Half Spectre, Full Exploit: Hardening Rowhammer Attacks with Half-Spectre Gadgets

Andrea Di Dio*, Mathé Hertogh* and Cristiano Giuffrida
* *Equal contribution joint first authors*
Vrije Universiteit Amsterdam
{*a.di.dio,m.c.hertogh,c.giuffrida*}*@vu.nl*

***Abstract*—Despite nearly a decade of mitigation efforts by both industry and academia, the community has yet to find comprehensive and efficient countermeasures against pernicious hardware vulnerabilities such as Spectre and Rowhammer. While Spectre mitigations have mostly focused on patching dangerous disclosure gadgets in high-value codebases such as the Linux kernel, mitigating Rowhammer in software is still challenging and security often hinges on the (im)practicality of real-world attacks. Indeed, some Rowhammer attacks are entirely *nondeterministic*, triggering random bit flips in the hope of corrupting victim data—but at the risk of corrupting critical data and crashing the system. More reliable attacks rely on techniques such as memory *templating* and *massaging*, but achieving fully deterministic behavior is still difficult in face of complex memory management abstractions in both hardware and software.**

In this paper, we show that fully deterministic Rowhammer attacks *are* feasible. To this end, we exploit synergies with Spectre and specifically focus our attention on so-called *half-Spectre gadgets*. We show these gadgets, previously deemed unexploitable on last-generation CPUs due to their inability to directly disclose secret data, *do* enable powerful disclosure primitives to harden other attacks such as Rowhammer. Specifically, we use half-Spectre gadgets to build PRELOAD+TIME, a generic primitive to monitor a controlled victim's physical memory activity at the *cache line granularity*, without sharing memory with the victim. We use this capability to craft ProbeHammer, the first *crash-free* end-to-end Rowhammer exploit that does not rely on templating or massaging. In detail, we spray physical memory with aggressor (i.e., user) and victim (i.e., page table) data and disclose their location with PRELOAD+TIME. This primitive allows us to select safe hammering patterns and avoid unintended bit flips that may crash the system. Our evaluation confirms ProbeHammer attacks yield no false positives (hence, no crashes) by construction and can compromise real-world systems in a matter of hours.

1. Introduction

Spectre attacks, originally disclosed in 2018 [1], demonstrated attackers can lure victim software into speculatively executing a *disclosure gadget* to (i) access secret data and (ii) transmit such data via a microarchitectural covert channel—which the attacker can then exploit for secret data disclosure. In response, vendors have deployed pervasive mitigations [2], resulting in exploitable gadgets being much harder to find in practice, especially for the original (v1) Spectre variant [1]. Indeed, recent gadget scanning campaigns on high-value targets such as the Linux kernel [3], [4] have only uncovered *half-Spectre gadgets*. Such gadgets miss the ability to disclose data and are thus assumed to be uninteresting for exploitation on last-generation systems [5].

```
1  void spectre_disclosure_gadget(int x) {
2      if (x < SIZE_A)
3          y = B[4096 * A[x]];
4  }
5
6  void half_spectre_gadget(int x) {
7      if (x < SIZE_A)
8          y = A[x];
9  }
```

Listing 1: Top: a classic Spectre-v1 disclosure gadget. Bottom: a half-Spectre gadget, alone insufficient to leak data.

In this paper, we show half-Spectre gadgets are much more powerful than previously assumed, enabling new capabilities for exploitation. To support this claim, we leverage such gadgets to build PRELOAD+TIME, a generic primitive to precisely recover physical addressing information of both attacker-owned pages and kernel data pages. To showcase its effectiveness, we use this primitive to harden state-of-the-art Rowhammer attacks. In particular, we present ProbeHammer, the first *crash-free* Rowhammer attack that is agnostic to both the physical memory allocator and the data encoding scheme adopted by the hardware.

Half-Spectre Gadgets. Classic Spectre gadgets are considered exploitable as long as they enable transmission of secret data via a covert channel. For example, the gadget at the top of Listing 1 transmits the accessed secret data (i.e., out-of-bounds value `A[x]`) to an attacker via a data cache covert channel (i.e., secret-dependent array access `B[...]`). A half-Spectre (v1) gadget (sometimes also described as *prefetch* gadget in literature [6]), on the other hand, is a "*crippled*" Spectre gadget missing an explicit transmitter. For example, the gadget at the bottom of List-

ing 1 still accesses the same secret data as the top gadget, but features no secret-dependent operation to transmit the data back to the attacker. As prior work observed, the secret data can still be leaked with CPU bugs such as MDS, which can leak recently accessed data from microarchitectural buffers [3]. But since MDS-like bugs have been subject of orthogonal mitigations [7], hardware vendors have deemed explicitly mitigating half-Spectre gadgets unnecessary [5].

PRELOAD+TIME. In this paper, we show that, although they do not directly enable data disclosure, half-Spectre gadgets are far from harmless on modern systems. In particular, we show that such gadgets lurking in operating system kernel codebases still allow an attacker to efficiently recover physical addresses of *both* attacker-controlled pages and of kernel sensitive data. Although common half-Spectre gadgets use *relative* addressing, an attacker can exploit a Super TLB side channel to still craft an *absolute* addressing primitive. In turn, this effectively allows them to speculatively fill any cache line in physical memory on modern operating systems such as Linux. The net result is the ability for an unprivileged attacker to eavesdrop a controlled victim's physical memory activity at the cache line granularity, with no requirement for explicit shared memory between attacker and victim. This primitive, which we term PRELOAD+TIME, generalizes existing address translation oracles [8], [9], allowing the attacker to precisely locate data of an *arbitrary* victim under their control in physical memory. We use this primitive to locate aggressor and victim data in physical memory and enable reliable Rowhammer attacks that relax many of the assumptions made in prior work.

ProbeHammer. Since its discovery in 2014 [10], the research community has devised a plethora of attacks based on Rowhammer [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. Some attacks are probabilistic in nature [19], [17], in that they spray memory with sensitive data and trigger random bit flips in the hope of finding a target—but at the risk of corrupting unintended data and crashing the system. To improve reliability, deterministic attacks [18], [24], [11] rely on a combination of physical memory *templating* and *massaging* techniques [11], which, however, can still induce unintended bit flips in case of unpredictable memory reuse behavior [27] and also incur unexploitable bit flips in case of unpredictable data encoding such as that of on-DIMM ECC [19].

Following the disclosure of the first practical Rowhammer attacks, major Operating Systems such as Linux, stopped exposing physical addressing information to unprivileged users [28] and restrict access to super (1GiB) pages to superusers. Therefore, modern attacks rely on techniques, such as transparent huge pages (THPs), that provide *partial* physical addressing information on *aggressor* rows. On the other hand, PRELOAD+TIME provides *full* knowledge of the physical addresses that map to *aggressor and victim* rows.

We show this knowledge allows an attacker to mount a deterministic Rowhammer attack without relying on templating or massaging. We present *ProbeHammer*, the first end-to-end Rowhammer attack that is structurally *crash-free*, in the sense that the attacker knows before hammering that they will not induce unintended bit flips. The key idea is to spray memory with *aggressor* data (i.e., user pages) and *victim* data (i.e., page table pages), rely on PRELOAD+TIME to probe physical memory for their exact location, and select safe patterns for Rowhammer accordingly.

As our experiments show, our approach does not incur any false positives when leaking exact physical addresses of either aggressor or victim pages. For our end-to-end exploit, this is crucial to preserve the crash-free property of ProbeHammer. While probing for spraying data is less efficient than controlled memory reuse via templating and massaging, ProbeHammer still yields realistic attack times. On average, finding a single double-sided hammering triplet (i.e., two aggressor rows and one victim row) takes 5 minutes. End-to-end exploitation, i.e., gaining write access to all of physical memory via a bit flip in a page table, takes an average of 28 hours on our (not very bit flip-prone) testbed.

Contributions. To summarize, our main contributions are:

1) We demonstrate half-Spectre gadgets are a more powerful exploitation primitive than previously assumed. We also show a Super TLB side channel can elevate the capabilities of common gadgets.
2) We introduce PRELOAD+TIME, leveraging half-Spectre gadgets to eavesdrop on a controlled victim's physical memory activity at cache line granularity, without shared memory requirements.
3) We present ProbeHammer, the first crash-free Rowhammer attack which does not rely on templating and memory massaging.

Testbed. The experiments described in this paper were conducted on our *testbed*: an Intel i9-13900K (Raptor Lake) CPU with microcode revision 0x123 and 16GiB of DDR4 RAM, running Ubuntu 22.04 with Linux kernel v6.7.10.

Availability. We have open sourced our code at https://github.com/vusec/half-spectre.

