhe Wang. 2025. HyperHammer: Breaking Free from KVM-Enforced Isolation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 545–559.

[6] Shing Hing William Cheng, Chitchanok Chuengsatiansup, Daniel Genkin, Dallas McNeil, Toby Murray, Yuval Yarom, and Zhiyuan Zhang. 2024. Evict+Spec+Time: Exploiting Out-of-Order Execution to Improve Cache-Timing Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024, 3 (2024), 224–248. https://doi.org/10.46586/TCHES.V2024.I3.224-248

[7] Yaakov Cohen, Kevin Sam Tharayil, Arie Haenel, Daniel Genkin, Angelos D. Keromytis, Yossi Oren, and Yuval Yarom. 2022. HammerScope: Observing DRAM Power Consumption Using Rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022.* https://doi.org/10.1145/3548606.3560688

[8] Lucian Cojocar, Jeremie S. Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. 2020. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* https://doi.org/10.1109/SP40000.2020.00085

[9] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* https://doi.org/10.1109/SP.2019.00089

[10] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2021. SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* https://www.usenix.org/conference/usenixsecurity21/presentation/ridder

[11] Andrea Di Dio, Mathé Hertogh, and Cristiano Giuffrida. 2025. Half Spectre, Full Exploit: Hardening Rowhammer Attacks with Half-Spectre Gadgets. In *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025.* https://doi.org/10.1109/SP61157.2025.00207

[12] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray A. Perlner, Arkady Yerukhimovich, and Daniel Apon. 2022. When Frodo Flips: End-to-End Key Recovery on FrodoKEM via Rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022.* https://doi.org/10.1145/3548606.3560673

[13] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* https://doi.org/10.1109/SP.2018.00022

[14] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* https://doi.org/10.1109/SP40000.2020.00090

[15] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* https://doi.org/10.1109/SP.2018.00031

[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings.* https://doi.org/10.1007/978-3-319-40667-1_15

[17] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* https://doi.org/10.1109/SP46214.2022.9833692

[18] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021.* https://doi.org/10.1145/3466752.3480110

[19] Wei He, Zhi Zhang, Yueqiang Cheng, Wenhao Wang, Wei Song, Yansong Gao, Qifei Zhang, Kang Li, Dongxi Liu, and Surya Nepal. 2025. WhistleBlower: A System-Level Empirical Study on RowHammer. *IEEE Trans. Computers* 74, 3 (2025), 805–819. https://doi.org/10.1109/TC.2023.3235973

[20] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. 2020. Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020.* https://doi.org/10.1109/MASCOTS50786.2020.9285962

[21] Alexander Van't Hof and Jason Nieh. 2022. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022.* https://www.usenix.org/conference/osdi22/presentation/vant-hof

[22] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. 2019. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.* https://www.usenix.org/conference/usenixsecurity19/presentation/hong

[23] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[24] Intel. 2023. 12th Generation Intel® Core™ Processors. https://cdrdv2-public.intel.com/682436/682436-031.pdf

[25] Intel. 2024. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://cdrdv2.intel.com/v1/dl/getContent/671488

[26] Intel. 2024. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2-public.intel.com/825743/325462-sdm-vol-1-2abcd-3abcd-4.pdf

[27] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.* https://www.usenix.org/conference/usenixsecurity19/presentation/islam

[28] Yeongjin Jang, Jae-Hyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017.* https://doi.org/10.1145/3152701.3152709

[29] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* https://doi.org/10.1109/SP46214.2022.9833772

[30] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024.* https://www.usenix.org/conference/usenixsecurity24/presentation/jattke

[31] JEDEC. 2020. DDR5 SDRAM Specification. https://www.jedec.org/standards-documents/docs/jesd79-5c01

[32] Marcin Kaczmarski. 2014. Thoughts on intel xeon e5-2600 v2 product family performance optimisation–component selection guidelines. *Santa Clara, CA, USA: Intel* (2014).

[33] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. 2024. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024.* https://www.usenix.org/conference/usenixsecurity24/presentation/kang

[34] Michael Jaemin Kim, Minbok Wi, Jaehyun Park, Seoyoung Ko, Jaeyoung Choi, Hwayong Nam, Nam Sung Kim, Jung Ho Ahn, and Eojin Lee. 2023. How to Kill the Second Bird with One ECC: The Pursuit of Row Hammer Resilient DRAM. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023.* https://doi.org/10.1145/3613424.3623777

[35] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.

In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. https://doi.org/10.1109/ISCA.2014.6853210

[36] Petr Kobalicek. 2023. asmjit: Low-latency machine code generation. https://asmjit.com/

[37] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. 2022. Half-Double: Hammering From the Next Row Over. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double

[38] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. https://www.usenix.org/conference/osdi18/presentation/konoth

[39] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*. https://doi.org/10.1145/3662010.3663451

[40] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading Bits in Memory Without Accessing Them. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. https://doi.org/10.1109/SP40000.2020.00020

[41] Eojin Lee, Ingab Kang, Sukhan Lee, G. Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. https://doi.org/10.1145/3307650.3322232

[42] J.-B. Lee. 2014. "Green Memory Solution," in Samsung Electronics, Investor's Forum, 2014.

[43] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin (Sherman) Shen. 2024. Yes, One-Bit-Flip Matters! Universal DNN Model Inference Depletion with Runtime Code Fault Injection. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. https://www.usenix.org/conference/usenixsecurity24/presentation/li-shaofeng

[44] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. https://www.usenix.org/conference/usenixsecurity22/presentation/lipp

[45] Haocong Luo, Ataberk Olgun, Abdullah Giray Yaglikçi, Yahya Can Tugrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. 2023. RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*. https://doi.org/10.1145/3579371.3589063

[46] Michele Marazzi and Kaveh Razavi. 2024. RISC-H: Rowhammer Attacks on RISC-V. In *4th Workshop on DRAM Security (DRAMSec), 2024*. https://www.research-collection.ethz.ch/handle/20.500.11850/678065

[47] John McCalpin. 2006. STREAM: Sustainable memory bandwidth in high performance computers. (2006). http://www.cs.virginia.edu/stream/

[48] Micron. 2016. "DDR4 SDRAM Datasheet," p. 380, 2016.

