Rubicon: Precise Microarchitectural Attacks with Page-Granular Massaging

Matej Bölcskei
ETH Zurich
mboelcskei@ethz.ch

Patrick Jattke
ETH Zurich
pjattke@ethz.ch

Johannes Wikner
ETH Zurich
kwikner@ethz.ch

Kaveh Razavi
ETH Zurich
kaveh@ethz.ch

***Abstract*—Microarchitectural attacks like Rowhammer and Spectre rely on precisely targeting specific memory page frames despite the inherent unpredictability of memory allocation. Due to the lack of a generic mechanism to accurately place the target data in the pages of interest, these attacks resort to spraying their target or scanning the entire physical memory for it. These approaches, however, suffer from unreliability and inefficiency. In contrast, the deterministic behavior of page allocators presents an opportunity to enhance existing attacks and enable new ones.**

This paper introduces Rubicon, a novel technique for page-granular physical memory massaging within the Linux kernel's Zoned Buddy Allocator (ZBA). Rubicon leverages three new primitives that enable placing a page frame at the head of any chosen ZBA free list, ensuring it is prioritized for allocation regardless of its initial state or per-CPU freelist association. Using Rubicon, we build the first deterministic privilege escalation Rowhammer exploit on x86 with a success rate of 100%. Our integration of Rubicon into a recent Spectre attack shows that the root hash of /etc/shadow can now be leaked in 27.8 and 9.5 seconds on AMD and Intel systems — a 6.8× and 284× speedup over the original attack, respectively. We also propose and evaluate practical mitigations for Rubicon, which limit page movement between ZBA lists with negligible performance and fragmentation impact.

1. Introduction

In many cases, the seemingly random page frame assignment of victim data makes microarchitectural attacks slow and unreliable. In attacks like Rowhammer [1]–[4], this can lead to corruptions that crash the system. In Spectre attacks on the kernel [5]–[7], this leads to searching vast amounts of physical memory with a slow and noisy side channel. What if the attacker could place the victim data on *any* desired free page in the system? This paper provides this generic capability on top of Linux for the first time and shows the significant advantage that it provides for microarchitectural exploitation, including possibilities for new Rowhammer and Spectre-like attacks.

Physical memory massaging. One of the original *memory massaging* concepts was described by SkyLined with heap spraying [8]. Broadly speaking, memory massaging is about interacting with a memory allocator until it creates a desired layout in memory to prepare for reliable exploitation. Razavi et al. [9] extended the scope of memory massaging beyond virtual memory, such as the heap, to *physical* page frames to exploit Rowhammer bit flips in DRAM [10], [11]. With physical memory massaging, the attacker aims to interact with low-level memory allocators, to force some victim data to be placed in a specific location in physical memory. This has been achieved by abusing page deduplication [9], user-accessible Direct Memory Access (DMA) APIs in Android [3], or thread-local and purpose-bound reuse of physical pages [12]. These massaging techniques are either limited or rely on specific features that have been disabled. Therefore, a universal page-granular physical memory massaging capability is currently missing.

Microarchitectural attacks. Reliable Rowhammer attacks make use of physical memory massaging to land security-sensitive data on a target page of interest before corrupting or leaking it [3], [9], [12]. In the absence of physical memory massaging, the alternative is spraying the security-sensitive data [1], which is unreliable for large memory spaces. Another class of microarchitectural attacks that could potentially benefit from physical memory massaging is Spectre-like attacks on privileged memory [5]–[7], [13]. These attacks look for secret data in physical memory by first forcing the secret data to be allocated at an unknown location. They then scan the physical memory to look for the secret data. Due to noisy and low-bandwidth channels, it can take a very long time to find the secret. Furthermore, the duration of the attack increases with the amount of physical memory installed in the system. We make a key observation that physical memory massaging can also significantly improve this class of attacks if the attacker can control the exact page frame to which the secret data is assigned.

Rubicon. This paper presents Rubicon, a new physical memory massaging technique that enables an attacker to allocate victim data in any desired page in the Linux Zoned Buddy Allocator (ZBA). Rubicon achieves this objective by pushing the desired page onto the head of any allocator free list that the victim data can use. Three new mechanisms make Rubicon possible:

(1) moving blocks of pages across different ZBA free lists of different orders,
(2) evicting a given page from its respective per-CPU pages (PCP) free list so that the victim's allocator can claim it, and finally,
(3) repurposing a given page, since Linux avoids reusing pages with a different purpose (e.g., anonymous memory) than the requested one (e.g., page table).

Rubicon uses the first mechanism to push the desired page to the correct free list, the second mechanism to make sure the next allocation is served from that free list, and the last mechanism to associate the desired page with the correct purpose (i.e., *migratetype*) of interest.

We demonstrate the benefits of the Rubicon primitive using two different microarchitectural attacks: we build the first reliable Rowhammer-based privilege escalation exploit on x86 without relying on special memory management features. Rubicon enables the exploit to trigger a page table allocation on a previously attacker-owned page, regardless of its previous purpose, i.e., its associated *migratetype*. We also integrate Rubicon into RETBLEED [7], a recent Spectre-style attack on the Linux kernel. In this case, Rubicon enables RETBLEED to land `/etc/shadow` on an attacker-provided page. Without the need for scanning physical memory, Rubicon-armed RETBLEED is, on average, 6.8× and 284× faster on AMD and Intel systems.

Lastly, we design and implement three different mitigations against Rubicon, targeting each of the mechanisms Rubicon relies on. These mitigations are based on randomization, better isolation of PCP lists, and more restricted type migration. Our evaluation of these mitigations shows that they have a minimal impact on performance and fragmentation.

Contributions. We make the following contributions:

- We present Rubicon, a novel technique for massaging arbitrary data into arbitrary physical memory locations from an unprivileged process in Linux using three mechanisms: *PCP Evict*, *Block Merge*, and *Migratetype Escalation*.
- We use Rubicon to build the first deterministic x86 Rowhammer exploit and to significantly speed up the leakage of `/etc/shadow` with the RETBLEED attack.
- We propose mitigations with low impact on performance and fragmentation that directly address the low-level ZBA features that enable Rubicon.

Open sourcing. Rubicon is fully open source, with a focus on ensuring easy integration into diverse attack scenarios. Further details are available on the project webpage: https://comsec.ethz.ch/rubicon.

2. Background

We introduce Rowhammer (Section 2.1) and Spectre attacks (Section 2.2) before providing some background on virtual memory management (Section 2.3) and physical memory massaging (Section 2.4).

2.1. Rowhammer Attacks

Dynamic random-access memory (DRAM) is a memory technology commonly used as main memory. It stores bits of information in memory cells consisting of a capacitor and access transistor, organized in a matrix structure of rows and columns. Before a read or write request, the entire row is brought into a row buffer by the `ACT` command. From there, data can be accessed at a smaller granularity. Since capacitors leak charge over time, each DRAM cell has to be refreshed (at least) once per refresh window (e.g., every 64 ms for DDR4), achieved by periodically sending a `REF` command to the DRAM chip, which refreshes the cells' charge to ensure data integrity.

As the density of DRAM cells approaches physical limits, their ability to reliably retain charges diminishes [14]. In 2014, it was shown that attackers can reliably induce disturbance errors in *victim rows* by repeatedly activating, or *hammering*, neighboring *aggressor rows* [10]. This poses a significant threat as it allows attackers to leak data [12], [15] or corrupt security-sensitive structures like page tables [3], [16]–[18], SSH keys [9], or object references [19], [20], assuming they can reliably be placed into physical memory locations vulnerable to Rowhammer. In response, vendors have deployed in-DRAM mitigations referred to as Target Row Refresh (TRR). However, these mitigations were proven to be vulnerable and can be bypassed with special access patterns [11], [21]–[23].

2.2. Spectre Attacks

Modern CPUs rely on a performance optimization technique called *speculative execution* to reduce stalls due to unresolved branch instruction dependencies. Rather than waiting for the dependencies to resolve, *branch predictors* serve predictions of the upcoming control flow based on the recorded control flow history. The execution then speculatively continues along the predicted instruction path. When the branch instruction dependencies are finally resolved, the speculative results are committed to the architectural state if the prediction was correct. Otherwise, the results are squashed, and the execution restarts at the correct instruction instead.

Although incorrect computations are squashed, they leave microarchitectural traces behind (e.g., in caches) that can be leaked with side channels [24]–[26], as shown by Meltdown [27] and Spectre [28]. In Spectre attacks, an attacker manipulates branch predictors to speculatively execute code to reveal secrets such as passwords or encryption keys through a microarchitectural side channel. A recent example is RETBLEED which bypasses the originally proposed *retpoline* mitigation [29]–[31]. As many speculative execution attacks, RETBLEED achieves a low leakage rate (219 B/s on Intel Coffee Lake). Even worse, and typical in such attacks, the location of the secret information is unknown, which forces the attacker to scan the entire physical memory for a known part of the secret.

2.3. Virtual Memory Management

On modern CPUs, software operates on virtual memory, which is an abstraction layer on top of physical memory. The memory management unit (MMU) transparently translates virtual to physical addresses by traversing *page tables*. Page tables consist of *page-table entries* (PTEs) that reference *page frames*, which are fixed-sized blocks of contiguous physical memory (usually 4 KiB). The x86-64 architecture typically uses a 4-level page table structure, where the last-level PTE stores the physical address corresponding to the virtual address.

The virtual address space is orders of magnitude larger than the available physical memory and is exclusive to a single process. Demand-paging enables processes to physically back only those virtual pages that are accessed.

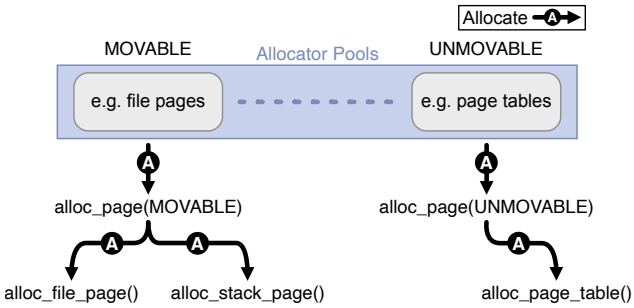

Figure 1: **Allocator pools.** Allocators partition memory into several pools tailored to serve allocations with specific requirements. In this example, one pool allows the underlying physical pages to be migrated (i.e., `MOVABLE`) while the other does not (i.e., `UNMOVABLE`).

