Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks

Andrea Di Dio
Vrije Universiteit Amsterdam
a.di.dio@vu.nl

Koen Koning
Intel*
koen.koning@intel.com

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

***Abstract*—Despite nearly decade-long mitigation efforts in academia and industry, the community is yet to find a practical solution to the Rowhammer vulnerability. Comprehensive software mitigations require complex changes to commodity systems, yielding significant run-time overhead and deterring practical adoption. Hardware mitigations, on the other hand, have generally grown more robust and efficient, but are difficult to deploy on commodity systems. Until recently, ECC memory implemented by the memory controller on server platforms seemed to provide the best of both worlds: use hardware features already on commodity systems to efficiently turn Rowhammer into a denial-of-service attack vector. Unfortunately, researchers have recently shown that attackers can perform one-bit-at-a-time memory templating and mount ECC-aware Rowhammer attacks.**

In this paper, we reconsider ECC memory as an avenue for Rowhammer mitigations and show that not all hope is lost. In particular, we show that it is feasible to devise a software-based design to both efficiently and effectively harden commodity ECC memory against ECC-aware Rowhammer attacks. To support this claim, we present Copy-on-Flip (CoF), an ECC-based software mitigation which uses a combination of memory *migration* and *offlining* to stop Rowhammer attacks on commodity server systems in a practical way. The key idea is to let the operating system interpose on all the error correction events and offline the vulnerable victim page as soon as the attacker has successfully templated a sufficient number of bit flips—while transparently migrating the victim data to a new page. We present a CoF prototype on Linux, where we also show it is feasible to operate simple memory management changes to support migration for the relevant user and kernel memory pages. Our evaluation shows CoF incurs low performance and memory overhead, while significantly reducing the Rowhammer attack surface. On typical benchmarks such as SPEC CPU2017 and Google Chrome, CoF reports a <1.5% overhead, and, on extreme I/O-intensive scenarios (saturated nginx), up to ∼11%.

I. Introduction

Since its discovery in 2014, Rowhammer [25] has evolved from a mere ‘side effect’ caused by ever-increasing DRAM density to a real security vulnerability which can be exploited on a wide variety of systems. Unfortunately, existing defenses have proven to be impractical (e.g., requiring complex hardware or software changes) and/or ineffective (e.g., only stopping specific Rowhammer variants). Research has shown that Rowhammer is a threat even on Error-Correcting Code (ECC) memory implemented by the memory controller [10], which is heavily deployed on server platforms. In this paper, we show ECC memory is not a lost cause in the fight against Rowhammer. To this end, we present Copy-on-Flip (CoF), a practical software mitigation which hardens ECC memory against Rowhammer. Our design takes advantage of ECC's ability to detect vulnerable pages, allowing our mitigation to render these unavailable to the attacker with a combination of memory *migration* and *offlining*[1].

Rowhammer. The research community has devised an abundance of Rowhammer exploits which compromise a wide array of different systems, including native environments on both x86 [16], [17], [22], [48], [50] and ARM [15], [57], virtualized environments [46], browser sandboxes via JavaScript [8], [13], [18] and networked systems [33], [51]. In addition, research has shown that the Rowhammer vulnerability can also be used as a powerful side channel for information leakage [30].

Early on, it was believed that Rowhammer attacks could be mitigated by deploying Error-Correcting Code (ECC) memory [25]. Typical ECC memory follows the SECDED scheme: *single error correction, double error detection*. Thus, it is able to correct single-bit errors per code word, by adding control bits to each chunk of data bits. This significantly hinders the effect Rowhammer can have, since bit flips are often automatically fixed. In the worst case, two or more bit flips in a word crash the system with high probability, effectively turning Rowhammer into a mere denial-of-service (DoS) attack vector. However, research by Cojocar et al. [10] shows that exploiting Rowhammer deterministically on ECC memory *is* attainable, by carefully combining three or more specific bit flips in the same word, effectively bypassing the ECC function.

The most critical step for an attacker to exploit Rowhammer on ECC-capable memory is the so-called *templating* phase, in which an attacker has to find vulnerable memory areas which are susceptible to bit flips at the desired offsets. In ECCploit-like attacks [10], the attacker has to be very careful to inject only single-bit errors in a code word in any given

*The work on this submission has been performed while Koen Koning was employed at the Vrije Universiteit Amsterdam.

[1] We use the term memory (or page) offlining as used in the Linux kernel source and by hardware vendors to refer to those pages which are poisoned and rendered unavailable to the system [19], [26].

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24337
www.ndss-symposium.org

https://t.me/learningnets

templating attempt (i.e., correctable errors), in order to avoid system crashes due to detectable but uncorrectable errors.

While many other mitigations beyond ECC have been proposed by both industry and academia, these all fall short in terms of comprehensiveness and practicality. Most of the mitigations have either been hardware-based [5], [23], [37], [44], [47], [61] *or* software-based [6], [21], [29], [43], [58], [63], with only a few proposals of hybrid defenses which involve cooperation between hardware and software components [3], [9]. Both approaches have pitfalls which make them ineffective at defending from Rowhammer attacks in realistic settings. New hardware-only defenses are impossible to deploy on commodity systems and are associated with higher costs to implement tracking algorithms. On top of that, hardware-based defenses have also been shown to be vulnerable to targeted attacks, with fixes unavailable for years to come [13], [16], [22]. In contrast, while software-only defenses are easier to deploy, comprehensive ones [29] require complex software changes and incur high performance/memory overhead.

Copy-on-Flip. In this paper, our key observation is that, while ECC by itself is insufficient to *protect* against Rowhammer attacks, it may still serve as an advance warning system to *detect* that an attack is in progress and trigger strong mitigating action in software to stop it in its tracks. In particular, with Copy-on-Flip (CoF), we augment the operating system (OS) kernel to detect templating attempts causing (correctable) ECC errors and safeguard the data in that area by *migrating* it out of harm's way before the attacker has caused a potentially dangerous number of corrections in an ECC code word. To prevent attackers from reusing partially templated memory post migration, we also immediately *offline* the vulnerable areas of memory at migration time.

Our defense prevents templating of ECC memory, thereby removing the ability to mount a targeted attack. In particular, CoF fully restores the security guarantees that ECC memory was assumed to have prior to ECCploit [10] (but did not) and reduces the Rowhammer threat to that of a DoS attack at worst.

The key challenge to implement our design is the need to support migration for memory that may be used by the attacker for templating purposes. This includes user memory (the common target in existing exploits), but also kernel memory such as slabs and page tables that may be allocated by the kernel on the attacker's behalf (a target in more advanced exploits). Unfortunately, while user memory is relatively easy to migrate, kernel memory is normally not migratable on commodity systems. In particular, the memory management of the kernel itself typically relies on physical addressing, so memory cannot be transparently remapped through virtual address translation. To address this challenge, we propose a simple design based on removing the kernel's strict dependency on physical addressing. We show our design is practical with a Linux-based CoF prototype of only around 600 LoC. While simple and amenable to further extensions, we show our design can support migration of all the relevant user and kernel pages, resulting in a quantifiable attack surface reduction of over 95%. Moreover, CoF incurs almost negligible runtime overhead for most programs (0.2% geomean for SPEC CPU2017 and 1.1% for browsers), with a worst case of $\sim$11% overhead for I/O intensive workloads on saturated systems.

Contributions. To summarize, our contributions are:

1) The design and open source implementation[2] of Copy-on-Flip (CoF), a new software mitigation to harden ECC memory and provide low-overhead protection from Rowhammer attacks.
2) An analysis of the spatial distribution of Rowhammer induced bit flips using state-of-the-art Rowhammer fuzzer data [22]. This analysis helped shape our design and determine its memory overhead guarantees, but the results are broadly applicable to other mitigations.
3) An evaluation of our CoF prototype, showing its practicality and comprehensiveness, running a variety of workloads including web browsers, showing little to no runtime overhead.

II. Background

In this section we describe several concepts crucial to understanding the remainder of the paper: the Rowhammer vulnerability, and how to apply it on ECC implemented by the memory controller.

The Rowhammer vulnerability allows attackers to flip a bit in a row of memory that they do not own by repeatedly activating ("hammering") one or more neighboring rows [25]. Attackers do not know in advance which, if any, bit will flip in the victim row, but once they find a bit that flips, they can flip the same bit again with high probability. Besides the number of activations, the occurrence of bit flips also depends on the data (if the cells in the neighboring row contain the same value, bit flips are unlikely), and on the specific activation patterns.

To enforce the desired activation patterns, an attacker needs some knowledge of memory addressing. From the CPU perspective, only a single *physical address space* is visible, abstracting away the underlying DRAM geometry. On top of this, software only sees its own *virtual address space*, where virtual addresses are dynamically mapped to physical *page frames* (e.g., 4 KB) by the operating system. While these levels of abstraction complicate Rowhammer attacks, an attacker can massage the memory mappings in such a way that the underlying physical memory is hammerable via virtual memory. While the suitable activation patterns to trigger Rowhammer bit flips depend on the target system and the memory modules used, even the latest memory modules have been shown to be vulnerable [13], [16], [17], [22]. In general, modern Rowhammer attacks (especially ECC-aware ones) consist of three steps:

1) *Memory templating:* Find vulnerable locations (where bits flip) in memory allocated by the attacker.
2) *Memory massaging:* Manipulate the memory layout so that sensitive data is now stored at a vulnerable location.
3) *Hammering:* Perform targeted, rapid, and uncached memory accesses to flip a bit in the sensitive data.

