SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks

Paper #312, 13 pages + references

Abstract—**The recent *Spectre attacks* have revealed how the performance gains from branch prediction come at the cost of weakened security. Spectre Variant 1 (v1) shows how an attacker-controlled variable passed to speculatively executed lines of code can leak secret information to an attacker. Numerous defenses have since been proposed to prevent Spectre attacks, each attempting to block all or some of the Spectre variants. In particular, defenses using taint-tracking are claimed to be the only way to protect against all forms of Spectre v1. However, we show that the defenses proposed thus far can be bypassed by combining Spectre with the well-known Rowhammer vulnerability. By using Rowhammer to modify victim values, we relax the requirement that the attacker needs to share a variable with the victim. Thus, defenses that rely on this requirement, such as taint-tracking, are no longer effective. Furthermore, without this crucial requirement, the number of gadgets that can potentially be used to launch a Spectre attack increases dramatically; those present in Linux kernel version 5.6 increases from about 100 to about 20,000 via Rowhammer bit-flips. Attackers can use these gadgets to steal sensitive information such as stack cookies or canaries, or use new *triple gadgets* to read any address in memory. We demonstrate two versions of the combined attack on example victims in both user and kernel spaces, showing the attack's ability to leak sensitive data.**

I. Introduction

Computer architecture development has long put emphasis on optimizing for performance in the common case, often at the cost of security. Speculative execution is one feature following this trend, as it provides significant performance gains at a detrimental security cost. This feature attempts to predict a program's execution flow before determining the correct path to take, saving time on a correct prediction, and simply rolls back any code executed in the case of a misprediction. However, such predictions may mistakenly speculate that malicious code or values are safe, allowing for attackers to temporarily bypass safeguards and run malicious code within misspeculation windows.

The potential of such speculative and out-of-order exploits was first demonstrated by Spectre [25] and Meltdown [31], which revealed a new class of vulnerabilities rooted in transient execution. These attacks have shaken the world of computer architecture and security, leading to a large body of work in transient execution attacks [4], [5], [24], [33], [42] and defenses [5], [38], [39], [46], [51].

Moving away from information leakage, Rowhammer [23] is a complimentary vulnerability that breaks the integrity of data and code stored in a machine's main memory. More specifically, the tight packing of transistors in DRAM DIMMs allows attackers to induce bit-flips in inaccessible memory addresses, by rapidly accessing physically-adjacent memory rows. Similarly to Spectre, Rowhammer has spawned numerous exploits [3], [11], [13], [17], [18], [27], [32], [34], [37], [40], [43], [45], [47], including the recent bypass of dedicated defenses, such as Targeted Row Refresh (TRR) [50] and Error Correcting Codes (ECC-RAM) [9].

While both Spectre and Rowhammer have been extensively studied individually, much less is known, however, about the combination of both vulnerabilities. Indeed, only one prior work, GhostKnight [55], has considered the new exploit potential resulting from combining both techniques. At a high level, GhostKnight demonstrates that despite their transient nature, speculative memory accesses can cause bit-flips in addresses that Rowhammer could not reach alone, resulting in bit-flips at those memory locations. However, GhostKnight only shows how Spectre can be used to enhance Rowhammer, and neglects to consider the complimentary question of how Rowhammer may be used to enhance Spectre. Noting that most modern machines are vulnerable to both Spectre and Rowhammer, in this paper we ask the following questions:

Can the Rowhammer vulnerability be used to strengthen Spectre attacks? In particular, can an attacker somehow leverage Rowhammer to alleviate Spectre's main limitation of having a gadget inside the victim's code with attacker controlled inputs? Finally, what implications do combined attacks have on existing Spectre mitigations?

A. Our Contributions

We demonstrate that Rowhammer and Spectre can, in fact, be combined to evade the proposed defenses and increase the number of exploitable gadgets in widely-used code. In what follows, we provide a high-level overview of this combined attack, called SpecHammer, and discuss our discovery of newly exploitable gadgets in the kernel.

Attack Methods. The core idea of SpecHammer is to trigger a Spectre v1 attack by using Rowhammer bit-flips to insert malicious values into victim gadgets. We present two forms of SpecHammer: the first relaxes the restrictions on ordinary Spectre gadgets (which will henceforth be called *double gadgets*), and the second uses new *triple gadgets* to provide arbitrary reads with just a single bit-flip.

Double Gadget Exploit. Ordinarily, Spectre v1 allows an attacker to send any malicious value to a Spectre gadget and read memory arbitrarily within the victim's address space. The main weakness of Spectre v1 is that it requires a gadget within the victim's code that uses an attacker controlled offset variable, limiting Spectre v1's attack surface. The target for the first version of SpecHammer, however, is a portion of code that meets all the requirements of a Spectre gadget, but *does not provide the attacker any direct way to control the victim offset*. By using Rowhammer, it is possible to modify the offset and trigger a Spectre attack on such victims to leak sensitive data. This attack eliminates Spectre v1's main weakness, allowing for exploits on a wider range of code.

Unfortunately, Rowhammer can be used to flip, at best, only a few bits for a given word of memory, limiting control the attacker has over the victim offset. Nonetheless, we demonstrate how the attacker, even with limited control, is still able to leak sensitive data. For example, it is feasible to flip bits in the offset such that it points to just past the bounds of an array. This allows for leaking secret stack data, such as stack canaries designed to protect against buffer-overflow attacks [10]. That is, we show how the double gadget exploit can be used to leak such secrets, bypassing stack protection mechanisms.