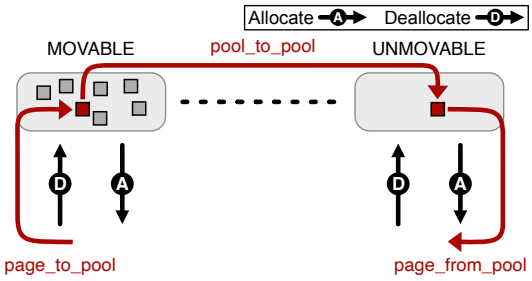

Figure 2: **Primitives for page-granular massaging.** `page_to_pool` places a target page inside a pool, `pool_to_pool` moves it between pools, and `page_from_pool` gets it back from a desired pool.

2.4. Physical Memory Massaging

Physical memory allocators inside an operating system are responsible for managing physical memory. Their main goal is to allocate memory efficiently to different processes or kernel components with specific requirements, such as specific physical memory ranges, physically contiguous memory, or physical memory that should not be moved (e.g., due to swapping). A common strategy of allocators is to partition memory into distinct pools tailored to allocation requirements, as shown in Figure 1: a general pool for user allocations and a specialized pool for kernel-level allocations.

Memory *massaging* is the process of manipulating memory allocation mechanisms so that allocations fall into predictable memory locations. As such, it is a common preliminary stage of many exploitation techniques [2]–[4], [9], [12], [19], [20], [32]–[39]. To achieve predictable memory allocations, the attacker may, for example, allocate and free large ranges of memory to defragment it, causing the memory allocator to return memory locations following predictable patterns. Once the memory allocator has been manipulated to issue predictable memory locations, the attacker can exploit this predictability to place the victim data at the targeted memory location.

Existing Rowhammer attacks rely on memory massaging for reliably placing the target data structure on a vulnerable location in physical memory, but they often rely on special memory management features. As an example, Drammer [3] relies on the ION allocator and Flip Feng Shui [9] on memory deduplication. Therefore, these attacks can (and have been) mitigated by simply disabling these special features. Instead of memory massaging, it is also possible for a Rowhammer attack to rely on spraying the target data structure in the hope that it lands on the desired physical memory location [16], [17]. Spraying results in attacks that are not always successful and requires creating multiple copies of the target data structure, which limits it to certain targets, such as page tables. Consequently, we currently lack a precise massaging technique using basic operations of the physical memory allocator that works on a variety of victims and systems.

3. Threat Model

We assume an attacker who is in control of an unprivileged process running on an up-to-date Linux kernel. The attacker intends to place a victim data structure in a specific block under their control. They can trigger the creation of at least one copy of the victim into memory at will. We do not assume any special memory management feature to be present such as page deduplication [9], user-level I/O allocators [3], or certain procfs files to help with physical memory massaging [12]. The additional attack-specific requirements of our examples are outlined below.

Rowhammer. We assume DRAM devices to be vulnerable to Rowhammer bit flips. Recent work shows that most DDR4 devices are vulnerable. We use a fuzzer to find effective patterns [21].

Spectre. We assume a CPU that employs speculative execution. Such attack on the kernel often provide a primitive for scanning kernel memory to look for a secret [6], [7], [13], [28]. As an example, we use RETBLEED [7] to show the benefit of precise memory massaging for Spectre-like attacks. We assume all mitigations against speculative execution attacks that were available at the time of the original study to be enabled, i.e., KPTI, retpoline, user pointer sanitization, and disabling unprivileged eBPFs.

4. Overview

Section 4.1 introduces a set of primitives that are needed to build our page-granular massaging technique Rubicon. Section 4.2 discusses how these primitives enable precise microarchitectural attacks, and Section 4.3 gives an overview of the challenges of implementing them using the operating principles of the pool allocator.

4.1. Page-Granular Massaging Primitives

We assume that the attacker is in control of a physical page `P`. The goal is to construct a massaging technique that enables an attacker to place the victim data on `P`. Assuming a generic pool-based allocator, we require three primitives to achieve this page-granular massaging technique shown in Figure 2: (i) a `page_to_pool` primitive to put the previously allocated page `P` back on a pool; (ii) a `pool_to_pool` primitive to move `P` between pools, if the attack requires victim data allocations from a different pool; and (iii) a `page_from_pool` primitive that enables the attacker to store the victim data on `P`.

4.2. Precise Microarchitectural Attacks

Microarchitectural attacks such as Rowhammer or Spectre, albeit quite different, would both benefit from our new page-granular massaging technique.

Rowhammer. To perform a privilege escalation Rowhammer attack, the attacker needs to find a page with a suitable bit flip that can corrupt a PTE. Once such a page `P` has been found, it is returned to the allocator with the `page_to_pool` primitive. Since page table pages are allocated from a different pool, the attacker invokes the `pool_to_pool` primitive to move `P` to the pool where page table pages are allocated from. Finally, the attacker invokes the `page_from_pool` primitive to store a page table page on `P`, achieving the desired objective.

Spectre. In a typical Spectre attack on the kernel, a sensitive file page (e.g., `/etc/shadow`) is leaked. Since the attacker does not know where this page is allocated, they need to scan the entire physical memory looking for it. Due to their low leakage rate, these attacks can take a long time before they can find the target page. An attacker equipped with our primitives releases any desired page `P` with the `page_to_pool` primitive. Since file pages are often allocated from the same pool, the attacker uses the `page_from_pool` primitive to land the desired file page onto `P`. Assuming that the physical address of `P` can be leaked, the attacker uses the side channel to directly leak from `P` instead of scanning the memory.

4.3. Challenges

Building page-granular massaging primitives on a real target, such as the Linux kernel, is complex. It requires effective methods to construct and implement these primitives within the constraints of a real-world operating system. This poses our first challenge:

> **Challenge (C1).** Exploit the memory (de)allocation operations in the Linux kernel to instantiate page-granular massaging primitives.

To tackle this challenge, we first provide in Section 5 a summary of the complex interactions in the ZBA, the heart of the Linux' kernel's memory management. Section 6 introduces three new massaging mechanisms to enable the primitives discussed in Section 4.1. We refer to this new page-granular massaging technique as *Rubicon*. Each of Rubicon's mechanisms has a specific goal and consists of multiple carefully designed massaging steps that together allow precise control over the page(s) to be allocated next.

To showcase that these primitives are powerful and widely applicable, we must prove their usefulness on real-world, end-to-end attacks. This is our second challenge:

> **Challenge (C2).** Prove the universality and power of Rubicon in real-world, end-to-end attacks.

In Section 7, we demonstrate Rubicon's potential in two real-world end-to-end attacks. First, we show how we build the first reliable and fully deterministic Rowhammer attack targeting PTEs on common x86 systems without requiring any special memory management features. Second, we explain how existing attacks can seamlessly integrate Rubicon, for which we take RETBLEED, a recent transient execution attack, as an example. We assess the performance and reliability of these attacks in Section 8.

The last challenge is mitigating Rubicon without intrusive changes in the way ZBA works while keeping the performance overhead negligible:

> **Challenge (C3).** Design, implement, and evaluate a lightweight mitigation against Rubicon.

We discuss in Section 9 why fully mitigating Rubicon in the ZBA or any deterministic pool allocator without reducing memory utilization is difficult. Nonetheless, we show that constraining Rubicon's three mechanisms with small and lightweight changes to the ZBA provides a good trade-off between security and memory utilization.

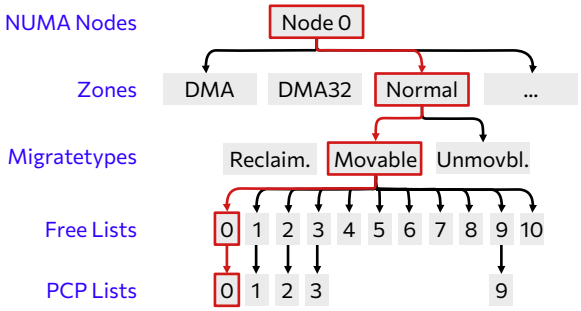

Figure 3: **Structure of the zoned buddy allocator.** The structure for managing physical memory in the Linux kernel, as an example for the NUMA node 0. *Zones* divide memory for different addressability requirements. *Migratetypes* describe a memory block's mobility. *Free lists* contain blocks with specific combinations of properties to satisfy allocation requests quicker. *PCP (per-cpu) lists* are blocks reserved for a certain CPU to avoid system-wide allocator locks. The highlighted path illustrates a list with properties matching a specific allocation request.

5. Zoned Buddy Allocator

The Linux kernel subsystem for managing and allocating memory at the page level is the *Zoned Buddy Allocator* (ZBA). In addition to directly serving memory to user processes and the kernel itself, it also feeds secondary allocators such as SLAB, SLOB, and SLUB, which handle allocations smaller than a single page. Therefore, the ZBA is a critical component of the kernel with significant implications for overall system security.

In this section, we provide an overview of the basic functionality of the ZBA. First, we discuss how the kernel organizes physical memory into a multi-level structure, designed to fulfill the requirements of various data structures (Section 5.1). Then, we explain how the ZBA enables efficient allocation of memory through the use of *free lists*, each designed to serve allocations with a specific combination of these requirements (Section 5.2). Finally, we outline how the system moves memory between these lists along a deterministic allocation path (Section 5.3).

Table 1: **Memory zones.** Overview of the Linux memory zones.

Zone	Usage
`DMA`	16-bit addressable memory.
`DMA32`	32-bit addressable memory.
`NORMAL`	Normally addressable memory.
`HIGHMEM`	Not directly addressable (high) memory.
`MOVABLE`	Limits `UNMOVABLE` allocations.