Existing defenses are unable to prevent Rowhammer attacks in today's memory chips, are expensive, or result in false positives. In particular, the TRR Rowhammer defense [23],

[2]https://github.com/vusec/Copy-on-Flip

https://t.me/learningnets

[32], [37] deployed in modern hardware is easily bypassable [13], [16], [22], while most of the software defenses assume detailed knowledge of DRAM geometry and either limit protection to specific memory areas [6], [9], [58] or convert half of the memory to (slow) swap space [29]. Defenses based on anomaly detection [3] not only depend on performance counters that are not always available, but, with increasing DRAM density, also suffer from many false positives.

In this paper, we are especially concerned with memory in high-value servers which is commonly protected against accidental bit flips through Error Correcting Code (ECC). With the typical *single error correction, double error detection* (SECDED) scheme, when a single bit flip occurs in an ECC word (typically 64 bits), it will be corrected automatically. If there are two bit flips, SECDED implementations can no longer correct them, but they will still detect that there is a problem and trigger a crash. Only the occurrence of three bit flips may go undetected.

Originally designed to handle accidental bit flips in non-adversarial conditions, ECC memory was long believed to offer an adequate defense against Rowhammer as well. After all, to an attacker a single bit is no good as it will be automatically corrected, a double bit flip is no good as it will lead to a crash, and trying to find an ECC word with three flippable bits without encountering one with two bit flips first is highly unlikely—in principle reducing a memory corruption vulnerability to a denial-of-service issue [25], [40].

However, Cojocar et al. [10] show that exploiting Rowhammer on ECC memory, while harder, is still possible. By using data patterns that ensure that at most a single (specific) bit can flip in an ECC word, in combination with a timing side channel that reveals that a single bit error has been corrected, they probe all the bits in an ECC word during the templating phase until they find a word where three bits are flippable. In the final attack, they then flip all three bits at once.

III. Threat Model

This paper considers an attacker able to launch an advanced Rowhammer exploit such as ECCploit on ECC memory [10]. In particular, we assume an attacker with detailed knowledge of the ECC function and the ability to trigger undetectable bit flips using ECCploit-like attacks. We further assume that the target system is equipped with orthogonal mitigations needed to address other classes of vulnerabilities and that the memory controller reports ECC correctable errors to the operating system correctly [10]. The goal is to stop exploitation (i.e., memory corruption) via Rowhammer on ECC memory by preventing templating of the memory. We do not aim to stop attackers from mounting denial-of-service attacks, as they are always possible (e.g., by blindly hammering ECC memory to cause a system crash), but generally considered less serious.

IV. Workflow

The core idea of Copy-on-Flip (CoF) is to use ECC bit error corrections as an early warning of templating and mitigate the attack by moving the victim page out of harm's way. Figure 1 shows the end-to-end workflow of our design. In particular, on a system equipped with ECC memory, the memory controller

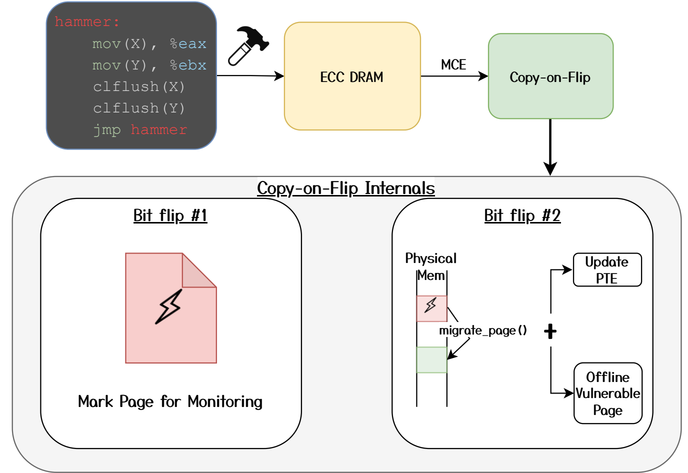

Fig. 1: High-level workflow of Copy-on-Flip.

informs the OS via a machine check exception (MCE) whenever a bit flip occurs. As previously observed by Cojocar et al. [10], this MCE is raised synchronously, meaning that error detection leads to immediate execution of the MCE handler in the OS kernel upon access. As detailed later, CoF responds by performing different actions depending on whether the number of observed bit flips in the offending page points to a successful templating attempt, specifically:

1) *First bit flip.* If this is the first bit flip reported for this page, CoF marks it as potentially under attack.
2) *Second bit flip.* On the second bit flip on the same page, CoF pauses all user threads and transparently migrates the data from the vulnerable page to a new physical page via virtual address translation. In addition, it offlines the vulnerable page to prevent further templating/exploitation attempts. Finally, it resumes the previously paused user threads.

Since the page migration/offlining occurs before the third bit flip, the attacker is no longer able to obtain a fully templated page for ECC-aware exploitation [10]. Blindly hammering memory without a template is still possible, but is almost guaranteed to result in a detectable error (and a system crash).

V. Copy-on-Flip

To enable the hardening of ECC memory pages against templating, CoF features several components, as shown in Figure 2. First, the *templating detector* uses information from the memory controller to detect successful attempts at templating areas of memory through Rowhammer. This will, in turn, request the *page protector* to render the victim page unusable. In response, this component must be able to transparently *migrate* victim data to a new page as necessary and *offline* the original victim page as soon as the page is found vulnerable to ECC-aware Rowhammer. To this end, this component supports page migration and offlining for all the memory pages an attacker could target for templating. This includes user and kernel pages allocated by the kernel on the user's (or attacker's)

https://t.me/learningnets

behalf, such as page cache, slab, stack, and page table pages—as detailed later. In the remainder of this section, we introduce a general design based on process/memory management facilities available on modern operating systems such as Linux, FreeBSD, etc. Later (Section VI), we discuss implementation details of our current CoF prototype targeting Linux.

A. Templating Detector

To detect templating, we require knowledge of every bit flip that occurs in the system, and its location. Luckily, the memory controller reports this information to the OS for every ECC error in the form of a machine check exception (MCE), including bit flips that were corrected (i.e., single-bit errors). Modern operating systems store these error reports in system logs for later re-evaluation. On Linux, *reporting* is handled by the error detection and correction (EDAC) kernel module. Similar reporting capabilities are available in other commodity operating systems such as FreeBSD [4]. While the exact information that is reported differs per memory controller, most (if not all) report the page frame number (PFN) of the physical page the error occurred in.

By hooking into the existing error reporting code of the OS kernel, CoF can thus monitor all correctable bit flips that occur. Large-scale analysis conducted by Meza et al. [36] on Facebook data centers has shown that, while accidental correctable memory errors are widespread, they depend on the workload and their prevalence is decreasing with time. Furthermore, recent research [34] shows that, while commodity workloads *can* exhibit memory access patterns that induce bit flips, this only happens in very specific circumstances. This means that (correctable) bit flips can serve as a robust mechanism to detect ECC-aware Rowhammer templating. Moreover, even in the rare case of multiple accidental bit flips that happen to match those of successful templating attempts, CoF would simply trigger migration and offlining of the offending page with no other consequences for the running system.

Identifying vulnerable memory. The templating detector requires keeping track of every bit flip in the system. In principle, we could consider an area of memory vulnerable only when *three* bit flips are observed in the same *ECC word* (typically, 64 bits)—since, again, an attacker needs to reliably observe at least three individual bit flips in a word to bypass ECC in common SECDED schemes [10]. Nonetheless, to rule out relatively low-entropy probabilistic Rowhammer attacks (with attackers having observed only two bit flips and attempting to guess the location of the third bit flip), CoF considers an area of memory vulnerable after observing only *two* bit flips in the same word. As we show in Section VII, this conservative strategy still leads to a low-overhead solution.

Tracking granularity. The second design decision we need to make concerns the tracking granularity. The most natural choice is to store metadata on observed bit flips on a per ECC word basis. However, tracking information at this granularity could be impractical: even on systems with relatively fewer flips, some flips are bound to happen throughout the tens to hundreds of gigabytes of DRAM, complicating metadata management and likely imposing performance/memory overhead tradeoffs. Another problem is that the report from the memory controller about bit flips might not include such detailed information: while the page frame number is typically included, the offset within the page is often not.

For these reasons, CoF instead tracks bit flips at the *page granularity*. This choice supports common memory controller implementations and drastically simplifies metadata management. Specifically, on the first bit flip observed anywhere in a page, CoF sets a flag in the page metadata (e.g., `struct page` on Linux). When a second bit flip occurs anywhere on the page, CoF conservatively considers the page templated and thus vulnerable. The advantage of this policy is that there is no additional overhead due to tracking. On the other hand, we may over-approximate the number of vulnerable pages, since the two bit flips may have happened in two different ECC words, and are thus not exploitable. However, as we will show in Section VII, the impact of this over-approximation is minimal even on "flippy" DIMMs that exhibit many bit flips.

B. Page Protector

In order to prevent templating, CoF offlines pages that are flagged by the templating detector. However, simply offlining a page would significantly disrupt the operation of the system. On the other hand, individually requiring every application and kernel subsystem to support memory offlining is infeasible. For this reason, CoF supports *transparent* memory offlining by means of page migration (i.e., migrating the victim data to a new page before offlining the original vulnerable one). The page protector facilitates this process for all potential templating targets in the system, in most cases by leveraging virtual memory remapping. To this end, as we will see, the key challenge is to support *migratable kernel memory* for a comprehensive mitigation.

Templating targets. In order to ensure safety for the system, CoF must classify and protect all memory an attacker can attempt to flip for templating purposes. In existing exploits, the typical target is user memory. However, for a comprehensive mitigation, CoF also considers other possible targets in kernel memory. Nonetheless, since kernel memory is normally not migratable in commodity operating systems, a practical design needs to carefully pick the kernel memory targets of interest to minimize changes. As such, we consider the key requirements an attacker needs to fulfill for any given templating target:

- **R1** The attacker must be able to allocate the target memory at will in order to template different pages in physical memory.
- **R2** The attacker must be able to allocate aggressor locations at a predictable physical distance from the target memory in order to implement the desired Rowhammer patterns.
- **R3** The attacker must be able to control/predict the victim data in the target memory in order to reliably perform one-bit-at-a-time ECC-aware templating.
- **R4** The attacker must be able to time the accesses to the target memory in order to infer if ECC corrected a bit flip.

