SoK: Rowhammer on Commodity Operating Systems

Zhi Zhang
zzhangphd@gmail.com
The University of Western Australia
Australia

Decheng Chen
midcchen@mail.scut.edu.cn
School of Microelectronics, South China University of Technology
China

Jiahao Qi
qiaoziggy@gmail.com
School of Microelectronics, South China University of Technology
China

Yueqiang Cheng
yueqiang.cheng@nio.io
NIO
China

Shijie Jiang
jsjscut@gmail.com
School of Microelectronics, South China University of Technology
China

Yiyang Lin
milinyyang@mail.scut.edu.cn
gao.yansong@hotmail.com
CSIRO’s Data61
Australia

Surya Nepal
surya.nepal@data61.csiro.au
CSIRO’s Data61
Australia

Jiliang Zhang
zhangjiliang@hnu.edu.cn
College of Integrated Circuits, Hunan University
China

Yang Xiang
yxiang@swin.edu.au
School of Software and Electrical Engineering, Swinburne University of Technology
Australia

ABSTRACT
Rowhammer has drawn much attention from both academia and industry in the past years as rowhammer exploitation poses severe consequences to system security. Since the first comprehensive study of rowhammer in 2014, a number of rowhammer attacks have been demonstrated against dynamic random access memory (DRAM)-based commodity systems to break software confidentiality, integrity and availability. Accordingly, numerous software defenses have been proposed to mitigate rowhammer attacks on commodity systems of either legacy (e.g., DDR3) or recent DRAM (e.g., DDR4). Besides, multiple hardware defenses (e.g., Target Row Refresh) from the industry have been deployed into recent DRAM to eliminate rowhammer, which we categorize as production defenses.

In this paper, we systematize rowhammer attacks and defenses with a focus on DRAM-based commodity systems. Particularly, we have established a unified framework demonstrating how a rowhammer attack affects a commodity system. With the framework, we characterize existing attacks, shedding light on new attack vectors that have not yet been explored. We further leverage the framework to categorize software and production defenses, generalize their key defense strategies and summarize their key limitations, from which potential defense strategies are identified.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASIA CCS ’24, July 1–5, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3656998

1 INTRODUCTION
Recent years have witnessed an infamous hardware vulnerability, termed as rowhammer, which is essentially a circuit-level interference problem that exists in several mainstream memories (e.g., NAND Flash Memory [14], MRAM [55] and DRAM [61]). Among them, rowhammer on DRAM has caused serious security implications as DRAM is widely used in commodity (operating) systems. In as early as 2014, Kim et al. [61] performed the first comprehensive study of rowhammer on DRAM, which showed that frequent access to DRAM (known as aggressor rows) could cause bit flips in their adjacent rows without accessing the victim rows. In cases where a victim row hosts sensitive data (e.g., page tables), an unprivileged adversary can induce rowhammer-based bit flips to corrupt the data even if legitimate access to the victim row is not allowed, implying that a rowhammer...
exploit can break memory management unit (MMU)-enforced memory isolation even in the absence of software vulnerabilities, posing severe consequences to the whole system security. Starting from Kim et al. [61], a myriad of rowhammer attacks have been demonstrated to break the software’s CIA, that is, confidentiality (e.g., leaking sensitive information [16, 64, 90]), integrity (e.g., gaining privilege escalation [15, 25, 34, 37, 91, 99, 105, 108, 119, 129]) and availability (e.g., causing denial-of-service to the system [34, 48]).

To this end, numerous countermeasures have been proposed and can be classified into two categories, i.e., software and production defenses1, among which software solutions have a better deployability by building a defense wall for commodity systems. For production solutions, they were believed to be effective in mitigating the rowhammer attacks until they have been reverse-engineered [18, 26, 38, 62]. In the foreseeable future, rowhammer is unlikely to be eliminated and thus continues its threat to system security, making itself continuously heats up the arms race between rowhammer attacks and defenses. To this end, we are interested in the following questions:

Can we present a unified framework outlining how a rowhammer attack affects a commodity system from the origin (where it starts) to the end (what it achieves)? With the framework, can we systematize all the existing works? If so, what are the primary techniques used to mount a rowhammer attack? To mitigate rowhammer attacks, by scrutinizing prior works, can we have any novel insights?

Our Work: To answer the questions above, we perform a systematic study of rowhammer on commodity systems. Specifically, a unified framework is established for mounting an attack, identifying unknown attack vectors and their impacts. Within the framework, we present a taxonomy of existing attacks with a qualitative analysis of their primary attack techniques. On top of that, we perform an experimental analysis of all the Intel instructions related to CPU cache control and find that clwb can be used for cache flush, which has not discovered by existing attacks.

Besides, we categorize existing software defenses based on the framework above, from which we can clearly see that all of them aim to mitigate/prevent the attack techniques. However, each category of the defenses has its own limitations in keeping its security guarantee and applicability to commodity systems. We further categorize existing production defenses based on their security objectives and prototype locations, followed by a summary of their limitations in terms of keeping their security guarantees.

Last, we provide explicit research directions for future attacks and defenses by identifying two potential attacks and three possible defenses. First, an attacker in a hardware-assisted VM is likely to trigger rowhammer, break hypervisor-enforced memory isolation and gain hypervisor privilege. Second, the attacker may launch an attack from an isolated GPU in x86, particularly in a real-world scenario where GPU and its memory are shared in a multi-tenant public cloud.

As a response, a defender may reinforce DRAM-aware memory isolation in hardware-assisted virtualization to address the potential threats from some rowhammer attacks. A practical isolation prototype may reside in a kernel-based virtual machine (KVM), as it is supported by mainstream Linux kernel and widely used by mainstream cloud platforms. Second, the defender may detect rowhammer attacks at a much broader range based on the Intel power-monitoring interface [69], as rowhammer always requires frequent memory accesses to DRAM rows and may generate an abnormal pattern of DRAM power consumption. Last, considering that rowhammer requires frequent access to DRAM rows, it will result in an abnormal number of row buffer conflicts. Thus, it is expected to be effective in detecting rowhammer via available Intel uncore performance events.

Summary of Contributions: In summary, we have made three main contributions:

• We establish a unified framework showing how a rowhammer attack affects a commodity system from the origin to the end. With the framework, we comprehensively characterize existing attacks and identify possible unknown attack vectors. Particularly, we have analysed primary attack techniques and CPU-cache-control instructions in Intel.

• We also rely on the unified framework to rigorously categorize software and production defenses, generalize their key defense strategies and summarize their key limitations.

• We show our insights about concrete research directions for future rowhammer attacks and defenses.

2 BACKGROUND AND RELATED WORKS

2.1 DRAM Organization

Figure 1 presents an overview of a modern DRAM organization. The memory controller (MC) communicates with dual inline memory modules (DIMMs) through channels and each channel consists of a command bus, an address bus and a data bus. A DIMM has one or more ranks (e.g., single/dual-rank). A rank has a set of chips that operate in lockstep to reply to commands from the MC. Each chip has several banks. In a typical case where the data bus is 64-bit wide, a rank has 8 chips with 8 banks in a chip to serve the data bus. If multiple ranks share the same channel, the ranks are multiplexed. A bank is composed of multiple subarrays and a global row buffer (made up of sense amplifiers). Each subarray is a two-dimensional array of cells with a local row buffer, storing data from a specific

---

1While a number of hardware-based defenses (e.g., [33, 60, 66, 73, 83, 97, 121]) have been proposed from the academia, they are beyond our scope as they have not yet been adopted by the industry and are non-applicable to protecting commodity systems.

2In Section Appendices, we provide a detailed discussion of existing hardware defenses.
row of cells that is recently accessed. As such, subsequent accesses to the row are served by the row-buffer. The cells in a row are connected horizontally through a wordline. The cells in a column are connected vertically through a bitline to the local row buffer. The local row buffers are wired to the global row buffer via global bitlines. A cell consists of an access transistor serving as a switch and a capacitor storing a single bit.

**DRAM Address Mapping:** It consists of two main steps. In the first step, the MC maps physical addresses to logical DRAM addresses. A logical DRAM address refers to a 3-tuple of (bank index, row index, column index), in which the bank index includes DIMM, channel, and rank. In the second step, DRAM internal mapping primarily remaps logical DRAM addresses to physical DRAM addresses. As such, logical rows that are adjacent to each other in the view of the MC are likely to be physically non-adjacent in DRAM. For the mapping in the first step, it is publicly undocumented but has been reverse-engineered by previous works [87, 104, 112, 119] based on a timing side channel [77]. The remapping in the second step is also kept as confidential and reverse-engineered [17, 58, 82, 104].

### 2.2 Rowhammer in DRAM

Prior work has identified characteristics that affect rowhammer-induced bit flips [52, 54, 58, 61, 82] in DRAM-based systems. In the following, we introduce the major characteristics that are of interest to rowhammer exploits or tests.

**Hammer Pattern:** Kim et al. [61] found that rowhammer requires repeated accesses (i.e., hammering) to a row to cause permanent charge leakage (i.e., bit flips) within cells of adjacent rows. As row buffer caches recently accessed rows, it must be bypassed/flushed to ensure memory accesses are directed to rows, i.e., hammering rows. A hammer pattern refers to how rows are hammered. As shown in Figure 2, there are five existing hammer patterns.

- **Double-sided Hammer** hammers two rows that have one row apart within the same bank. As shown in Figure 2(a), all rows adjacent to the hammered rows are likely to be victims, among which the row sandwiched by the two aggressor rows is the most likely to be flipped. Single-sided Hammer randomly selects two rows for hammering, with the hope that the selected rows are within the same bank and thus the row buffer is bypassed [99]. When the rows being hammered happen to be in the same bank, each row neighboring the aggressor row is likely to be a victim. This is shown in Figure 2(b). One-location Hammer is illustrated in Figure 2(c) where it only picks one row for hammering [34]. It only applies to specific systems where the MC employs a close-page policy, that is, the MC preemptively closes an accessed row without opening another row, implicitly forcing the DRAM to flush the row buffer. Many-sided Hammer, shown in Figure 2(d), hammers more than two rows within the same bank one after another, proposed by TRRespas [26], resulting in a number of bit flips in DDR4 modules where target row refresh (TRR) [75] is deployed. Following [26], Blacksmith [49] is a variant of this pattern where many aggressor rows are hammered in a non-uniform way, unlike the hammer patterns above where each aggressor row is hammered at the same frequency. TRR-aided Many-sided Hammer shown in Figure 2(e) abuses TRR to induce bit flips [62]. When an aggressor row with red is hammered, TRR will be triggered to refresh its neighboring rows with pink. Thus, slightly hammering (i.e., a few activations) the pink rows can flip their adjacent victim rows.