Table 2: **Migratetypes.** Overview of the migratetypes available in Linux.

Migratetype	Description
`UNMOVABLE`	Memory that cannot be moved, e.g., kernel mem.
`MOVABLE`	Memory that can be moved.
`RECLAIMABLE`	Memory that can be moved and reclaimed.
`HIGHATOMIC`	Reserved for high-order allocations.
`CMA`	Reserved for the Contiguous Memory Allocator.
`ISOLATE`	Non-allocatable memory.

5.1. Physical Memory Organization in the ZBA

The Linux kernel relies on various data structures that make specific assumptions about the properties of underlying physical memory. When these assumptions are violated, the data structures may break, potentially leading to a system crash. The kernel must therefore ensure that the allocated memory meets the requirements of the requesting data structure. To achieve this, it organizes physical memory into a multi-level structure, designed to maintain access to suitable memory at all times. Each level of the structure partitions the memory based on certain *properties* as depicted in Figure 3. At the highest level, there are Non-Uniform Memory Access (NUMA) *nodes*, dividing memory according to access latency for different CPUs in multiprocessor systems. Nodes are further divided into several *zones*, which are fixed ranges in memory separated by their addressability. `ZONE_DMA`, for example, contains memory addressable by DMA devices capable of using 16-bit addresses only. In Table 1, we provide an overview of all available zones.

Data structures residing in memory exhibit varying mobility properties due to their interactions with each other. For instance, files are mapped into the virtual memory space and can be conveniently relocated in physical memory by the kernel via a simple alteration of their PTE. However, relocating the page tables themselves would require modifying a higher-level page directory, rendering them unmovable to simplify page table management. As a result, the ZBA uses *migratetypes* to provide memory allocation for data structures with comparable mobility properties, as listed in Table 2. Unlike zones or nodes, which are fixed physical memory ranges, migratetypes are defined over *blocks*, i.e., groups of 2^{order} contiguous physical memory pages, where *order* ranges from 0 to 10. These blocks are created and managed by the buddy system. Whenever a block is released back to the allocator, it checks if any neighboring blocks of the same order are available, and if so, merges them into a larger block. If there are no more blocks of a requested order and migratetype available to satisfy an allocation request, larger blocks are repeatedly split in half to produce a block of the requested order. The Linux kernel therefore manages physical memory using a multi-level, tree-like structure to always keep suitable memory at hand.

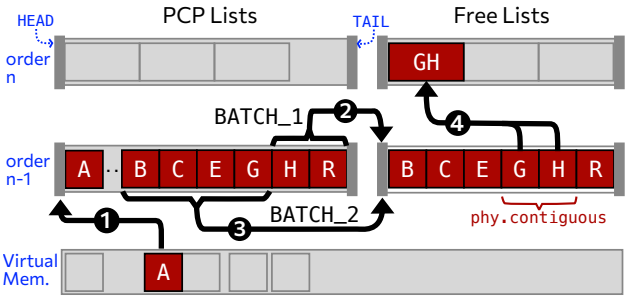

Figure 4: **Block deallocation.** Releasing a block pushes it to the head of its corresponding PCP list (❶). If the PCP list set exceeds its capacity, a batch (`BATCH_1`) is moved (order-preserving) to the head of the corresponding free list (❷), and the batch size is doubled (`BATCH_2`) s.t. the next time twice as many blocks are moved (❸). If any of the blocks pushed onto the free list has a free buddy, the ZBA merges them and pushes them onto a higher-order free list (❹; blocks G,H).

5.2. Allocator Lists

When receiving an allocation request, the ZBA is provided with a set of flags representing the purpose of the requested memory, from which the following properties are derived:

(a) the preferred **node** with suitable access time,

(b) limitations on the addressability that are used to choose the **zone**,

(c) information on the block's mobility that may affect the **migratetype**, and

(d) the requested block's (contiguous) size which determines the **order**.

Instead of searching for a block with specific properties upon an allocation request, the ZBA maintains a *free list* for each combination of properties, from which blocks can be allocated directly. The allocator uses the flags provided to traverse the tree-like structure (red in Figure 3) until it reaches the list matching the request's properties.

To minimize contention on the free lists, the ZBA also maintains *per-CPU pages* (PCP) lists for frequently used combinations of properties (depending on kernel version and configuration, e.g., Transparent Huge Pages). These lists serve as small, CPU-local caches populated with blocks drawn from the corresponding global free lists. Allocations are first attempted from these PCP lists, and only if they are empty, the system falls back to the global free lists. We will refer to the collection of all PCP lists associated with a single CPU within a given zone as a *PCP list set*, irrespective of the migratetype. To manage each PCP list set, the system maintains lightweight statistics, such as the set's capacity and the current number of cached blocks. When a PCP list is depleted, or the number of cached blocks exceeds the list's capacity, a *batch* of blocks is transferred between the PCP and the corresponding free list to keep the number of blocks within a desired range.

5.3. Block Allocation Pipeline

Putting these concepts together, we now explain how the ZBA handles block deallocations and allocations.

Block deallocation. Upon receiving a deallocation request, the ZBA pushes the block onto a list selected according to the block's properties (Figure 4-❶). This

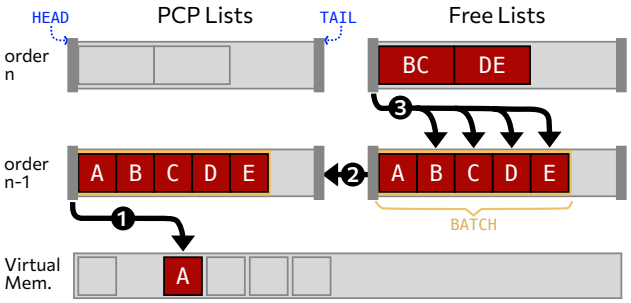

Figure 5: **Block allocation.** The allocator first tries fulfilling an allocation request using the block at the head of the list selected according to the request's properties (❶). In case the selected list is a PCP list and it is empty, it is filled by moving a batch of blocks from the free list of matching properties (❷). Empty free lists, are filled by splitting up blocks from the next higher order corresponding free list (❸).

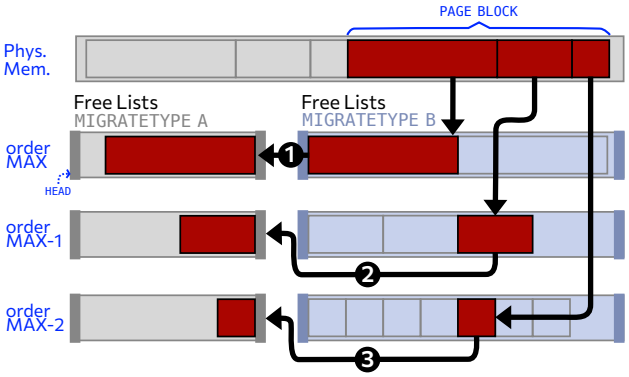

Figure 6: **Block stealing.** When all pages of a certain migratetype run out, the ZBA steals a block of the highest available order from another migratetype (❶). Thereafter, the ZBA performs a heuristic check (Listing 1) to determine whether to steal the whole page block. If successful, all other free blocks in that page block are stolen as well (❷ and ❸).

```
if (order >= pageblock_order / 2
   || start_mt == MIGRATE_RECLAIMABLE
   || start_mt == MIGRATE_UNMOVABLE
   || page_group_by_mobility_disabled)
  return true;
```

Listing 1: **Page block stealing condition [40].** The heuristic condition used in `can_steal_fallback` of `mm/page_alloc.c` of the Linux kernel to decide whether to steal a whole page block.

Table 3: **Victim data structures.** Examples of victim data structures, attacks they were used in, their migratetype, and their order. The variations in migratetypes and orders highlight the importance of a universally applicable massaging technique.

Data Structure	Existing Attacks	Migratetype	Order
Page Tables	Seaborn et al. [16], Drammer [3], Half-Double [17]	`UNMOVABLE`	0
Page Cache	RETBLEED [7], Flip Feng Shui [9], BlindSide [5]	`MOVABLE`	0
Kernel Stack	SpecHammer [38]	`UNMOVABLE`	1 (x86)

could be either a PCP list or a free list, with slightly different behaviors. For PCP lists, the ZBA must ensure they stay within a certain capacity to avoid unnecessarily committing memory to a specific CPU. If freeing the block causes the PCP list set to exceed its capacity, a *batch* of blocks is moved from the tail of the PCP list onto the head of the corresponding free list (Figure 4-❷). The batch size is calculated based on the following formula: $b \cdot 2^F$, where b is a base size and F is a free factor that is incremented each time the set is overfilled. This means that the next time the PCP list set is overfilled, twice as many blocks are moved (Figure 4-❸). During periods of frequent deallocations, this exponential growth in batch size reduces the frequency of accesses to the free list, thereby improving scalability. If the batch exceeds the number of blocks on the selected list, the ZBA starts emptying other lists in the same PCP list set, one at a time, in a round-robin fashion. Finally, as blocks are released onto a free list, the ZBA searches for their buddies and merges them if they are available (Figure 4-❹). In the absence of a PCP list, the pages are pushed onto the free list directly and merging is performed immediately.

Block allocation. Upon receiving an allocation request, the ZBA attempts to allocate a block from a list selected based on the required properties. If the selected list is non-empty, the allocation is straightforward: the block is simply taken from the head of the list (Figure 5-❶). However, if the list is empty, the allocator's behavior depends on whether it is a PCP list or a free list. Empty PCP lists are refilled in batches from the corresponding free lists (Figure 5-❷). When the free list itself is empty, the ZBA refills it by splitting blocks from higher-order free lists, following the textbook buddy algorithm (Figure 5-❸). For properties with no PCP list, the ZBA allocates blocks directly from the free list, splitting higher-order blocks as needed.