We observe that all the requirements are trivially satisfied by user memory, but harder to satisfy for kernel memory.

https://t.me/learningnets

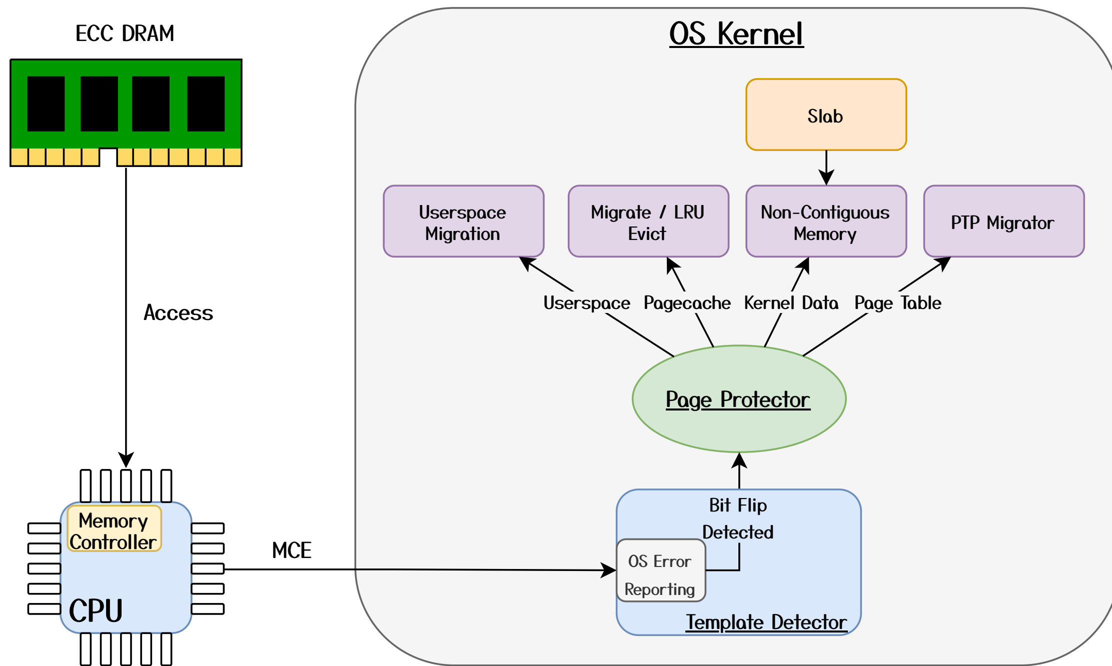

Fig. 2: Components of Copy-on-Flip.

Nonetheless, an advanced attacker can satisfy such requirements for many kernel targets. Let us consider page table pages as an example. The attacker can allocate page table pages at will by simply allocating user memory (**R1**), massage the page frame allocator to conveniently co-locate user pages with page table pages (**R2**), leak the content of page table entries using orthogonal hardware vulnerabilities such as MDS [59] (**R3**), and finally time controlled page table walks to check for bit flips (**R4**).

The next question is how to generalize this analysis to arbitrary kernel memory targets. While we could in principle focus on any of the requirements above, not all of them lead to a clear way to identify the attack surface. For instance, for **R2**, should we include only targets that are managed by allocators prone to user-controlled massaging? Where do we draw the line? For **R3**, should we include all the possible targets an attacker might potentially leak data from (and wildly overapproximate) or only all the possible targets that contain user data (and wildly underapproximate)? Finally, for **R4**, should we include all the targets that can be accessed and timed through user-controlled events (page table walks, syscalls, page faults, etc.)? Again, it is difficult to draw a clear line. In short, our analysis shows most of the requirements lead to a fuzzy definition of attack surface. Luckily, **R1** is of exception, given that kernel memory that is allocated on the user's behalf is already carefully defined and accounted for by the kernel. As such, CoF solely focuses on **R1** and user-accounted memory to identify the relevant targets of interest.

User-accounted pages. To identify user-accounted memory pages (and our targets), we turn our attention to the memory accounting infrastructure implemented by the kernel to manage containers [41], [54]. Intuitively, such infrastructure needs to fully capture all types of memory that can be allocated on the user's behalf so that a malicious or buggy program part of a container can be prevented from repeatedly allocating excessive memory (i.e., using limits and out-of-memory killing). In Section VII-C, we validate this qualitative intuition with quantitative measurements that confirm our choice of templating targets drastically reduces the attack surface.

On Linux, such infrastructure is part of the *memory resource controller* (or *memcg*) [54]—although similar controllers are available on other OSes such as FreeBSD [41]. As expected, memcg performs accounting of *user* memory, but also of a number of kernel memory types allocated on the user's behalf: *page cache* (including file-backed pages, swap, shared memory, etc.), *vmalloc*, *slab*, *kernel stack*, and *page table* memory [54]. In the next section, we discuss how CoF can protect all these memory types by migrating data in memory before offlining (discussed in the next subsection). To minimize kernel changes, our design attempts to reuse existing migration and offlining mechanisms whenever possible, and piggyback on well-known design principles such as virtual memory-based indirection and composable allocators otherwise.

Page offlining. The naive approach to page offlining is to deallocate a vulnerable page post migration. Unfortunately, this strategy is susceptible to trivial memory reuse attacks. Indeed, previous research has demonstrated that the allocation patterns of kernel memory allocators are very predictable and vulnerable to physical memory massaging [57]. Thus, an attacker can easily force the kernel to reuse a vulnerable page through memory massaging and then resume templating after an offlining event. An option is to rely on randomized allocation patterns to deter reuse attacks [43], but it is difficult to apply this strategy to arbitrary kernel memory without exceedingly reducing the entropy. As a result, CoF instead isolates the vulnerable page and simply marks it as *poisoned*,

https://t.me/learningnets

Type of Memory	Migration Strategy
User Pages	Migrate Page and Update PTE
Mapped Page Cache	Migrate Page and Update PTE
Unmapped Page Cache	Evict from Page Cache
Non-contig. Mem (NCM)	Add Migration Support
Slab Pages	Back with NCM
Kernel Stacks	Back with NCM
Page Table Pages	Update Physical References

TABLE I: Page Migration in CoF for different memory types

effectively leaving the page permanently unusable. As shown in Section VII, offlining events (due to two bit flips observed in a page) are rare and this simple strategy has low impact on the memory footprint in practice.

C. Comprehensive Page Migration

We now highlight how CoF handles the migration for the different (attacker-controlled) memory types. A summary of the page migration techniques is included in Table I.

User pages. Typically, user memory is oblivious to the underlying physical memory that backs its virtual address space. Since in non-uniform memory architecture (NUMA) systems each processor has its own DRAM (to which accesses are much more performant), most commodity operating systems implement *page migration* of (only) user memory, to relocate physical memory transparently [1]. Migration of pages is also required to support other memory management features such as page deduplication (available on Linux [2], Windows [8], etc.). Page migration is done by allocating a new physical page (on the desired node), copying over the data from the old page, and updating the page tables of the process to point to the new physical page. CoF can reuse this mechanism to move data on a vulnerable page: the data is migrated to (any) new page by the kernel, after which the old page is offlined.

Page cache. Data loaded from disk is kept in memory even after a process might be finished with it. This cache is called the *page cache* and makes up a large part of the memory used by the kernel. While a page cache page is mapped to one or more user processes, it is "promoted" to user memory and can thus be easily migrated as explained earlier. However, when a page is no longer mapped and is left in the page cache for future use, it can no longer be easily migrated. Luckily, since the page no longer has any users, CoF can simply evict the page from the cache before offlining, without the need for additional migration mechanisms.

Non-contiguous memory. To combat the effects of external fragmentation, modern OS kernels provide overlay allocators on top of the (page) frame allocator to allocate memory which is virtually, but not physically, contiguous. The frame allocator allocates physical memory areas and gives access to these through the *direct map*—a reserved region in the kernel address space that provides a one-to-one mapping of virtual to physical memory. Non-contiguous allocators (e.g., `vmalloc` for Linux [11], `kmem_malloc` for FreeBSD [52] or memory allocated from the *paged pool* on Windows [39]) rely on the underlying frame allocator to allocate individual page frames and support large virtually-contiguous allocations to store arbitrary kernel data. They work similarly to the user-level `mmap`: a set of random physical pages is allocated and mapped into the (kernel) virtual address space.

In CoF, we enable migration for the pages allocated by these non-contiguous allocators similarly to user page migration. In other words, CoF takes ownership of the virtual-to-physical mappings for these pages and updates the kernel page tables to redirect the vulnerable virtual mapping to a new physical page. Not only does this simple extension support migration for all the pages allocated by these allocators, but also provides a building block we can reuse to migrate other kernel memory types.

Slab. The *slab* allocator is another key overlay allocator composing on top of the frame allocator to reduce internal fragmentation. Originally proposed in SunOS [7], slab allocators are now implemented by all the major operating systems, including FreeBSD [14] and Linux [31]. Linux even supports several slab implementations [31]—SLUB being the default. The slab allocator can place multiple objects in a slab cache (allocated by the frame allocator), offers high performance, and is also often used as a backing allocator for higher-level allocators (e.g., Linux' `kmalloc`). This allocator is used to manage all the frequently accessed small objects in the kernel, including many (e.g., network) buffers storing user data. To enable migration of slab pages, CoF changes the target slab allocator to use the non-contiguous allocator instead of the frame allocator to allocate its slab caches. The result is that slab pages are now backed by *migratable* virtual mappings rather than direct physical memory mappings.