2. Background

2.1. Virtual Memory and Caching

Modern architectures manage the available physical memory by means of a level of indirection known as virtual memory. Software issues memory reads and writes using exclusively *virtual addresses*, which are then translated to physical addresses by the Memory Management Unit (*MMU*). The Operating System (OS) manages this translation by maintaining a radix tree-like data structure known as *multi-level page tables* (PTs) resident in memory. On a typical x86-64 system with 4-level paging, the virtual address space is limited to the lowest 48 bits of an address. The lowest 12 bits represent the offset within a page (4KiB), leaving the remaining 36 bits of a virtual address to retrieve

the corresponding page frame via the PTs. Each page table level is itself a page which holds 512 8-byte *page table entries* (PTEs). As such, 9 bits are required to address a PTE within each level of the page table hierarchy, resulting in a 4-level page table structure. On Linux, these four levels are known as the Page Global Directory (PGD), Page Upper Directory (PUD), Page Mid-level Directory (PMD) and Page Table Entry (PTE), from highest to lowest level, respectively.

Upon any virtual memory reference, the MMU uses the page table hierarchy to perform a *page table walk* and retrieve the physical address. This operation, however, means that a memory access requires additional memory accesses to reference each level of the page table hierarchy, making virtual address resolution a slow operation. To speed up address translation, modern systems are equipped with special caches known as *Translation Lookaside Buffers* (TLBs) and *translation caches*, which cache full and partial address translations, respectively. Furthermore, from the CPUs perspective, PTEs are normal data and as such are cached in the system's cache hierarchy. This results in a faster translation whenever the virtual address does not have an entry in the dedicated TLBs or translations caches.

2.2. The Direct Map

On x86-64 systems, Linux and other modern operating system kernels set up a special memory region called the *direct map* in the kernel memory address space. This region has a one-to-one mapping to every page frame starting from Page Frame Number (PFN) zero to the highest (`MAX_PFN`). The purpose is to quickly be able to access any location in physical memory and also simplify physical-to-virtual address translation. Kernel heap allocators such as SLUB and `vmalloc`, which overlay the page frame allocator (i.e., the *buddy allocator*), return pointers to the direct map. Therefore, data dynamically allocated by the kernel such as page tables fall within this memory region. Given that the direct map maps the *entire* physical memory, every page in the system mapped to user space has also an *alias* in the direct map which translates to the same page frame in the physical address space. The starting address of the direct map is known as the `page_offset_base` and its value is randomized at boot time with *KASLR* enabled.

2.3. Spectre

In 2018, Kocher et al. [1] demonstrated how an attacker can take advantage of speculative execution to leak data across security domains. The variant of interest for this paper is Spectre-PHT (also known as Spectre-v1), which exploits the speculative execution window which occurs after a mispredicted conditional branch. Within this window, an attacker can force the CPU to perform a speculative load at a desired address, with the loaded value used to effect changes to the microarchitectural state. Since microarchitectural changes are not rolled back by the CPU when speculation aborts, the attacker can use a (e.g., cache) covert channel to leak the data stored at that loaded address. A typical Spectre-v1 gadget is shown at the top of Listing 1.

2.4. Rowhammer

The Rowhammer vulnerability [10] allows attackers to flip bits in rows of extraneous physical memory by repeatedly activating i.e., "*hammering*", one or more neighboring rows. Rowhammer-induced bit flips are notoriously reproducible and data-dependent [29], [30]. The first Rowhammer end-to-end exploit was presented by Seaborn and Dullien [17], *probabilistically* targeting bit flips in PTEs to escalate privileges. Probabilistic attacks rely on the ability of the attacker to spray memory with victim (e.g., PTE) data. However, since memory also contains other critical data, such attacks cannot guarantee the bit flips will not accidentally corrupt unintended data and crash the system. Follow-on work demonstrated more *deterministic* attacks on different architectures (x86 architectures [11], [12], [13], [14], [15], [16] and ARM [18], [19], [20]) and environments (browser sandboxes [21], [22], [23], clouds [24], kernels [27], and networked systems [25], [26]).

Typically, a deterministic Rowhammer exploit comprises three main steps namely, *templating*, *memory massaging*, and *hammering* [24]. In the first step, the attacker hammers their own memory to find bit flips at a specific page offset (a so-called *template*). In the memory massaging phase, the attacker manipulates the memory layout so that the target data is stored at the vulnerable template(s). Finally, the attacker re-issues the original hammering patterns to trigger a bit flip in the target data. Unlike probabilistic attacks, deterministic attacks can be in principle *crash-free*. However, this is subject to the reliability of memory massaging—which requires the attacker to fully predict allocation behavior [24] and, more importantly, to that of templating—which is unreliable on modern DIMMs with ECC in face of non fully predictable victim data [19].

3. Threat Model

We consider a classic local exploitation scenario, with an unprivileged userland attacker seeking to disclose confidential information, escalate privileges, etc. To this end, we assume the attacker targets a half-Spectre gadget in the operating system kernel, reachable via a system call. We assume a modern up-to-date kernel with all default mitigations against (microarchitectural) attacks applied and with no software bugs. Finally, we assume the kernel to have a *direct map* of physical memory, as done by commodity kernels such as Linux.

4. Half-Spectre Gadgets

To gather insights into the attack surface of half-Spectre gadgets in the Linux kernel, we rely on existing gadget scanners which have been developed to identify Spectre-v1 and half-Spectre gadgets in the kernel. Table 1 presents

Tool	Half-Spectre gadgets reported
Smatch [32]	234
Google CodeQL [4]	290
Kasper [3]	722

TABLE 1: Half-Spectre Gadgets in Linux v6.7.10

an overview of the number of *unique* half-Spectre gadgets found by these tools. These tools rely on either static or dynamic data-flow analysis to identify half-Spectre gadgets, an approach that can successfully pinpoint potential gadgets but cannot provide insights into their practical exploitability [31]. In other words, exploitability analysis must be conducted manually. From the results gathered by these tools, we manually selected two running examples (Listings 2 and 3), one for each tool that has explicitly sought to uncover half-Spectre gadgets [3], [4], and closely studied their exploitability. In the following, we first discuss the characteristics that affect exploitability and then present an analysis of our two running examples.

4.1. Half-Spectre Gadget Properties

Transient execution. Similar to their Spectre-v1 counterparts, half-Spectre gadgets abuse *conditional branch prediction* present on modern CPUs. As such, exploitation requires the attacker to (i) mistrain the target branch using *in-place* or *out-of-place* training [6], [33] to induce transient execution and (ii) delay branch resolution as much as possible to obtain a sufficiently large speculation window. Delaying branch resolution is as simple as evicting a cache line if the branch depends on data loaded from memory [1]. However, many real-world gadgets branch on constant data and attackers need to resort to SMT resource contention to lengthen the window [34], [31].

Addressing. Half-Spectre gadgets can either be *absolute* or *relative*. In the former case, an attacker directly provides the address to be speculatively loaded in the kernel. In the latter case, an attacker only controls an offset to an uncontrolled *base* address, allowing them to only address the speculative loads relative to this base. Absolute gadgets often arise from pointer dereferencing, whereas relative gadgets usually arise from array indexing. Due to the prevalence of array index bounds checking, relative gadgets appear much more common. For example, all the gadgets reported by Smatch and Google CodeQL in Table 1 are relative.

Range. Not all half-Spectre gadgets may be able to speculatively load *any* address. A gadget's *range* is the set of virtual addresses that an attacker can let it speculatively load. For a gadget arising from an array being speculatively indexed with a 32-bit integer, the range is limited to the 32-bit address region behind the start of the array. But the range may be even much smaller than that. In general, the range depends on the complete interaction between attacker and victim. This includes the gadget's *body*, the code between

```
1  static int do_prlimit(struct task_struct
   ↪  *tsk, unsigned int resource, struct
   ↪  rlimit *new_rlim, struct rlimit
   ↪  *old_rlim)
2  {
3    struct rlimit *rlim;
4    int retval = 0;
5
6    if (resource >= RLIM_NLIMITS) // BCB
7      return -EINVAL;
8    ...
9    rlim = tsk->signal->rlim + resource;
10   task_lock(tsk->group_leader);
11   if (new_rlim) {
12     if (new_rlim->rlim_max > rlim->rlim_max
       ↪  && !capable(CAP_SYS_RESOURCE))
13   ...
14   }
15 }
```

Listing 2: An example of a *relative* half-Spectre gadget in the Linux kernel (`kernel/sys.c`) found by Google [4]. We use this gadget in our end-to-end exploit.

the mispredicted branch and the speculative load, which may for example mask off certain bits of the loaded address. But it also depends on the full code path between system call entry and the gadget: sanitization of user arguments at the start of a system call may limit the attacker's control over the address of a speculative load. And even cross system call interaction may be relevant for the range, as we will see in the `break_lease` example gadget in Section 4.2.

4.2. Exploitability of Running Examples

Our two running examples (Listings 2 and 3) consist of a relative and an absolute half-Spectre gadget (respectively) found in the Linux kernel. In order to assess their exploitability, we mapped a 4KiB page in userland (i.e., a reload buffer), retrieved its physical address via the `pagemap` interface, and triggered the gadgets to perform a speculative load to the direct map alias of our user page.