Block stealing. As a last resort, when there are no blocks of the desired migratetype available, the ZBA attempts to *steal* blocks from the free lists of other migratetypes (Figure 6). However, since blocks of different migratetypes cannot be merged, stealing small blocks can lead to severe fragmentation. The ZBA, therefore, prefers stealing the largest available blocks, as this helps keep neighboring pages of the same migratetype. To further reduce fragmentation, physical memory is divided into 2 MiB segments, referred to as *page blocks* (not to be confused with buddy blocks described in Section 5.1). The ZBA tries to keep all pages within a given page block assigned to the same migratetype. Hence, all stolen (buddy) blocks revert to the migratetype of their page block upon deallocation.

To accommodate dynamic workload changes, the system uses a heuristic condition (Listing 1) to reassign page block migratetypes. This check is triggered during block stealing, which serves as a signal of pressure on the desired migratetype (Figure 6-❶). If the condition is met, the system reassigns the surrounding page block's migratetype and steals all other free blocks within it (Figure 6-❷ and ❸). This ensures that even if a large block cannot be allocated, blocks of different migratetypes do not spread throughout memory.

6. Rubicon

In this section, we introduce *Rubicon*, a novel memory massaging technique that implements the high-level primitives presented in Section 4.1 for the ZBA in the Linux kernel. Rubicon exploits the deterministic behavior of the

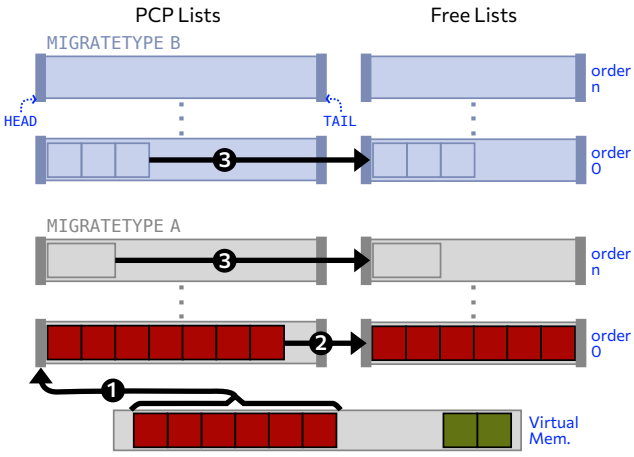

Figure 7: **PCP Evict.** We evict a PCP list set by overfilling it with a large amount of memory through one of its lists (❶). The ZBA starts releasing batches of blocks from the set to limit their total, initially taking blocks from the list we used to overfill the set (❷). Subsequent batches grow in size, eventually outgrowing that list and even the whole list set, thus releasing all blocks from their respective lists (❸).

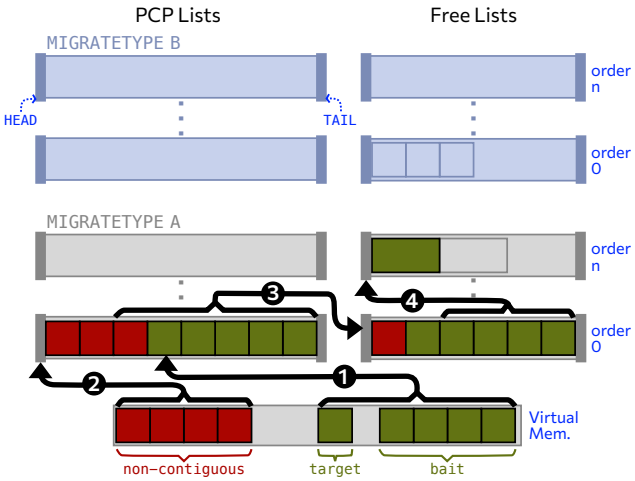

Figure 8: **Block Merge.** We release the target block, initially split into pages (❶) and push it onto the free lists by releasing additional memory (❷). When the pages reach the free list (❸), they are merged into a single block and moved to the head of the higher-order free list (❹).

ZBA, which always allocates memory from the head of a specific list selected based on the properties (e.g., order, migratetype) required by the recipient data structure. By positioning blocks at the heads of these lists, Rubicon ensures that subsequent allocations, such as those for victim objects, land at attacker-controlled locations. As potential victim data structures can have an arbitrary combination of properties (Table 3), Rubicon must be able to massage blocks onto any list in the ZBA. Rubicon achieves this through three mechanisms:

- **PCP Evict (Section 6.1)** clears all blocks from the PCP lists, forcing the allocator to fetch attacker-controlled blocks from shared free lists. This implements the `page_from_pool` primitive.
- **Block Merge (Section 6.2)** places a target block at the head of a desired free list within the same migratetype, positioning it to be allocated. This implements the `page_to_pool` primitive.
- **Migratetype Escalation (Section 6.3)** moves blocks across different migratetypes by exploiting the page block stealing condition (Listing 1). This implements the `pool_to_pool` primitive.

6.1. Mechanism #1: PCP Evict

Rubicon moves blocks to the head of free lists to place them in line for the next victim allocation. However, the ZBA prioritizes allocations from PCP lists instead (Figure 5-❶). It only falls back to a free list when the corresponding PCP list is empty (Figure 5-❷). To this end, we present *PCP Evict*, a mechanism that evicts all blocks from the PCP lists, forcing the allocator to fetch blocks from the shared free lists.

PCP Evict exploits the dynamic batch size scaling described in Section 5.3 to evict all blocks from the PCP lists. We begin by releasing a large number of pages to deliberately overfill the PCP list set (Figure 7-❶). This triggers a batch release of blocks from the overfilled PCP list to the corresponding free list. The batch size is initially small, so all released blocks originate from a single list (Figure 7-❷), but with each subsequent batch, the size doubles. After several iterations, the batch size exceeds the total capacity of all PCP lists in the set, causing all remaining blocks to be released (Figure 7-❸). At this point, the PCP lists are empty[1], so the next allocation will be drawn from the free list, thus implementing our `page_from_pool` primitive. Releasing more memory than strictly necessary simply repeats this cycle multiple times, with no adverse side effects. This makes the mechanism highly robust, as it tolerates wide overestimation while still ensuring that the PCP lists are reliably emptied.

6.2. Mechanism #2: Block Merge

With PCP Evict, we ensure that the next allocation will be drawn from a free list. To complement this, we introduce *Block Merge*, a mechanism that moves the target block to the head of that list (within the same migratetype), thus ensuring it is used for the next allocation.

Block Merge exploits a key property of the ZBA: blocks that are merged during deallocation are always inserted at the head of a free list (Figure 4-❹). We begin by releasing the target block, which initially exists as a collection of individual pages in virtual memory. Upon release, these pages are moved to the order-0 PCP list (Figure 8-❶). To push them onto a free list, we continue releasing additional memory (Figure 8-❷). As the PCP list reaches capacity, it begins to evict pages from its tail, moving them to the corresponding free list (Figure 8-❸). Eventually, the target pages are evicted as well, at which point they are merged into a contiguous block and moved to the head of the appropriate higher-order free list (Figure 8-❹). This puts the target block in line for the next allocation, thus implementing our `page_to_pool` primitive. The amount of memory used

1. Since PCP lists are per-CPU, PCP Evict must either be performed on all CPUs or on the same CPU used for the victim allocation. This is not a major limitation, as most existing attacks target data structures allocated via system calls or standard library functions, which are colocated with the attacker's process (Section B).

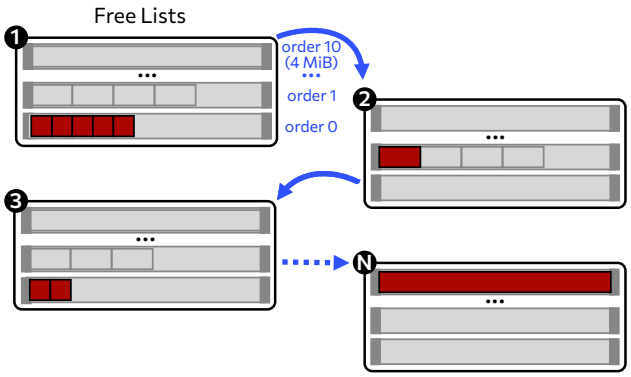

Figure 9: **Contiguous memory allocation.** We begin by allocating a large amount of memory, nearly exhausting the system's capacity, which initiates a deterministic sequence of operations within the ZBA. To minimize fragmentation, the allocator first serves the smallest available order-0 blocks (❶). As these are depleted, larger blocks are recursively split to fulfill the allocation request (❷→❸), progressively consuming all available orders. Eventually, the allocator must split order-10 4 MiB blocks (Ⓝ), making the final portion of the allocated memory area consist of contiguous blocks. By selectively drawing memory from the end of the area and releasing the rest, we can reliably obtain contiguous blocks from userspace without relying on transparent huge pages (THPs) or `procfs`.

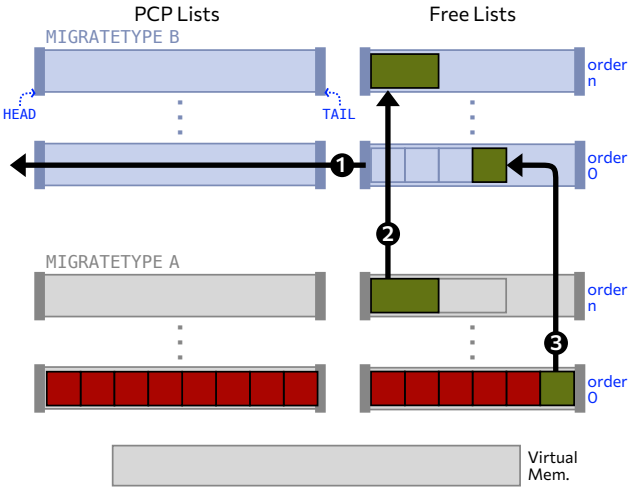

Figure 10: **Migratetype Escalation.** Building upon Figure 8, where we placed the target and bait block onto correct free list, we depict here the rest of migratetype escalation. After repeatedly triggering allocations of the desired migratetype (❶), the desired migratetype runs out of blocks and steals our bait block (❷). Since our bait block satisfies the page block stealing condition, our target page is also stolen (❸).

to push the target onto the free list does not have to be precise, it is sufficient if it exceeds the capacity of the PCP list set. This allows us to overestimate it by a wide margin, which makes this mechanism very reliable.