**Hammer Count:** Hammer Count (HC) refers to the number of activations that hammer aggressor rows adjacent to a victim row. Take the double-sided hammer as an example, the total number of activations to two aggressor rows is referred to as HC [58]. DRAM’s susceptibility can be quantified by the minimum hammer count to induce the first bit flip, i.e., HC$_{\text{first}}$, which can be in the order of 20 K on DDR3 modules and 10 K on DDR4 modules [58].

**Data Pattern:** Values stored in the aggressor and victim rows also significantly affect bit-flip effectiveness, coined as data pattern [61]. There are four main data patterns, i.e., Solid (all cells store ‘0’ or ‘1’), RowStripe (rows storing ‘0’ are interleaved with rows storing ‘1’), ColStripe (columns storing ‘0’ are interleaved with columns storing ‘1’), and Checkered (cells storing either ‘0’ or ‘1’ in a checkerboard way). Among these patterns, RowStripe is the most effective [61]. For a single vulnerable cell, whether it can be flipped depends on not only itself but also the logic values of cells above and below [61]. For a true cell that is in a charged state and thus stores ‘1’, it can be flipped to ‘0’ most likely when the cells above and below it store ‘0’, but less likely when both cells store ‘1’. If a true cell is in a discharged state and thus stores ‘0’, it cannot be flipped to ‘1’ no matter what the above and below values are. For anti cells, their bit-flip direction is also monotonic and works in the opposite way. With this key observation, Ji et al. [51] develop effective data patterns at the granularity of a single cell to trigger bit flips in targeted cells while suppressing bit flips in other unwanted cells.

(Figure 2: Five existing hammer patterns in the literature. (We note that rows that are either adjacent or non-adjacent to aggressor rows can be vulnerable [58, 61] and the row distance from an aggressor row to a row with bit flips is called blast radius.)
Bank-level parallelism: The aforementioned characteristics focus on hammering a single bank. SledgeHammer [54] can hammer different banks simultaneously with a key observation that while memory accesses to different banks are serialized from the developer’s perspective, they are optimized by hardware to be parallel. Compared to previous works, it can induce much more bit flips.

2.3 Related Works

Mutlu et al. [78] provided a retrospective of rowhammer in 2019. While they surveyed published rowhammer attacks and defenses at the time of their writing, they discussed each of them succinctly and focused on their previously proposed hardware solution (i.e., PARA [61]). Besides, they discussed the futurespective of rowhammer by describing other potential vulnerabilities besides rowhammer in memory and advocating a principled system-memory co-design approach for rowhammer elimination, which was proposed in their original work [61] in 2014. In 2021, Loughlin et al. [72] presented a taxonomy of existing mitigations: isolation-centric, frequency-centric and refresh-centric, and proposed new hardware primitives to enable software defenses. Compared to them, our work has performed a systematization of existing rowhammer attacks and defenses on commodity systems, identified a new instruction for explicit cache flush and described concrete directions for possible attacks and defenses in real-world.

3 ROWHAMMER ATTACKS

3.1 Overview of rowhammer attack

In this section, we present a unified framework, in Figure 3, to show how a rowhammer attack affects commodity systems. It consists of four components, which are introduced below.

Origin: Generally, starting a rowhammer attack can be from either local or remote. For local origin, a rowhammer attacker requires crafting their code and executing it in the target system. The code is carefully written by different programming languages running on different microarchitectures, resulting in different file formats, e.g., malicious code can be written in Java and C/C++ and compiled into an Android APK [108] on ARM processors, in C/C++ and compiled into an executable binary on x86 processors [15, 34, 99], and in JavaScript running on either x86 or ARM [25, 37]. Note that the attacker-crafted code execution is performed assuming the attacker resides within the system with an unprivileged access, e.g., a native process, a browser sandbox, a para-virtualized or hardware-assisted VM, etc. In contrast, the attacker from the remote origin does not craft code on their own and instead sends network packets only to the target system. It was until [70, 105] that network packet transmission was implemented as a viable source of remote origin.

Environment: Starting from the origin, the attacker can reside in various system-defined environments following the principle of least privilege. Thus, the attacker is presented with different attack surfaces and permissions, resulting in different levels of ease to conduct an attack. For local origin, the environment is diverse. For example, it can be an unprivileged user process or VM, which allows native binary execution [15, 119]. In such environments, the attacker is allowed to leverage explicit CPU cache-flush instructions, cache eviction or Direct Memory Access (DMA) to access DRAM directly. For JavaScript code running in a browser sandbox, they have no such permissions and are restricted to cache eviction only. For remote origin-based attacks, they are restricted to DMA.

Methodology: The primary goal of this component is to trigger attacker-desired bit flips, which is achieved through two main steps.

1. The attacker aims to place sensitive objects (e.g., cryptographic keys) onto desired rows such that the sensitive objects have close proximity in DRAM to attacker-induced (target) objects, so-called memory massaging. The sensitive objects belong to a victim and their integrity or confidentiality can be compromised when placed onto desired rows. The attacker-induced objects refer to either unprivileged data that are explicitly created and accessed by the attacker, or privileged data that are implicitly created and accessed by the attacker (i.e., a confused deputy). For example, page tables are implicitly created for the attacker’s memory mapping and they can be accessed implicitly by the attacker through address translation [126]. This step can be achieved by abusing various built-in features of either software or hardware, resulting in different memory massaging techniques. To evaluate each technique qualitatively, three metrics are defined, i.e., stealthy referring to a technique that never causes out-of-memory (OOM), deterministic indicating a technique that is definitely able to achieve the aim of this step, and available showing a technique that is available by default in the system and has not yet patched by the system.

2. The attacker frequently accesses (i.e., hammer) attacker-induced objects in aggressor rows bypassing processor caches and row buffer and achieving desired bit flips. As a modern processor has a cache hierarchy to effectively reduce the time of accessing requested data in DRAM memory, access to attacker-induced objects will be first served from the processor caches. To enable accessing the objects from the memory, the attacker has to refrain from using the cache of the processor. Alternatively, the attacker can abuse DMA to bypass the cache. As row buffer stores recently accessed data in a DRAM bank and acts like the cache, the attacker must also bypass row buffer before each target memory access. With bypassing cache and row buffer, the attacker can hammer the attacker-induced objects when accessing them is frequent enough.

We note that the target objects can be either in the same row with the sensitive objects or adjacent to them as shown in Figure 3. When they are in the same row, sensitive objects cannot be corrupted. Instead, their information are inferred by flipping bits in their adjacent rows [64].

Objective: The consequences of desired bit flips are devastating to the system security, e.g., availability compromise (e.g., denial-of-service), integrity compromise (e.g., privilege escalation) and/or confidentiality compromise (e.g., cryptographic key leakage). By scrutinizing existing rowhammer attacks via the aforementioned framework, they are categorized based on their origin, environments, memory massaging techniques, and their objectives, as shown in Table 1. In the following, we discuss in detail about the methodology and objective.
3.2 Placing Sensitive Objects onto Desired Rows

To implement memory massaging, most existing attacks\(^3\) have leveraged different system features to propose different techniques. In this section, we discuss the techniques based on the features that have been exploited. We also evaluate each technique using the aforementioned three metrics as shown in Table 1.

**mmap:** It is a POSIX user interface, by which a user can specify a file to be memory-mapped and access it from her own address space. In the case of Linux, the kernel loads the file content into physical memory and allocates page-table entries accordingly in an on-demand manner, that is, the physical memory and page-table entries are not allocated until the file is accessed by the user process. Thus, the attacker can repeatedly invoke this feature, forcing the kernel to create a large number of page-table pages. These pages are sensitive objects and some are highly likely to be physically adjacent to the attacker pages in aggressor rows [15, 99, 126]. However, if mmap is abused excessively, available memory will be exhausted, resulting in OOM [99].

**fork:** It is a system call, by which a (parent) process can create a new child process. Similar to the invoking process, the child process has its own kernel structures, such as struct task_struct and struct cred. Among the structures, struct cred is of interest to an attacker, as it stores the user id of a process. An attacker can repeatedly invoke fork to create a large number of processes and thus sprays the memory with sensitive structures, with the hope that some struct cred will be placed next to the attacker pages in aggressor rows [15, 126].

**WebGL:** It is a JavaScript API for web developers to accelerate 2D and 3D graphics rendering in mainstream browsers. WebGL reserves a memory cache pool for processing textures in integrated GPUs, and the pool contains 2048 pages. To prevent vulnerable pages of textures from being freed to the pool, GLitch [25] releases 2048 previously-allocated textures to the pool. Right after releasing vulnerable textures, it applies for a large memory allocation of target JavaScript objects ArrayObjects. Thus, some ArrayObjects are predicted to reuse the freed vulnerable texture pages and thus corrupted by rowhammer.

**Page deduplication:** To improve memory utilization, a running system merges physical pages that have the same content, coined as page deduplication. This feature allows cloud service providers to run more VMs with the same amount of physical memory. To abuse this feature, the attacker crafts an attacker page at a specified vulnerable page. This page has the same content as the target sensitive page of another VM in a cloud setting. Thus, the underlying KVM hypervisor is lured into merging the two pages into the attacker-specified page [91]. Similarly, Bosman et al. [12] abused this feature for rowhammer attacks against Microsoft Edge browser. However, this feature has been disabled in commodity systems [76, 92], defeating rowhammer attacks exploiting this feature [12, 41, 91].

**Per-CPU page-frame cache update policy:** Linux buddy allocator divides system memory into multiple non-overlapping zones, e.g., ZONE_DMA, ZONE_DMA32 and ZONE_normal in the x86-64 architecture. If a process running on a CPU frees a page frame, the buddy allocator does not return the freed page frame to its corresponding zone. Instead, it pushes the freed page frame into a per-CPU page-frame cache (i.e., per-CPU pageset), which maintains a list of free page frames released by the CPU using the policy of Last-In-First-Out (LIFO). A consequence of this design is that the freed page frame is likely to be reused by the CPU and the page content is still hot in the per-CPU cache. RAMBleed [64], DeepHammer [124] and SpecHammer (user exploit) [106] first hold vulnerable page frames. Then they free the page frames and trigger targeted processes to run. With the update policy, the page frame cache will allocate the freed page frames for targeted sensitive pages of the processes.