Triple Gadget Exploit. While the first exploit poses a threat to a common defense against buffer-overflow attacks, its scope is more limited than the original Spectre attack which leaked arbitrary memory in the victim's address space. The second type of SpecHammer attack, however, can be used to dump the data of any address in memory. This method relies on a *triple gadget*, which has similar behavior to the Spectre v1 gadget, except that it features a triple nested access. Using this, the attacker can modify an offset to point to attacker-controlled data. This data can be set to point to secret data, which leads to the use of secret data in a nested array access, just as is done in Spectre v1. The attacker-controlled data can be modified to point to any secret within the attacker's address space, including kernel memory when exploiting a triple gadget residing in the kernel. Thus, a single bit-flip allows for arbitrary memory reads, as opposed to the double gadget which is more restricted in what addresses it can leak.

Challenges. Implementing these SpecHammer attacks presents several key challenges:

1) We must find addresses containing useful Rowhammer bit-flips that can force a victim to access secret data under misspeculation.
2) We need to massage memory to force victims to allocate their array offset variables at addresses that contain these useful flips. For targets residing in the kernel, this means massaging kernel stack memory.
3) We must demonstrate that flipping an array offset value in a Spectre v1 gadget can leak data under misspeculation.
4) Finally, we need to find gadgets in sensitive real-world code to understand the impact of relaxing gadget requirements.

Challenge 1: Producing Sufficient Rowhammer Flips. SpecHammer requires bit-flips at specific page offsets in order to leak secret data. To that aim, we used the code repositories attached to prior work [16], [44], [48], [50] in order to test the susceptibility of DRAM DIMMs to Rowhammer attacks. Unfortunately, the amount of flips produced by these repositories suggests it is hard to find a DIMM with enough bit-flips to practically execute SpecHammer.

However, as we show in Section IV, we observe that all of these repositories make a key oversight regarding cached data: they first initialize victim rows, and then induce bit-flips *in DRAM* (not caches), but neglect to flush the victim cache line before checking for flips. This leads them to observe *cached data* when checking for flips, leaving many flips in the DRAM arrays unobserved. By correcting these oversights, we are able to increase the number of bit flips by 248x in the worst case and 525x in the best case on DDR3, and 16x in the best case on DDR4, demonstrating bit-flips are much more common than previous work would suggest. Not only does this allow us to run SpecHammer, but it also makes Rowhammer attacks more practical than previously thought.

Challenge 2: Stack Massaging. For the SpecHammer attack, the target for Rowhammer bit-flips is a variable used as an index into an array. Such offsets are most often allocated as local variables, meaning they are located on the stack. Rowhammer attacks rely on massaging targets onto physical addresses that are vulnerable to bit-flips. However, to the best of our knowledge, only one prior work [40] has demonstrated hammering stack variables, relying on memory deduplication to massage stack data as needed. With deduplication now disabled by default, SpecHammer thus requires a new way of massaging a victim stack into place. Furthermore, the most attractive targets for this attack are gadgets residing in the kernel, as they can be used to leak kernel data, and hence a *kernel stack massaging* primitive is highly desirable.

Yet, the prior examples of kernel massaging focused on PTEs, rather than the stack [43], or were performed on mobile devices, taking advantage of features exclusive to Android [47]. Thus, we develop new primitives for massaging both user and kernel stacks, in order to allow for stack hammering without the use of deduplication (Section V).

Challenge 3: Proof-of-Concept (PoC) Demonstration. As a proof of concept, we demonstrate (in Section VI) the variations of the attack on example artificial victims in both user and kernel spaces. We demonstrate the double gadget attack in user space and the triple gadget attack in kernel space due to each attack's applicability in its respective space. These PoC attacks act as the basis for eventual attacks on the gadgets already found in widely-used code, and show that the attack is capable of leaking data at a rate of up to 24 bits/s on DDR3 and 19 bits/min on DDR4.

Challenge 4: Kernel Gadgets. In order to better understand the effects of relaxing gadget requirements, we found the number of gadgets present in the Linux kernel, with the original Spectre v1 restrictions compared to the amount of SpecHammer gadgets. As shown in Section VII, we find that with the original requirements, there are about 100 ordinary, double gadgets, and only 2 triple gadgets. Modifying the function to search for gadgets vulnerable to our SpecHammer attack leads it to report about 20,000 double gadgets, and about 170 triple gadgets. Thus, we show the number of potential gadgets in the kernel is greater than previously understood.

Summary of Contributions. This paper makes the following contributions:

- Combining Rowhammer and Spectre to relax the crucial requirement of an attacker-controlled offset for Spectre gadgets, discovering more than 20,000 additional gadgets in the Linux kernel (Section III & Section VII).

- Development of new methods for precisely massaging a victim stack in user space, and for massaging kernel memory, allowing an attacker to exploit the numerous gadgets present in the Linux kernel (Section V).
- Correcting oversights made by prior Rowhammer techniques to improve bit-flip rate by 525x in the best case (Section IV).
- Demonstrating how SpecHammer gadgets can be used to obtain stack canaries for buffer-overflow attacks and how triple gadgets can be used to provide arbitrary reads from any memory address on example user and kernel space victims, respectively (Section VI).

II. Background

We present the necessary background information on Spectre and Rowhammer needed to understand the new combined attack, SpecHammer. Since Spectre relies on previous cache side-channels, relevant cache attacks are explained as well.