Importantly, none of the additionally released pages can be merged into a block of the same order as our victim. If such a merge were to occur, the resulting block would be placed ahead of the target in the free list. To prevent this, we utilize the method illustrated in Figure 9 to allocate physically contiguous blocks. By releasing every other page from these blocks, we ensure that their buddies remain unavailable, effectively preventing any merging.

6.3. Mechanism #3: Migratetype Escalation

Until now, our focus has been solely on massaging victims within a single migratetype. However, many victim data structures targeted in real-world attacks (Table 3), do not match the migratetype returned by user-accessible allocators (e.g., `MOVABLE` for `mmap`). To overcome this limitation, we introduce *Migratetype Escalation*, a new massaging mechanism that enables deterministically moving blocks between different migratetypes.

Migratetype Escalation exploits the page block stealing condition described in Listing 1, specifically the fact that stealing a sufficiently large block causes the entire surrounding page block, and thus all smaller blocks within it, to be stolen. We begin by taking control of an entire page block, which can be achieved as shown in Figure 9. We then split this page block into two halves, one of which we use as *bait* to trigger the stealing condition, while the other half can be used to place the target block as needed. To ensure that our bait block becomes the largest available candidate, we eliminate all larger free blocks either through fragmentation or allocation. Using Block Merge, we move the target and bait block onto their respective free lists. With the blocks in place, we can start to repeatedly trigger allocations of the desired migratetype (Figure 10-❶). Importantly, we do not need to allocate the actual victim; any allocation that consumes memory of the same migratetype suffices. For instance, to target the `UNMOVABLE` migratetype, we can spray page tables by mapping pages at 2 MiB intervals, as demonstrated in Drammer [3]. As we continue triggering allocations, the desired migratetype eventually runs out of blocks, forcing the ZBA to resort to stealing. Since we have removed all larger blocks, it steals the provided bait block (Figure 10-❷) and checks the page block stealing condition (Listing 1). Our bait block was formed from half of a page block, so it always satisfies the condition `order >= pageblock_order/2`. Consequently, the entire page block and all free blocks within it, including the released target block, will also be stolen and assigned the desired migratetype (Figure 10-❸), which implements our `pool_to_pool` primitive.

7. Rubicon in Practical Attacks

We use Rubicon to build the first deterministic x86-64 privilege escalation Rowhammer attack (Section 7.1) and significantly speed up a Spectre attack (Section 7.2).

7.1. Deterministic Rowhammer on x86-64

Similar to previous work [3], [9], [12], we divide the attack into three steps: ① memory templating, ② memory massaging, and ③ exploitation. We discuss these steps more in detail next.

① Memory templating. If a Rowhammer access pattern triggers a bit flip in a vulnerable memory location, repeating the access pattern likely causes the same bit flip(s) again. We template the memory using the Blacksmith Rowhammer fuzzer to generate TRR-evading access patterns and find repeatable bit flips [21]. The original fuzzer uses 1 GiB superpages as backing memory. As we do not want to rely on any special memory features, we

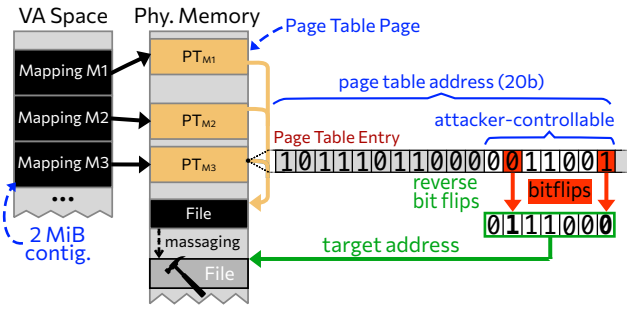

Figure 11: **File remapping.** We remap a file multiple times (**Mapping M1-M3**) with 2 MiB offsets to spray page table pages ($\mathbf{PT_{M1\text{-}3}}$). We then check if any of the templated bit flips are useful, i.e., fall into the **attacker-controllable** address part of the PTE belonging to the mapping. If so, we **reverse the bit flips** and combine the resulting address with the file mapping's address to determine the **target address** for massaging the file. Hammering the file at this address with the previously determined pattern will give us access to the page table's page.

modify the fuzzer to use virtual memory backed by 4 MiB contiguous blocks, obtained as described in Figure 9.

② **Memory massaging.** The attack's goal is corrupting a page table entry such that it maps its own page table, as depicted in Figure 11 [3]. Once we find a vulnerable memory location in one of the 4 MiB blocks, we apply the inverse of the bit flip(s) to the known part of the vulnerable page's physical address. If the bit flip direction allows the resulting address to become the vulnerable page again, we can massage it into a PTE at the vulnerable location. Using Block Merge, we first free and reallocate the order-0 page of the resulting address with a file, as described in Section 6.2. Using a file, we can spray page tables that all reference this file. We control the location of the victim PTE within the victim page table by controlling the virtual address of the new mappings to the file.

Because the migratetype of page tables is different (`UNMOVABLE`), we use Migratetype Escalation so that the victim page (`MOVABLE`) can become a page table. We then spray page tables through 2 MiB-offsetted file mappings until the victim page gets recycled as a page table.

③ **Exploitation.** As our file resides in a carefully chosen location, retriggering the bit flip(s) makes the victim PTE point to its own page table. We identify the corrupted mapping as it is the file mapping that no longer contains the file contents. This gives us access to a page table through the file mapping, resulting in arbitrary read and write access to the entire system memory.

7.2. Accelerating Spectre with Rubicon

We discuss how Rubicon can accelerate Spectre attacks against the operating system. Such attacks demonstrate their capability by leaking the `/etc/shadow` file as an unprivileged user [5]–[7], [13]. Our key observation is that the majority of time is spent on scanning for this file in physical memory. Given the limited and inaccurate side channel of Spectre attacks, being able to use Rubicon to massage the secret file into a predetermined memory location could accelerate such attacks substantially.

Prerequisites. To allocate `/etc/shadow` in memory, we trigger a SUID binary to open and read `/etc/shadow`. The SUID binary we use for this is `expiry` of the `shadow-utils` project, available in most Linux distributions. We also confirmed that the attack works with other SUID binaries that read `/etc/shadow`, such as `passwd`. Because files remain allocated in the *page cache* of Linux after they are closed, we first evict all pages in the page cache. By causing memory pressure on the system, we ensure that `/etc/shadow` is evicted from the page cache. To make as few allocations as possible when invoking `expiry`, we read all the shared objects that `expiry` depends on (thereby caching them in the page cache) before invoking `expiry`. This way, only a few page allocations are necessary before `expiry` allocates `/etc/shadow`, which remains in page cache after `expiry` exits.

To determine the physical address that `/etc/shadow` will be allocated to, we leak a physical address before releasing it in such a way that it gets assigned `/etc/shadow`. Obtaining a physical address of a given memory page is possible using Spectre itself [7].

Massaging. Using the allocation method in Figure 9, and by knowing a single physical address inside a contiguous memory block, we can infer the beginning of the block and, thereby, all physical page addresses within it. Using Block Merge from Section 6.2, we release pages from the contiguous block to the PCP list that will be used to allocate `/etc/shadow` when we execute `expiry`. Releasing pages from the contiguous block ensures that `/etc/shadow` lands within. The actual page of `/etc/shadow`, however, may be a few pages off from the expected one. In our experiments, the allocation can occur up to 16 pages away from the expected page, primarily due to ASLR and stack randomization.

Table 4: **Microbenchmark results.** We report the success rate (**Succ. Rate**) for each of our mechanisms for a number of repetitions (**#Reps.**) that is chosen based on the average execution time (**Avg. Duration**).

Primitive	#Reps.	Succ. Rate	Avg. Duration
PCP Evict (§ 6.1)	1 000	100%	736.805 µs
Block Merge (§ 6.2)	100 000	100%	10.046 µs
Migratetype Escal. (§ 6.3)	100	100%	5976.37 ms

8. Evaluation

In this section, we assess the performance of our massaging mechanisms (Section 8.1), our end-to-end Rowhammer attack (Section 8.2), and how Rubicon improves the RETBLEED attack (Section 8.3).

8.1. Massaging Mechanisms

We first perform three *microbenchmarks*, each specifically designed to test one of the massaging mechanisms introduced in Sections 6.1 to 6.3. The results for all three mechanisms are summarized in Table 4.

PCP Evict. In this test, we measure the speed and success rate of the PCP eviction mechanism. To evict, we allocate and release a sufficiently large number of pages as described in Section 6.1. We establish a baseline by checking the total number of all pages on the PCP list. We then instantiate a single page table. If the total number

Table 5: **DDR4 DIMMs.** We report the SPD data of the DDR4 DIMMs used in the evaluation of the Rowhammer attack. Devices indicating a *n/a* as manufacturing Date (Mf. Date) did not report any date. We report the DIMM's frequency (Freq.), size, and geometry (Geom.), i.e., the number of ranks (RK), bank groups (BG), banks per bank group (BA), and rows (R). All DIMMs are from vendor Samsung.

ID	Mf. Date [MM-YYYY]	Freq. [MHz]	Size [MiB]	DIMM Geom. (RK,BG,BA,R)
1	03-2020	2666	8192	(1, 4, 4, 2^{16})
2	*n/a*	2132	8192	(1, 4, 4, 2^{16})
3	*n/a*	2132	8192	(1, 4, 4, 2^{16})
4	*n/a*	2132	8192	(1, 4, 4, 2^{16})
5	*n/a*	2666	8192	(1, 4, 4, 2^{16})

Table 6: **Blacksmith results on 4 MiB blocks.** For each of the tested devices (ID), we report in both cases the number of patterns found during fuzzing ($\mathbb{P}_F$), the number of bit flips found during fuzzing ($\mathbb{F}_F$) and during the best pattern's sweep ($\mathbb{F}_S$). For the latter we additionally report the number of one-to-zero bit flips ($\mathbb{F}_{S,1\to0}$).