The gadget shown in Listing 2 is straightforward to trigger via the `setrlimit` system call. The user has direct control over the 32-bit integer `resource` parameter (in bold). Line 9 adds `resource` as an offset to an array of `struct rlimit` elements allocated on the kernel heap for each user process. Hence, the gadget is relative. Its range consists of the 64GiB region of virtual address space behind the base `tsk->signal->rlim`. To reach the out-of-bounds speculative load on line 12, we have to enlarge the speculation window using SMT contention, since the gadget branches on a constant value.

The absolute gadget shown in Listing 3, on the other hand, is more complicated to exploit. Indeed, exploiting absolute gadgets typically requires the attacker to land controlled data in the kernel via memory massaging [3]. Note that on x86-64 systems, one cannot simply pass a pointer to the kernel via a system call and trick it into speculatively dereferencing such pointer since

```
1  static bool leases_conflict(struct
   ↪  file_lock *lease, struct file_lock
   ↪  *breaker)
2  {
3    bool rc;
4
5    if
     ↪  (lease->fl_lmops->lm_breaker_owns_lease
6      &&lease->fl_lmops->lm_breaker_owns_lease(
       ↪  lease))
7      return false;
8    ...
9  }
10
11 static bool any_leases_conflict(struct
   ↪  inode *inode, struct file_lock
   ↪  *breaker)
12 {
13   struct file_lock_context *ctx =
     ↪  inode->i_flctx;
14   struct file_lock *fl;
15
16   lockdep_assert_held(&ctx->flc_lock);
17
18   list_for_each_entry(fl, &ctx->flc_lease,
     ↪  fl_list) { // BCB
19     if (leases_conflict(fl, breaker))
20       return true;
21   }
22   return false;
23 }
```

Listing 3: An example of an *absolute* half-Spectre gadget in the Linux kernel (`fs/locks.c`) found by Kasper [3].

user pointer dereferences are serialized via SMAP [4]. As shown by Kasper [3], list iterators as implemented in the Linux kernel are prone to speculative type confusion, inducing out-of-bounds speculative loads past the last element of the list depending on the relative type sizes. In this case, `struct file_lock_context` is smaller than `struct file_lock`, hence, if we can massage a valid kernel pointer next to the SLUB-allocated `file_lock_context` object, the speculative type confusion allows the gadget to load from an absolute controlled location in the direct map (in bold).

We implemented two proof-of-concepts to trigger both gadgets with controlled input—i.e., offset passed via a system call for the relative gadget and address injected via SLUB memory massaging for the absolute gadget—and evaluated the number of successful gadget invocations per second while creating SMT contention as suggested in prior work [34], [31]. Table 2 presents our results. The `do_prlimit` gadget has considerably higher throughput than the `break_lease` gadget, because `do_prlimit` is very 'shallow' in the call stack, it does not require any memory massaging for exploitation, and the code architecturally returns to userland immediately after the condition check is performed (on line 7). Therefore, for the remainder of the paper and for our end-to-end exploit (Section 7), we will exclusively focus on this half-Spectre gadget—which is also more representative of common (relative) gadgets. However, being a relative gadget, this choice raises an additional challenge, which we discuss in Section 4.3.

Gadget	Reload buffer hits per second
`do_prlimit`	493,600
`any_leases_conflict`	1,500

TABLE 2: Throughput of the gadgets in Listing 2 and 3.

4.3. From Relative To Absolute with Super TLB

The rest of this paper will focus on exploiting half-Spectre gadgets with *absolute* addressing being a requirement. However, as mentioned, *relative* gadgets in which a speculative load is performed on a constant kernel address (the *base*) plus an attacker-controlled value (the *offset*), are much more prevalent. This section shows that an attacker can effectively obtain *absolute* addressing capabilities from a *relative gadget*. To this end, the attacker must know the exact kernel virtual address of the relative gadget's (uncontrolled) base. Given that we are targeting the direct map, the first step is to break KASLR. We achieve this by means of the `prefetch` side channel [8], which provides the attacker with an oracle to determine whether a given kernel address is mapped or not. For an attacker with knowledge of the true offset of the direct map (i.e., `page_offset_base`), leaking the uncontrolled base is trivial if the base is a static kernel variable, e.g., a global variable. However, in the vast majority of the gadgets reported in Table 1, this is more complicated because the uncontrolled base points to a heap object which is dynamically allocated at runtime. For instance, the `do_prlimit` gadget (Listing 2) used in our end-to-end exploit has a base `tsk->signal->rlim` allocated via the slab allocator.

We present a new technique that uses *differential analysis* and *sliding* [35] to leak the *exact* address of a relative half-Spectre gadget's base via a Super TLB side channel. At a high level, we perform TLB Evict+Reload [36] against the half-Spectre gadget itself. However, the Linux kernel heap resides in the direct map, which is backed by 1GiB super pages [37]. This requires a new generalization of Evict+Reload to the *Super TLB*, the TLB's separate partition holding entries for super pages. Our differential analysis distinguishes the gadget's signal from systematic syscall noise. Despite the signal being only super page granular, we can pinpoint the exact address of the base via a sliding technique. Both the differential analysis and the sliding technique take advantage of the attacker's fine-grained control over the half-Spectre gadget's speculative execution.

First we discuss Evict+Reload on the (Super) TLB. A userspace attacker *evicts* a kernel page from the TLB, then triggers a system call, and afterwards *reloads* the backing TLB entry—timing the operation—to determine whether the system call accessed the page. For regular pages, TLBs have been extensively reverse engineered [38]. Evicting a (kernel) TLB entry can be done by accessing a number of user pages

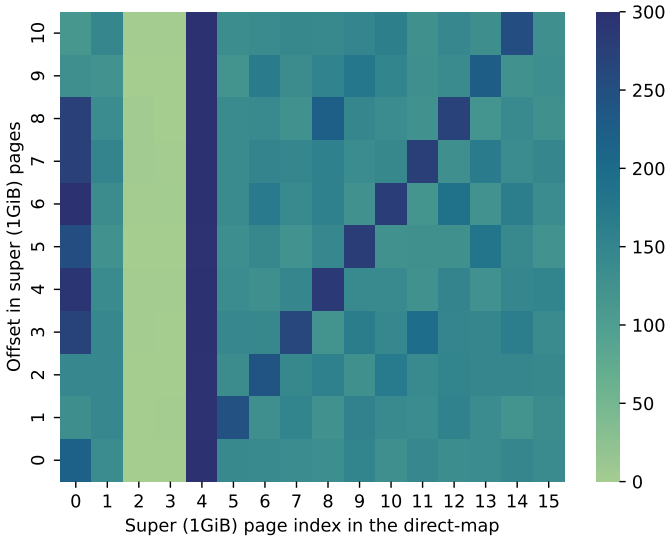

Figure 1: Super TLB Evict+Reload signal on Linux' direct map, upon triggering the `do_prlimit`'s half-Spectre gadget with different offsets. The heatmap shows the number of Super TLB *hits* measured out of 300 repetitions, per super page and per offset. The blue diagonal originates from the half-Spectre signal, and indicates the base to be within super page 4.

forming a TLB eviction set. Reloading can either be done via prefetch instructions [8], or page fault handlers [39].

Hence, to generalize Evict+Reload to the Super TLB, we need to be able to implement Reload as well as build eviction sets. For the former, despite the lower translation latency incurred by super pages, our experiments show a measurable timing difference between a Super TLB hit and miss via both prefetches and page faults on our testbed.

Super TLB evictions are more challenging. Super pages are restricted to superusers on Linux. Hence, unprivileged attacker cannot build their own Super TLB eviction sets. To address this challenge, we reverse engineered the Super TLB behavior in our testbed, by relying on a classic blackbox eviction set algorithm [40] adapted to the Super TLB and disclosed its properties. The algorithm found only a single eviction set with 4 entries, which reveals the Super TLB behaves like a fully-associative, 4-way cache—with LRU-like replacement policy. This is small enough to let the kernel *self-evict* its own super pages. That is, an attacker can repeatedly trigger the half-Spectre gadget to perform speculative loads on other super pages. Alternatively, on platforms such as our testbed where a user-mode prefetch invocation can trigger TLB fills for kernel addresses, we can directly prefetch the kernel's super pages (e.g., in the direct map) to trigger evictions.