Kernel stacks. The kernel allocates a dedicated kernel stack for each user thread on the system. Hence, much like the memory types discussed earlier, the attacker can repeatedly allocate kernel stack memory at will (i.e., by spawning additional threads). Kernel stack pages are normally allocated by a predetermined backing allocator, typically a slab or frame allocator depending on the platform. To enable migration of slab pages, CoF again switches the backing allocator to the non-contiguous memory allocator in order to allocate kernel stacks on migratable pages.

Page tables. Page table pages are part of the page table hierarchy for one of the virtual address spaces. These pages are a prime target for Rowhammer *exploitation* [48], but, as exemplified earlier, they are also convenient for templating, since they can easily be repeatedly allocated by user-space and accesses can be triggered and measured through the MMU. Page table pages are normally allocated by the frame allocator and cannot easily be managed via the virtual mappings of a non-contiguous allocator—since the pages reference each other with *physical addresses*. To enable page table page migration, CoF scans the page tables of the owning process and updates any physical references to the migrated page table page.

https://t.me/learningnets

VI. Implementation

To evaluate our design, we implemented CoF on Linux (v5.4.1), requiring around 600 LoC of kernel changes, the bulk of which related to migration of `vmalloc` (i.e., the non-contiguous allocator) and SLUB (i.e., the default slab allocator) pages. In the following, we provide more details on the implementation of the main CoF components.

A. Templating Detection and Vulnerable Page Offlining

Whenever the memory controller detects a correctable bit flip, it raises a machine check exception and informs the EDAC module of the kernel. We hook into this module with our templating detector. In order to keep track of per-page bit flips, we introduce a new page flag: `PG_flip`. On the first bit flip in a page, we set the page flag to remember that a single bit flip was successfully injected in that page. This allows us to detect when an attacker manages to flip two bits in the same page. At that point, we set the existing `PG_hwpoison` flag to permanently offline the vulnerable page. By using a single (unused) page flag, we can do this tracking efficiently with just one bit of information in the existing `struct page`.

B. Migration Support for `vmalloc` Pages

As discussed earlier, the first building block to support the migration of kernel pages is to add migration capabilities to the `vmalloc` allocator. To reduce overall kernel changes, we aim to make the migration functionality compatible with the existing user migration interface (i.e., `migrate_pages`).

On Linux, pages subject to migration need to have an `address_space` structure associated to them in order to specify the memory mapping. We allocate an anonymous inode to represent the `vmalloc` memory mappings, as it contains the `address_space_operations` structure. This structure stores callbacks to the three functions that the page migration routine calls to move a physical page. The `vmalloc`-specific migration callback copies the data and flags of the old page to a new page, unmaps the old page from the kernel page table, and replaces it with a new page table entry. Finally, we invalidate the TLB entry for the old page.

Furthermore, since the `vmalloc` allocator is not aware of the underlying migration, we must keep all the data structures associated with a memory allocation consistent. Therefore, we add a reverse mapping which allows us to get hold of the corresponding `vm_struct` from a `vmalloc` page. This is done by overloading the `private` field in `struct page` to store the pointer to the data structure. This is required to update the `pages` array in the `vm_struct`, which stores the pointers to all the order-0 pages backing the `vmalloc` allocation.

C. Migration Support for SLUB Pages

To support migration of SLUB pages, our goal is to change the backing allocator managing slab caches to our migratable `vmalloc` variant. However, as a few special slab caches (e.g., DMA buffers) rely on physical addressing, we need to be able to selectively preserve the original allocator backend as needed. Therefore, our modified SLUB allocator supports using both `vmalloc` and the original page frame allocator as backends, on a per-cache basis. We add support to allocate slab memory with `vmalloc` in `alloc_slab_page`, which is the function interfacing with the page frame or *buddy* allocator to request new pages.

To avoid changing too many code paths in SLUB, we define a special page structure which holds a `struct page` and a `struct vm_struct`, so that we can cast the pointer to a `struct page` and allow the code in SLUB to use a normal page structure. Depending on the slab flag, we can then cast the pointer back to our special page in order to access the metadata of the vmalloc allocation. This design, similar to subclassing in object-oriented languages, allows us to reuse the existing SLUB code with minimal changes.

There are three main ways in which SLUB can allocate memory: *named caches*, *anonymous caches*, and *large allocations*. We consider each case in detail in the following.

Named caches. In the Linux kernel, data structures that are allocated/deallocated often have a named slab cache to make their allocations efficient (and trackable). In CoF, we use a combination of the slab flag and GFP flags to specify that the slab pages should be allocated with `vmalloc`. Our current prototype allocates most of the objects in the named caches listed in `/proc/slabinfo` with `vmalloc`.

Anonymous caches. The vast majority of memory allocations made through the slab allocator are satisfied with the `kmalloc` interface. The Linux kernel creates a set of anonymous slab caches during the initialization phase of the memory management subsystem. These are multipurpose caches with predefined power-of-two sizes in the range of $[8 - 8,192]$ bytes. The 5.4.1 kernel is equipped with three different types of anonymous caches, `kmalloc` for general purpose allocations, `kmalloc-dma` for DMA allocations and `kmalloc-rcl` to allocate reclaimable memory. With CoF, we introduce a new general-purpose cache (`kmalloc-cof`) which is backed by `vmalloc` memory. With this design, we can exclude the few special `kmalloc` calls from using `vmalloc`.

Large Allocations. Whenever a subsystem requests an allocation through `kmalloc` which is larger than the maximum cache size defined in the kernel, SLUB turns to the buddy allocator to complete the allocation. In this case, SLUB does not keep track of the allocation by means of a `kmem_cache` structure. Upon a `kfree` operation, the allocator simply checks if the pointer to the object has a `kmem_cache` associated to it, and if it does not, SLUB simply calls the buddy allocator's routine to free the pages (i.e., `__free_pages`). In CoF, when a caller requests a large allocation, `kmalloc` returns a `vmalloc` pointer and upon a free operation, we check if the pointer of the object is in the `vmalloc` space; if so, we call `vfree`.

D. Migration Support for Other User-accounted Pages

For the other types of pages that are covered by the page protector, CoF relies on existing kernel mechanisms or the ones presented earlier in this section. In particular, *User* pages (anonymous or file pages in page cache) are migrated using the existing `migrate_pages` interface. Unreferenced *page*

https://t.me/learningnets

cache pages are directly evicted from the page cache, using the existing interfaces used by the memory reclaiming subsystem. *Kernel stack* pages are allocated with `vmalloc` by means of the `CONFIG_VMAP_STACK` configuration option, after which they can be migrated like any other `vmalloc` area.

Finally, *page table pages* are made migratable via some simple changes: in `struct page` (using an existing field unused for page table pages), each page table page is given either a pointer to the owning `mm_struct` (for top-level page table pages), or a pointer to the parent table's entry. Then, after creating the migrated page table page, we update all references (either in `mm_struct` or the parent table) by following these pointers in `struct page`. We also update any pointers in the `struct page` of its child page table pages, to keep our metadata consistent. We then flush the TLB to complete the migration process.

E. Addressing Race Conditions

Although page migration is an application-transparent process (i.e., user processes are not aware of the remapping of their page tables), it still takes non-zero execution time. This introduces a vulnerability window in which a stealthy attacker could potentially race against page migration from another thread to continue the attack—i.e., inject a third bit flip with Rowhammer to evade the ECC function before the migration completes and the page is offlined.

To address this race condition, upon the execution of the MCE handler (called *synchronously* by the memory controller as soon as an access happens to an address which has suffered a correctable error event), CoF immediately pauses all the user threads until the migration completes. To this end, we rely on the pausing/resuming mechanism part of hibernation support in modern OSes and specifically `freeze_processes` and `thaw_processes` on Linux. As we confirmed experimentally, this strategy is efficient and does not introduce visible overhead even in adversarial conditions.

VII. Evaluation

We evaluate CoF along three different dimensions: memory overhead, runtime overhead and security. Additionally, we present an empirical analysis on the distribution of bit flips in DRAM modules from different vendors. This data underlies some design choices of CoF, and will be used in the remainder of the evaluation. We conduct all the experiments on an Intel Xeon Silver 4110 CPU with 32 GB of DDR4 ECC RAM. We compare the aforementioned aspects of CoF to a vanilla 5.4.1 Linux kernel.

A. Distribution of Rowhammer Induced Bit Flips

The optimal design that minimizes overhead without compromising security depends on the distribution of bit flips. For instance, if page frames rarely incur multiple bit flips, per-page tracking is sufficient and, being more efficient, preferable to tracking per ECC word. Therefore, we conduct an analysis on the spatial distribution of Rowhammer-induced bit flips, based on the DDR4 flip database produced by the state-of-the-art Rowhammer fuzzer presented by Jattke et al. [22].

The data from the fuzzer is a number of "flip tables", one per memory module, showing the distribution of flips inside a large buffer (256 MB). The dataset contains these tables for 40 different memory modules, from four different vendors that collectively produce most of the memory modules in the world. The names of the vendors are blinded, but are representative of memory modules found in commodity systems. The memory chips in the data set are the same for ECC and non ECC memory. Table II shows the data for all memory modules. Here, the total bit flips column refers to the bits flips found in the 256 MB buffer, where a higher number implies that a module is more vulnerable to Rowhammer.