**Page frame allocation policy:** Linux buddy allocator manages system memory as memory blocks and each block has a continuous physical memory region, the size of which is a power-of-two number of page frames. For blocks that have the same size, they are maintained in the same block list. To serve a memory allocation request, Linux first searches each block list iteratively for a block that satisfies the request. If none of the blocks have the requested size, Linux splits a larger-sized block into two blocks, returns one

---

\(^3\)A few attacks (i.e., SGX-Bomb [48], HammerScope [16] and Netherhammer [70]) do not need memory massaging, as they only require flipping bits in coarse-grained memory locations rather than surgical locations.
Table 1: A taxonomy of rowhammer attacks against commodity systems based on the framework in Figure 3. Particularly, the memory massaging technique used by each attack is evaluated via three metrics, i.e., whether it is stealthy, deterministic and/or available (N/A indicates that a work did not use any memory massaging technique). For the discussion about future explorations, please refer to Section 5.1.

| Origin | Environment | Software | Hardware | Real-World Attack | Memory Massaging | Objective |
|--------|-------------|----------|----------|-------------------|-------------------|-----------|
|        |             | NIC w/ RDMA | x86 | Throwhelammer [105] | Stealthy | Deterministic | Available | Gain app. priv. |
|        |             | CPU       |         | Nethammer [90]     | ✓       | ✕         | ✕         | Cause DoS |
|        |             |           |         | Bosman et al. [12] | ✓       | ✓         | ✕         | Escape sandbox |
|        |             |           |         | SMASH [21]         | ✕       | ✓         | ✕         | Escape sandbox |
|        |             |           |         | Seaborn et al. [99] | ✕       | ✕         | ✕         | Escape sandbox |
|        |             |           |         | Rowhammer.js [36]  | ✕       | ✕         | ✕         | Escape sandbox |
|        |             |           |         | CATTmew [15]       | ✓       | ✕         | ✓         | Gain kernel priv. |
|        |             |           |         | FTHammer [126]     | ✓       | ✕         | ✓         | Gain kernel priv. |
|        |             |           |         |                   |         |           |           |               |
| Local  |             | x86 | CPU |                   |         |           |           |               |
|        | Native Process | x86 | CPU |                   |         |           |           |               |
|        | Native Process | x86 | CPU |                   |         |           |           |               |
|        | Remote Network Protocol | x86 | CPU |                   |         |           |           |               |
|        | Native Process | x86 | CPU |                   |         |           |           |               |
|        | Native Process | x86 | CPU |                   |         |           |           |               |

Block to fulfill the request, and adds the other block into an appropriate block list. Upon a memory deallocation request, Linux tries to merge the freed page frames with their neighboring free page frames (if possible), generating a bigger block and updating it into a block list. This allocation policy can minimize external fragmentation of physical memory.

Drammer [108] and RAMpage [109] abuse this allocation policy by exhausting and freeing memory blocks in a predictable way, with the goal of placing a page-table page onto an available and vulnerable page frame. However, they are highly likely to cause OOM. Besides, to abuse the policy, both attacks must use the Android ION interface, which has been deprecated by recent android kernel versions. SpecHammer (kernel exploit) [106] and GadgetHammer [107] leverage the policy to create memory pressure, which will probably force the kernel to use a vulnerable user page to store targeted kernel stack variables. SMASH [21] consumes all the memory blocks that are smaller than 2 MB and then requests additional 4 KB page frames for JavaScript ArrayBufferS, which forces the buddy allocator to split a previously freed vulnerable 2 MB memory block. The split vulnerable 4 KB page frame is predicted to host JavaScript ArrayBuffer. If many blocks smaller than 2 MB are exhausted, it might cause OOM. Besides, 2 MB memory blocks are supported by transparent huge pages (THPs), which are not available by default in the recent browsers. This non-default configuration also renders another browser-oriented exploit (i.e., rowhammer.js [36]) ineffective. CATTmew [15] exhausts the memory blocks that are smaller than or equal to the DIMM row size (e.g., 256 KB). Then the buddy allocator is forced to split a large memory block that is twice the row-size when the attacker allocates 4 KB attacker pages and uses mmap for 4 KB page-table page allocations. By doing so, page-table pages are likely to be placed onto vulnerable pages next to the attacker pages.

**memcached item allocation policy:** A memcached architecture manages key-value pairs, which are stored in-memory as *items*. The corresponding data structure for each item is *struct stritem* and it has different sizes ranging from 96 bytes to 1 MB. The memcached maintains a singly linked list, i.e., hash chain, which stores different keys but with the same hashes (i.e., colliding keys). The hash is computed by a hash function with the key as an input. However, the function is not cryptographically secure and thus its computed hash value can be resulted from different keys. First, the attacker [105], as a remote client process, issues numerous SET requests to the memcached for crafting many 1 MB items in which each key is hashed to the same value. These items are chained together and can be read by the attacker. Then, the attacker sends a DELETE request to force the memcached to free a target 1 MB item, and then issues GET requests to lure the memcached into reusing the 1 MB item for smaller-sized item allocations, with the hope that one smaller item will land on a vulnerable DRAM location. However, DELETE is not supported by default in recent versions of RDMA-based memcached [105].

**Page cache eviction policy:** To boost system performance, Linux manages unused page frames as page cache. If a memory-mapped file is released by a process, Linux still keeps its data in the page
Table 2: A taxonomy of techniques proposed by rowhammer attacks in avoiding CPU cache, GPU cache and row buffer.

| Objective               | Technique                          |
|-------------------------|------------------------------------|
| Avoid using CPU cache   | explicit cache-flush instructions  |
|                         | non-temporal instructions          |
|                         | (Intel CAT-assisted) cache eviction|
|                         | direct memory access               |
| GPU cache               | cache eviction                     |
| row buffer              | different hammer patterns          |

cache. If the file is accessed again, Linux does not have to load the file from non-volatile storage such as disks and instead serves the access quickly from page cache. Linux considers pages in the page cache as available memory and thus will evict obsolete file data from these pages to load recently accessed files. Gruss et al. [34] observe that if Linux evicts file data from a page-cache page frame, it reloads the evicted file data onto a different page frame upon access. By abusing the eviction policy repeatedly, they relocate sensitive code of a setuid process onto vulnerable pages [34] deterministically.

Page table update policy in paravirtualization: In paravirtualization, guest OS inside each VM is modified to work with its underlying hypervisor. In Xen paravirtualization, MMU of a paravirtualized VM is modified to fill its page table entries with physical page frame numbers (PFN)s rather than pseudo PFNs. To isolate each VM, Xen hypervisor enforces an invariant that a VM cannot write its page tables directly. Instead, the VM must invoke a hypercall for page table updates. However, the VM is allowed to specify page frames that host its page tables. With this key observation, Xiao et al. [119] place a target page-table page onto a specified vulnerable page frame without requiring much memory.

3.3 Hammering Attacker-Induced Objects

3.3.1 Refraining from using CPU/GPU cache. As shown in Table 2, there are four techniques that have been proposed to avoid using either CPU or GPU cache and thus induce a DRAM memory access, that is, explicit cache-flush instruction, non-temporal instruction, cache eviction and direct memory access. In the following, each technique is introduced in detail.

Explicit cache-flush instruction: Modern CPUs have multiple levels of cache. For each level of cache, it consists of cache sets and each cache set consists of multiple cache lines (also known as ways). Existing x86-based rowhammer attacks have identified a couple of unprivileged instructions to explicitly flush a CPU cache line specified by a memory address from all levels of cache hierarchy, i.e., clflush and clflushopt. clflush is available in all Intel-based microarchitectures while clflushopt is only available since the recent Intel 6th generation microarchitectures (e.g., Skylake). Besides, we perform an analysis of all the Intel instructions relevant to CPU cache control [46] (i.e., clflush, clflushopt, clwb and cldemote) and observe that clwb, as an unprivileged instruction, can also be used for explicit cache flush. Specifically, clwb is used to write a modified cache line (represented by a linear address) back to memory. After that, the cache line may be retained in the cache hierarchy or may be invalidated from the cache hierarchy by hardware (an indicator of cache flush).

Our empirical observation is performed on a Supermicro server with Intel Xeon Gold 6230 CPU (a Cascade Lake microarchitecture) as clwb is only available in certain Intel microarchitectures (e.g., Intel Xeon Processor Scalable Family and 11th Generation Intel Core processor family). Based on Mastik4, we design a finite loop that runs 1,000,000 times, within which mfence (a memory barrier instruction in Intel x86 used to ensure memory ordering,) is followed by measuring the access latency to a linear address from an allocated memory buffer. The access latency is then averaged. As a comparison, mfence is replaced by an instruction sequence of clwb and mfence. The results show that the access with clwb takes about 245 CPU cycles while the access without clwb costs around 20 CPU cycles, showing that clwb has explicitly flushed a specified cache line. Further, we measure the elapsed CPU cycles of invoking each explicit cache-flush instruction. Each invocation is also repeated for 1,000,000 times in a finite loop, generating an averaged measurement. For clwb, clflushopt and clflush, their respective averaged cycles are 252, 263 and 287, showing that clwb outperforms the other two in their hammering efficiency.

Non-temporal instruction: Given that unprivileged non-temporal instructions used for non-temporal data references employ cache bypass, the attacker can also invoke non-temporal store instructions to perform write access to data directly into memory without CPU cache involvement. Qiao et al. [88] and BitMine [130] thoroughly analyze non-temporal store instructions on Intel microarchitectures and present a string of available instructions, i.e., movnti, movntdq, movntpd, movntps, movntq, maskmovq and maskmovdq.

Cache eviction: As CPU/GPU cache has limited capacity, the attacker can evict a target address from either CPU [36] or GPU [25] by accessing a set of many enough virtual addresses. The addresses in the set are congruent in a way that they are mapped to the same cache set and the same cache slice as the target address. This technique is particularly useful when launching an attack in a browser sandbox environment where the cache-flush and non-temporal instructions are not available. As Intel cache allocation technology (CAT) [40] allows system software to partition the CPU cache, each subset of the cache is dedicated to a process or virtual machine to mitigate cache thrashing. In such a case, the cache capacity for a process decreases significantly, indicating that the number of congruent addresses in the eviction set will decrease. Aga et al. [1] have abused Intel CAT to facilitate rowhammer attacks.