ID	Contiguous Memory				4 MiB Memory Blocks			
	$\mathbb{P}_F$	$\mathbb{F}_F$	$\mathbb{F}_S$	$\mathbb{F}_{S,1\to0}$	$\mathbb{P}_F$	$\mathbb{F}_F$	$\mathbb{F}_S$	$\mathbb{F}_{S,1\to0}$
1	47	1 061	82 183	41 471	117	7 002	87 607	44 589
2	42	7 771	113 190	57 665	111	14 524	95 958	48 479
3	102	17 790	98 425	49 296	73	6 696	81 127	40 181
4	66	3 415	32 090	15 988	86	6 267	50 860	25 207
5	126	12 689	80 601	40 876	169	3 269	56 584	28 969

increased, it means that the system had to move a batch onto the PCP lists to fulfill the allocation request. This confirms we have successfully evicted the PCP list. On the other hand, if the count decreased, it indicates there were still unmovable pages of order 0 on the PCP list and eviction failed. We achieve a success rate of 100% over 1 000 repetitions with an average duration of 736.805 µs.

Block Merge. In this test, we measure the reliability of our block merging mechanism (see Section 6.2) by repeatedly massaging a temporary file. The test follows the procedure for massaging `/etc/shadow` as outlined in Section 7.2, except that we now read the massaged file directly instead of through an SUID binary. This allows us to test Block Merge in a noise-free environment. To perform the test, we release a single target page onto the PCP list using Block Merge and populate it with our file by mapping it immediately afterwards. We confirm the success of each test repetition by comparing the physical addresses (obtained via `pagemap`) of the target page and the file after massaging. We achieve a success rate of 100% over 100 000 repetitions with an average duration of 10.046 µs, thus proving deterministic memory massaging.

Migratetype Escalation. In this test, we evaluate the reliability of the migratetype escalation mechanism by massaging a page table into a specific page frame. We have chosen a setup similar to the one used in the Rowhammer attack, namely:

(a) we allocate a single page block as described,
(b) we exhaust memory to remove blocks that are larger than 1 MiB,
(c) we push both the bait block and the target page frame onto their free lists, and finally,
(d) we instantiate a sufficient number of page tables such that the bait block is stolen.

We check whether a repetition has been successful by directly reading physical memory through `/dev/mem` and comparing the returned value with an expected page table entry. To ensure our mechanism is tested in a worst-case scenario, we lock the page frames adjacent to the target. We achieve a success rate of 100% over 100 repetitions with an average duration of 5976.37 ms.

Portability. We tested our massaging mechanisms on a newer kernel version (6.8.0-51) and found no significant differences regarding performance and reliability.

Stability under load. We evaluated the performance of our memory massaging mechanisms under parallel memory allocations using `stress-ng` in its `vm` stressor mode. The system was stressed with four threads and memory allocations of up to 128 MiB. Despite the increased load, the results were consistent with those from the single-threaded tests.

8.2. Rowhammer Attack

We evaluate our end-to-end Rowhammer attack on the five DDR4 DIMMs listed in Table 5 on a system equipped with an Intel Core i7-8700K, running Ubuntu 18.04 (kernel 5.4.0-125). Before evaluating our attack, we compare the original Blacksmith using 1 GiB contiguous memory to our modified version using 4 MiB memory blocks only. The results in Table 6 show a comparable performance for our modified version. On 4 out of 5 DIMMs, we find a higher number of patterns during the fuzzing run. The number of bit flips is comparable but expectedly, on average slightly lower (8.45%) than the original version. These results demonstrate that Rowhammer remains effective on smaller 4 MiB memory blocks.

Load levels. In order to thoroughly assess the attack's reliability, we repeatedly run it under various load conditions. We employ *stress-ng's* default CPU workers (which cycle through all of the available CPU tests) to synthetically generate three varying levels of system load:

(a) idle: stress-ng is turned off;
(b) medium: we use 8 workers to load two-thirds of the available CPU threads; and finally,
(c) high: all of the available threads are fully loaded by 12 workers.

We perform 20 repetitions per stress level on a test pool consisting of five DDR4 DIMMs with in-DRAM Rowhammer mitigations.

Results. The results of the evaluation can be found in Figure 12. On an idle system, we achieve an average success rate of 87%. As we increase the system load to medium and high levels, Blacksmith was unable to trigger any bit flips on DIMMs 4 and 5, respectively. Since our main focus is the reliability of our massaging primitive, we can conclude that it is at least equally robust against interference as Rowhammer on the affected DIMMs. For the rest of the test pool, the success rate remained stable at medium load (83.75%) and even increased at high load (91.7%). Excluding the runs where Blacksmith could not trigger any bit flips due to increased load on the system, we achieve an overall success rate of 87.3% over 240 runs.

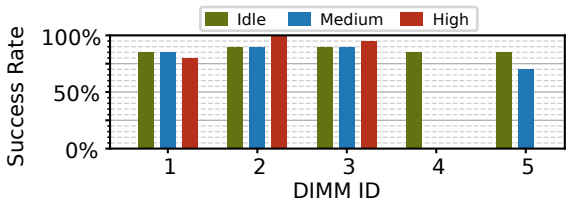

Figure 12: **Impact of system load on success rate.** For each DIMM, we report the success rate under three stress levels: idle (■), medium (■), and high (■). The results show that the reliability of our massaging primitive remains constant under increased system load. Missing bars indicate the inability to trigger any bit flips.

Discussion. It is important to note that our massaging primitive was always successful in placing the victim page table into the vulnerable page frame. The failed attempts were caused solely by failing to reproduce the templated bit flips on the victim due to the additional load. Hence, we did not record any crashes due to bit flips on data structures other than page tables during our evaluation. Additionally, we ran the attack alongside stress-ng's default memory stressors and the WebKit regression test suite, which served as a realistic workload. Nevertheless, the attack remained fully functional.

8.3. RETBLEED Attack

We evaluate the total duration of RETBLEED to leak `/etc/shadow` on an AMD EPYC 7252 (Zen 2, microcode `0x8301038`) and an Intel Core i7-8700K (Coffee Lake, microcode `0xea`). Both systems run Ubuntu 20.04 with Linux kernel 5.8.0-63, allowing us to reuse the disclosure gadgets presented in the original paper [7]. Rubicon improves RETBLEED by providing contiguous physical memory on systems where THPs are unavailable and by massaging the secret (i.e., `/etc/shadow`) into a known location. We evaluate the following scenarios:

(a) the baseline configuration with THPs, the same as in the original RETBLEED work;
(b) the baseline configuration without THPs, which makes use of contiguous memory; and finally,
(c) Rubicon, which makes use of both contiguous memory and `/etc/shadow` massaging.

Results. The results in Table 7 show the median time to leak `/etc/shadow` over 10 invocations. The Intel baseline only depends on THPs to leak a physical address, which turned out to be faster with 4 KiB pages in the baseline configuration without THPs. The AMD variant became slower in this configuration because of the overhead caused by the contiguous memory allocation. Both Intel and AMD variants show a significant improvement with Rubicon, indicating a 6.8× and 284× speedup on AMD and Intel, respectively.

As we have shown, precise control over physical memory allocations provided by Rubicon leads to significant performance improvements on Spectre-like attacks and enables new reliable Rowhammer attacks. It is, therefore, important to design mitigations that prevent Rubicon's mechanisms, which we will present in the next section.

Table 7: **Leaking `/etc/shadow` using RETBLEED.** Median execution time of all three test configurations. As shown, disabling THPs (Baseline, No THPs) does not have a significant impact on the results. The Rubicon-accelerated attack with massaging `/etc/shadow` shows a dramatic speedup of 6.8× and 284× on AMD and Intel, respectively.

CPU Model	Baseline		Rubicon
	THPs	No THPs	No THPs
Intel Core i7-8700K	45 m 2 s	41 m 52 s	9.5 s
AMD EPYC 7252	3 m 10 s	5 m 40 s	27.9 s

9. Mitigation

We discuss, implement, and evaluate possible strategies for mitigating Rubicon. Since the ZBA is a core component of the kernel, we focus on achieving security guarantees with minimal performance overhead. We examine the design flaws that enable the massaging mechanisms introduced in Section 6 and propose suitable mitigations. First, we prevent PCP list eviction by keeping an individual block count for every PCP list (Section 9.1). Next, we fully mitigate migratetype escalation, and thus the attack introduced in Section 7.1, by replacing the insecure page block stealing condition (Section 9.2). After that, we discuss randomization as a possible mitigation of deterministic memory massaging (Section 9.3). Finally, we demonstrate the viability of these mitigations by evaluating their impact on system performance with several benchmarks (Section 9.4).

9.1. Individual List Counters

When a PCP list set reaches its capacity, the batch of blocks released onto the free lists is capped to prevent releasing more blocks than are currently in the set. However, due to the lack of individual list counters, the cap cannot be restricted to the number of blocks on the specific list that caused the set to overflow. As a result, all PCP lists within a set are emptied once the batch grows sufficiently large. We exploited this to create PCP Evict, as described in Section 6.1.

To mitigate this, we propose maintaining a separate block counter for each PCP list. This ensures that when memory is released onto a PCP list and the set becomes overwhelmed, only the blocks from the specific list that caused the overflow are released, leaving the others untouched. By enforcing this restriction, we prevent the inadvertent eviction of blocks from other lists within the set and effectively eliminate the possibility of PCP Evict.