		% of pages		
DIMM	Total Bit Flips	1 bit flip	2 bit flips	2 bit flips in ECC word
$\mathcal{A}_0$	82,183	9.3%	9.0%	2.8%
$\mathcal{A}_1$	12,134	7.3%	4.9%	0.1%
$\mathcal{A}_2$	134,702	5.1%	4.7%	4.2%
$\mathcal{A}_3$	1,746	2.4%	0.2%	0.0%
$\mathcal{A}_4$	5,132	5.2%	1.8%	0.0%
$\mathcal{A}_5$	113,190	9.3%	9.3%	4.2%
$\mathcal{A}_6$	98,425	9.4%	9.3%	3.5%
$\mathcal{A}_7$	32,090	8.8%	8.0%	0.6%
$\mathcal{A}_8$	92,660	9.3%	9.3%	3.2%
$\mathcal{A}_9$	4,889	5.9%	1.3%	0.1%
$\mathcal{A}_{10}$	3,051	4.1%	0.5%	0.0%
$\mathcal{A}_{11}$	3,171	1.6%	1.2%	0.0%
$\mathcal{A}_{12}$	43,581	4.7%	4.7%	1.4%
$\mathcal{A}_{13}$	59,721	4.8%	4.7%	2.3%
$\mathcal{A}_{14}$	64,083	4.7%	4.7%	2.4%
$\mathcal{A}_{15}$	52,580	4.7%	4.6%	2.0%
$\mathcal{A}_{16}$	99,552	5.0%	4.8%	3.7%
$\mathcal{A}_{17}$	138,601	5.3%	5.1%	4.9%
$\mathcal{A}_{18}$	80,601	9.3%	9.2%	2.7%
$\mathcal{A}_{19}$	11,599	4.4%	3.6%	1.6%
$\mathcal{B}_0$	63	0.1%	0.0%	0.0%
$\mathcal{B}_1$	506	0.7%	0.1%	0.0%
$\mathcal{B}_2$	15	0.0%	0.0%	0.0%
$\mathcal{B}_3$	111	0.2%	0.0%	0.0%
$\mathcal{B}_4$	1,107	1.4%	0.2%	0.0%
$\mathcal{B}_5$	14	0.0%	0.0%	0.0%
$\mathcal{B}_6$	78	0.1%	0.0%	0.0%
$\mathcal{B}_7$	70	0.1%	0.0%	0.0%
$\mathcal{B}_8$	258	0.4%	0.0%	0.0%
$\mathcal{B}_9$	1,223	1.3%	0.4%	0.1%
$\mathcal{C}_0$	26	0.0%	0.0%	0.0%
$\mathcal{C}_1$	28	0.0%	0.0%	0.0%
$\mathcal{C}_2$	2,551	2.6%	0.9%	0.1%
$\mathcal{C}_3$	636	0.9%	0.0%	0.0%
$\mathcal{C}_4$	769	1.1%	0.1%	0.0%
$\mathcal{C}_5$	1,028	1.2%	0.3%	0.1%
$\mathcal{D}_0$	10,646	6.4%	4.0%	0.1%
$\mathcal{D}_1$	6,655	3.5%	2.3%	0.1%
$\mathcal{D}_2$	2,030	2.2%	0.7%	0.1%
$\mathcal{D}_3$	6,797	5.3%	2.4%	0.0%

TABLE II: Spatial distribution of bit flips on DDR4 DIMMs (anonymized). We show the number of pages that are susceptible to Rowhammer, those could be considered vulnerable by CoF (i.e., have 2 bit flips), and those that are susceptible to ECCploit (have 2 bit flips in the same ECC word).

Next, we calculate the fraction of pages that are vulnerable to Rowhammer: pages that contain at least one bit flip. These are the pages that CoF monitors (i.e., for which it can set the `PG_flip` flag) to check for any further correctable errors which would invoke the page protector. The median number of

https://t.me/learningnets

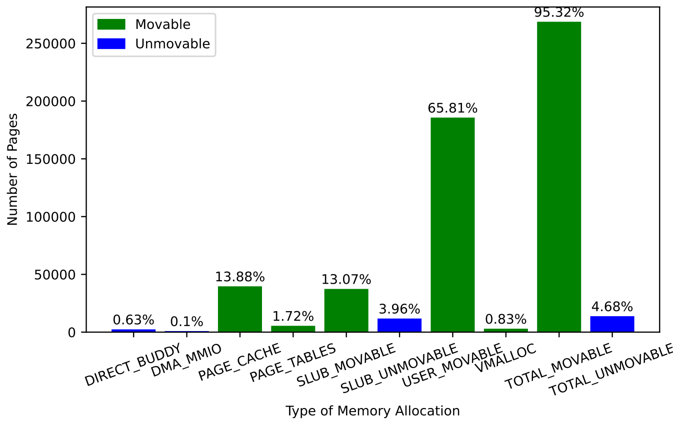

Fig. 3: A histogram to show the share of Migratable (or *movable*) pages in the system after a run of the LMBench benchmark.

	Baseline	CoF	
	seconds	seconds	Δ
`perlbench_s`	379	380	0.3%
`gcc_s`	544	547	0.6%
`mcf_s`	864	870	0.7%
`lbm_s`	1,538	1,538	0.0%
`omnetpp_s`	530	534	0.8%
`xalancbmk_s`	295	296	0.3%
`x264_s`	458	458	0.0%
`deepsjeng_s`	459	460	0.2%
`imagick_s`	1,406	1,407	0.1%
`leela_s`	595	595	0.0%
`nab_s`	485	485	0.0%
`xz_s`	2,938	2,938	0.0%
geomean			0.2%

TABLE III: SPEC CPU2017 results for CoF

pages monitored by CoF (i.e., pages which contain *at least* one bit flip) is 3.8%. We can see a high variance in the data, with some memory modules far more vulnerable to Rowhammer. In particular, the vast majority of bit flips are concentrated in DIMMs $\mathcal{A}_{0-19}$ and $\mathcal{D}_{0-3}$.

We then extend this analysis to pages containing *two* bit flips—the pages CoF might migrate due to templating attempts. We can see this is less than the previous figures, since it is less likely that two bit flips occur in the same page. The median number of vulnerable pages, assuming two bit flips are required, is only 1.26% of all pages. Finally, we calculate the fraction of pages that contain at least one *ECC word* with two bit flips. With a median of 0.08% across all pages, this is very unlikely to happen even with active hammering. However, overall, even on the most vulnerable DIMMs, fewer than 10% of the pages have multiple flips, and in the worst case, fewer than 5% incur multiple bit flips in the same ECC word.

Next, we use this data to calculate the average time an attacker would need to template ECC memory. We can later use this number to analyze adversarial situations, i.e., where the system is under attack. We assume that an attacker can inject one bit flip per ECC word in one DRAM refresh window, which is typically 64 ms. In practice, this is hard to achieve and represents the absolute best-case timing for an attacker. We further assume that a DRAM row is 8 KB wide and can store the equivalent of 2 page frames (4 KB) from the OS perspective. These values allow us to derive the following formula to calculate the expected templating time in seconds:

$$T_{template} = \frac{T_{page}}{P(template)} = \frac{T_{refw} \times size(ECCword)}{2 \times P(template)} \quad (1)$$

In this formula, T_{page} is the time needed to template all the bits on a single page and $P(template)$ the probability that this page is a page that is suitable for a Rowhammer attack (as obtained from Table II). We can template one bit in every ECC word of a row at the same time. Since we do this for every refresh window, we need to multiply by the number of bits in an ECC word (and divide by 2, since we do 2 page frames in a single row). Substituting $P(template)$ in our formula with the highest and the median values of Table II, yields an average templating time of 22 and 163 seconds, respectively.

Given both the distribution of bit flips and the attack rates, we expect very rare migration events during execution even in the worst case. Hence, we expect low overheads for CoF, as confirmed experimentally in the remainder of this section.

B. Memory Overhead

Measuring the exact memory overhead of CoF is hard, as it depends on the overall state of the kernel, which in turn depends on all processes in the system. Instead, we define the overhead as a function of several system variables as well as the overhead resulting from our modified allocators. We measure the overhead for two different situations: (1) under normal operation, and (2) under adversarial conditions where an attacker is actively hammering the system.

Normal conditions. One of the advantages of CoF is that we can rely on the memory controller to detect Rowhammer templating attempts, meaning that we do not have to keep track of any metadata ourselves. As mentioned in Section VI, the templating detector marks pages by setting a single flag of `struct page`. No further state is required to detect an ongoing Rowhammer attack.

Our modification to make SLUB migratable, however, does introduce some overhead. Every 4 KB page used by SLUB has 72 bytes overhead from the special page struct, and an additional 64 + 72 byte overhead from the `vmalloc` metadata. Overall, this results in a 208 byte overhead, or around ∼5% overhead per slab page. On an average system we measured 572 pages used by SLUB, meaning an additional 116 KB of memory was used by SLUB. This represented about ∼0.01% of the total memory used in the system.