Next, we leak the gadget's base page by using (Super) TLB Evict+Reload against the system call triggering the relative half-Spectre gadget. Triggering the gadget with offset 0 inserts the base into TLB, potentially along with other pages which are accessed by the system call, resulting

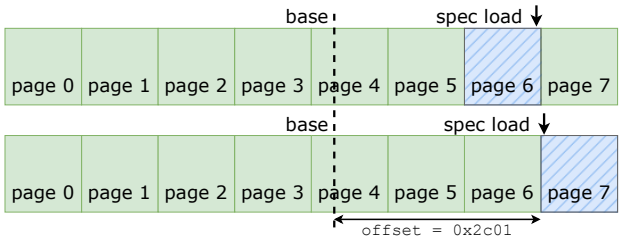

Figure 2: Sliding the speculative load higher and higher, by providing higher and higher offsets to the half-Spectre gadget, caused the TLB signal to cross the page boundary, revealing the base to be at offset 0x3ff within page 4.

in Evict+Reload measuring systematic noise. To distinguish the correct base's page from this noise, we take advantage of the flexibility the attacker has with respect to triggering the gadget. The attacker triggers the gadget with multiple different offsets, spanning across multiple pages, and performs differential analysis on the results. As the speculative load moves across different pages, while the systematic noise stays (mostly) static, our differential analysis is able to distinguish the base's page. An example run of this technique for the `do_prlimit` gadget is illustrated in Figure 1. Since `do_prlimit` lies inside Linux' direct map (backed by super pages), we resort to the Super TLB. Although the gadget's base resides on super page 4, we see some systematic noise on super page 0 as well. Even if more systematic noise would be present, say both super pages 1 and 5 would have dark blue columns as well, the diagonal signal still reveals the base to be on super page 4.

Now that we know the base's page, we can leak the base's exact location via *sliding*. Our differential analysis has already revealed an offset which causes the half-Spectre gadget to hit a target (super) page that is otherwise never hit. The attacker then slides this offset, i.e., increase byte-by-byte, until the speculative load crosses the next (super) page boundary, as depicted in Figure 2. The offset upon which this happens reveals the exact (byte-precise) location of the base within the (super) page. As an optimization, we perform sliding in binary search fashion, instead of linearly.

We implemented Super TLB Evict+Reload using the prefetch method on x86_64 Linux, and evaluated the differential analysis and sliding techniques for the relative `do_prlimit` gadget on our testbed. Out of 20 runs, the correct base address was leaked each time, in an average of 49 seconds ($\sigma = 16s$).

5. Attack Overview

In this section, we provide an overview of ProbeHammer, which attackers can use to gain read/write access to the entirety of physical memory on a victim system. Unlike previous Rowhammer attacks which commonly rely on memory templating and massaging as their main steps in the exploit chain, ProbeHammer relies on PRELOAD+TIME, allowing the exploit to consist of

```
for (pa = page_offset(va); pa < MEM_SIZE;
    pa += PAGE_SIZE) {
    clflush(va);
    kernel_speculatively_load(pa);
    if (access_time(va) < THRES)
        return pa
}
```

Listing 4: Disclosing the physical address `pa` of the user virtual address `va` via speculative loads in the kernel.

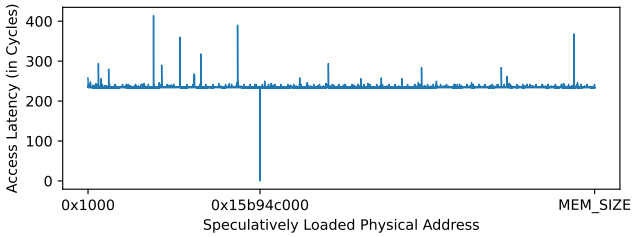

Figure 3: An example run of Listing 4, using a synthetic system call that speculatively loads a user provided physical address, and 16GiB of RAM. We see a low access time, i.e., cache hit, only at physical address 0x15b94c000, revealing it as the correct physical address.

three phases: (1) interleaved memory spraying, (2) pattern search and, (3) hammering.

PRELOAD+TIME. Our approach relies on half-Spectre gadgets to craft a primitive that leaks the physical addresses of the attacker's user data pages and those of the page table pages. Furthermore, we use this information to precisely hammer *only* PTEs—the target data which we want to safely corrupt with Rowhammer bit flips.

Interleaved memory spraying. In the first phase of the attack, we spray physical memory by interleaving allocations of page table pages *and* user pages. The former contain the data that we want to place in the victim rows, while the latter the data that we want to place in the aggressor rows.

Pattern search. Once physical memory is filled with page table pages and user pages that we control, we scan through the physical memory address space looking for double-sided hammering pairs. Specifically, we use PRELOAD+TIME to leak the physical addresses of both the user data pages and the page table pages, until we find a situation where the allocated pages follow the typical hammering pattern (i.e., Aggressor-Victim-Aggressor) over three DRAM rows in the same bank. Note that this requires knowledge of the memory controller's mapping between physical and DRAM addresses, which we discuss in Section 7.3.

Given that exploiting Rowhammer on DDR4 systems requires building *many*-sided patterns, we keep searching for double-sided patterns until we can form a so-called n-double-sided pattern. Furthermore, as we skip the templating phase, we do not know a priori whether the patterns we find are effective at triggering bit flips in the victim rows. Therefore, we need to test many n-double-sided patterns in a given bank before we find an exploitable pattern. As we will later show in Section 7.4, our optimized spraying algorithm allocates ∼30k double-sided pairs evenly spread across banks on our testbed with 16GiB of RAM.

Hammering. Finally, while we keep finding new n-double-sided patterns in a given bank, we hammer the aggressor rows. In order to maximize our probability of finding an exploitable pattern, we combine the single double-sided patterns in many different n-double-sided patterns and hammer all the new combinations as we keep finding new hammering pairs. Upon a bit flip, we check our read/write access to a page table by using an oracle [19] that only speculatively accesses the page, to suppress any potential crashes. Upon success, we have effectively fully compromised the system. Otherwise, we keep searching for patterns.

6. PRELOAD+TIME

We now present PRELOAD+TIME, a generalization of the address translation oracle [8], [9], re-enabling the oracle on modern systems, significantly improving its performance and (absolute vs. relative addressing) flexibility, and—most importantly—extending its application to *victim* (not just attacker) data. PRELOAD+TIME can monitor a victim's memory activity at the cache line granularity (similar to Flush+Reload [41]), while being more widely applicable (similar to Evict+Time [42]). We focus on a (Rowhammer oriented) application of PRELOAD+TIME, by targeting the MMU to leak the location of page tables in physical memory. We later evaluate PRELOAD+TIME's accuracy.

6.1. Address Translation Oracle

The combined work of Gruss et al. [8] and Schwarzl et al. [9] describes an address translation oracle, enabling an unprivileged attacker to locate their own data in physical memory. The oracle can be summarized in three main steps. Let V be the virtual address of an attacker's user page, and let P be a (guessed) physical address. The attacker (1) flushes the cache line backing V, (2) forces the kernel to speculatively load the cache line at physical address P via the direct map, and (3) measures the access time to V. The most important insight is that the second step caches V's data if and only if V is backed by the physical memory at P. Hence, a fast access time indicates V translates into physical address P. Upon a slow access, the attacker takes a new guess for P, and repeats. Listing 4 shows an example implementation and Figure 3 depicts the raw timing data of an example run.

Initially, Gruss et al. [8] implemented step (2) with a prefetch instruction issued from userland. Later, Schwarzl et al. [9] showed the prefetches to actually be irrelevant, and determined the root cause of the cache signal to be speculative kernel loads triggered via Spectre-v2 gadgets. The attacker stores a direct map address into a register, and

```
int preload_time(physaddr_t pa) {
  victim();
  evict_cache_set(pa);
  t_base = time(victim);
  evict_cache_set(pa);
  preload(pa);
  t_preload = time(victim);
  return t_preload < t_base - THRES;
}
```

Listing 5: PRELOAD+TIME determines whether the `victim` function accesses physical address `pa`.

	Flush+Reload	Evict+Time	Preload+Time
Victim Control	no	yes	yes
Victim Latency	irrelevant	stable	stable
Shared Memory	yes	no	no
Granularity	cache line	cache set	cache line

TABLE 3: Comparison of side channels.

then issues many (random) system calls. The slim chance of indirect branch mispredictions in the kernel, combined with many kernel indirect call targets dereferencing registers, resulted in the direct map address sometimes being speculatively dereferenced.

Since these occurrences are rare, this blackbox strategy only resulted in 5 to 60 hits per second [9]. To put this in perspective, scanning through 16GiB of RAM, as done in Figure 3, would take at least 19.4 hours to translate *one* address. Moreover, since the deployment of Spectre-v2 mitigations, this attack has been fully mitigated [9].

6.2. PRELOAD+TIME: Attacker Data

We now first show PRELOAD+TIME can disclose the physical location of *attacker* data (à la address translation oracle). We later target *victim* data (Section 6.3).