Direct memory access: Last, the attacker can access uncached memory directly by abusing the DMA feature. The DMA memory is marked as uncached and thus accessing it will bypass the CPU cache. Unlike the non-temporal instructions that require write access, DMA access can be any type, i.e., write, read or execute. In the x86 architecture, both Throwhammer [105] and Nethammer [70] launch the attack by sending network packets directly into DMA memory. In the ARM platform, Dramer [108] can access DMA memory even in a local unprivileged process.

3.3.2 Bypassing Row buffer. As row buffer resides in each bank, the processor and peripheral devices cannot clear row buffer explicitly.

4Mastik is a microarchitectural side-channel toolkit and available at https://github.com/vtADE1A1DE/Mastik
To this end, the attacker can leverage the hammer patterns introduced in Section 2.2 to clear row buffer implicitly. To implement double-sided hammer or many-sided hammer, the attacker must know which rows to access and thus requires (partial) mappings from virtual addresses to physical addresses, physical addresses to logical DRAM addresses and logical to physical DRAM address.

For the last two mappings, they are either available from public documentation or have been reverse-engineered as discussed in Section 2.1. For the first mapping, an unprivileged process can acquire it through the pagemap interface but the interface has been restricted to root users since Linux version 4.0 [101] to mitigate rowhammer. Alternatively, the attacker can abuse the feature of huge page (e.g., 2 MB) in the x86 architecture or Android Ion in the ARM architecture to request a large block of contiguous physical memory. Within the large memory, it is highly likely to find multiple same-bank rows. However, huge page is disabled by default in the browser sandbox environment. To address this issue, SledgeHammer [54] leverages a contention-based cache side channel in verifying the contiguity of memory blocks. By doing so, they are able to find many 2 MB blocks that physically contiguous.

3.3.3 Frequent Accessing Attacker-Induced Objects. Most attacks require putting objects explicitly created by the attacker onto aggressor rows and applying different techniques in Table 2 to avoid processor caches and row buffer for hammering the objects, which is coined as explicit hammer [126].

For objects that are implicitly created, the attacker cannot access them explicitly and instead can use a benign entity to implicitly access the objects, the so-called implicit hammer [126]. The entity can be hardware (e.g., the processor) or software (e.g., system call handler). For instance, PThammer uses the page-table-walk feature of CPU while GhostKnight [127] abuses the speculative execution feature of CPU for implicit hammer. To trick the benign entity into frequently accessing the objects, the effective way for bypassing the caches is restricted to cache eviction [126].

3.4 Objective

3.4.1 Causing Denial-of-service. An attacker can demonstrate a DoS attack either locally or remotely. Both SGX-Bomb [48] and Gruss et al. [34] have demonstrated local DoS attack by abusing a feature of Intel SGX [19], i.e., data integrity check. Specifically, an SGX enclave has a physically contiguous memory region, which is encrypted in DRAM and protected from non-enclave memory accesses. If data in the enclave memory is corrupted, the corruption will be detected by SGX’s memory encryption engine (MEE), resulting in a system halt. With this observation, the attacker can simply apply for an SGX enclave and induce a single bit flip in the enclave memory to make the whole system unresponsive.

Compared to local attacks, remote attacks do not require code execution from the attacker. Specifically, Nethammer [70] simply sends crafted network packets repeatedly to a target system. To handle the packets, the system frequently accesses (read, write or execute) relevant memory. If the system applies techniques in Table 2, bit flips can be induced remotely in the system and some of them can occur in the file system, thus crashing the entire system.

3.4.2 Gaining Privilege Escalation. Various rowhammer attacks break different forms of memory isolation on different microarchitectures with the goal of achieving privilege escalation. These attacks are discussed based on their targeted privilege software.

Targeting browser: A sandbox provides a browser-controlled environment where an untrusted program is running with restricted permissions, separating untrusted code from trusted code within the same process. There are five attacks that have leveraged a malicious program to break the sandbox environment and gain privilege escalation. Among them, one uses native code and four are based on JavaScript code.

Google native client provides a sandboxing environment to run validated instructions from mainstream architectures such as x86 and ARM. Seaborn et al. [99] demonstrate a Chrome sandbox escape. Particularly, a malicious native client program uses bit flips to corrupt opcodes of validated instructions and turns them into unsafe ones, enabling arbitrary calls of host OS’s syscall. Gruss et al. [37] escape a Firefox browser by crafting JavaScript code in a website and corrupting page tables through bit flips. SMASH [21] corrupts metadata of a JavaScript ArrayBuffer and compromises the latest Firefox browser of 2021. Bosman et al. [12] compromise a Microsoft Edge browser through rowhammer. They abuse the page deduplication feature to craft counterfeit JavaScript Uint8Array objects and corrupt the objects with bit flips, resulting in an arbitrary read/write primitive within the browser. Glitch [25] is an attack against integrated graphics processing unit (GPU) on a mobile platform, which corrupts elements of JavaScript ArrayBuffer and escapes a Firefox sandbox.

Targeting memcached server application: A memcached server application allows a remote client to send network packets directly into a specified memory region, i.e., remote DMA (RDMA). Throwhammer [105] uses RDMA-enabled NIC to compromise the entire memcached server application with arbitrary write and code execution. It abuses the memcached item allocation policy to corrupt memcached items by simply issuing network packets without requiring any local code execution.

Targeting setuid process: setuid is for set user ID on execution, which is a type of file permission in Unix-like operating systems such as Linux. When an unprivileged user launches a binary with setuid, the binary will run with root privilege rather than the user privilege. Gruss et al. [34] mount an attack from an unprivileged process against a setuid process. They break MMU-enforced inter-process isolation and bypass all existing defenses prior to their proposed attack. Particularly, they abuse the page cache eviction policy to corrupt opcodes of critical branches in the setuid process, resulting in root privilege escalation.

Targeting Kernel: Kernel is one of the most appealing targets. There are four attacks breaking user-kernel isolation and corrupting page tables, resulting in kernel privilege escalation from an unprivileged native process.

Specifically, Seaborn et al. [99] mount the first rowhammer attack against kernel in the x86 architecture. They propose abusing the mmap interface to spray the last-level page-table pages, some of which will be placed adjacent to attacker-accessible pages. CATTmew [15] demonstrates an attack against CATT [13], the first DRAM-aware user-kernel isolation. They have observed the
so-called memory-ownership weakness of CATT, that is, there exist kernel buffers that are still accessible to unprivileged users in the presence of CATT. With this observation, they hammer video buffers or SCSI Generic buffers and trigger bit flips in some page tables created by mmap. PThammer [126] introduces implicit hammer-based attack, which tricks the CPU into hammering page tables, resulting in other page table corruption. Drammer [108] performs an attack against the ARM architecture. It abuses the page frame allocation policy of Linux buddy allocator to surgically induce bit flips in a target page-table entry.

**Targeting paravirtualized hypervisor:** In Xen paravirtualization, a VM has no write access but only read access to its own page tables. Thus, the VM’s page table updates must be approved by the underlying hypervisor. Xiao et al. [119] propose an attack against the paravirtualized hypervisor. They abuse the page table update policy in paravirtualization to flip bits in page tables and gain write access to them, breaking the MMU-enforced VM-hypervisor isolation and gaining the hypervisor privilege.

**Targeting victim HVM:** Compared to the paravirtual VM, a hardware-assisted VM (HVM) is unaware of underlying hypervisor and other HVMs. DFFS [91] demonstrates an attack against a KVM-based HVM from another HVM. It abuses the page deduplication feature to corrupt targeted files (i.e., OpenSSH public keys, Debian URLs in sources.list and trusted public keys in trusted.gpg) residing in the page cache of a victim HVM, breaking inter-HVM isolation and gaining full control of the victim HVM.

### 3.4.3 Leaking sensitive information. This category of attacks aims to leak different types of sensitive information.

**Targeting cryptographic algorithm:** RSA, as a public-key cryptosystem, is one of the most widely used asymmetric cryptography algorithms and has been the attack target for three rowhammer attacks. Specifically, Bhattacharya et al. [10] manage to induce a single bit flip fault in the secret exponent of RSA public key exponentiation in the GNU-MP big integer library. Similarly, JackHammer [114] induces bit flip faults in the OpenSSL RSA implementation, resulting in faulty signatures. RAMBleed [64] is the first attack that uses rowhammer-induced bit flips as a read primitive to break inter-process isolation and leak an RSA key from the OpenSSH daemon.

**Targeting kernel data:** Kernel data is a high-value target to an unprivileged attacker. Both Spechammer (kernel exploit) [106] and GadgetHammer [107] primarily target kernel stack variables to leak kernel data. Their major difference is: Spechammer [106] uses rowhammer to modify attacker-controlled offset shared with the kernel for Spectre gadgets, thus discovering more Spectre gadgets and reviving Spectre attacks. For GadgetHammer [107], it looks for gadgets of a particular code pattern, i.e., kernel code using nested pointer dereferences to return benign data to the user space. By flipping the pointers, arbitrary kernel data can be leaked.

**Targeting KASLR:** Kernel Address Space Layout Randomization (KASLR) is a Linux kernel feature that randomizes the base address of kernel’s text segment to defend against code-reuse attacks. Particularly, the base address can have 512 possibilities every time the kernel boots up. While there have been many attacks that can leak KASLR (e.g., Prefetch [35]), HammerScope [16] exploits rowhammer as a side channel to leak the run-time base address. Its key observation is that higher DRAM power consumption will increase the hammer count that is required to flip a bit.

**Targeting DNN model:** As training a model is resource-intensive and the relevant training data is sensitive, how to protect intellectual property of a DNN model on both server and mobile platforms has attracted much attention from security researchers. Similar to RAMBleed, DeepSteal [90] uses the rowhammer bit flip as a read primitive and they leak weights from a target DNN model through this primitive.

### 3.4.4 Degrading DNN model. DNN has been pervasively used recently due to its stunning inference accuracy. Hong et al. [41] mount a rowhammer attack from an unprivileged process against a full-precision DNN model process. They exploit the page deduplication feature to change model weight and thus degrade model inference accuracy. DeepHammer [124], assuming a white-box of a target quantized model, abuses the per-CPU page-frame cache update policy to corrupt specified weights in each model layer, reducing the model’s inference to random guess. Different from them, Li et al. [68] leverages the page cache eviction policy to inject faults into underlying third-party machine learning library, thus requiring no model knowledge and achieving model inference depletion.

### 4 ROWHAMMER DEFENSES

#### 4.1 Software Defenses

Existing software defenses are proposed to defend commodity systems against rowhammer attacks and they can be classified into three categories, i.e., ad-hoc attempts, DRAM-aware isolation and RowHammer(RH)-triggered detection, shown in Table 3. Each of them aims to mitigate step 1 or 2 in the methodology component of Figure 3. In the following, we discuss each category in detail.