9.2. Secure Stealing Condition

Migratetype escalation is enabled by the page block stealing condition (Listing 1). The condition is always evaluated when a block is stolen across migratetypes. If the condition is met, the page block's migratetype is changed, and *all* free blocks within that page block are stolen. This is done so that the block's migratetype can become homogeneous again as soon as all blocks still in use (and with the original migratetype) are released. As demonstrated in Section 6.3, this mechanism can be

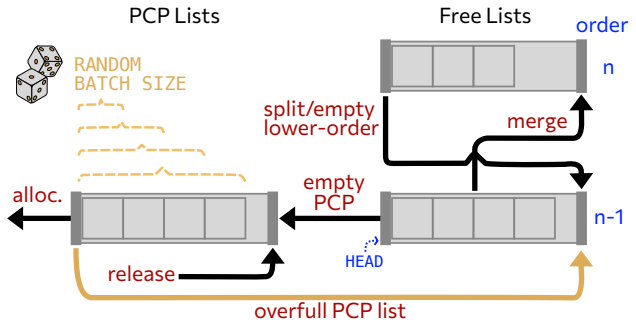

Figure 13: **Reordered allocation pipeline in our mitigation.** We fully randomize the position by reordering the allocation pipeline to eliminate the determinism of allocations and stop our attack mechanisms. We show for all supported events (e.g., merge) the flow of blocks between the different lists.

exploited to deterministically move blocks between migratetypes that would otherwise be unlikely to be stolen, such as individual pages.

To address this issue, we propose removing the heuristic condition and instead only stealing page blocks as a whole. Since the condition is exclusively used in case of severe fragmentation or lack of memory, its removal has no effect on performance under normal operation. Furthermore, eliminating this condition does not explicitly prevent small blocks from being stolen. Instead, it ensures blocks are stolen individually, preventing multiple blocks within a page block from being stolen together in a manipulable way, which completely eliminates Migratetype Escalation.

9.3. Batch Randomization

In contrast to the two other mitigation mechanisms (Sections 9.2 and 9.3), mitigating Block Merge requires addressing a deep-rooted vulnerability of deterministic memory allocators. In short, deterministic memory allocators are finite state machines operating on a state shared across processes and privilege domains. Given their deterministic nature, each operation has a predictable effect on the state. An attacker can, therefore, perform a sequence of memory operations that bring the allocator to a state where it is certain to use a desired target block for a specific victim allocation. They can then trigger the allocation in another process or a higher privilege domain, e.g., through a syscall. The allocator, being deterministic, predictably acts on the state and allocates the desired target block.

This vulnerability can be mitigated using randomization [41], provided there is enough entropy to sufficiently slow down attackers. Our goal is to randomize the position of the released block within its PCP list to prevent deterministic reuse. Since PCP lists are implemented as linked lists, randomizing the position of a released block would involve traversing a list, which is linear in complexity and thus unacceptable for performance-critical operations such as memory (de)allocations. Instead, we can closely emulate a fully randomized position by reordering the allocation pipeline as shown in Figure 13. By placing the released block at the tail of the PCP list, we force the attacker to remove an unknown number of blocks from the PCP list before reaching the target block. Under normal operation, the randomness comes from the system itself as the initial state of the PCP list is unknown. However, to stop attackers from trying to remove the randomness by either exhausting or overflowing the PCP list, we randomize the batch size of blocks transferred between the PCP lists and free lists. The batch size B is sampled from a discrete uniform distribution $B \sim \mathcal{U}\{B_{min}, B_{max}\}$ where B_{min} and B_{max} denote the minimum and maximum batch sizes, respectively. Consequently, the probability of an attacker correctly guessing the batch size is given by $1/(B_{max} - B_{min} + 1)$.

9.4. Evaluation

We implemented these three mitigations in the Linux kernel version 5.15.0 (the latest version at the time of implementation), running on Ubuntu 20.04. To demonstrate their effectiveness, we started our evaluation by rerunning the testers introduced in Section 8.1. All testers were run for the same number of repetitions, yielding a success rate of 0% and confirming that our mitigations indeed prevent Rubicon. Afterwards, we measured their performance overhead by running the following benchmarks in the Phoronix test suite [42]:

(a) Pmbench (1.0.2) to evaluate the system's paging performance,
(b) UnixBench (byte-1.2.2) to evaluate general system performance, and finally,
(c) TensorFlow Lite (1.1.0 mobilenet_v1_1.0_224.tflite) as a real-world memory-heavy workload.

The performance of our mitigated kernel was within 0.1% of the stock kernel for all three benchmarks. This falls well within the margin of error, demonstrating the low cost of our mitigations.

10. Discussion

We discuss observations made throughout our work on Rubicon and suggest possible future directions.

Security implications of procfs. System interfaces like `/proc/zoneinfo` and `/proc/buddyinfo` provided by the process filesystem (procfs) offer valuable insights into how different workloads affects memory usage and fragmentation. While such details can be useful for software optimization, they also expose the internal state of the ZBA to unprivileged users. Rubicon does not rely on these interfaces for its massaging primitives, but these interfaces could be used to squash any attempt at using randomization techniques to prevent memory manipulation. As we believe that benign applications do not require access to such information, we strongly recommend restricting access to users with elevated privileges.

Investigating possible further use cases. One of the core contributions of this paper is the generality and adaptability of the presented massaging primitives. They can be used as building blocks of more complex massaging procedures. As such, they facilitate experimentation with massaging where researchers can implement and test a specific procedure in a matter of hours. Therefore, we feel this paper opens up the opportunity of finding unexpected ways of using memory massaging. This includes,

assessing their applicability outside of hardware attacks, for example, in traditional software attacks.

Rubicon in the cloud. The cloud represents a high-value target for Rubicon, as it hosts critical data and services. While the primitives outlined in Section 6 remain applicable in the cloud scenario, they require alternative implementations. Such implementations, however, are challenging as hypervisors employ specialized resource-sharing mechanisms to manage physical memory, which influence how memory is allocated and deallocated. Large contiguous memory blocks cannot be easily exhausted due to resource limit enforcement, and memory management techniques such as balloon drivers complicate deallocation for guest machines [43].

11. Related Work

In this section, we discuss work related to Rowhammer (Section 11.1), Spectre (Section 11.2), and physical memory massaging (Section 11.3).

11.1. Rowhammer

The Rowhammer vulnerability [10] in DRAM devices attracted strong interest due to its serious impact on system security. Over the years various studies showed the practical feasibility of Rowhammer attacks in diverse settings. For example, across VMs in the cloud [9], [44], over the network [20], [45], on mobile phones [3], [36], in the browser via JavaScript [4], [34], using memory deduplication [9], [19], and even on ECC-protected DRAM [46].

DRAM manufacturers started deploying in-DRAM mitigations on DDR4 devices based on supplemental refreshes to predicted victim rows, known as TRR [11]. Since then, Rowhammer access patterns became more complex but attacks have not been stopped [11], [21], [47]. More recently, novel effects [17], [48], [49] were discovered that demonstrate the large number of variables involved in Rowhammer [50].

11.2. Speculative Execution Attacks

Since the inception of Spectre in 2018 [28], [51], a plethora of additional Spectre attacks have emerged attacking indirect branch target predictors [6], [7], [28], [51], [52], return target predictors [7], [33], [53], [54], branch condition predictors [5], [28], [51], [55], and the memory disambiguator [56]. Some of these attacks consider physmap to leak arbitrary data [5]–[7], [51], [54]. Göktas et al.'s Blindside [5] leaked the page cache entry of `/etc/shadow` through physmap, and BHI [6], RETBLEED [7], and Inception [13] followed this approach to demonstrate end-to-end attacks. CPU vendors have deployed mitigations against various variants of Spectre in software [29]–[31] and hardware [57], [58]. A weakness in the aforementioned attacks is the amount of time they take to complete due to the vast search space in physmap. In this work, we showed that Rubicon can reduce the search space from millions of candidate pages (approx. 17 M with 64 GB of RAM) to at most 16 pages.

Table 8: **Memory massaging techniques.** We summarize existing massaging techniques based on their determinism (**Det.**), their requirements (**Req.**), i.e., if they need special memory management features or access to `procfs`, and their ability to massage different migratetypes (**mtypes**).

Technique	Det.	Req.	mtypes
Rubicon	✓	–	✓
SpecHammer [38] (2022)	✗	`procfs`	✓
SMASH [34] (2021)	✗	THP	✗
RAMBleed [12] (2020)	✗	–	✗
RAMpage [2] (2018)	✓	ION	✗
Drammer [3] (2016)	✓	ION	✗
Flip Feng Shui [9] (2016)	✓	KSM,THP	✗
Seaborn & Dullien [1] (2015)	✗	–	✓

11.3. Physical Memory Massaging

The physical memory massaging techniques proposed by previous work can be divided into techniques exploiting page spraying [1], [17], special DMA allocators [2], [3], the buddy allocator's inherent properties [12], [35], [38], and techniques relying on memory management or OS features [4], [9], [34]. Table 8 summarizes these memory massaging techniques and their requirements.

Spraying. Seaborn and Dullien [1], and later Half-Double [17], used a probabilistic page-spraying technique by filling as much of the available memory as possible with page tables in the hope that one of them would fall into Rowhammer-vulnerable physical locations. In comparison, Rubicon is fully deterministic, and as such, more reliable without causing unwanted side effects like system instability due to corrupted kernel data structures.

DMA allocators. Van der Veen et al. introduced *Phys Feng Shui* [3] that deterministically forces allocation of a victim data structure (page table) at a previously attacker-controlled location, verified to be vulnerable to Rowhammer. This was done by performing a series of large, physically contiguous allocations via ION, android's DMA allocator. While Drammer originally proposed THPs as a possible replacement for contiguous DMA allocators, Van der Veen et al. [2] further clarifies that these allocations must be performed in the same migratetype as the target, making THPs inadequate. *RAMpage* [2] extended Phys Feng Shui by allowing massaging with non-contiguous DMA allocators, but did so by exhausting all available small blocks through the DMA allocator. As such, both techniques rely on the ability to trigger huge DMA allocations, and are thus easily mitigated by limiting the amount of user-accessible DMA memory. Besides that, these attacks are not applicable to x86-64 where user programs cannot allocate DMA memory. In contrast, Migratetype Escalation of Rubicon allows us to deterministically massage kernel data structures via the default ZBA allocator, making our technique architecture-agnostic and harder to mitigate.