While PRELOAD+TIME on attacker data still follows the algorithm from Listing 4, it instead triggers a speculative load in the kernel via half-Spectre (v1) gadgets, which have been left largely unmitigated [5]. PRELOAD+TIME deliberately adopts a whitebox strategy, targeting a predetermined half-Spectre gadget and ensuring its speculative execution, for example via branch mistraining and SMT contention. After breaking KASLR, the attacker knows the location D of the direct map in the kernel's virtual address space. After acquiring absolute addressing capabilities for the chosen half-Spectre gadget, the attacker can trigger the gadget to speculatively load the kernel virtual address $D + P$, i.e., physical address P. This re-enables the address translation oracle, with much better performance. For example, using the `do_prlimit` gadget lowers translation time to 8.5 seconds for one address (cf. Table 2). Our optimizations in Section 6.4 even bring this down to merely 0.3 seconds.

6.3. PRELOAD+TIME: Victim Data

We now apply PRELOAD+TIME to eavesdrop on the physical memory activity of a controlled *victim*. Unlike existing techniques, PRELOAD+TIME can monitor a victim's memory activity at the *cache line granularity*, even if the attacker and victim *do not explicitly share memory*. Without access to the victim data being monitored, the attacker cannot directly flush the data, or measure its access time, as in Listing 4. Instead, the attacker resorts to *cache eviction* and *victim timing*, similar to Evict+Time [42].

Suppose the attacker wants to know whether the victim accesses physical address `pa`. Then, as shown in Listing 5, the attacker first executes the victim, caching its code and data. Next, the attacker evicts `pa`'s cache set, by walking a cache eviction set, and times the total run time of the victim, as a baseline measurement. Next, the attacker again evicts the cache set of `pa`, but now *preloads* `pa` itself: triggering a half-Spectre gadget in the kernel to speculatively load `pa`'s cache line. Timing the victim's total run time again, and comparing it against the baseline timing, reveals whether the victim accessed `pa`. Note that the victim's latency should be stable across runs, in order to distinguish the victim's potential cache hit vs. miss on `pa` reliably (similar to Evict+Time [42]). In general, depending on the variance of the victim's run time, the attacker must perform enough measurements to see a statistically significant lower `t_preload` timing across runs.

PRELOAD+TIME can be seen as a refinement of Evict+Time, having the exact same applicability. As for requirements, that means the attacker needs direct control over the victim, and the victim's run time variance should be low. The main advantage is that both Evict+Time and PRELOAD+TIME are applicable *without* shared memory between attacker and victim. PRELOAD+TIME is more powerful than Evict+Time though. Whereas Evict+Time monitors *cache sets*, PRELOAD+TIME extracts information of the victim's physical memory activity at the *cache line* granularity, similar to Flush+Reload. For example, on a virtually indexed L1 data cache, Evict+Time can disclose whether the victim accesses the first cache line of *any* page, while PRELOAD+TIME can reveal whether the victim accesses the first cache line of *one particular* page. In summary, as also listed in Table 3, PRELOAD+TIME is as widely applicable as Evict+Time, while disclosing information as detailed as Flush+Reload. Moreover, PRELOAD+TIME's more fine-grained information disclosure leaves it unmitigated by some cache attack defenses, like Page Coloring [43], that protect against *cache set* granular side channels, but not *cache line* granular ones.

6.4. Locating PTEs

This section shows how to physically locate PTEs, by applying PRELOAD+TIME to the MMU's page table walk. Let `va` be a virtual address in the attacker's address space,

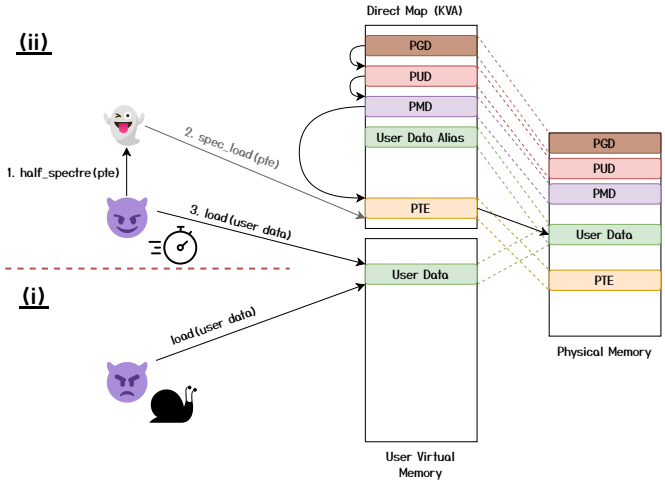

Figure 4: Leaking the physical address of a PTE with PRELOAD+TIME. Bottom-left (i): Slow load of user data with PTE evicted from cache and the user data address evicted from the TLB. Top-left (ii): Fast load of user data due to the faster page table walk, after preloading (hence, caching) the correct PTE.

whose PTEs we want to locate. Upon a TLB miss for `va`, the MMU will perform a page table walk, consequently loading all `va`'s PTE levels into the cache. PRELOAD+TIME can leak their physical addresses up to cache line granularity, as described by Listing 5, where `victim` is a load from `va` that misses the TLB—which the attacker should ensure by purposefully evicting `va` from the TLB beforehand. Figure 4 illustrates this application of PRELOAD+TIME to locate PTEs.

As an optimization, we can avoid checking *all* cache lines in physical memory, if we have a priori physical address knowledge. The virtual address bits of `va` determine the offsets of its PTEs in the different page tables, hence the attacker already knows the page offset of all the PTEs. For example, on our testbed with an x86_64 CPU and 16GB of RAM, this reduces the search space from 28 bits (256M cache lines) to 22 bits (4M cache lines), a 32x speedup. Moreover, we can use Evict+Time to break these 22 bits of entropy in two separate steps. Our testbed's L2 cache is indexed with the 11 physical address bits 16:6 (the lower six are the cache line offset). A PTE's lower 12 bits (the page offset) are known a priori, and the next 5 bits 16:12 can be leaked via Evict+Time on the L2 cache against the MMU's page table walk. Breaking this 5 bit entropy is a quick operation, leaving us with only 18 bits of entropy left to break with PRELOAD+TIME, allowing us to gain another 32x speedup. Replacing the L2 cache with the bigger L3 cache, would result in an additional 24x speedup on our testbed, which we did not implement due to the higher complexity of the proprietary hash functions used for L3 slice indexing. Note that the same search space reduction can be applied when locating attacker data with PRELOAD+TIME, as in Section 6.2. Also note that none of the Evict+Time methods come close to leaking *full* physical addresses. PRELOAD+TIME is the crucial step to enable full disclosure.

Leakage Target	True Pos.	False Pos.	False Neg.
User Data Page	943 (94.3%)	0 (0%)	57 (5.7%)
PTE Page	928 (92.8%)	0 (0%)	72 (7.2%)

TABLE 4: Accuracy Results for PRELOAD+TIME after running the experiment outlined in Section 6.5.

6.5. Accuracy Evaluation

We implemented PRELOAD+TIME to locate both user pages (i.e., attacker-owned data) as well as their PTEs (i.e., victim data) on our testbed. As an evaluation, we allocate 1,000 normal user pages, and we apply PRELOAD+TIME to leak the physical address of these user pages, as well as their lowest level PTEs (pointing to the user pages). The results of this experiment, reported in Table 4, show that PRELOAD+TIME is a highly precise primitive for recovering the physical addresses of both user pages and PTEs (with a 94.3% and 92.8% True Positive Rate, respectively). We attribute the false negatives to noise in the system, which accidentally evicts the 'preloaded' data from the cache before the timing step. Nevertheless, such false negatives only result in *slower* attacks—as we are missing exploitation opportunities whenever we cannot locate a user page or a PTE in physical memory—but do not affect their *reliability*.

On the other hand, any false positives would weaken the crash-free guarantees of ProbeHammer, as they could lead an attacker to incorrectly interpret critical system data as a user page or PTE. However, ProbeHammer eliminates false positives in the practical cases of interest by design. Indeed, as we are targeting attacker-controlled user pages or PTEs pointing to user pages, we do not expect any independent accesses to such pages. Only hypothetical features such as a hardware prefetcher *systematically* accessing user data pages and the corresponding PTEs would result in false positives (which we could not filter out by means of repeated measurements). We have not encountered any false positives in all our experiments nor in our end-to-end exploit.

In terms of performance, locating a user data page takes ∼2 minutes on average, while locating a PTE page takes ∼5 minutes, the latter requiring more repetitions for high-accuracy measurements.

As we will show in Section 7, for our end-to-end exploit, we will rely on these primitives in order to *selectively* flip bits in memory only in our target data, without causing system crashes.