In our current design, every slab page has its own `vmalloc` area, while technically a single `vmalloc` area can hold multiple pages, and thus one vmalloc area per slab cache would be sufficient. This would mean only a 72 byte overhead per page, and a 64 + 72 byte overhead per cache. On our system we measured ∼150 caches, resulting in an overhead

https://t.me/learningnets

	Baseline	CoF	
	ms	ms	Δ
`astar`	101.1	101.4	0.3%
`beat-detection`	82.4	83.5	1.3%
`dft`	117.8	121.3	3.0%
`fft`	59.3	60.7	2.3%
`oscillator`	72.5	73.1	0.8%
`gaussian-blur`	220.9	221.9	0.5%
`darkroom`	218.7	218.9	0.1%
`desaturate`	63.2	63.2	0.0%
`parse-financial`	34.8	35.2	1.1%
`stringify-tinderbox`	27.9	28.0	0.3%
`crypto-aes`	90.1	91.9	2.0%
`crypto-ccm`	93.7	95.1	1.5%
`crypto-pbkdf2`	85.1	85.8	0.8%
`crypto-sha256-iter`	44.4	45.2	1.8%
geomean			1.1%

TABLE IV: Mozilla Kraken results for CoF

	Baseline	CoF	
	μs	μs	Δ
`null call`	0.1	0.1	0.0%
`null I/O`	0.2	0.2	0.0%
`stat`	0.6	0.6	0.0%
`open/close`	1.2	1.2	0.0%
`select TCP`	3.4	3.4	0.9%
`fork proc`	230.8	245.3	6.3%
`exec proc`	777.3	783.4	0.8%
`sh proc`	2,314.0	2,393.0	3.4%
`pipe latency`	5.0	5.1	0.2%
`TCP latency`	11.1	11.6	4.8%
`TCP conn. latency`	14.2	15.0	5.4%
`signal install`	0.2	0.2	0.0%
`signal handle`	1.4	1.4	0.0%
`prot fault`	0.8	0.8	1.3%
	MB/s	MB/s	
`pipe bw`	2,832	2,667	5.8%
`TCP bw`	6,106	5,984	2.0%
geomean			1.9%

TABLE V: LMBench results for CoF

of ∼2.5% for SLUB. Since this still accounts for only about ∼0.01% of memory, we did not implement this optimization.

Adversarial conditions. In order to calculate the memory overhead of CoF on a system which is under attack, we refer to the results reported in Table II. The data shows that the amount of memory which is offlined under attack highly depends on the memory module used by the system. The median value of pages that CoF offlines under adversarial conditions is 1.26%. The highest values were found for DIMMs $\mathcal{A}_5$, $\mathcal{A}_6$ and, $\mathcal{A}_8$ for which CoF would offline 9.3% of physical memory.

As discussed in Section V, in our current design CoF tracks the bit flips on a per-page basis. However, if memory controllers were to consistently report the locations of the bit flips at a finer granularity when it raises an MCE, we could implement the tracking per code word. While the data in Table II suggests that this approach would incur lower memory overhead on a system under attack, we would have to employ a more expensive tracking system that requires 64-bits of extra information per page in the most optimistic scenario. Specifically, we could use one bit per code word assuming a page frame can store 64 data words which are 64-bit wide.

In a more realistic scenario, it would not be practical to add data to `struct page`, as it is a very compact structure. Extending it would be detrimental for memory overhead, but also performance, since it would no longer be cacheline aligned. Therefore, we would have to store the page frame number and the offset in the page of a correctable error out-of-band in a separate data structure. Furthermore, this approach has the downside of 'wasting' memory to store the metadata representing the code words in software.

C. Security

In order to evaluate the security guarantees of CoF, we calculated the residual attack surface, which is represented by the number of non-migratable pages in the system—even if most such pages are not useful for attackers. In order to collect the data required to make these calculations, we used the `page owner` tracing functionality in the Linux kernel [55]. This debug facility records a stack trace for every page frame allocation in the system. For our analysis, we parsed all these traces and categorized the memory allocations in buckets to provide an accurate overview of all memory used in the system. We then annotated this data with whether each type of memory was protected by CoF.

The plot in Figure 3 shows that, on our system, after a run of a system call heavy benchmark such as LMBench, 95.32% of pages in the whole system were protected. Most importantly, the graph shows that the pages in a system which are attacker-controllable as defined in Section V are protected by CoF: the `vmalloc` areas (including kernel stacks), page tables, page cache, and SLUB. For SLUB, we still have a small fraction of caches that cannot be migrated, due to some reliance on physical (contiguous) addressing. However, these caches are all special-purpose ones and we have not observed cases where the allocations are attacker-controlled.

D. Performance

Experimental setup. To evaluate the performance of CoF, we tested the prototype with standard benchmarking suites, such as SPEC CPU2017, LMBench, Nginx and Mozilla Kraken. These results show the performance overhead of our solution under normal conditions in various settings. Moreover, to test the performance of CoF under attack conditions we devised a microbenchmark to measure the impact of page migration.

SPEC CPU2017. To evaluate the performance of the CoF kernel, we first ran the SPEC CPU2017 benchmarking suite. All benchmarks in SPEC CPU2017 are either CPU or memory bound, and perform little to no I/O. The binaries are compiled with `-fno-strict-aliasing` to avoid commonly known portability issues for the `perlbench` and `gcc` binaries [49]. Furthermore, all the benchmarks were compiled with the `-O3`

https://t.me/learningnets

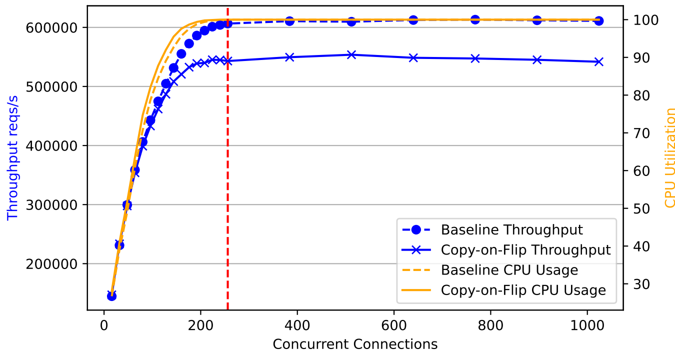

Fig. 4: A plot to show Nginx throughput at saturation.

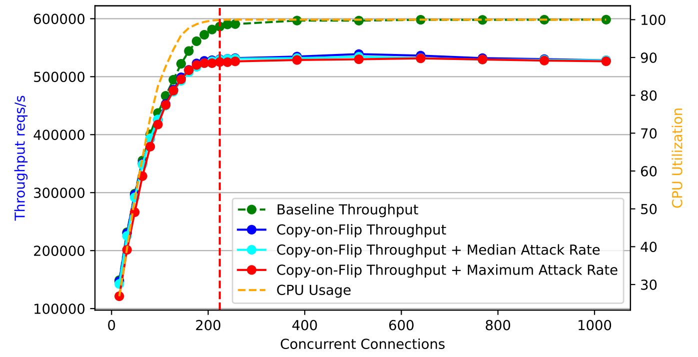

Fig. 5: A plot to show Nginx throughput at saturation under adversarial conditions.

flag and ran with transparent huge paging *off* to reduce noise. We ran all C and C++ benchmarks from both the integer and floating point sets of the SPECspeed suite. For benchmarks that supported it, OpenMP parallelization was enabled with 16 threads. The results, shown in Table III, are reported as the median runtime value in seconds of five runs, with negligible standard deviation. The graph shows negligible overhead ($< 1\%$) of the CoF kernel when compared to the vanilla 5.4.1 Linux kernel across all binaries, with a geometric mean (geomean) overhead of 0.2%. These results are in line with our expectations, because CoF does not modify user memory allocations as those are already migratable. Moreover, the benchmarks do not significantly stress the kernel allocators, due to the scarcity of I/O and system calls.

Web browser performance. To further test commodity programs and real-world use cases and libraries, we ran Google Chrome (version 102.0.5005.61) on top of our CoF kernel. Chrome works reliably, including I/O-heavy websites such as YouTube, demonstrating our kernel changes are compatible with highly complex software. To evaluate the performance of the web browser, we ran the Mozilla Kraken benchmarking suite. This suite consists of a number of different JavaScript benchmarks, including scripts for audio processing and image filtering. Table IV shows the median results of five runs of Kraken for the baseline and our prototype. Similar to SPEC CPU2017, Kraken shows that CPU- and memory-intensive workloads in *user-space* are largely unaffected by CoF, with a geomean of only 1.1%.

Nginx. To demonstrate CoF in I/O-intensive situations, we ran benchmarks on the popular high-performance `nginx-1.18.0` web server. We set up a second (identical) machine to run the `wrk` HTTP benchmarking tool to make the requests. These machines are connected using Mellanox ConnectX 100G NICs to ensure we can reach CPU saturation on the server. We configured `nginx` to use 16 workers, and use 16 `wrk` client threads to request a 64-byte file containing static HTML. We ran the benchmark 3 times for 30 seconds and reported the median value.

Figure 4 shows the throughput of `nginx` in requests per second for the baseline kernel and the CoF kernel. All cores are saturated for ≥ 256 connections. The median throughput at saturation is 10.56% lower in CoF compared to the baseline, while increasing the 90th percentile latency by 12.6%.

LMBench. To dig down further into the overhead CoF can cause, we ran the LMbench benchmarking suite. This suite contains a number of I/O and syscall microbenchmarks. Table V shows the median runtime of five runs compared to the baseline kernel. In general, the results show that the overhead for the majority of tests is minor, with the peak overheads being `fork proc` (6.3%) and the TCP benchmarks ($\sim$5%). This overhead is due to the fact that these system calls make heavy use of the `vmalloc`-backed SLUB allocator. The geomean overhead for LMBench is 1.9%.

Performance overhead under attack. Finally, we measure the performance of CoF under stress, when an attacker is actively mounting an attack. When an attacker is hammering memory, this might induce bit flips. If a physical page frame experiences two bit flips, CoF migrates its data, potentially introducing overhead to the running applications. For this experiment, we set up the system to simulate bit flips and migrations at a deterministic rate. For this, we used our analysis of Section VII-A, where we found attack rates of flips every 22 and 163 seconds, for worst-case and typical memory modules respectively.

For our experiment, we measured the overhead of the `nginx` benchmark (our most performance-sensitive benchmark), similar to our earlier experiments, using both this median and a worst-case attack rate. As Figure 5 shows, the impact of this is very minimal, at only 0.22% and 0.69% throughput degradation for the median-case and the worst-case adversarial condition, respectively, compared to the throughput achieved in CoF under normal conditions. This shows that an attacker cannot abuse CoF's protection to significantly disrupt the performance of the system.