[49] Inc Micron. 2020. 8Gb: x4, x8, x16 DDR4 SDRAM Features-Excessive Row Activation. https://www.micron.com/products/dram/ddr4-sdram

[50] Hwayong Nam, Seungmin Baek, Minbok Wi, Michael Jaemin Kim, Jaehyun Park, Chihun Song, Nam Sung Kim, and Jung Ho Ahn. 2024. DRAMScope: Uncovering DRAM Microarchitecture and Characteristics by Issuing Memory Commands. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*. https://doi.org/10.1109/ISCA59077.2024.00083

[51] Ravan Nazaralıyev, Yicheng Zhang, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. 2025. Not so Refreshing: Attacking GPUs using RFM Rowhammer Mitigation. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. https://www.usenix.org/conference/usenixsecurity25/presentation/nazaraliyev

[52] Lois Orosa, Abdullah Giray Yaglikçi, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. 2021. A Deeper Look into RowHammer's Sensitivities: Experimental Analysis of Real DRAM Chipsand Implications on Future Attacks and Defenses. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. https://doi.org/10.1145/3466752.3480069

[53] Joongun Park, Seunghyo Kang, Sanghyeon Lee, Taehoon Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2024. Hardware-hardened Sandbox Enclaves for Trusted Serverless Computing. *ACM Trans. Archit. Code Optim.* 21, 1 (2024), 13:1–13:25. https://doi.org/10.1145/3632954

[54] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W. Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. https://doi.org/10.1109/MICRO50266.2020.00014

[55] Yuke Peng, Hongliang Tian, Junyang Zhang, Ruihan Li, Chengjun Chen, Jianfeng Jiang, Jinyi Xian, Xiaolin Wang, Chenren Xu, Diyu Zhou, Yingwei Luo, Shoumeng Yan, and Yinqian Zhang. 2025. ASTERINAS: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB. In *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*. https://www.usenix.org/conference/atc25/presentation/peng-yuke

[56] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[57] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

[58] Finn De Ridder, Patrick Jattke, and Kaveh Razavi. 2025. Posthammer: Pervasive Browser-based Rowhammer Attacks with Postponed Refresh Commands. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. https://www.usenix.org/conference/usenixsecurity25/presentation/de-ridder

[59] Gururaj Saileshwar, Bolin Wang, Moinuddin K. Qureshi, and Prashant J. Nair. 2022. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. https://doi.org/10.1145/3503222.3507716

[60] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15, 71 (2015), 2.

[61] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. https://www.usenix.org/conference/atc18/presentation/tatar

[62] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. 2022. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. https://doi.org/10.1109/SP46214.2022.9833802

[63] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. 2024. Go Go Gadget Hammer: Flipping Nested Pointers for Arbitrary Data Leakage. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 2024)*. https://www.usenix.org/conference/usenixsecurity24/presentation/tobah

[64] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. https://doi.org/10.1145/2976749.2978406

[65] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. 2020. DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. https://doi.org/10.1109/DAC18072.2020.9218599

[66] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. 2023. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. https://doi.org/10.1109/HPCA56546.2023.10070966

[67] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J. Nair. 2023. Scalable and Secure Row-Swap: Efficient and Safe Row Hammer Mitigation in Memory Systems. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. https://doi.org/10.1109/HPCA56546.2023.10070999

[68] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao

[69] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. https://www.usenix.org/conference/usenixsecurity20/presentation/yao

[70] Chuqi Zhang, Rahul Priolkar, Yuancheng Jiang, Yuan Xiao, Mona Vij, Zhenkai Liang, and Adil Ahmad. 2025. Erebor: A Drop-In Sandbox Solution for Private Data Processing in Untrusted Confidential Virtual Machines. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*. https://doi.org/10.1145/3689031.3717464

[71] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. 2024. The HitchHiker's Guide to High-Assurance System Observability Protection with Efficient Permission Switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. https://doi.org/10.1145/3658644.3690188

[72] Zhi Zhang, Decheng Chen, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. 2024. SoK: Rowhammer on Commodity Operating Systems. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. https://doi.org/10.1145/3634737.3656998

[73] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. https://doi.org/10.1109/MICRO50266.2020.00016

[74] Zhi Zhang, Yueqiang Cheng, Yinqian Zhang, and Surya Nepal. 2020. GhostKnight: Breaching Data Integrity via Speculative Execution. *CoRR* abs/2002.00524 (2020). https://arxiv.org/abs/2002.00524

[75] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. 2023. Implicit Hammer: Cross-Privilege-Boundary Rowhammer Through Implicit Accesses. *IEEE Trans. Dependable Secur. Comput.* 20, 5 (2023), 3716–3733. https://doi.org/10.1109/TDSC.2022.3214666

[76] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Minghua Wang, Kang Li, Surya Nepal, and Yang Xiang. 2021. BitMine: An End-to-End Tool for Detecting Rowhammer Vulnerability. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 5167–5181. https://doi.org/10.1109/TIFS.2021.3124728

[77] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Bo Li, Peter Volgyesi, and Xenofon Koutsoukos. 2020. Leveraging EM Side-Channel Information to Detect Rowhammer Attacks. In *2020 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP40000.2020.00060