A. Cache Side-Channel Attacks

The cache was initially designed to bridge the gap between processor speeds and memory latency, but inadvertently led to a powerful side-channel exploited for numerous attacks [25], [35], [36], [52], [53]. By timing memory accesses, an attacker can tell whether data is being pulled from the cache (a fast access) or DRAM (a slow access), and can therefore observe a victim's memory access patterns.

Most relevant to SpecHammer is the Flush+Reload technique [53]. The goal is to use the cache to observe a victim's access patterns on memory shared by the victim and attacker. For example, if a victim accesses particular addresses dependent on a secret value (e.g., using bits of a secret key as an array index), understanding which addresses the victim accesses can leak valuable secret information.

The technique first prepares the cache by flushing any cache lines the victim may potentially access using the `clflush` instruction. Then the victim is allowed to run, and will only access particular addresses dependent on secret data, loading *only* the corresponding blocks into the cache. Next, the attacker accesses all blocks of memory the victim *may have* accessed, while timing each access. If the access is slow, it implies data needs to be moved from DRAM to the cache, meaning the victim did not access any addresses within the block. However, if the access is fast, data is being pulled from the cache, meaning the victim must have accessed an address corresponding to the same cache line. Thus, by taking advantage of the drastic timing difference in latency between a cache hit versus a cache miss, attackers can accurately discern which addresses a victim interacts with and, consequently, any secret data used to control which addresses were accessed.

B. Spectre

Speculative and Out-of-Order Execution. In order to improve performance, modern processors utilize out of order execution to avoid necessarily waiting for instructions to complete when subsequent instructions are ready to be run. In the case of linear execution flow, processors utilize *out of order* (OoO) execution, running instructions out of program order, and only committing instructions once all preceding instructions have been committed as well. When a program has branching execution paths that depend on the result of certain instructions, the processor uses *speculative execution*, predicting which path the branch will take. If the prediction is incorrect, any code run in the speculation window is simply undone, causing negligible performance overhead compared to not speculating at all.

Transient Execution Attacks. Running instructions before prior instructions have committed, due to OoO or speculative execution, creates a period of transient execution. Such transient execution windows have long been considered benign, as any code that should not have run is rolled back, and only proper code is committed. However, through the Meltdown [31] and Spectre [25] attacks, researches have recently demonstrated how OoO and speculative execution, can be used by attackers to force programs to run using malicious values, uninhibited by safe guards that only take effect *after* the transient execution is complete. By the time the code is rolled back, the malicious values have left architectural side effects (e.g. placed data in the cache) that can be used to leak data even through transient execution. SpecHammer focuses on Spectre and the domain of speculative execution.

```
if(x < array1_size){
  y = array1[x]
  z = array2[y * 4096];
}
```

Listing 1: Spectre v1 Gadget

Spectre Attacks. Spectre [25] presents multiple ways in which an attacker can exploit speculative execution. We focus on Spectre v1, which is illustrated with the following example. Assume the victim contains the lines of code shown in Listing 1 and `x` is an attacker-controlled variable. The attack requires first training the branch predictor to predict that the *if* statement will be entered. The attacker can then change `x` such that reading `array1[x]` accesses a secret value beyond the end of `array1`. Even though `x` may be out of bounds, the secret value will still be accessed thanks to speculative execution, as the branch predictor has been trained accordingly. While the data read from `array2` is never committed to `z`, speculative execution still causes `array2` to use the secret value `y` as an index and load data at ("secret" * 4096) + `array2 base address` into the cache.

The attacker then uses Flush+Reload [53] to check what cache line was pulled, to reveal the `array2` index, exposing the secret value. One key assumption this attack makes is that the attacker controls `x`, as she needs to change `x` to the malicious value used to access secret data via `array1`.

Prevalence of Gadgets. Since Spectre attacks rely on the presence of a *gadget* in the victim code, the prevalence of gadgets in sensitive code becomes a crucial question. Researchers have developed tools [19], [29], [51] to automate the process of finding gadgets within target code. For example, smatch [29], a kernel debugging tool, was extended with the capability to report Spectre v1 gadgets within the Linux kernel. On kernel version 5.6, smatch reports about 100 gadgets.

Followup Attacks. Upon Spectre's discovery, numerous papers emerged detailing how alternate variants could be used for new attack vectors [4], [7], [20], [24], [26], [33], [41], [42]. These included performing speculative writes [24], running a Spectre attack over a network [42], and combining Spectre with other side-channels to exploit "half gadgets" that require a single array access within a conditional statement [41].

C. Rowhammer

The Rowhammer bug [23] presents a way of modifying values an attacker does not have direct access to. The exploit takes advantage of the fact that DRAM arrays use capacitors to store bits of data, where a fully-charged capacitor indicates a 1 and a discharged capacitor indicates a 0. As transistors became smaller, DRAM became more dense, packing the capacitors closer together. [23] found that by rapidly accessing values in DRAM, causing them to be quickly discharged and restored to their original values, disturbance effects can increase the leakage rate of capacitors in neighboring rows. Thus, by rapidly accessing (or "hammering") an aggressor row, an attacker can discharge neighboring capacitors flipping 1s to 0s (or 0s to 1s) in neighboring memory locations.