Buddy allocator massaging. Kwong et al. presented *Frame Feng Shui* [12] which massages victims to target physical memory locations by exploiting the page frame cache. Unlike Drammer [3], Frame Feng Shui targets x86-64 and does not rely on DMA. However, it does not allow massaging kernel data structures like page tables.

SpecHammer [38] extended Frame Feng Shui with *limited spraying* to achieve cross-migratetype massaging. As with Seaborn and Dullien [1], this resulted in unreliable massaging with a success rate of only 66% for massaging across migratetypes. Moreover, they relied on tracking the number of free `UNMOVABLE` blocks through `/proc/buddyinfo` to make their spraying more effective. Contrary to these techniques, our technique does not require access to procfs (e.g., `/proc/{pagetypeinfo,buddyinfo}`) and achieves a success rate of 100%.

Memory management and OS features. Instead of actively massaging memory, certain attacks rely on special memory management features. For example, Flip Feng Shui [9] targeted RSA keys and abused memory deduplication. The attacker tricks the OS into mapping an attacker-controlled page and a victim page with known contents into the same Rowhammer-vulnerable physical memory page. While this approach is fully deterministic, it excludes many valuable victims with unknown contents, such as `/etc/shadow` and page tables.

Memory waylaying [37] enables replacement-aware page cache evictions to lure the OS into bringing the victim page into the target physical location. Since it relies on the page cache for bringing the victim into memory, the group of potential victims is limited to files only. Our Rubicon mechanisms, however, place no such restrictions on the victim data structure.

Rowhammer.js [4] used THPs to show Rowhammer on page tables from the browser. To do so, it relies on severe memory pressure to force the system into landing the page table page in templated THPs. Like spraying, this approach is probabilistic and might even result in crashes due to out-of-memory conditions. Other browser-based attacks [19], [34], [36] rely on reusing of browser objects for corrupting pointers inside JavaScript.

Mitigations. Clearly, physical memory massaging is a critical component of many attacks, particularly for Rowhammer. Hence, defenses preventing the aforementioned techniques have been proposed to isolate the area of sensitive physical pages [59] or the victim data structures with guard pages [2], [20], [60]. Further, techniques have been proposed to harden page deduplication [61] and memory allocations to increase the time needed for massaging [41]. Our mitigation presented in Section 9 builds upon the ZBA of the stock kernel and is lightweight, making it easily deployable.

12. Conclusion

We presented Rubicon, a new page-granular physical memory massaging technique built on the Linux Zoned Buddy Allocator. Rubicon leverages three new mechanisms that enable precise control over page reuse, allowing an attacker-provided page to be deterministically allocated for victim data. Rubicon enables creating exploits that were previously impossible. For instance, we developed a reliable Rowhammer-based privilege escalation exploit on x86 systems without relying on specialized memory management features such as DMA allocations, huge pages, or page deduplication. Additionally, we demonstrated that Rubicon can significantly accelerate existing Spectre-like attacks on the kernel. By integrating Rubicon with the recent RETBLEED attack, we achieved a 6.8× speedup on AMD systems and a 284× speedup on Intel systems for leaking `/etc/shadow`, compared to the original method. Our proposed mitigations for Rubicon introduce targeted restrictions on page movement within the Zoned Buddy Allocator, resulting in a minimal impact on system performance and memory fragmentation.

Ethical considerations. This paper introduces a new exploitation technique for existing vulnerabilities like Rowhammer and Spectre, which are regularly mitigated in modern systems. We are nevertheless in contact with Linux kernel maintainers about Rubicon and our proposed mitigations.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

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

[2] 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.

[3] 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.

[4] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[5] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the Spectre era," in *CCS*, 2020.

[6] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *USENIX Security*, 2022.

[7] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022.

[8] SkyLined, "Internet explorer IFRAME parameter BoF remote compromise," 2005, https://web.archive.org/web/20070716023801/http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php.

[9] K. Razavi, B. Gras, C. Giuffrida, E. Bosman, B. Preneel, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security*, 2016.

[10] 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.

[11] P. Frigo, E. Vannacc, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *S&P*, 2020.

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

[13] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *USENIX Security*, 2023.

[14] J. S. Kim, M. Patel, A. G. Yaglikci, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques," in *ISCA*, 2020.

[15] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "Go go gadget hammer: Flipping nested pointers for arbitrary data leakage," in *USENIX Security*, 2024.

[16] S. Mark and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges: How to cause and exploit single bit errors," Black Hat USA, 2015, https://www.youtube.com/watch?v=0U7511Fb4to.

[17] 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.

[18] K. Yoshioka and S. Akiyama, "GbHammer: Malicious Interprocess Page Sharing by Hammering Global Bits in Page Table Entries," in *DRAMSec*, 2024.

[19] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.

[20] A. Tatar, R. K. Konoth, C. Giuffrida, H. Bos, E. Athanasopoulos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC*, 2018.

[21] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *S&P*, 2022.

[22] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms," in *USENIX Security*, 2024.

[23] M. Marazzi and K. Razavi, "Risc-h: Rowhammer attacks on risc-v," in *DRAMSec*, 2024.

[24] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014.

[25] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *S&P*, 2015.

[26] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA*, 2006.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[28] P. Kocher, J. Horn, A. Fogh, a. 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 *S&P*, 2019.

[29] P. Turner, "Retpoline: A software construct for preventing branch-target-injection," 2018, https://support.google.com/faqs/answer/7625886.

[30] Advanced Micro Devices, Inc., "AMD64 TECHNOLOGY INDIRECT BRANCH CONTROL EXTENSION," 2018, https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf.

[31] Intel Corp., "Retpoline: A branch target injection mitigation," 2018, https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf.

[32] M. Fahr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich, and D. Apon, "When frodo flips: End-to-end key recovery on FrodoKEM via rowhammer," in *CCS*, 2022.

[33] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, "Spring: Spectre returning in the browser with speculative load queuing and deep stacks," in *WOOT*, 2022.

[34] 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.

[35] F. Yao, A. S. Rakin, and D. Fan, "Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *USENIX Security*, 2020.

[36] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, 2018.

[37] 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 *S&P '18*, 2018.

[38] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "Spechammer: Combining spectre and rowhammer for new speculative attacks," in *S&P*, 2022.

[39] I. Kang, W. Wang, J. Kim, S. van Schaik, Y. Tobah, D. Genkin, A. Kwong, and Y. Yarom, "Sledgehammer: Amplifying rowhammer via bank-level parallelism," in *USENIX Security*, 2024.

[40] L. Torvalds, "Linux Kernel: mm/page_alloc.c," 2023, https://github.com/torvalds/linux/blob/v6.2/mm/page_alloc.c.

[41] M. Wiesinger, D. Dorfmeister, and S. Brunthaler, "MAD: Memory allocation meets software diversity," in *DRAMSec*, 2021.

[42] Phoronix Test Suite Developers, "Phoronix test suite," 2024, https://www.phoronix-test-suite.com.

[43] W. Chen, Z. Zhang, X. Zhang, Q. Shen, Y. Yarom, D. Genkin, C. Yan, and Z. Wang, "Hyperhammer: Breaking free from kvm-enforced isolation," in *ASPLOS*, 2025.

[44] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security*, 2016.

[45] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing Rowhammer Faults through Network Requests," in *EuroS&P*, 2020.

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

[47] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications," in *MICRO*, 2021.

[48] Z. Lang, P. Jattke, M. Marazzi, and K. Razavi, "Blaster: Characterizing the Blast Radius of Rowhammer," in *DRAMSec*, 2023.

[49] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "RowPress: Amplifying Read Disturbance in Modern DRAM Chips," in *ISCA*, 2023.

[50] O. Mutlu, A. Olgun, and A. G. Yağlıkcı, "Fundamentally Understanding and Solving RowHammer," in *DAC*, 2023.

[51] J. Horn, "Reading privileged memory with a side-channel," 2018, https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[52] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection," *arXiv preprint arXiv:2203.04277*, 2022.

[53] G. Maisuradze and C. Rossow, "Ret2Spec: Speculative execution using return stack buffers," in *CCS*, 2018.

[54] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," in *WOOT*, 2018.

[55] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing," in *S&P*, 2023.

[56] J. Horn, "Issue 1528: Speculative execution, variant 4: Speculative store bypass," 2018, https://bugs.chromium.org/p/project-zerot/issues/detail?id=1528.

[57] Intel Corp., "Indirect branch restricted speculation," 2018, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[58] ——, "Speculative execution side channel mitigations," 2018, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[59] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory," in *USENIX Security*, 2017.

[60] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *USENIX Security*, 2018.

[61] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion," in *SOSP*, 2017.

Appendix A. Data Availability

We make Rubicon publicly available at: https://github.com/comsec-group/rubicon. The repository includes all the files necessary to reproduce the results presented in this paper, along with detailed instructions on how to build and execute the code.

Appendix B. Victim Allocation

One potential concern is that Rubicon's effectiveness may be constrained by the precision of triggering victim allocations. In practice, however, this does not pose a significant issue, as the majority of existing attacks target data structures allocated via system calls or standard library functions. Because these interfaces are often colocated with the attacker's process and rely on minimalist implementations, they follow deterministic allocation patterns, which makes predicting victim allocations much easier.

In fact, the attack introduced in Section 7.2 represents a worst-case scenario, where the victim allocation occurs in a separate process that requires loading an entire binary along with multiple shared libraries into memory, significantly increasing entropy. Yet, even under these unfavorable conditions, Rubicon maintains a precision of 16 pages, demonstrating its robustness. A summary of allocation precision and colocation for victim data structures targeted by various attacks is provided in Table 9.

Table 9: **Victim allocation.** Examples of victim data structures, associated attacks, allocation precision in pages, and colocation with the attacker process.

Victim	Existing Attacks	Precision	Colocated
Page Table	Seaborn et al. [16] Drammer [3] Half-Double [17]	1	✓
Password file	RETBLEED [7] Flip Feng Shui [9] BlindSide [5]	<16	✓
Kernel Stack	SpecHammer [38]	1	✓
Userspace Binary	Gruss et al. [37]	1	✓