7. End-to-End Exploit

To show how an attacker can take advantage of half-Spectre gadgets to build a more complex exploit, we present ProbeHammer: a Rowhammer attack which targets PTEs and takes advantage of half-Spectre gadgets to bypass

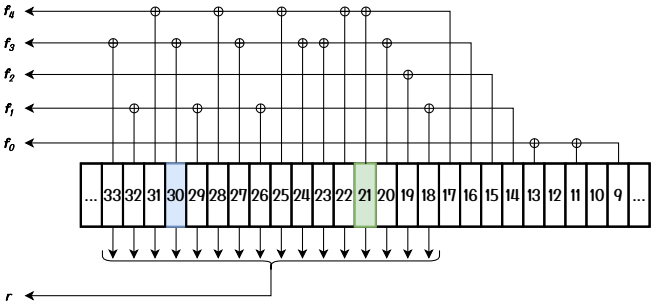

Figure 5: Reverse-engineered DRAM addressing functions for the Intel i9-13900K machine with a single-channel, single-DIMM, dual-rank DRAM configuration. Highlighted in blue ( ) and green ( ) the number of lowest bits shared between the physical and virtual address for Super (1GiB) pages and Huge (2MiB) pages, respectively.

the templating and memory massaging stage of typical Rowhammer attacks, while structurally avoiding system crashes by precisely targeting *only* page frames storing PTEs. In this section, we explain the steps to build our end-to-end exploit and gain read/write access to the entirety of physical memory.

7.1. Experimental Setup

To evaluate the components we need to build the end-to-end exploit, we run all experiments on our testbed, equipped with a G Skill F4-3200C14-16GTZR DDR4 DIMM. In order to use the high-throughput half-Spectre gadget shown in Listing 2, we reverted the patch mainlined by Google [4].

7.2. Crash-Free Rowhammer

To implement crash-free Rowhammer semantics, ProbeHammer relies on a combination of *interleaved memory spraying* (Section 7.4) and the PRELOAD+TIME primitives. Using these two primitives, an attacker can derive plenty of double-sided hammering triplets and combine them to achieve 'n-double-sided hammering', a generally effective Rowhammer variant on modern DRAM [13]. In our attack, we target PTEs as our victims as they have been repeatedly targeted by prior exploits for the purpose of privilege escalation. Thanks to PRELOAD+TIME, we can reliably leak their addresses in physical memory (along with the addresses of user page aggressors).

7.3. Raptor Lake DRAM Addressing

We reverse-engineered the DRAM addressing functions for our target machine using bank conflicts as a side channel similarly to previous work [44]. The resulting functions which map a physical address to a DRAM address are depicted in Figure 5. As shown by Kang et al. [16], the memory controller on more recent Intel microarchitectures (Alder Lake) uses more bits and higher bits of the physical address to complete the address translation. Our work shows that this is also the case on the latest Raptor Lake microarchitecture. We compare our results to the state-of-the-art tool, *DARE* [15], and find consistent results.

In order to further investigate the quality of our reverse engineered functions, we ran state-of-the-art Rowhammer fuzzers [13], [14], [16]. We first ran these three tools on a Coffee Lake CPU, a legacy microarchitecture with a simpler memory controller (but not as prone to SMT contention as modern ones) and observed a significant number of bit flips. We then ran the tools on our Raptor Lake setup as well as on other modern Intel microarchitectures using the same DIMM. Over 24 hours, we consistently observed no bit flips or an insignificant (and unstable) number of bit flips compared to our Coffee Lake runs. We attribute this behavior to more complex memory controller behavior on modern microarchitectures, hindering state-of-the-art Rowhammer patterns. Thereafter, we halved the refresh rate in the BIOS, resulting in consistent bit flips again. Since our focus here is on the exploitability of Rowhammer bit flips and ProbeHammer can compose with any hammering patterns, we leave the investigation of more effective Rowhammer patterns on modern memory controllers to future work and present experiments with the refresh rate halved hereafter.

With all the tools mentioned earlier, both on standard and halved refresh rate, we always observe the bit flips to be three rows away from the first of the two aggressors, revealing non-standard DRAM geometry. In other words, the victim row was not logically sandwiched between the aggressors but happened to be 3 rows away from the first aggressor row. Our exploit takes this finding into account and, in the remainder of the paper, we will refer to such a triple of rows as a *hammering triplet*. Also note that an attacker would do this reversing of the DRAM geometry only once, in an offline phase.

7.4. Interleaved Memory Spraying

Armed with our PRELOAD+TIME primitive which can precisely recover the physical memory location of both our victim pages and our aggressor pages, we have devised *interleaved memory spraying*, a new memory spraying technique to increase the likelihood of finding safe hammering patterns. The key idea is to optimize spraying to maximize the chances of (i) colocating aggressor (user) data with victim (page table) data and (ii) entirely filling victim rows with victim data. To this end, in the memory spraying stage, we interleave allocations of both page table pages *and* user data pages until we exhaust physical memory. By spraying most of the physical memory available on the system, we span over a large number of bits within the physical address space resulting in allocations which are equally distributed across all the DRAM banks on our system. This allows us to target *any* bank in the system in the hammering stage of our exploit.

To allocate as many page table pages as needed we resort to similar techniques as previous work [17], [21], [19], [18]

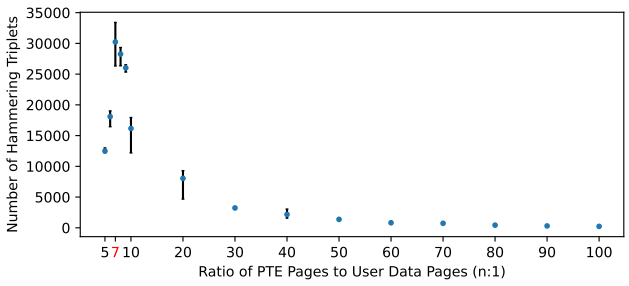

Figure 6: Average number of hammering triplets (i.e., Aggressor-Victim-Aggressor tuples) over 6 runs of our interleaved memory spraying technique with a varying ratio of page table pages to user data pages ($n:1$). Highlighted in red, the ratio (i.e., $7:1$) resulting in the most triplets.

and use shared file mappings in order to efficiently have many PTEs pointing to the same physical pages. To allocate user data pages, we allocate a large buffer with read/write permissions and fault in pages as needed. One key aspect we have to take care of is that our aggressor pages (i.e., the user data pages) must be resident in memory at all times and should not be swapped out to disk because, when they are paged back into memory, they might no longer be at the same physical location. Given that by exhausting physical memory we are putting the system under heavy memory pressure, this issue is very likely to arise as the kernel tries to free up physical memory to reclaim pages for other processes. To circumvent this problem, we use the `mlock` system call to force the OS to keep those pages resident in memory. In theory, the memory compaction daemon on Linux (`kcompactd`) *could* still migrate the aggressor pages and change the backing page frame for those particular mappings because page migration does not break `mlock` semantics. In practice, however, due to the fact that most of physical memory is allocated with *unmovable* page table pages and the *movable* freelists are almost depleted, we have never observed this behavior during our memory spraying experiments. We have also ensured that all user data pages remained resident in memory with the `mincore` system call.

In order to optimize the spraying phase for our attack, we run an experiment to find the best ratio of page table pages to user data pages. We run multiple iterations of our spraying techniques by varying the number of page table pages (n) to user data pages. The results of this experiment are shown in Figure 6. By averaging the number of 'hammering triplets' over 6 runs we see that the best ratio is $7:1$, which results in 30,238 patterns. As we decrease the number of user data pages (i.e., increase the PTE to user data ratio), the number of hammering triplets gradually decreases, reaching a value of 0 when $n \geq 80$. Therefore, we use the $7:1$ ratio for our end-to-end Rowhammer exploit.

It is important to note that by skipping the templating phase, we cannot know where the bit flips are going to occur within a specific row. Therefore, we want to make sure that when spraying we fill an *entire* victim row i.e., 8 KiB, with our target data (PTEs). Theoretically, a triplet in which the victim row has one PTE page co-located with a user data page in the same row is also a 'safe' hammering pattern, i.e., would not end up causing memory corruptions in memory that is not attacker-controlled. Another instance of a 'safe' triplet is one in which the aggressor rows only contain one user data page each and can be co-located with any other data on the same row and the victim row stores at least one page table page. However, in order to maximize the chances of bit flips, in the remainder of this section we will only take into account hammering triplets in which all the data stored in the aggressor rows is represented by user data pages and the victim row only contains PTEs. We will go into further detail on how we target 'interesting' bits within the PTEs in Section 7.5.

Over 20 runs of our spraying algorithm, on average, we can spray physical memory as desired with PTEs and user data pages on our experimental testbed in 144 seconds.

7.5. Rowhammer Patterns

Pattern Search. Once we have sprayed most of physical memory with page table pages and user data pages, we proceed with the search of the hammering triplets using PRELOAD+TIME. Firstly, we break KASLR using the prefetch side channel [8]. Then we start the search by picking one of the pointers which we use to map the PTEs and leak the physical address of the lowest level page table page as described in Section 6.4. Next, using our knowledge of DRAM addressing, we compute the physical addresses of the other 5 pages that make up a hammering triplet. Then, we use PRELOAD+TIME again to find the virtual address of the 'mate' PTE page i.e., the PTE page which is co-located on the *same* row as the first PTE page found in the earlier step. If we find two PTE pages which satisfy this condition, we proceed to find the virtual addresses of the remaining 4 user data pages which form the aggressor rows of the hammering triplet. Note that, this time, we fix a physical address because the pages forming a hammering triplet must be in specific DRAM locations in order to be useful, and use PRELOAD+TIME to search through all virtual addresses, rather than the other way around. In the event where we cannot find PTEs or user data pages at our desired physical addresses, we discard the address we picked initially and start the search again.