DRAM Organization & Double-Sided Rowhammer. A DRAM array consists of multiple channels, each of which corresponds to a set of ranks, where each ranks holds numerous banks. Each bank consists of an array of rows made of capacitors containing the individual bits of data. While it is possible to cause flips by rapidly accessing single DRAM rows [17], it is much more efficient to use double-sided Rowhammer (i.e alternating between hammering two aggressor rows surrounding a single victim row). By increasing the number of adjacent accesses, the capacitor's leakage rate increases, drastically improving the efficiency of inducing flips. Double-sided Rowhammer requires hammering adjacent DRAM rows within the same bank. However, attackers cannot directly see the DRAM addresses of values they interact with. Instead, they can only see the virtual addresses. These are mapped to physical address, which are mapped to DRAM addresses.

Exploits. As with Spectre, Rowhammer inspired numerous exploits taking advantage of the ability to modify inaccessible memory. This began with Seaborn and Dullien [43] demonstrating how a flip can be used both to perform a sandbox escape, as well overwrite page table entries. Many exploits followed [1], [3], [17], [27], [32], [34], [37], [40], [45], [47], demonstrating how Rowhammer can be used for privilege escalation on mobile devices [47], flipping bits through a web browser using JavaScript [16], as well as remotely attacking a victim over a network [32], [45]. Gruss et al. [17] additionally showed how many Rowhammer defenses can be defeated.

GhostKnight. To the best of our knowledge, only one prior work, GhostKnight [55], has demonstrated how Spectre and Rowhammer can be combined for a more powerful attack. Since Spectre allows for accessing arbitrary memory within a given address space, GhostKnight made the observation that rapidly accessing a pair of aggressor addresses can cause flips in the speculative domain. This effectively increases Rowhammer's attack surface by allowing for bit-flips at addresses only reachable under speculative execution.

III. SpecHammer

Our combined SpecHammer attack shows how Rowhammer can be used to enhance Spectre, bypassing proposed defenses and relaxing the requirements for a Spectre v1 gadget. We present two versions: a double gadget attack and triple gadget attack, each striking a different trade-off between the attack's capabilities and the assumptions made regarding the availability of gadgets in the victim's code.

A. Double Gadget Attack: Removing Attacker Control

As discussed in Section II, a key limitation of Spectre v1 is that the attacker must control a variable used as a victim array index. We relax this restriction by using Rowhammer to modify the index variable without direct access.

```
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

Listing 2: Pseudocode double gadget

Attack Overview. At a high level, the goal of the double gadget exploit is to mount Spectre v1 attacks even if the attacker does not have direct control over the array offset. We use Rowhammer to modify this offset value, causing an array to access secret data and leak it via a cache side-channel.

Listing 2 presents a gadget exploited by the first version of our attack, which uses the same gadgets as Spectre v1. In addition to assuming the presence of such code gadgets in the victim's code, we also assume that the victim's address space contains some secret data. Finally, unlike the Spectre v1 attack, we do not assume any adversarial control over the values of `x`. Rather than controlling `x` directly, the attacker instead exploits Rowhammer to trigger a bit-flip in the value of `x`, such that `array1[x]` accesses the secret data.

Step 1: Memory Templating. The first step in any Rowhammer-based attack is to template memory in order to find victim physical addresses that contain useful bit-flips, i.e., a flip that will cause `x` to point to the desired data. As described in Section II, templating essentially consists of hammering many physical addresses until finding a pair of aggressors that correspond to a victim row with a useful flip. After finding a physical address with a suitable flip, our memory massaging technique (see Section V) is used to ensure that the value of `x` resides in this physical address, making it susceptible to Rowhammer-induced bit-flips.

Step 2: Branch Predictor Training. After placing the victim's code in a Rowhammer-susceptible location, the attacker trains the victim's branch predictor by executing the victim code normally. As we are executing the victim's code with legal values of `x`, it is the case where `x < array1_size`, which results in the CPU's branch predictor being trained to

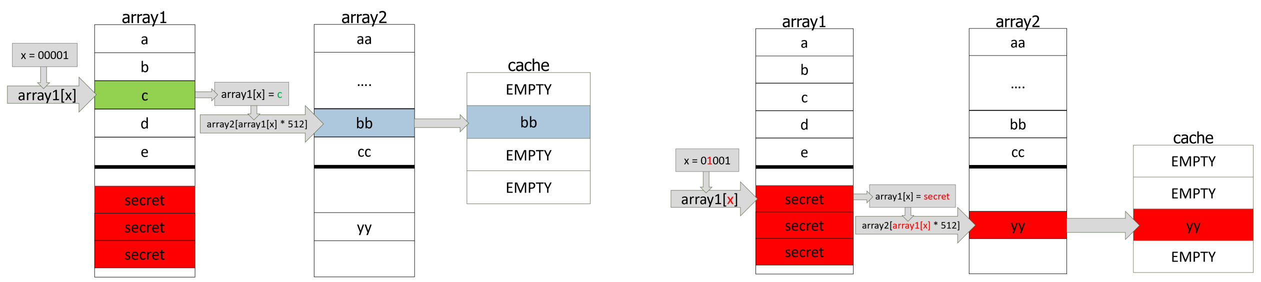


Fig. 1: Example attack scenario. (left) Training phase with legal value. (right) Attack phase with malicious value.

predict that the `if` in the first line of Listing 2 is taken. See Figure 1(left) for an illustration.

Step 3: Hammering and Misspeculation. Next, the attacker hammers `x`, leading to the state in Figure 1(right), where a bit-flip (marked in red) increases the value of `x` such that it points to the secret data past the end of `array1`. It is also necessary for the attacker to evict the value of `x` from the cache beforehand, ensuring the next time it is read, the flipped value in DRAM is used, as opposed to the previously cached value. After evicting `array1_size`, the attacker triggers the victim's code. As `array1_size` is not cached, the CPU uses the branch predictor, and speculates forward assuming that the `if` in Line 1 of Listing 2 is taken. Next, due to the bit-flip affecting `x`, the access to `array1` uses a malicious offset, resulting in `secret` being used as `array2`'s index, thereby causing a secret-dependent memory block to be loaded into the cache. Finally, the CPU eventually detects and attempts to undo the results of the incorrect speculation, returning the victim to the correct execution according to program order. However, as discovered by Spectre [25], the state of the CPU's cache is not reverted, resulting in a `secret`-dependent element of `array2` being cached. See Figure 1(right).

Step 4: Flush+Reload. To recover the leaked data from the speculative domain, the attacker uses a FLUSH+RELOAD side channel [53] in order to retrieve the secret. More specifically, the attacker accesses each value of `array2` while timing the duration of each memory accesses. Since all values of `array2` were previously flushed from the cache, the attacker's timed access should be slow if no accesses happened between the eviction and this stage of the attack. However, if a timed access is fast, that memory block must have been recently accessed. In this case, due to the access to `array2[secret*512]` during speculation, the attacker should observe a fast access when measuring the offset `secret*512`, thereby learning the value of `secret`.

B. Triple Gadget Attack: Enabling Arbitrary Memory Reads

The attack presented in Section III-B assumes that the attacker can use Rowhammer to flip arbitrary bits in the victim's physical memory. In practice, however, Rowhammer-induced bit-flips are not sufficiently common to flip the number of bits required for leaking arbitrary addresses. An attacker can flip, at most, a few bits of the array offset, limiting the addresses she can reach. In order to provide for arbitrary reads even with the limited control provided by Rowhammer, we develop another variation that utilizes "triple gadgets". With just a single bit-flip, an attacker can use a triple gadget to point an array offset to attacker controlled data. This data can then be set to point to any value in memory, allowing an attacker to leak arbitrary data with a single flip, as detailed below.