**4.1.1 Ad-Hoc Attempts.** We introduce four major countermeasures in this category. First, multiple computer manufacturers such as Apple [3], HP [42], and Lenovo [67] provide BIOS updates to double the DRAM refresh rate, that is, the DRAM refresh period is decreased from 64 ms to 32 ms. However, ANVIL [4] still can induce bit flips in the doubling refresh rate. Even worse, Kim et al. [58] observe that a bit flip can occur in mainstream DDR4 chips within 1 ms. Second, Google chrome browser has disallowed the use of both explicit cache-flush instructions [74] and non-temporal instructions [94], mitigating the NaCl exploit from [99]. Still, there are other available techniques in avoiding CPU cache in Table 2, e.g., cache eviction and direct memory access. Third, since version 4.0, Linux kernel removes non-root access to the pagemap interface [101], which prevents potential attackers from acquiring the mapping from a virtual address to a physical address. Clearly, the countermeasure is not effective, as discussed in Section 3.3.2. Last, the page deduplication feature has been turned off by default only to mitigate specific attacks [12, 91] in commodity cloud platforms [111].

**4.1.2 DRAM-Aware Domain Isolation.** This category of defenses separates security-sensitive memory from attacker-accessible memory in DRAM by putting guarding rows in between. By doing so, the attacker-induced bit flips are absorbed by the guarding rows.
Reinforcing user-kernel isolation: To defend against rowhammer attacks targeting the kernel, CATT [13] partitions the DRAM rows within each bank into two parts— one part for the user domain and the other part for the kernel domain, and uses at least one empty row to separate the two domains. As the empty rows will absorb the bit flips from the user domain, the kernel’s integrity is protected and little runtime overhead (within 1%) is incurred.

Reinforcing inter-process isolation: To prevent rowhammer attacks targeting security-sensitive process such as setuid, RIP-RH [11] enforces DRAM-aware memory isolation for the targeted processes by segregating their physical memory into dedicated DRAM areas. As such, these processes are not allowed to share memory with other processes, otherwise, the attacker can hammer the shared memory. Their evaluated benchmarks report a modest run-time overhead (within 4%).

Reinforcing DMA buffer isolation: There are two reinforced DMA isolation schemes, that is, ALIS [105] on x86 and GuardION [109] on ARM. ALIS surgically isolates RDMA-enabled DMA memory with guarding rows and thus every attacker-generated bit flip is confined to guarding rows, defending against Throwhammer [105] and introducing almost no run-time overhead to the protected system. Similar to ALIS, GuardION [109] mitigates DMA-based attacks particularly on mobile devices by isolating ION-enabled DMA buffers using guarding rows. Also, its overall performance overhead is acceptable, i.e., within 6% on average performance increase and 7% decrease in its worst case.

Reinforcing page-table isolation: As all existing kernel-privilege-escalation attacks target corrupting page tables, CTA [118] aims to reinforce DRAM-aware page table isolation. It proposes a two-step approach. In the first step, page-table pages are allocated from a dedicated memory region that has high physical addresses and resides at the end of the memory space. In the second step, rows with true cells are selected from the memory region for page-table allocation. As the dedicated memory region is not separated from the rest of memory region with guarding rows, a bit flip can occur in the page-frame-number field of a page-table entry (PTE). However, the corrupted PTE will only point to a new physical address lower than the original one and it will never point to another PTE. Thus, the attacker cannot gain access to a page-table page. Base on their evaluated benchmarks, little overhead (within 1%) is reported.

Reinforcing Intra-HVM isolation: ZebRAM [63] leverages hardware assisted hypervisor to split DRAM rows of a target HVM into two regions in a zebra pattern, that is, even rows for safe system processes and odd rows for unsafe swap space. Thus, hammering even rows is effective in inducing bit flips in adjacent odd rows of swap space. However, hammering odd rows is too slow to incur a single bit flip in adjacent even rows where system data reside. For potentially corrupted data in the swap space, ZebRAM performs data integrity check before using them. Its evaluated benchmark reports an overhead of 5%.

4.1.3 RH-Triggered Detection. For step 2 in methodology of Figure 3, existing rowhammer attacks require hammering target objects, which results in abnormal software/hardware side effects. We note that MAD [116] can detect many rowhammer attacks that have abnormal behaviors in memory allocations. However, it has not been evaluated on a commodity system (e.g., security effectiveness and performance overhead) and thus is not listed in Table 3.

Table 3: A taxonomy of software defenses. For the discussion about future explorations, please refer to Section 5.2.

| Objective                  | Software Defense | Requirement               |
|----------------------------|------------------|---------------------------|
| DRAM-Aware Reinforce       |                  |                           |
| Reconf Page-Table          |                  |                           |
| Inter-HVM Isolation        |                  |                           |
| RH-Triggered Detect        |                  |                           |
| Abnormal CPU cache misses  | ANVIL [4]        | Add-on kernel module      |
| Abnormal EM emissions       | RADAR [31]       | Software-defined radio device |
| Abnormal row-access patterns| SoftTRR [128]    | Add-on kernel module      |
| Abnormal DRAM power         |                  |                           |
| Abnormal row-buffer conflicts|                |                           |

and will not affect target security domain. Existing schemes reinforce MMU-enforced isolation at different granularities to protect different security domains [11, 13, 63, 105, 109, 118].
overall runtime overhead is low within 1% but it incurs a worse-case overhead of 8%.

**Detecting abnormal binary code:** Most rowhammer attacks use explicit cache-flush instructions, non-temporal instructions or cache eviction sets to avoid CPU caches. With this observation, Irazoqui et al. [47] implement a static analysis tool, coined MASCAT, to identify the typical rowhammer attributes within target binary code. They note that the tool is an extensible framework and can include other rowhammer attributes. As this tool performs a static analysis, it does not introduce run-time overhead.

**Detecting abnormal EM emanations:** With a key observation that hammering activities emanate distinguishable electromagnetic (EM) signals, RADAR [131] leverages a $299 radio-based external device to capture EM signals from DRAM. After the EM emanations are processed, RADAR can expose recognizable hammering-correlated sideband patterns in the spectrum of the DRAM clock signal. To achieve effectiveness and robustness classification, RADAR feeds hammering-correlated spectrograms into a convolutional neural network (CNN) for training. If the CNN’s inference indicates an ongoing rowhammer activity, RADAR can kill suspicious processes. Since it works outside of the protected system, it imposes almost no overhead to the system.

**Detecting abnormal row-access patterns:** SoftTRR [128] is a software target-row-refresh technique that prevents page tables from being corrupted. It leverages the row bits in page table entries to frequently track accesses to any rows that are adjacent to rows hosting page tables. When the tracked access counter reaches a predetermined threshold, refreshes will be performed to corresponding rows with page tables. Its runtime overhead is negligible within 1%.

4.1.4 Summary of Software Defenses. Clearly, the ad-hoc attempts have security issues that are discussed in Section 4.1.1. Below, we discuss the major limitations of the other two categories in Table 3.

**DRAM-aware domain isolation schemes:** First, all of the schemes defend against multiple (not all) row-hammer rowhammer attacks. Besides, they require modifying the memory allocator either in the kernel or the hypervisor to enforce DRAM-aware memory allocation, making them hard to adopt in commodity systems. Second, as rowhammer affects non-adjanct rows, the blast radius from aammered row is dependent on specific DRAM chips. From the observation [58], the burst radius of some chips can be up to 6. Thus, the defenses need to determine the blast radius for given chips and waste the rows within the blast radius as the guarding rows or the swap-space memory. When the blast radius grows, the wasted rows increases. Last, all of these defenses except CTA [118] rely on the invalid assumption that logical and physical DRAM addresses are the same, ignoring the DRAM row remapping. For CTA, its security guarantee is broken by the data-scrambling feature enforced by the memory controller [39].

**RH-triggered detection schemes:** RADAR [131] requires a specialized radio device, which is unlikely to be purchased and deployed in practice by individual laptops, company servers, and clouds. Besides, both ANVIL [4] and MASCAT [47] cannot detect attacks that originate from non-CPU hardware components, e.g., GLitch [25]. SoftTRR [128] implementation is ineffective in cases where the rowhammer blast radius is greater than 6.

| Objective                  | Production Defense | Prototype Location |
|---------------------------|--------------------|--------------------|
| Data Integrity Check      | MC-Aware DRAM-based ECC [95] | MC & Extra DRAM |
|                           | On-die ECC [79]    | *DRAM*
|                           | (Future Exploration) | |
| Row Activation Count      | on-die TRR [75]    | DRAM |

4.2 Production Defenses

Here, we classify existing production defenses into two categories, shown in Table 4, which are introduced below (Section Appendix provides a more detailed discussion).

4.2.1 Data Integrity Check. This category uses different algorithms to detect and correct bit flips by performing data integrity check.

**MC-Aware DRAM-based ECC:** ECC (Error Checking and Correcting) is proposed to mitigate data-corruption by introducing additional bits to store the parity of data that is written into DRAM. A production-level case of ECC is single-error-correction-double-error-detection (SECDED) hamming code. In this case, ECC functions are implemented in the MC on commodity processors, that is, the MC has 8-bit hamming SECDED code with every 64-bit data. Thus, the data bus between DRAM and MC is extended from 64 bits to 72 bits and each rank has an extra chip to store ECC code.

**On-die ECC:** Different from the aforementioned ECC, on-die ECC, as its name suggests, is integrated directly into the same DRAM chips and invisible to the MC [79, 85]. As the on-die ECC operates entirely within the same DRAM chips, it is adopted by mainstream DRAM manufacturers to reduce the data-corruption rates in new generations of DRAM chips (e.g., LPDDR4 [81] and DDR5 [50]). As the DRAM manufacturers consider the on-die ECC functions as proprietary, its details cannot be found in public documentation and have been reverse-engineered in recent works [84–86].

4.2.2 Row Activation Count. DRAM manufacturers implement different schemes of row-level on-die Target Row Refresh (TRR) [75] to prevent data in DRAM from being flipped. As its name suggests, TRR counts rows that are being activated and refreshes their adjacent rows when the counts reach a predefined threshold. Typically, a TRR scheme implements a bank-level counter table, and counts a limited number (e.g., 16) of different rows within the same bank.