E. Detection Accuracy

To further evaluate the effectiveness of CoF, we now consider its detection accuracy. In particular, we are interested in circumstances where (i) the template detector may classify a safe page as vulnerable, leading to unnecessary offlining and thus memory overhead (False Positive) or (ii) the template detector may classify a vulnerable page under attack as safe, leading to residual attack opportunities (False Negative).

To evaluate the former, we measured the number of migrated pages when running benign applications such as

https://t.me/learningnets

the ones used for our performance evaluation, namely SPEC CPU2017, `nginx-1.18.0`, LMBench, and Mozilla Kraken. During the repeated execution of our benchmarks, we did not encounter any false positives due to spurious bit flips. While our time-bounded experiments cannot rule out false positives in the general case, Meza et al. [36] have conducted an analysis of correctable memory error rates in large-scale production data centers. Their work has shown that there is a decreasing trend in the incidence of such errors compared to the previous decades and that the DIMM architecture characteristics affect the overall error rate. Moreover, although spurious bit flips may cause the template detector to (over)mark some pages as vulnerable, this should only happen rarely. Given that false positives and (over)offlining can only occur in face of at least two spurious bit flips within the same page frame, we expect to have negligible impact on memory or performance overhead.

To evaluate false negatives for the template detector, we conducted an experiment in which we induce bit flips in every type of memory page that CoF protects as discussed in Section V-B. Initially, we ran publicly available Rowhammer PoCs [16], [22]; however, they reported no flips in our setup. Therefore, to stress-test our solution, we induced bit flips with hardware error injection mechanisms. We ran this experiment on an Intel Xeon Silver 4310 machine which has ECC error injection capabilities and induced the correctable errors using the EINJ interface in the Linux kernel [53]. During our experiments, we used such interface to repeatedly inject bit flips in all the supported memory types and verified that CoF was able to detect the flips and trigger page migration (and offlining) in all cases.

VIII. Discussion

Protecting residual allocations. In Section V-B, we outlined the requirements for which memory should be protected by CoF. Moreover, Figure 3 confirms CoF protects most of the user-accountable memory, and the vast majority of memory in the system.

One type of memory CoF currently does not protect is that of DMA buffers. Since different systems load different drivers, some of which have custom memory allocators, it is difficult to standardize these allocations and protect them. Given specific DMA allocations, in principle, one could make them migratable by using `vmalloc`. This strategy can take advantage of a modern IOMMU that supports virtual addressing. Conveniently, most modern OSes, including Linux [56], FreeBSD [42] and Windows [38] support DMA remapping with modern IOMMUs. However, legacy devices that only operate on physically-contiguous DMA buffers cannot be easily supported. As suggested in prior work [58], an alternative is to protect DMA buffers by means of row isolation.

While pages outside user control (Section V-B) are not good targets for attacks, one might still want to protect them to provide defense in depth. Doing so is mostly a matter of engineering. For pages that only rely on virtual addressing, one can again rely on `vmalloc`. The few other pages that cannot be easily migrated could be isolated in dedicated memory areas and protected by guard rows [9], [29], [51], [58].

Virtualization-based implementation. We decided to implement CoF on bare metal in order to offer protection to a wider range of (native or virtualized) systems. However, our CoF design can also be implemented by relying on Second Level Address Translation (SLAT)—a hardware virtualization feature available on modern processors [20]. Rather than changing the individual kernel allocators to support page migration, one can interpose on SLAT mappings to dynamically remap and offline vulnerable pages of a guest virtual machine in a guest-transparent way. While this strategy would likely simplify the implementation of our design and is also a natural match for virtualized environments, it would force a bare-metal environment to run on a virtualized stack—unnecessarily incurring virtualization (performance and security) costs.

Offlining DoS attack. In our current prototype, a page is migrated and offlined when two bit flips occur in the same page. Due to our optimized metadata for marking pages—setting a single bit in the page struct—the system cannot distinguish flips of different bits versus two flips of the same bit in a given page. Therefore, an attacker could target CoF to try to offline as many pages as possible. As shown in our analysis in Table II, the impact of this attack is minimal in practice. Moreover, we consider denial-of-service attacks to be preferable over arbitrary memory corruptions through Rowhammer. Finally, we excluded such attacks from the threat model since an attacker armed with Rowhammer on an ECC-equipped system can easily mount denial-of-service attacks by means of blind hammering anyway.

The Half-Double Attack. Recently, Kogler et al. presented Half-Double [28], an end-to-end Rowhammer attack which escalates Rowhammer to rows beyond adjacent neighboring rows. Moreover, Half-Double does not resort to traditional memory templating in order to bypass on-DIMM ECC memory. On the contrary, it introduces the concept of *blind hammering*. With this technique, an attacker tries to inject bit flips in page table entries with Rowhammer and relies on stealthy oracles such as Spectre [27] to verify that the bit flip occurred at the right offset in a page table entry without causing a crash.

Nonetheless, Half-Double attacks can still be detected by CoF. First, CoF does not target specific Rowhammer patterns to protect the system. That is, we can protect page frames regardless of their distance to the aggressor rows. Furthermore, CoF relies on memory controller-based (MC-based) ECC rather than the on-DIMM ECC targeted by blind hammering. While the latter entirely hides errors from software, making ECC-agnostic exploitation possible, MC-based ECC raises machine check exceptions (MCEs) to the OS upon access. This forces the attacker to resort to ECCploit-style [10] attacks, but also enables CoF-style bit flip interposition. Finally, with the experiment outlined in Section VII-E, we experimentally confirm that the Spectre oracle used as a building block for blind hammering still raises an MCE on MC-based ECC systems and is thus detected by the template detector in CoF.

CoF vs. Other Software-Based Defenses. Since it is hard to directly compare CoF performance to that of other software-based defenses fairly—as other solutions were each evaluated with a different set of benchmarks—we focus here on a qualitative comparison. First, we observe that CoF is among the few solutions which aim to provide system-wide protection, with

https://t.me/learningnets

the only other defense being ZebRAM [29]. However, in order to guarantee system-wide protection, ZebRAM trades off system memory and performance, incurring nontrivial overhead for large working set sizes. For instance, the ZebRAM paper reports a 3x overhead in execution time on a saturated Redis reaching 70% working set size, with the solution configured to provide protection to only near-aggressor-based Rowhammer attacks. The performance impact (and/or memory overhead) would even be more significant to address far-aggressor-based Rowhammer attacks such as Half-Double [28]. Other software-based defenses are more efficient and can compete with CoF performance, but also limit protection guarantees to specific data [43], [50], [58], [63], specific security domains [9], or specific Rowhammer patterns [3].

Moreover, some existing solutions rely on primitives which are often unavailable in practice. For instance, most isolation-based defenses rely on precise knowledge of DRAM geometry [9], [29], [63], which is hard to obtain precisely on all systems [50]. Other defenses rely on advanced performance counters [3], which are often unreliable [12] and are not supported by all processors. CoF, , only relies on well-established process/memory management capabilities of modern operating system kernels and on memory controller-based ECC which is widespread on server platforms.

IX. Related Work

A. Rowhammer Attacks

Since the discovery of Rowhammer [25], a plethora of different attacks which exploit this vulnerability have been documented, both by industry and by academic researchers.

The first privilege-escalation Rowhammer attack was presented by Seaborn and Dullien [48], which targeted page tables as an attack vector to gain privileged access to physical memory. Shortly after this proof of concept, attackers showed that Rowhammer is exploitable from virtualized environments [46], on ARM architectures [57], over the network [33], [51], from browser sandboxes [8], [13], [15], [18] and also as an information leak primitive [30].

As more researchers started to turn their attention to Rowhammer, the community gained more understanding of how to amplify the Rowhammer effect even more on modern memory modules. Efforts in reverse engineering of how physical addresses map to DRAM [45], [50], and of the inner workings of mitigations implemented by hardware vendors [16], [22], have allowed attackers to build more powerful "hammering patterns" which result in higher error injection rates. For example, Tatar et al. [50] showed that with precise knowledge of the underlying DRAM geometry, an attacker could find bit flips in up to 99% of modules in DDR3 modules. Frigo et al. [16] and Jattke et al. [22] brought to light the weaknesses of the mitigations adopted by hardware vendors, and found that, especially for DDR4 modules, certain "hammering patterns" are far more effective than others.

B. Rowhammer Defenses

In order to aid in the protection of modern systems, numerous defenses have been proposed both at the software and at the hardware layer.

Most of the existing software mitigations against Rowhammer target the memory massaging phase of Rowhammer attacks by isolating, in DRAM, attacker-controlled memory from sensitive memory by means of guard rows [6], [9], [29], [58]. Unfortunately, doing so is difficult not just because it requires accurate knowledge of DRAM geometry (which is challenging to obtain [50]), but especially because boundaries are often blurry. For instance, the isolation of kernel memory proposed in G-CATT [9] is incomplete due to the Linux page cache and other user-kernel shared areas [17], [58]. Moreover, isolation-based defenses typically do not offer comprehensive protection, but rather protect specific memory areas [6], [9], leaving much of the memory vulnerable to Rowhammer attacks. Conversely, ZebRAM [29], does offer comprehensive protection by using every other row as a guard row, but at a nontrivial cost in terms of performance and/or memory.

Other software mitigations focus on the hammering phase—using performance counters to detect unusual cache miss rates (or other indications of uncached accesses) and interpreting those as a Rowhammer attack [3]. The problem with this approach is threefold. First, performance counters vary from CPU to CPU and many systems do not support a counter for uncached accesses. Second, the performance counters are indirect indicators at best. For instance, a large number of cache misses may easily stem from benign access patterns in DRAM. Third, the number of activations required for a bit flip keeps dropping as the density of DRAM increases. Where earlier attacks required hundreds of thousands of activations per refresh interval, recent ones require only a few tens of thousands. This means that the threshold must be lowered also, further increasing the likelihood of false positives.