```
1  if(x < array1_size){
2    attacker_offset = array0[x]
3    victim_data = array1[attacker_offset]
4    y = array2[victim_data*512];
5  }
```

Listing 3: Pseudocode triple gadget

Attack Overview. For the triple gadget attack, we utilize a new type of code gadget; see Listing 3 for an example. At a high level, while the original Spectre v1 assumed that an attacker controlled variable `x` is used by the victim for a nested access into two arrays (e.g., `array2[array1[x]]`), here we assume that the victim performs a triple nested access using `x`, namely, `array2[array1[array0[x]]]`.

By using such gadgets, the attacker can modify the inner-most array offset (`x`) such that `array0[x]` points to attacker controlled data. This, in turn, allows her to send arbitrary offsets to `array2[array1[ ]]`, resulting in the ability to recover arbitrary information from the victim's address space. More specifically, our attacks proceeds as follows.

Steps 1+2: Memory Profiling and Branch Predictor Training. As in Section III-A, the attacker starts by profiling the machine's physical memory, aiming to find physical addresses that contain useful bit-flips. The attacker then executes the victim's code normally, thus training the branch predictor to observe that the `if` in Line 1 of Listing 3 is typically taken.

Step 3: Hammering and Misspeculation. Next the attacker hammers `x`, leading to the state in Fig. 2, in which a bit-flip (marked in red) increases the value of `x` such that it points past the end of `array0`, into attacker controlled data. As in the case of Section III-A, the attacker triggers the victim's code after evicting `array1_size`, which causes the CPU to fall back onto the branch predictor, speculatively executing the branch in Line 1 of Listing 3 as if it was taken. The attacker controls the value in address `array0+x`, which results in an attacker-controlled value being loaded as the output of `array0[x]` in Line 2. Proceeding with incorrect speculation, the CPU executes `array1[array0[x]]` (Lines 2 and 3), resulting in the attacker controlling (through `array0[x]`) which address the victim loads from memory. The value of `array1[array0[x]]` is then leaked through the cache side channel, following the access to `array2` in Line 4.

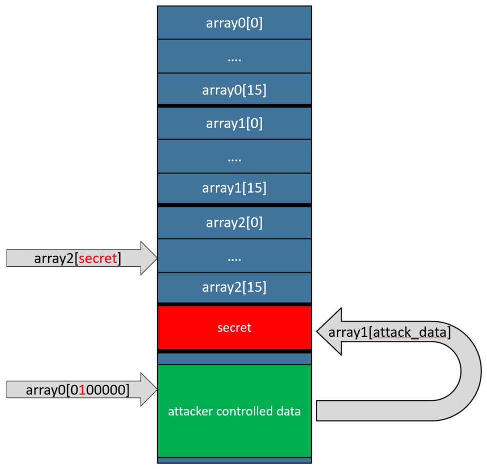


Fig. 2: **Triple gadget example**

Step 4: Flush+Reload. Finally, as in the case of Section III-A, the attacker uses a FLUSH+RELOAD side channel in order to leak the value accessed during speculation.

Comparison to Double Gadgets. While the triple gadgets require a triple-nested array access inside the victim's code, they also offer the advantage that multiple precise bit-flips are no longer needed for reading the victim's data. In particular, as only one bit-flip is used to point `array0[x]` into attacker-controlled data, multiple values can be read using the same bit-flip value. By varying the value of `array0[x]` and launching the attack repeatedly, the attacker can dump the entire victim address space using a single carefully controlled bit-flip.

Kernel Attacks. This attack is particularly dangerous when performed on a gadget residing in the kernel, as a single bit-flip can be used to read the entire kernel space. At first blush, it may seem that Supervisor Mode Access Prevention (SMAP), which prevents *kernel-to-user* accesses, will prevent the attack by disallowing the kernel from accessing the user-controlled data on line 2 of Listing 3. However, in Section VI-B we show how to bypass this mitigation, demonstrating how an attacker can use syscalls to inject data into the kernel, and afterwards use a single bit-flip to point from the gadget to this controlled kernel data. Since SMAP does not block *kernel-to-kernel* reads, this technique allows for performing the triple gadget attack even with SMAP enabled.

IV. MEMORY TEMPLATING

The high level description provided in Section III assumes two key prerequisites. First, the *memory templating* step is used to find useful flip-vulnerable address. Next, the *memory massaging* step is used to force the target victim variable to use this address. In this section, we describe the memory templating process, deferring stack massaging to Section V.

The goal of templating is to obtain "useful" bit-flips, meaning they can be used to flip an array offset variable and trigger a SpecHammer attack. Vulnerability to bit-flips depends on the nature of an individual DIMM, requiring hammering many addresses to learn which ones contain useful flips. The techniques used for templating borrow largely from existing work, and we therefore keep the descriptions high-level, referring readers to the appropriate prior work [27], [37] and giving a more detailed description in Appendix A.

A. Obtaining DRAM row indices from virtual addresses

As explained in Section II, Rowhammer is drastically more effective when two aggressor rows that pinch a victim row are hammered in succession, a technique called double sided hammering [23]. Finding flips via double sided Rowhammer requires controlling three consecutive DRAM rows. However, as unprivileged attackers, we have no direct way of determining how our virtual pages map to DRAM rows, preventing us from performing double sided hammering. We must therefore reverse engineer this mapping before we can begin hammering. Since virtual address map to physical addresses, which in turn map to DRAM rows, we must obtain both the *virtual to physical* and *physical to DRAM* mappings.