4.2.3 Summary of Production Defenses. All existing production defenses except the on-die ECC have been proved vulnerable to rowhammer. Specifically, the MC-Aware DRAM ECC was believed to deter rowhammer-based attacks until ECCploit [18] in 2019 demonstrated that it was susceptible to rowhammer by inducing enough (e.g., 3) bit flips in a single code word (e.g., 72 bits). For on-die ECC, given its functions have been reverse engineered as discussed above, it is worth investigating whether it is also susceptible to rowhammer. For on-die TRR implementations in recent DRAM, they are publicly undocumented. However, different TRR implementations from three major DRAM manufacturers have been reverse-engineered recently and DRAM based on them are still vulnerable to rowhammer [26, 38, 62], breaking their security guarantee.
5 OUR INSIGHTS ON FUTURE EXPLORATIONS

In this section, we first discuss attacks followed by a DRAM-aware memory isolation in commercial clouds. We then present how to detect rowhammer motivated by its key characteristics.

5.1 Potential Rowhammer Attacks

Gaining hypervisor privilege: The only rowhammer attack against HaV employed by public clouds is from Razavi et al. [91]. However, this attack has been rendered unavailable since page deduplication was disabled. Also, the attack achieves a control of a target HVM without compromising the hypervisor. Thus, it is worth exploring rowhammer to break MMU-enforced HVM-hypervisor isolation and gain hypervisor privilege in public clouds.

In native environments, MMU maintains a single layer of page tables, which maintains the mappings between virtual addresses and physical addresses. In HaV, mainstream microarchitecture vendors (e.g., Intel [44] and AMD [2]) modifies MMU to maintain two layers of page tables. The first-layer page table, i.e., Guest Page Table (GPT), is managed by HVM, and the second layer, i.e., Extended/Nested Page Table (EPT/NPT), is managed by the hypervisor. For a memory access from the HVM, MMU performs page-table walk at both levels (if combined TLB and host TLB searches miss) and retrieve relevant page-table entries (PTEs) from memory (if PTEs are cached). If the attacker can produce frequent implicit accesses to memory locations where EPTs/NPTs reside, they can corrupt other adjacent PTEs and gain the hypervisor privilege.

Exploiting isolated GPUs in x86: As isolated GPUs in the x86-based processors are widely used for training DNN model tasks, mainstream cloud providers (e.g., Amazon, Google, Oracle, and Alibaba) offer cloud GPU services in a multi-tenant manner [80]. With this service, individuals or small businesses are charged for the time they have used the shared cloud GPUs for their DNN model training instead of purchasing their own physical GPUs. Thus, in a real-world scenario where an attacker shares cloud GPUs and GPU memory with a victim tenant, it is worth exploring whether the attacker can refrain from using GPU caches, induce bit flips in GPU memory of a special DRAM type (e.g., GDDR5 is vulnerable to rowhammer [67]) and mount end-to-end attacks, such as compromising the DNN model of a victim tenant.

5.2 Potential Rowhammer Defenses

Reinforcing DRAM-aware isolation in hardware-assisted virtualization: To counteract the aforementioned cloud-based rowhammer candidates, we can reinforce a DRAM-aware isolation for different security domains, such as inter-HVM and HVM-hypervisor. As kernel-based virtual machine (KVM) in the Linux kernel is widely used as a hypervisor, the isolation prototype can be based on KVM by extending its physical memory allocator to be rowhammer-aware. This KVM-based isolation is expected to be transparent to the HVMs and incur small performance overhead. A good example is Silox [71] that was published recently. It enforces DRAM-aware isolation in KVM by placing each VM’s and hypervisor’s data onto different DRAM subarray groups with guarding rows in between.

Detecting abnormal DRAM power: To improve performance, the processor always tries to serve memory-access requests with its caches. For rowhammer attacks, they induce frequent memory accesses to DRAM, which is expected to exhibit an abnormal distribution of power consumption from DRAM over a specific time period, thus being a possible solution to rowhammer detection. Intel running average power limit (RAPL) may be leveraged. Specifically, it is a mechanism of setting power and thermal limits on the processor packages and DRAM [46], which is available since Sandy Bridge. Intel RAPL has a set of model-specific registers (MSRs) to monitor the power consumption over a short time interval for different domains, e.g., package, power planes, and DRAM. For user-space access, Intel implements a power capping framework (i.e., powercap), which exposes the MSRs through sysfs. Note that since the RAPL-based power side-channel has been discovered [69], only root-privilege users have legitimate access to the sysfs interface. As such, it is worth investigating Intel RAPL to detect abnormal DRAM power induced by rowhammer.

Detecting abnormal row-buffer conflicts: Given that rowhammer requires accessing one or multiple rows frequently to trigger bit flips, it will result in an abnormal number of row buffer conflicts, which is thus an indicator of an ongoing rowhammer activity. We observe that an Intel server can be of help by providing the statistics of row buffer conflicts based on the Intel manual [45]. Specifically, there exists a critical part outside Intel cores and it is called "Uncore". The uncore part includes PCI-express, memory controller, etc. Intel implements a list of performance counter events to monitor the performance of the uncore part, among which the PRE_COUNT.PAGE_MISS event counts DRAM PRE events caused by page misses [45]. Here, a page miss occurs when a row buffer is open but has a wrong row in it and thus it is resulted from a "row-buffer conflict". Besides PRE_COUNT.PAGE_MISS, the CAS_COUNT.RD event counts all DRAM read requests and the CAT_COUNT.WR event records all DRAM write requests. To this end, another event called PCT_REQUESTS_PAGE_MISS reports the percentage of memory requests that result in row buffer conflicts, i.e., PRE_COUNT.PAGE_MISS / (CAS_COUNT.RD + CAS_COUNT.WR). Consequently, how to leverage PCT_REQUESTS_PAGE_MISS to detect rowhammer activities in real-time is worth investigating.

6 CONCLUSION

Motivated by Kim et al. [61] in 2014, a large number of rowhammer attacks have been proposed against applications, kernels and clouds. As a response, software and hardware defenses from both academia and industry are developed for commodity systems. As rowhammer is unlikely to be addressed in the near future, this work can help the community deeply understand rowhammer on commodity systems with production DRAM chips deployed. We also believe new attack vectors and countermeasures (besides Section 5) will be developed to study the causes and impacts of rowhammer in real-world.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This research was supported partly by the SCUT Research Startup Fund No. K3200890. Z. Zhang and D. Chen contribute equally. Y. Gao is the corresponding author.
REFERENCES

[1] Musiker Tadesse Aga, Zelalem Birhanu Aweke, and Todd Austin. 2017. When good protections go bad: Exploiting anti-DoS measures to accelerate Rowhammer attacks. In Hardware Oriented Security and Trust. 8–13.

[2] AMD. Inc. 2005. Secure Virtual Machine Architecture Reference Manual. (Dec. 2005).

[3] Apple. Inc. 2015. About the security content of Mac EFI Security Update 2015-001. https://support.apple.com/en-au/HT204934.

[4] Zelalem Birhanu Aweke, Salseassii Ferede Yithabere, Ruo Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL. Software-based protection against Next-Generation Rowhammer Attacks. In Architectural Support for Programming Languages and Operating Systems. 743–755.

[5] K Bains and J Halbert. 2016. Row hammer monitoring based on stored row hammer threshold value. US Patent App. US15/633,604.

[6] K Bains, J Halbert, C Mozack, T Schoenbohm, and Z Greenfield. 2015. Row Hammer Refresh Command. US Patent App. 14/068,677.

[7] Kuljit S Bains and John B Halbert. 2016. Distributed row hammer tracking. US Patent 9,299,400.

[8] Kuljit S Bains, John B Halbert, Suneeta Sah, and Zvika Greenfield. 2015. Method, apparatus and system for providing a memory refresh. US Patent 9,038,903.

[9] Tanj Bennett, Stefan Saroiu, Alec Wolman, and Lucian Cioocjar. 2021. Panopticoin: A Complete In-DRAM Rowhammer Mitigation. In Workshop on DRAM Security.

[10] Saraam Bhattacharya and Debdeep Mukhopadhyay. 2016. Curious case of rowhammer: flipping secret exponent bits using timing analysis. In Cryptographic Hardware and Embedded Systems. 602–624.

[11] Carsten Bock, Ferdinand Brasser, David Gens, Christoph Liebchen, and Ahmad-Reza Sadeghi. 2019. RIP-RH: Preventing Rowhammer-based Inter-Process Attacks. In Asia Conference on Computer and Communications Security. 561–572.

[12] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup es: machine memory deduplication as an advanced exploitation vector. In IEEE Symposium on Security and Privacy. 987–1004.

[13] Ferdinand Brasser, Lucas Davi, David Gens, Christoph Liebchen, and Ahmad-Reza Sadeghi. 2017. CAN'T Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In USENIX Security Symposium.

[14] Yu Cai, Saugata Ghose, Xun Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. 2016. ANVIL: Software-based Protection against Rowhammer. In USENIX Security Symposium. 1–13.

[15] Yuqiang Cheng, Zhi Zhang, Surya Nehal, and Zhi Wang. 2019. Cattme: Defeating software-only physical key isolation. IEEE Transactions on Dependable and Secure Computing (2019).

[16] 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 ACM SIGSAC Conference on Computer and Communications Security.

[17] Lucian Cioocjar, Jeremie Kim, Mineesh Patel, Lilian Talar, Stefan Saroiu, Alec Wolman, and Onur Mutlu. 2020. Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers. In IEEE Symposium on Security and Privacy.

[18] Lucian Cioocjar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: on the effectiveness of ECC memory against rowhammer attacks. In IEEE Symposium on Security and Privacy. 55–71.

[19] Victor Courtan and Sinivasan Devadas. 2016. Intel SGX Explained. IACR Cryptology ePrint Archive 2016 (2016). 86.

[20] Timothy B. Cowlci, Dean D. Gans, Xuan Zhang, and Kaveh Razavi. 2017. Rowhammer attacks on Intel® Xeon® processor E5-2690 v3 product family. In IEEE High Performance Computer Architecture. 657–668.

[22]$http://www.ieee-esc.org/systems/security/previous-issues/vol2 issue6/2002/2245–261$. 245–261.

[23] Sankhyan Hong, Pietro Frigo, Yi-jian Can Yaya, Cristiano Giuffrida, and Tudor Dumitrash. 2019. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In USENIX Security Symposium. 497–514.

[24] HP, Inc. 2015. HP Moonshot Component Pack. https://support.hpe.com/hpsc/doc/public/display/docId/c04676403.