Other software-based defenses have been presented to target specific types of attack primitives. For instance, Oliverio et al. [43] presented a defense which stops an attacker from using memory deduplication as a memory massaging primitive or as a side channel for information leakage. Zhang et al. [63] on the other hand, aim to stop bit flips from happening on pages which store page tables because many attacks corrupt page tables to gain unsupervised access to physical memory.

Current hardware-based mitigations similarly try to stop the bit flips from happening by tracking the number of activations per row, and refreshing adjacent rows when a threshold is reached. Such technique is generally known as *Target Row Refresh* (TRR) [23], [32], [37] and is widely deployed. However, it has been shown to be easily bypassable by means of improved Rowhammer patterns [13], [16], [22]. Other defenses proposed by academic researchers aim to proactively stop an attacker from launching a Rowhammer attack by eradicating at least one of the essential primitives required for exploitation [5], [21], [35], [60], [62]. Another recent hybrid defense [24] suggests using a cryptographic MAC as a more secure alternative to normal ECC memory. The defense enables error detection for an unbounded number of bit flips in hardware and error correction of up to 8 bits every 256 bits in both hardware and software.

Although more advanced hardware mitigations may eventually reach production, this will take time, leaving memory that will be in use for 5–10 years at the mercy of attacks.

https://t.me/learningnets

X. Conclusion

In this paper, we presented Copy-on-Flip (CoF), a simple, effective, and efficient design for hardening ECC memory implemented by the memory controller against ECC-aware Rowhammer attacks. By leveraging the ECC error corrections as warning signs of an attacker templating ECC memory, we can accurately migrate and offline vulnerable pages before the attacker can obtain exploitable templates. With our Linux-based prototype, we demonstrated we can transparently protect all the relevant templating targets, including those in kernel memory, with only small kernel changes. In combination with our low overhead—negligible in most cases—we believe CoF can be adopted by the kernel in practice, helping protect vulnerable ECC-equipped servers everywhere. In such a spirit, we have open sourced our prototype[3].

Acknowledgements

We thank the anonymous reviewers for their feedback. We'd also like to thank Daniel Andriesse for his help with the ECC error injection testbed. This work was supported by Intel Corporation through the Side Channel Vulnerability ISRA and by NWO through projects "Theseus" and "INTERSECT".

References

[1] Page migration. [Online]. Available: https://www.kernel.org/doc/html/v5.9/vm/page_migration.html

[2] A. Arcangeli, "Kernel Samepage Merging," 2008. [Online]. Available: https://www.kernel.org/doc/html/v5.0/vm/ksm.html

[3] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation rowhammer attacks," in *ASPLOS*, 2016.

[4] J. Baldwin. Support for x86 machine check architecture. [Online]. Available: http://fxr.watson.org/fxr/source/x86/x86/mca.c?v=FREEBSD-10-0

[5] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete In-DRAM rowhammer mitigation," in *DRAMSec*, 2021.

[6] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "RIP-RH: Preventing rowhammer-based inter-process attacks," in *AsiaCCS*, 2019.

[7] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *USENIX ATC*, 1994.

[8] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory deduplication as an advanced exploitation vector," in *IEEE S&P*, 2016.

[9] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't Touch This: Practical and generic software-only defenses against Rowhammer attacks," *arXiv preprint arXiv:1611.08396*, 2016.

[10] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the effectiveness of ECC memory against Rowhammer attacks," in *IEEE S&P*, 2019.

[11] J. Corbet and A. Rubini. vmalloc and friends. [Online]. Available: https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch07s04.html

[12] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *IEEE S&P*, 2019.

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

[14] J. Dyson, J. Roberson, and D.-E. Smorgrav. Uma(9) — freebsd kernel developer's manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=uma&sektion=9

[3] https://github.com/vusec/Copy-on-Flip

[15] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU," in *IEEE S&P*, 2018.

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

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of Rowhammer defenses," in *IEEE S&P*, 2018.

[18] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *DIMVA*, 2016.

[19] Intel. Intel & samsung: Improving memory reliability at data centers. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/intel-and-samsung-mrt-improving-memory-reliability-at-data-centers.pdf

[20] ——, "Intel® 64 and IA-32 architectures software developer's manual."

[21] G. Irazoqui, T. Eisenbarth, and B. Sunar, "MASCAT: Preventing microarchitectural attacks before distribution," in *CODASPY*, 2018.

[22] P. Jattke, V. van der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the frequency domain," in *IEEE S&P*, 2022.

[23] JEDEC Standard, "DDR4 SDRAM Registered DIMM Design Specification," 2014.

[24] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, M. Lipp, and D. Gruss, "CSI:Rowhammer - cryptographic security and integrity against rowhammer," in *IEEE S&P*, 2023.

[25] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.

[26] A. Kleen and F. Wu. Memory failure. [Online]. Available: https://elixir.bootlin.com/linux/v5.4.1/source/mm/memory-failure.c

[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

[28] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering from the next row over," in *USENIX Security*, 2022.

[29] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and compatible software protection against Rowhammer attacks," in *OSDI*, 2018.

[30] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *IEEE S&P*, 2020.

[31] C. Lameter, "Slab allocators in the linux kernel: SLAB, SLOB, SLUB," in *LinuxCon*, 2014.

[32] J.-B. Lee, "Green Memory Solution," *Samsung Investors Forum*, 2014. [Online]. Available: http://aod.teletogether.com/sec/20140519/SAMSUNG_Investors_Forum_2014_session_1.pdf#page=15

[33] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer faults through network requests," in *EuroS&PW*, 2020.

[34] K. Loughlin, S. Saroiu, A. Wolman, Y. A. Manerkar, and B. Kasikci, "MOESI-Prime: Preventing coherence-induced hammering in commodity workloads," in *ISCA*, 2022.

[35] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "ProTRR: Principled yet optimal In-DRAM Target Row Refresh," in *IEEE S&P*, 2022.

[36] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *DSN*, 2015.

[37] Micron Technology, "DDR4 SDRAM datasheet," 2015. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf

[38] Microsoft. IOMMU DMA remapping. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/display/iommu-dma-remapping

[39] ——. (2021) Overview of windows memory space. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-memory-space

https://t.me/learningnets

[40] O. Mutlu and J. S. Kim, “Rowhammer: A retrospective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, 2020.

[41] E. T. Napierala. RCTL(8) — FreeBSD System Manager’s Manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=rctl&sektion=8

[42] NetApp. IOMMU — freebsd. [Online]. Available: http://fxr.watson.org/fxr/source/amd64/vmm/io/iommu.c?v=FREEBSD-10-0;im=10#L223

[43] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, “Secure page fusion with VUsion,” in *SOSP*, 2017.

[44] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. Ho Ahn, and J. W. Lee, “Graphene: Strong yet lightweight RowHammer protection,” in *MICRO*, 2020.

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

[46] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, “Flip Feng Shui: Hammering a needle in the software stack,” in *USENIX Security*, 2016.

[47] G. Saileshwar, B. Wang, M. K. Qureshi, and P. J. Nair, “Randomized Row-Swap: Mitigating row hammer by breaking spatial correlation between aggressor and victim rows,” in *ASPLOS*, 2022.

[48] M. Seaborn and T. Dullien, “Exploiting the DRAM Rowhammer bug to gain kernel privileges,” in *Black Hat*, 2015.

[49] Standard Performance Evaluation Corporation. 500.perlbench_r SPEC CPU®2017 Benchmark Description. [Online]. Available: https://www.spec.org/cpu2017/Docs/benchmarks/500.perlbench_r.html

[50] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, “Defeating software mitigations against Rowhammer: A surgical precision hammer,” in *RAID*, 2018.

[51] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, “Throwhammer: Rowhammer attacks over the network and defenses,” in *USENIX ATC*, 2018.

[52] A. Tevanian and M. W. Young. Kernel Memory Management. [Online]. Available: http://fxr.watson.org/fxr/source/vm/vm_kern.c?v=FREEBSD-10-0#L308

[53] The Kernel Development Community. APEI Error INJection. [Online]. Available: https://docs.kernel.org/firmware-guide/acpi/apei/einj.html

[54] ——. Control group v2. [Online]. Available: https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v2.rst

[55] ——. page owner: Tracking about who allocated each page. [Online]. Available: https://www.kernel.org/doc/html/v5.4/vm/page_owner.html

[56] ——. x86 IOMMU support. [Online]. Available: https://docs.kernel.org/x86/iommu.html

[57] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic Rowhammer attacks on mobile platforms,” in *CCS*, 2016.

[58] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, “GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM,” in *DIMVA*, 2018.

[59] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” in *IEEE S&P*, 2019.

[60] Y. Wang, L. Orosa, X. Peng, Y. Guo, S. Ghose, M. Patel, J. Kim, J. Gómez-Luna, M. Sadrosadati, N. Ghiasi, and O. Mutlu, “FIGARO: Improving system performance via fine-grained in-DRAM data relocation and caching,” in *MICRO*, 2020.

[61] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi *et al.*, “Block-Hammer: Preventing RowHammer at low cost by blacklisting rapidly-accessed DRAM rows,” in *HPCA*, 2021.

[62] Z. Zhang, Z. Zhan, D. Balasubramanian, B. Li, P. Volgyesi, and X. Koutsoukos, “Leveraging EM side-channel information to detect Rowhammer attacks,” in *IEEE S&P*, 2020.

[63] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wang, N. Surya, Y. Gao, K. Li, Z. Wang, and C. Wu, “SoftTRR: Protect page tables against RowHammer attacks using software-only Target Row Refresh,” in *USENIX ATC*, 2022.

https://t.me/learningnets