For the latter, we use Pessl's DRAMA technique [37]. For the former, we only need the physical address bits used to determine the corresponding rank, bank, and channel. For a Haswell processor using DDR3, these are the lowest 21 bits. Thus, we can use the techniques presented in RAM-Bleed [27], to obtain a conitguous 2MiB page, giving us the lower 21 physical address bits. Since this technique relies on the recently restricted `pagetypeinfo` file, we use a new technique that relies on the world-readable `buddyinfo` file instead (see Appendix A) The time required for this step is unaffected by using the new `buddyinfo` technique.

For newer architectures that use DDR4 memory, we follow the methodology of TRRespass [14], using transparent hugepages which are enabled by default in Linux kernel version 5.14, the latest version at the time of writing. Note that for a one-DIMM configuration, only up to bit 21 is needed, even on newer architectures. For two-DIMM configurations, it is possible to use memory massaging techniques to obtain up to 4MB of contiguous memory.

B. Hammering Memory

With all the obtained memory sorted into rows, we initialize the aggressors and victims with values reflective of our desired flips. In our case, we seek to increase an array offset value to point to secret data, meaning we want to flip a particular victim bit from 0 to 1. We therefore initialize potential victim rows to contain all 0s. Since double sided hammering is most effective when the victim bit is pinched between two bits of the opposite value [23], [27], we set aggressor rows to all 1s, giving a 1-0-1, aggressor-victim-aggressor stripe configuration.

Inducing Flips. As done in prior work and existing Rowhammer templating code [16], [44], [50], [54], we repeatedly read and flush aggressor rows from the cache to ensure each read directly accesses DRAM and causes disturbance effects on neighboring rows. After doing a fixed number of reads, we read the victim row to check for any bit-flips, which in this case would mean a bit set to 1 anywhere in the victim row's value. We save addresses containing useful flips (i.e., a bit-flip that would cause an array offset to point to a secret), and move onto the memory massaging phase. Note that the above steps *neglect to flush the victim address cache lines*. Consequently, when we try to read the victim to check if we induced a flip, we will likely be reading cached initial data.

The Need for Useful Flips. Upon running existing Rowhammer code [16] on numerous DDR3 DIMMs, we experienced a somewhat low flip-rate of approximately 2 to 5 flips per hour. However, for our SpecHammer attack, we require specific bit-flips (a single bit position out of a 4KiB page), to point from an array to a secret, meaning it would take an infeasibly long amount of time to find the required bit in the average case. One option to overcome this would be to test many DIMMs until finding one particularly susceptible to Rowhammer, limiting the attack only to such susceptible DIMMs. However, we observed an oversight in existing Rowhammer repositories pertaining to the issue of cached victim data, which causes a susceptible DIMM to *appear* sturdy against flips, when, in fact, a vast majority of flips are simply being *masked* by cached data. By modifying these existing repositories, we found that the same DIMMs are vulnerable to thousands of flips per hour, allowing us to perform our attack on DIMMs that were previously thought to be safe.

Under-reported Flip-rate in Prior Work. Upon inspection of numerous public Rowhammer repositories [16], [44], [50], [54] designed to test a DIMM's vulnerability to Rowhammer, we observed that they all made the victim row cache oversight mentioned in the previous paragraph. By performing the above steps, reading a victim row to check for a bit-flip will likely result in reading the cached initialization data, leading to severe under-reporting of the actual number of flips obtainable on any tested DIMMS. Any flips that are reported are likely due to victim data being unintentionally evicted from the cache due to other memory accesses replacing those cache lines. In Appendix C we describe experiments we conducted to prove that cache effects are indeed responsible for masking bit flips.

Comparison of Rowhammer Techniques. In order to fully understand the effect this oversight had on finding bit-flips, we compared prior work with our victim cache flush modification.

The results are presented in Table I. We ran each program using double sided hammering over a two hour period with a 1-0-1 stripe configuration, then for 2 hours testing for using 0-1-0. The total flips over both runs are shown in the table.