[27] SK Gautam, SK Manhas, Arvind Kumar, Mahendra Pakala, and Ellie Yieh. 2019. Row hammering mitigation using metal nanowire in saddle fin DRAM. IEEE Transactions on Electron Devices 66, 10 (2019), 4770–4715.

[28] Satendra Kumar Gautam, Arvind Kumar, and Sanjeev Kumar Manhas. 2018. Improvement of row hammering using metal nanoparticles in DRAM: A simulation study. IEEE Electron Device Letters 39, 9 (2018), 1286–1289.

[29] Satendra Kumar Gautam, Sanjeev Kumar Manhas, Arvind Kumar, and Mahendra Pakala. 2020. Mitigating the Passing Word Line Induced Soft Errors in Saddle Fin DRAM. IEEE Transactions on Electron Devices 67, 4 (2020), 1902–1905.

[30] Hector Gomez, Andres Amaya, and Elkim Roa. 2016. DRAM rowhammer attack reduction using dummy cells. In IEEE Nordic Circuits and Systems Conference. 1–4.

[31] Zvika Greenfield, Kuljit S Bains, Theodore Z Schoenbohm, Christopher P Mozak, and John B Halbert. 2015. Row hammer condition monitoring. US Patent 8,938,573.

[32] Zvika Greenfield, John B Halbert, and Kuljit S Bains. 2014. Method, apparatus and system for determining a count of accesses to a row of memory. US Patent App. 13/626,479.

[33] Zvika Greenfield and LEVY Tomer. 2016. Thrrottling support for row-hammer counters. US Patent 9,251,885.

[34] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioi O’Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In IEEE Symposium on Security and Privacy. 245–261.

[35] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks. Bypassing SMAP and kernel ASLR. In ACM SIGSAC Conference on Computer and Communications Security. 368–379.

[36] 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. 308–321.

[37] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Program for testing the DRAM rowhammer problem using eviction. https://github.com/IAIK/rowhammerjs.

[38] Hasan Hassan, Tatyana Can Yura, Sanjeev Kumar Manhas, Arvind Kumar, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In IEEE/ACM International Symposium on Microarchitecture. 1198–1213.

[39] Wei He, Zhi Zhang, Yuezhong Cheng, Wenhuo Wang, Wei Song, Yanzong Gao, Qi-Fei Zhang, Kang Li, Dongxi Liu, and Surya Nehal. 2023. WhitestBlower: A System-level Empirical Study on RowHammer. IEEE Trans. Comput. (2023).

[40] Andrew Hendrick, Edwin Verplanken, Priya Autre, Ramesh Illikka, Chris Gionas, Ronak Singhla, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2680 v3 product family. In IEEE High Performance Computer Architecture. 657–668.

[41] Golake Amin, Sohail Dabbaghian, and Bianca Schroeder. 2012. Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design. ACM SIGPLAN Notices 47, 4 (2012), 111–122.

[42] Intel, Inc. 2011. Intel 64 and IA-32 Architectures Software Developer’s Manual. Combined Volumes: 1, 2A, 2C, 3A, 3B and 3C. (Oct. 2011).

[43] Intel, Inc. 2012. Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide. (March 2012).

[44] Intel, Inc. 2016. Intel 64 and IA-32 Architectures Software Developer’s Manual. 497–514.

[45] Yeongjin Jang, Jaebuyeok Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In Workshop on System Software for Trusted Execution. 1–6.

[46] Patrick Jatke, Victor van der Veen, Pietro Frigo, Stijn Gunther, and Kaveh Razavi. 2022. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In IEEE Symposium on Security and Privacy. 2023.

[47] Jadid ECDC. 2020. DDR3 SDRAM Specification. 560.

[48] Sangwoo Ji, Youngjoon Ko, Sarahy Oh, and Jong Kim. 2019. Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks. In Asia Conference on Computer and Communications Security. 549–560.

[49] Yichen Jiang, Hui Feng, Dean Sullivan, Xiaolong Guo, Xuan Zhang, and Yier Chen, 2021. Quantifying Rowhammer Vulnerability for DRAM Security. In Design Automation Conference.

[50] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maryspiel Eichelseder, Moritz Lipp, and Daniel Gruss. 2023. C3L: Rowhammer-Cryptographic Security and Integrity. In IEEE Symposium on Security and Privacy. 236–252.
[54] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tohab, Daniel Genkin, Andrew Kwong, and Yuval Yarom. 2024. SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism. In USENIX Security Symposium.

[55] Mohammad Nasir Imtiaz Khan and Zhi Zhang. 2017. Analysis of Row hammer attack on sttram. In International Conference on Computer Design. IEEE, 75–82.

[56] Jong-Hyun Kim. 2015. Semiconductor device and operation method thereof. US Patent 9,214,216.

[57] Due-Hyun Kim, Prashant J. Nair, and Moinuddin K. Qureshi. 2014. Architectural support for mitigating row hammering in DRAM memories. IEEE Computer Architecture Letters 14, 1 (2014), 9–12.

[58] Jerome S. Kim, Mïnesh Patel, A. Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. 2020. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In International Symposium on Computer Architecture.

[59] Moosoo Kim, Jungwoo Choi, Hyn Kim, and Hyuk-Jae Lee. 2019. An Effective DRAM Address Remapping for Mitigating Rowhammer Errors. IEEE Trans. Comput. 68, 10 (2019), 1428–1441.

[60] Michael Jaemin Kim, Jeeyoun Park, Jeon-hong Park, Wanjoo Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2022. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In IEEE High Performance Computer Architecture.

[61] Youngku Kim, Ross Daly, Jeremie Kim, Chris Fallin, J Hyee Lee, Donghyuk Lee, Chris Wallkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In International Symposium on Computer Architecture. 361–372.

[62] Andrzej Kogler, Jonas Jüffinger, Cristian Giuffrè, and Kaveh Razavi. 2018. ZEBRAM: comprehensive and compatible software protection against rowhammer attacks. In Operating Systems Design and Implementation. 697–710.

[63] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAM-Bled: Reading Bits in Memory Without Accessing Them. In IEEE Symposium on Security and Privacy.

[64] Leon Lantz. 1996. Soft errors induced by alpha particles. IEEE Transactions on Reliability 45, 2 (1996), 174–179.

[65] Eojin Lee, Ingab Kang, Sukhan Lim, E. Gary Suh, and Jung Ho Ahn. 2019. TWce: preventing row-hammering by exploiting existing time window counters. In International Symposium on Computer Architecture. 385–396.

[66] LENOVO, Inc. 2015. Row Hammer Privilege Escalation Lenovo Security Advisory: LEN-2015-069. https://support.lenovo.com/en_us/product_security/row_hammer.

[67] Shaofeng Li, Xinxi Wang, Minhui Xue, Haejin Zhu, Zhi Zhang, Yansong Gao, Mark Seaborn, and Thomas Dullien. 2015. Exploiting the DRAM rowhammer fault through network requests. In IEEE European Symposium on Security and Privacy. 355–371.

[68] Mortiz Lipp, Andres Kagler, David Oswald, Michael Schwarz, Eamonn Sheehan, and Daniel Gruss. 2020. DoubleHammer: A New Method for Rowhammer Attack. In USENIX Security Symposium. 565–581.

[69] Peter Persil, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In USENIX Security Symposium. 1–18.

[70] Red Hat, Inc. 2017. Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide. https://docs.redhat.com/en-us/redhat-enterprise-linux/7/html-virtualization-tuning-guide/00057495/.

[71] William Ryan and Sha Lin. 2009. Channel codes: classical and modern. Cambridge university press.

[72] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigation of the DRAM rowhammer using a non-temporal instruction. https://www.micron.com/products/dram/ddr4-sdram.

[73] Onur Mutlu and Jeremie S. Kim. 2019. Rowhammer: A retrospective. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 9 (2019), 1555–1571.

[74] Prashant J. Nair, Vilas Sridharan, and Moinuddin K. Qureshi. 2016. XED: Exposing on-die error detection information for strong memory reliability. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture. 341–353.

[75] Red Hat, Inc. [n. d.]. CPU Cloud Computing Solutions from Nvidia. https://www.nvidia.com/en-gb/data-center/gpu-cloud-computing. Accessed Dec 26, 2021.

[76] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlos, and Moinuddin K. Qureshi. 2016. XED: Exposing on-die error detection information for strong memory reliability. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture. 341–353.

[77] Onur Mutlu and Jeremie S. Kim. 2019. Rowhammer: A retrospective. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 9 (2019), 1555–1571.

[78] Prashant J. Nair, Vilas Sridharan, and Moinuddin K. Qureshi. 2016. XED: Exposing on-die error detection information for strong memory reliability. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture. 341–353.

[79] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigation of the DRAM rowhammer using a non-temporal instruction. In USENIX Security Symposium. 623–640.
A HARDWARE DEFENSES OF ROWHAMMER

In this section, we analyze existing hardware-based defenses of rowhammer from both academia and production. As shown in Table 5, we classify existing hardware defenses into multiple categories, which are introduced below.

A.1 Data Integrity Check

This category of defenses uses different algorithms to detect or even correct bit flips by performing data integrity check.

MC-AWARE DRAM-based ECC: As cosmic rays or alpha particles [43, 65] can corrupt data in DRAM chips accidentally, ECC is proposed to address the data-corruption problem by storing additional parity bits in separate chips next to the data bits in the original chips. When data is accessed, ECC calculates its parity and compares it against the stored one. Theoretically, ECC can correct $n \geq 1$ bit flips and detect more than $n$ bit flips. A production-level use case of ECC is single-error-correction-double-error-detection (SECDED) hamming code. In this case, ECC functions are implemented in the MC on commodity processors, that is, the MC has an extra chip to store ECC code. This MC-Aware DRAM ECC was believed to deter rowhammer-based attacks until ECSPloit [18] in 2019 demonstrated that it was susceptible to rowhammer by inducing enough (e.g., 3) bit flips in a single code word (e.g., 72 bits).

To reinforce the MC-Aware DRAM-based ECC against rowhammer, a couple of new defenses [59, 113] have been proposed. Particularly, they use row-remapping schemes to distribute bit-flips to different rows, which significantly reduces the occurrences where undetectable multiple bit-flip errors occur in a single word. Besides, cryptographic Lightweight Message Authentication Codes (MAC) are stored in the ECC memory or MC to support detecting and correcting rowhammer-induced bit flips [23, 53, 98]. Further, the design principle of future ECC is expected to put more weight on its detection capability rather than its correction capability [89], thus achieving a better detection of rowhammer.
On-die ECC: As introduced in Section 4.2, on-die ECC, is implemented in the DRAM chips and invisible to the MC [79, 85]. It is adopted by mainstream DRAM manufacturers to reduce the data-corruption rates in recent DRAM chips (e.g., LPDDR4 [81] and DDR5 [50]). While the on-die ECC functions are publicly unavailable, they have been reverse-engineered [84–86].