Aggressor Row Striping Data. As shown in prior research, including the first paper describing the Rowhammer phenomenon [10], bit flips are heavily data-dependent with a striping pattern, i.e., aggressors rows having the inverse value of the data in the victim rows, being the most effective. In order to maximize the probability of triggering bit flips in our victim page table pages, we can take advantage of PRELOAD+TIME to fill our user data pages with a data pattern to selectively target the page frame number (PFN) bits of the PTEs.

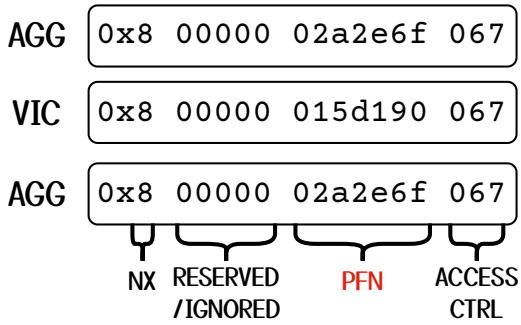

Figure 7: A striping pattern used for hammering, enabled by PRELOAD+TIME, where the bit flips that *could* cause our process to crash are 'masked out'.

The PTEs stored in the last level page table page can be easily predicted after having sprayed memory as explained in Section 7.4. Most of the access control bits such as the present, writable, NX bits etc... are known to our user process because we pass specific access rights to the `mmap` system call and use `mlock` to pin the pages in memory. The biggest source of entropy lies in the PFN bits (i.e., bits $[40:12]$). However, we can fully leak those bits with PRELOAD+TIME as explained in Section 6.2, meaning we can *fully* recover the PTE data, not merely its location. Once we know the data stored in the PTEs, we can effectively mimic a striping data pattern by storing the inverse value of the PFN bits in the user data (aggressor) pages. This allows us to precisely test for both bit flip directions ($0 \rightarrow 1$ and, $1 \rightarrow 0$) in the PFN bits in the PTE i.e., maximizing the chance that within one hammering round we can get useful bit flips. Furthermore, in order to minimize the chances of the attacker's process crashing, we make sure that the resulting PFN is *within* the physical address space therefore, we also 'mask out' the upper PFN bits to stop the PTE from pointing to any address above the largest PFN possible on the system. We do this by storing the same value (i.e., 0) in the bits which are needed to address content above the system memory limit.

An example of such a pattern is depicted in Figure 7. In the victim (middle) row, the first 64 byte value (i.e., a PTE) is stored. We can infer the control bits and we can leak the PFN bits with PRELOAD+TIME in order to have full knowledge of the PTE content. In order to maximize the chance of useful bit flips, in the aggressor rows, we store the same PTE value with the PFN bits inverted to mimic the striping pattern. I.e., the bit values in the columns matching those of the PFN bits in the victim rows are calculated by XORing the value with a mask $(PFN(vic) \oplus ((1 << \log_2(MAX\_PFN)) - 1))$.

7.6. Page Locality

On our experimental testbed, performing TLB evictions and triggering the half-Spectre gadget (due to contention) was very noisy. As such, reliably leaking PTE and user data physical addresses required many PRELOAD+TIME measurements per guess, considerably slowing down the attack. Finding one double-sided Rowhammer pattern as described above takes $\sim$10 minutes on average. This means that finding 5 hammering triplets to form a 10-sided pattern takes nontrivial time (i.e., 50 minutes).

As a speed optimization, we rely on the *locality* of the kernel page allocator to find Rowhammer patterns more efficiently without the need to pessimistically scan the whole direct map range with PRELOAD+TIME. Simply put: due to its internal page management, pages close in physical memory are more likely to be handed out by the allocator closely to each other. In order to confirm this behavior, we assign an allocation ID (*AID*) to both page table pages and user data pages—which we increment linearly during our spraying phase. To investigate this heuristic, we run our interleaved spraying strategy 20 times with the optimal PTE to user data pages ratio found in Section 7.4, i.e., $7:1$ and run a script which parses the data by finding the hammering triplets synthetically and records the AID of each page forming a triplet.

The results show that for 99.2% of triplets found, the two pages forming an aggressor row have a consecutive AID. In essence, this shows that once we are able to leak the physical address of one user page in the aggressor row, the other page residing in the same row in DRAM is at AID ± 1 with a probability close to 1. Similarly, for the page table pages which reside in the victim row of our hammering triplets, in 99.8% of the triplets found, the two PTE pages are at AID distance of 1. Restricting our search space to only those two pages allows for a speedup of almost 2x to find a suitable triplet.

7.7. Hammering the Triplets

Every time we find a pattern which satisfies our length requirements (in our case 10-sided) as described in Section 7.5, we start hammering the aggressor rows using the same hammering algorithm proposed by Frigo et al. [13]. If we find a bit flip and we can now read page table data by dereferencing the victim pointers using a crash-resistant oracle [19], we stop and we now have read/write access to the entire physical memory. If the hammering round does not succeed, we keep searching for hammering triplets and combine the newly found triplets with the old ones to form new 10-sided patterns.

We run our end-to-end exploit 5 times. On average, we find the first valid hammering pattern after $\sim$1 hour. The first successful bit flip (i.e., a bit flip in the PFN bits of a target PTE which now points to another PTE page in the system) occurs, on average, after 28 hours.

8. Mitigation & Discussion

Mitigating PRELOAD+TIME. Developing a comprehensive defense for Spectre-v1 in hardware is much more challenging than for other Spectre variants [6]. As such, Spectre-v1 mitigations leave the burden of reducing the attack surface

by means of software patches to OS vendors and programmers [2]. Similarly, there is no simple solution to mitigate half-Spectre gadgets in hardware without heavily crippling the performance benefits gained with branch prediction on modern CPUs.

In the context of PRELOAD+TIME, we identify two potential routes to protect existing and future systems. The first option, also proposed as a first-cut defense to mitigate MDS [45] is to disable SMT. Even though exploitation of half-Spectre gadgets does not strictly depend on SMT being enabled, disabling it would reduce the attack surface, since gadgets which need SMT contention to be exploited are no longer relevant. Another option is to include half-Spectre gadgets in the operating system's threat model. Similarly to Spectre-v1, mitigation entails patching dangerous gadgets with speculation-aware serialization or masking semantics, e.g., adding `array_index_nospec` in half-Spectre gadgets exploiting unsafe array access.

Mitigating Rowhammer. Even though numerous hardware- and software-based [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56] mitigations have been proposed since the discovery of the Rowhammer phenomenon, modern DIMMs are still vulnerable [15], [14], [13]. Ultimately, the best way to fundamentally mitigate Rowhammer would be by means of hardware defenses [48], [47], [49], [50], [53], [54] implemented by the memory vendors. Newer generation DIMMs (DDR5) are shipped with Refresh Frequency Management (RFM), which tracks the number of activations issued to a specific bank and, once this exceeds a predefined threshold, the memory controller issues refresh commands to DRAM. Unfortunately, the details of this defense are proprietary and recent work has demonstrated bit flips on DDR5 DIMMs as well [15]. This leaves a gap in the landscape of Rowhammer defenses, which is yet to be filled with a fully comprehensive solution to this vulnerability.

Future Work. In this paper, we have shown how an attacker can combine memory spraying and an half-Spectre enabled PRELOAD+TIME to build effective patterns to exploit Rowhammer. However, DRAM does not distinguish between architectural and speculative loads [57]. Therefore, an attacker could simply spray physical memory with page tables and, instead of having user data pages in the aggressor rows, they could trigger loads at arbitrary (aggressor) physical memory addresses via a half-Spectre gadget. This would enable an attacker to hammer memory directly via the direct map, including aggressor rows that are out of reach of (user page) memory spraying (e.g., because they already contain persistent kernel data). Nevertheless, for this approach to work, one would need a very high throughput gadget in which the attacker can mistrain the branch and issue tens of thousands of speculative loads to each aggressor row within a typical DRAM refresh window (i.e., 64ms). Even with the high-throughput gadget in Listing 2, we were unsuccessful in directly inducing bit flips via speculative loads. To further increase the half-Spectre gadget surface, future work should devise more sophisticated half-Spectre gadget scanners and reason about their exploitability characteristics.

Disclosure

We disclosed the issues detailed in the paper to the Linux kernel security team. The team replied that Linux does not intend to take any new actions for Rowhammer issues.