Note that the repository for Rowhammer.js [16] contains an error that uses *virtual addresses* rather than *physical addresses* when determining which addresses reside on the same bank, and is thus split into 2 entries: one for the unmodified Rowhammer.js and the other for the same code with the error removed excluding the cache flush oversight. Finally, we used TRResspass [14], the latest Rowhammer templating repository, exclusively for DDR4, since it uses techniques designed to bypass DDR4 exclusive defenses. The changes we made to these repositories are detailed in Appendix B.

We perform our DDR3 experiments on a Haswell i7-4770 CPU with Ubuntu 18.04 and Linux kernel version 4.17.3. For the DDR4 experiments, we use a Coffee Lake i7-8700K CPU with Ubuntu 20.04 and Linux kernel version 5.8.0.

Results. For DDR3, when compared to Rowhammer.js with the addressing error removed, our code improved the flip rate by 248x in the worst case, and by a factor of x525 in the best case. As for TRResspass, we found that modifying the the code to include victim cache flushes resulted in 6x to 8x flips on DDR4 DIMMs. While prior Rowhammer surveys have found larger numbers of flips [8], [22], they did so using techniques unavailable on general purpose machines. In the case of [22], the goal was to understand DIMMs vulnerability to Rowhammer at the *circuit* level, and thus DIMMs were tested via FPGAs to remove higher-level sources of interference that may have reduced the number of flips. Similarly, [8] sought to achieve flips on servers, and their techniques can only work on multi-socket systems. In contrast, we use code that is designed for users to test their own machines for Rowhammer bugs, and show how ensuring that the victim row is flushed before it is checked can drastically increase the number of flips.

In order to verify these additional flips were a result of cache flushing, we performed additional experiments to verify that data was in fact being pulled from memory and not the cache for each flip. These experiments are detailed in Appendix C.

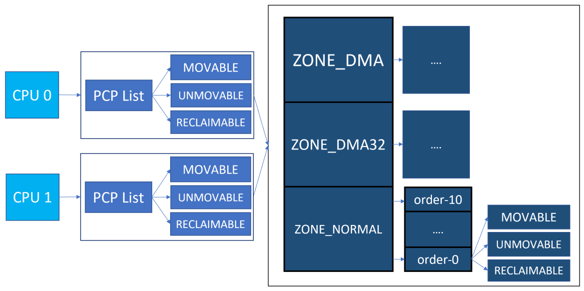


Fig. 3: **Linux memory organization**

V. Memory (Stack) Massaging

With possession of a useful, flip-vulnerable address, the next step is to force the victim variable into this address. The target victim is a variable used as an offset into an array. Such variables are most often allocated as local variables, and hence reside on the victim's stack. Therefore, in order to flip such variables and trigger the attack, we need to place the victim's stack on the flip-vulnerable page obtained from the templating step. Only one prior work has demonstrated stack massaging [40], and used (the now-disabled) page deduplication to do so.

Note that bit-flips correspond to particular DRAM addresses, which are fixed to specific *physical address*. Physical addresses, however, can be mapped to various different virtual addresses through a page table mapping. Thus, the goal is to force the victim to use a particular *physical page*.

Furthermore, if the victim resides in kernel code, the attacker needs to massage *kernel stacks* which adds an additional layer of complexity compared to massaging user space stacks, since an unprivileged attacker cannot directly manipulate (allocate and deallocate) kernel pages. While prior work has demonstrated kernel massaging by forcing PTEs to use certain pages, they use methods too imprecise for kernel stack massaging [43]. This existing technique simply unmaps the flip-

Model	Samsung (DDR3)	Axiom (DDR3)	Hynix (DDR3)	Samsung (DDR4)	Samsung (DDR4)	Samsung (DDR4)
rowhammer-test [44]	1	0	0	-	-	-
rowhammerjs [16]	4	9	2	-	-	-
rowhammerjs (corrected addresses)	15	38	32	-	-	-
rohammerjs with victim flushes	7,883	11,005	7,943	-	-	-
TRResspass [50]	-	-	-	947	2,976	2,134
TRResspass with victim flushes	-	-	-	7,916	17,958	15,611

TABLE I: Comparison between prior Rowhammer techniques and our new cache-flushing technique. Since the techniques listed in the top 4 rows are designed for DDR3, we did not run them on DDR4 DIMMs. Similarly, TRResspass is designed for DDR4 and was not run on DDR3. Note that rowhammerjs refers to the code in its "native" directory.

vulnerable page and fills physical memory with PTEs until one uses the recently unmapped page. For kernel stack massaging, new threads need to be spawned to allocate kernel stacks. Since spawning new threads is resource-intensive (relative to PTE allocation), we cannot spray a majority of memory with stack threads and must manipulate memory into a state that that maximizes the odds of a *limited* spray using the target page. Other prior work has demonstrated more deterministic techniques, but uses methods exclusive to Android [47].

In this section we develop a novel technique for massaging *kernel* memory by taking advantage of Linux's physical page allocator, the "buddy allocator" (see Appendix A), and its per-CPU (PCP) list system. Before describing our technique, we provide background on the memory structures we manipulated to achieve our result. An overview is shown in Figure 3.

Memory Zones. Within the buddy allocator, pages of memory are organized Within the buddy allocator, in addition to being sorted by order, free pages are also sorted by their *zone*. Zones represent ranges of physical addresses. Each zone has a particular *watermark level* of free pages. If the zone's total free memory ever drops below the watermark level, requests are handled by the next most preferred zone. For example, a process may request pages from ZONE_NORMAL, but, if the number of free NORMAL pages is too low, the allocator will attempt to service the request from ZONE_DMA32 [15].