Dynamic Skewed Hash Tree: Vig et al. [110] propose a lightweight scheme within the MC to check data integrity. Particularly, they apply a sliding window protocol to identify potentially corrupted victim rows and use a dynamic integrity tree structure with SHA-3 Keccak hash functions for bit-flip detection.

 Dummy Cells: Gomez et al. [30] propose the so-called dummy cells to perform data integrity checks. Compared to regular cells, the dummy cells are intrinsically more susceptible to rowhammer as they have smaller capacitance and larger transistors. Every row will contain a dummy cell that is fully charged during the DRAM refresh time. If the MC reads out a row with its dummy cell value bit-flipped, it indicates an early warning of data loss and thus the MC can refresh it and its adjacent rows.

A.2 Row Activation Count

In this category, defenses rely on hardware counters to count the number of activations per row within a DRAM refresh period. If the count for a row exceeds a predefined threshold, it will refresh its adjacent rows, or throttle further activations to the row, or swap the row with another random row. We discuss the defenses based on where the counters are implemented in the DRAM hierarchy.

Row-Level Count with One Counter per Row: CRA [57] maintains a counter per row in DRAM, and updates the counter when the row is activated. To reduce the access latency to each counter, it caches recently updated counters within the MC. Similar to CRA, Panopticon [9], a recent scheme in 2021, also maintains a counter in DRAM for each row. Unlike CRA, Panopticon rearranges DRAM’s subarrays’ layout in a staggered way, and thus a counter can be updated alongside a row activation.

Row-Level Count with One Counting Logic per Bank: Besides the on-die Target Row Refresh (TRR) [75], some academic works are also in this category. In particular, TWiCe [66] manages a counter table for each DRAM bank in the register clock driver (RCD), which is inside a DIMM but separated from chips. An entry in the counter table has an activation counter. When a row is being activated, TWiCe increments its activation counter if it exists in the table, otherwise allocates a spare entry. If the counter reaches a predefined threshold, TWiCe introduces a new command called Adjacent Row Refresh to refresh the adjacent victim rows. Graphene [83], implemented in the MC, also maintains a counter table for each DRAM bank. It leverages the Misra-Gries algorithm to identify a set of frequently accessed rows and introduces new nearby row refresh command to refresh the row’s adjacent rows. Similarly, Mithril [60] leverages a newly introduced DRAM command called refresh management (RFM) in DDR5 [50] to implement a customized Misra-Gries algorithm. ProTRR [73] proposes an in-DRAM Misra-Gries algorithm to optimize TRR.

Instead of refreshing the potential victim rows in advance, BlockHammer [121] proactively throttles memory accesses that are considered malicious. Implemented in the MC, BlockHammer contains two components: RowBlocker and AttackThrottler. RowBlocker employs two counting bloom filters per bank to alteratively track the activation rates of all DRAM rows in a rolling time window. If a row’s activation rate exceeds a predefined threshold, RowBlocker will blacklist this row to limit its further activations. Based on the results from RowBlocker, AttackThrottler cuts the memory bandwidth width provided to threads that are identified as potential attackers.

Table 5: A taxonomy of hardware defenses. (Whether on-die ECC is effective against rowhammer has not been studied.)

| Objective                          | Hardware Defense                     | Prototype Location |
|------------------------------------|--------------------------------------|--------------------|
| Data Integrity Check               | Correct/Detect bit flips             | DRAM and MC        |
|                                    | CM [33], PT-Guard [98]               | MC                 |
|                                    | On-die ECC [79] (Future Exploration) | DRAM               |
| Detect bit flips                   | Dynamic Skewed Hash Tree [110]       | MC                 |
|                                    | Dummy Cells [30]                     | DRAM               |
| Row Activation Count               | Refresh specific rows                | DRAM               |
|                                    | CRA [57]                             | DRAM and MC        |
|                                    | Corm [100], Graphene [83], Bains et al. [7], Fisch et al. [24], Kim [56] | DRAM and/or MC |
|                                    | Cowles et al. [20], Greenfield et al. [31] | DRAM or MC |
|                                    | CR-Aware ECC [95], SafeGuard [23]    | MC                 |
|                                    | PARA [61], ProHT [102], MRloc [125] | DRAM and MC        |
|                                    | LightRoAD [103]                     | DRAM and/or MC     |
|                                    | BlockHammer [121], Greenfield et al. [33] | DRAM or MC |
|                                    | Swap aggressor rows with random rows  | DRAM and MC        |
|                                    | Sailseshwar et al. [97], Woo et al. [117], SHADOW [115] | DRAM and MC |
|                                    | ProRT [73]                           | DRAM and MC        |
|                                    | Reduce acceptor-type traps           | DRAM               |
|                                    | Ryu et al. [96]                      | DRAM               |

451
exceeds a predefined threshold, either its victim rows will be refreshed [5–8, 24, 31, 32], or subsequent ACT commands sent to the row will be throttled [33].

Besides refresh- and throttling- based defenses, Saisleshwar et al. [97] propose to swap aggressor rows and other random rows when row activation count reaches a predefined threshold. By doing so, they can break the physical proximity between the aggressor row and its original victim rows, thus mitigating rowhammer, so-called row shuffle. Further, Woo et al. [117] and SHADOW [115] respectively enhance the security and performance of the row-shuffle defense.

RowSet-Level Count: CAT [100] applies one counter to a set of rows and every access to rows within the set will be counted. As benign workloads have memory access locality and some rows ("hot") are accessed more often than others ("cold"), CAT leverages adaptive trees to dynamically update the number of rows one counter manages. By doing so, more counters are allocated for a "hot" memory region and less counters are for a "cold" memory region. Silver Bullet [22, 120] assigns a counter to a subbank (i.e., a rowset), size of which can be tuned from two times the blast radius to the row number of a bank. A subbank size by Silver Bullet is decided when designing a DRAM chip and cannot change thereafter, which is different from CAT. Every time a counter reaches a predefined value, Silver Bullet will refresh one row in the corresponding subbank and clear the counter value to zero. The rows in a subbank are refreshed one by one in a round-robin way. Kim [56] assigns a counter to each DRAM mat (e.g., a subarray), and rows in a mat are refreshed when its counter reaches a predetermined value. Cowles et al. [20] assign a counter to a bank (a rowset here thus refers to all the rows in a bank) and the counter counts all row activation sent to the bank. When the counter reaches a predefined value, an RFM command will be issued to refresh this bank and the counter value will be decremented accordingly. Thus, Cowles et al. apply a counting granularity of a bank, which is different from the previous solutions that count at the row-level using one counting logic per bank where the counting granularity is a single row in a bank.

A.3 Row-Activation-Triggered Probabilistic Refresh

In contrast to the previous category that relies on the row activation count to refresh adjacent rows deterministically, this category refreshes rows with a fixed or adjustable probability, indicating that they cannot enforce a rowhammer-free security invariant.

Fixed Refreshing Probability: PARA [61] is one of the earliest probabilistic solutions to rowhammer. At every memory access, it decides whether to perform a refresh with a low probability \( P \) (e.g., 0.001). If yes, it refreshes one of two rows adjacent to the activated row, and either adjacent row is selected with equal probability. To achieve low area overhead, only a simple and compact probability generation circuit is needed to implement PARA. To enhance security, \( P \) can be tuned to a relatively higher value, which, however, causes more refreshes as well as energy consumption.

Adjustable Refreshing Probability: As PARA does not consider memory-access history, it does not perform well on most benchmarks. As such, ProHIT [102] uses two tables labeled “hot” and “cold” respectively to track the access history. ProHIT manages the tables in a probabilistic manner and only refreshes the row recorded in the highest position in the hot table at every regular refresh command (issued every 7.8 \( \mu s \)). MRLoc [125] improves the reliability of PARA while reducing the additional refreshes from ProHIT. Similar to ProHIT, MRLoc leverages memory-access locality by implementing a queue to track the memory-access history. Unlike ProHIT, MRLoc leverages the queue to dynamically decide a refresh probability at every memory access.

A.4 Cache Miss/Flush and DMA Access Count

Unlike the software detection in Section 4.1.3, LightRoAD [103] is a hardware scheme to detect abnormal memory-access behaviors. It resides in the MC and assigns three counters to count cache misses, cache flushes and DMA accesses, respectively. When the sum of the three counters reaches a predefined threshold, an alarm is raised. If the DMA-access count is the major cause of the alarm, the DMA access can be disabled for a while. If it is due to the cache, targeted victim data can be reallocated in the memory.

A.5 Fabrication Process Improvement

As the root causes of rowhammer are at the circuit-level, this category of solutions proposes new fabrication techniques [27–29, 96, 122] to mitigate rowhammer.

Mitigating electromagnetic coupling: One root cause is the electromagnetic coupling where the voltage fluctuation of a toggled aggressor wordline injects electromagnetic noise into its adjacent victim wordline and the noise can partially turn on the access transistor of victim cells, exacerbating the charge leakage of the capacitors in victim cells [93]. To reduce the leakage as much as possible, Yang et al. [122] use additional phosphorus implantation between two adjacent wordlines and junction depth optimization to form a localized shield from the electric field. Alternatively, Gautam et al. [28] introduce a low work function metal nanoparticles (MNPs) at the gate metal/gate oxide interface. To make it applicable on a DRAM structure of 3-D saddle fin recessed channel access transistor (S-RCAT), Gautam et al. [27] replace the MNPs with metal nanowire. On top of that, they investigate the zero-failure in S-RCAT, resulting from hammering the passing word line (PWL) [29]. To this end, they propose a high work function PWL to suppress the impact of electric field interaction between the PWL and the charge storage node, which thus significantly mitigates the zero-failure.

Reducing acceptor-type traps: The other root cause is the acceptor-type trap [123]. Specifically, in a trap-assisted charge pumping process, a trap located near the gate of an aggressor wordline captures charges and then emits them. Thus, the emitted charges will migrate to adjacent victim wordline and cause a potential drop, leaking the charge in the capacitors of victim cells. As a single trap may exacerbate the charge leak by a factor of 60 in a 2y-nm node, it will eventually cause bit flips if this process repeats. To mitigate this problem, Ryu et al. [96] propose a silicon migration technique of hydrogen annealing.