9. Related Work

9.1. Rowhammer

Since its discovery in 2014, Rowhammer has had plenty of attention from both academia and industry, leading to numerous exploits being developed. Following the first practical exploit by Seaborn and Dullien [17] which targeted page table pages in order to obtain privilege escalation, researchers have demonstrated how to leverage Rowhammer bit flips in order to hinder both integrity [18], [25], [26], [11], [24], [20], [58], [21], [23], [16] and confidentiality [30], [27], [59], [60].

ProbeHammer is the first end-to-end crash-free Rowhammer attack that hinders *integrity*. Other efforts have demonstrated crash-free guarantees in other scenarios. For instance, RamBleed [30] features an attacker hammering their own memory (i.e., in a crash-free fashion) and exploiting the data-dependent nature of Rowhammer bit flips to leak data. ECCploit [29] hammering patterns avoid inducing bit flips in memory locations which would be detected by ECC and potentially cause crashes during memory templating. Pinpoint Hammer [61] similarly exploits the data-dependent properties of Rowhammer to suppress unwanted bit flips. The latter also relies on memory templating which has been shown by Kogler et. al. [19] to be unreliable on modern DIMMs which implement on-die ECC.

As more researchers started to work on novel Rowhammer attacks, the community gained more insight on how to amplify the Rowhammer effect on various generations of memory modules. In particular, heavy reverse engineering efforts have been devoted to understanding how to construct the most efficient hammering patterns by precisely recovering DRAM geometry information [44], [58] and by uncovering the weaknesses in the mitigations implemented by hardware vendors [13], [14], [15], [19].

9.2. Memory Massaging

Both Drammer [18] and SpecHammer [59] devised techniques to massage the Linux buddy allocator into allocating page table pages vulnerable to Rowhammer next to attacker-controlled data. However, Drammer-style massaging is no longer feasible on modern Linux kernels due to the internal separation of the buddy allocator's freelists in "migrate types" (with user pages and page table pages being served by different pools). SpecHammer [59] overcomes this issue by means of memory exhaustion techniques, which,

however, are slow, not fully reliable, and not applicable to memory-limiting scenarios (e.g., containers). In contrast, ProbeHammer relies on interleaved memory spraying, which is more effective in quickly forming hammering triplets.

9.3. Spectre

Following the original Spectre disclosure [1], researchers have produced plenty of work studying and building new attacks exploiting Spectre gadgets in order to leak sensitive information [62], [45], [63], [64], [36]. Given the high impact of the vulnerability, a lot of analysis has been conducted on the gadgets that enable exploitation [3], [4], [31] in order to aid in mitigation efforts. Recent efforts by Wiebing et al. [31] and Hertogh et al. [36] have shown that gadgets that were previously deemed unexploitable by the community can still be used to build different Spectre attacks showing the need for more sophisticated analysis in order to properly defend existing and future systems from such vulnerabilities. In line with these recent findings, our PRELOAD+TIME primitive shows that half-Spectre gadgets can also be exploited in order to leak sensitive information which can be used to build more complex attacks.

9.4. Rowhammer Meets Spectre and Side Channels

In the last decade, researchers turned their attention to hardware failure mechanisms which are exploitable from software such as Rowhammer [10] and transient execution vulnerabilities, including Spectre [1]. In recent years, work combining both Spectre and Rowhammer has shown the power of such vulnerabilities in breaking confidentiality [59] and integrity [57], [19]. For example, Kogler et al. [19] showed how an attacker armed with a Spectre oracle can perform 'blind hammering' in order to bypass the limitations of the templating phase present in most Rowhammer attacks. Tobah et al. [59] demonstrated that an attacker can flip bits in victim data in order to relax the requirement that an attacker needs to have shared memory with the victim in order to leak data using a FLUSH+RELOAD-style cache-based covert channel. Another work by Tobah et al. [27] highlighted the presence of Rowhammer gadgets which allow an attacker to leak kernel data by flipping bits in kernel pointers to redirect them to sensitive data and use PRIME+PROBE in order to reduce (but not eliminate) the chances of their memory massaging technique resulting in crashes. In contrast to prior work, PRELOAD+TIME enables an attacker to mount a Rowhammer attack which does not crash the system and does not rely on memory templating and massaging. Finally, Zhang et al. [57] show that speculative loads can also be used to trigger Rowhammer bit flips on DDR3 memory.

10. Conclusion

In this paper, we reconsidered the security impact of half-Spectre gadgets by using them to build PRELOAD+TIME. We have shown how one can use PRELOAD+TIME as a powerful attacker primitive in order to build ProbeHammer, a crash-free Rowhammer attack which does not rely on memory templating or massaging. ProbeHammer structurally discards hammering patterns that may induce unintended bit flips and thus prevents corruption of unintended data (and system crashes). As countermeasures, we suggest kernel developers to include half-Spectre gadgets in their threat model and mitigate such gadgets, similarly to Spectre-v1 gadgets, by issuing code patches to further reduce the attack surface of transient execution vulnerabilities.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by NWO through project "INTERSECT" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] 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.

[2] Intel, "Affected processors: Guidance for security issues on intel® processors," https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html#tab-blade-1-1.

[3] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *NDSS*, 2022.

[4] J. Zomer and A. Sandulescu, "Finding gadgets for CPU side-channels with static analysis tools," https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md, 2023.

[5] "Kasper information page," https://vusec.net/projects/kasper.

[6] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019.

[7] Intel, "Microarchitectural data sampling," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/microarchitectural-data-sampling.html

[8] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *CCS*, 2016.

[9] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative dereferencing: Reviving foreshadow," in *FC*, 2021.

[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] 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.

[12] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-user-kernel-boundary Rowhammer through implicit accesses," in *MICRO*, 2020.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. Shin, "Go Go Gadget Hammer: Flipping nested pointers for arbitrary data leakage," in *USENIX Security*, 2024.

[28] J. Corbet. Pagemap: security fixes vs. abi compatibility. [Online]. Available: https://lwn.net/Articles/642069/

[29] 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.

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

[31] S. Wiebing, A. de Faveri Tron, and C. Giuffrida, "InSpectre Gadget: Inspecting the residual attack surface of cross-privilege spectre v2," in *USENIX Security*, 2024.

[32] D. Carpenter, "Finding spectre vulnerabilities with smatch," https://lwn.net/Articles/752408/, 2018.

[33] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&Half: Demystifying intel's directional branch predictors for fast, secure partitioned execution," in *IEEE S&P*, 2023.

[34] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing mitigations for branch target prediction attacks," in *IEEE EuroS&P*, 2023.

[35] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, 2017.

[36] M. Hertogh, S. Wiebing, and C. Giuffrida, "Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation," in *IEEE S&P*, 2024.

[37] The Kernel Development Community. Amd64 specific boot options. [Online]. Available: https://www.kernel.org/doc/Documentation/x86/x86_64/boot-options.txt

[38] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB; DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *USENIX Security*, 2022.

[39] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE S&P*, 2013.

[40] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE S&P*, 2019.

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

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

[43] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *IEEE DSN-W*, 2011.

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

[45] 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.

[46] A. Di Dio, K. Koning, H. Bos, and C. Giuffrida, "Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks," in *NDSS*, 2023.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] JEDEC, "Jesd79-5, ddr5 specification," 2024.

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

[55] 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.

[56] 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.

[57] Z. Zhang, Y. Cheng, Y. Zhang, and S. Nepal, "Ghostknight: Breaching data integrity via speculative execution," https://arxiv.org/pdf/2002.00524v1, 2020.

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

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

[60] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "Hammerscope: Observing dram power consumption using rowhammer," in *CCS*, 2022.

[61] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint rowhammer: Suppressing unwanted bit flips on rowhammer attacks," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.

[62] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *CCS*, 2019.

[63] E. Goktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative Probing: Hacking Blind in the Spectre Era," in *CCS*, 2020.

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

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

In this paper the authors present a deterministic, crash-free Rowhammer exploit that does not require any memory massaging or templating. To accomplish this, they developed a technique called Preload+time, which uses a "half-Spectre" gadget to recover the physical addresses of data. Using this new primitive, they are able to precisely flip bits in targeted memory locations, allowing them to flip bits in Page Table Entries without risking crashing the system.

A.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) Identifies an Impactful Vulnerability: This paper describes how "half-Spectre" gadgets, previously thought to be harmless, can be leveraged to exploit a modern Linux kernel.
2) Provides a Valuable Step Forward in an Established Field: ProbeHammer advances the state-of-the-art in Rowhammer attacks by enabling attackers to deterministically conduct Rowhammer attacks. Additionally, the Preload+time primitive for recovering physical addresses has significance to side-channel research more broadly.

A.4. Noteworthy Concerns

1) The paper demonstrates an attack on only a single DIMM, on one CPU model, and with the refresh rate artificially halved. This suggests that conducting a deterministic ProbeHammer attack may be substantially harder in practice.