Page Order. Within each zone, pages are sorted into blocks by *size*, also called their *order*, where an order-x block contains 2^x contigous pages. The allocator always attempts to fulfill requests from the smallest order possible, but if no small order blocks are available, a larger block will be broken in half, and one half is used to fulfill the request [15].

Migrate-types. Pages are further organized by *migrate-type*. Migrate-types determine whether the virtual-to-physical address mapping can be changed while the page is in use. For example, if a process controls virtual pages that map to physical pages with the migrate-type `MOVABLE`, it is possible to replace the physical page, by mapping the same virtual address to a different physical address [28].

PCP Lists. Finally, the PCP list (also referred to as the *Page Frame Cache*) [6] is essentially a cache to store recently freed order-0 pages. Each CPU corresponds to a set of first-in-last-out lists organized by zone and migrate-type. Whenever an order-0 request is made, the allocator will first attempt to pull a page from the appropriate PCP list. If the list is empty, pages are pulled from the order-0 freelist of the buddy allocator. When pages are freed, they are always placed in the appropriate PCP list. Even if a contiguous higher-order block is freed, each individual page is placed on a PCP list, and they are merged only when they are returned from the PCP list to the buddy allocator freelist. Thus, the system serves to quickly fetch pages that were recently freed on the same CPU, rather than needing direct access to the buddy allocator.

A. User Space Stack Massaging

Building on existing user space massaging techniques [6], [27], the main goal is to free the flip-vulnerable page currently in the attacker's possession, and then force a victim allocation that will use the recently freed page. In the case of stack massaging, this means forcing a new stack allocation. The techniques presented here follow similar steps as those done in prior work [6], [27]. While prior works use this process to massage pages allocated via `mmap`, we massage victim stacks.

Stack allocation. User space stacks are allocated upon spawning a new process or thread, and use ZONE_NORMAL, migratetype MOVABLE memory. Additionally, even though they typically use more than one page, the request is handled as multiple order-0 requests, meaning pages are pulled from a PCP List. Pages obtained from `mmap` calls in user space also use NORMAL, MOVABLE memory, meaning stack pages and the controlled flip-vulnerable page are of the same type. Therefore, freeing the flip-vulnerable page via `unmap` will place the page in the same PCP list used for stack allocation.

Massaging Steps. Now understanding Linux stack allocation, stack massaging is performed using the following steps:

Step 1: Fodder Allocations. First, we make "fodder" allocations to account for any allocations made by the victim before allocating the stack. It is possible the target variable does not reside on the first page of the victim's stack. Therefore, we must first calculate how many pages will be used by the victim before the victim allocates the stack page containing the target, and allocate such number of fodder pages.

Step 2: Unmapping Pages. We then free the flip-vulnerable page, placing it in the PCP List, and then free the fodder pages, placing them in the same list above the flip-vulnerable page.

Step 3: Victim Allocation. Finally, we spawn the victim process, forcing it to perform the predicted allocations, and target stack allocation. Any allocations that occur prior to the target allocation will remove the fodder pages from the PCP List, forcing the stack to use the target page.

Results. This technique works with about 63% accuracy, which is acceptable since it only needs to be done once to mount the attack. If this step fails, we can attempt massaging again, and expect it to succeed within two tries. We can check for a massage failure by running the subsequent steps of the attack (i.e. calling the victim containing the gadget and hammering our aggressors) and checking for data on the cache side-channel. If no data is observed, we re-attempt massaging.

B. Kernel-Space Stack Massaging

Targeting gadgets in the kernel similarly requires forcing stack variables to use specific, flip-vulnerable pages. Like with user-space stack allocation, a kernel stack is allocated upon creation of a new thread or process, and that stack is used for all syscalls made by that thread or process. However, unlike user-space stacks, kernel stacks use UNMOVABLE memory, meaning they pull pages from PCP list different from that used by user space `mmap` and `unmap` calls. Therefore, the attacker needs a method to force the kernel to use "user pages" (MOVAVABLE pages) instead of "kernel pages" (UNMOVABLE pages). We observe from Seaborn [43] that the kernel does use user pages when memory is under pressure, and build on Seaborn's techniques to allow for a more precise memory massaging technique that allows for massaging kernel stacks.

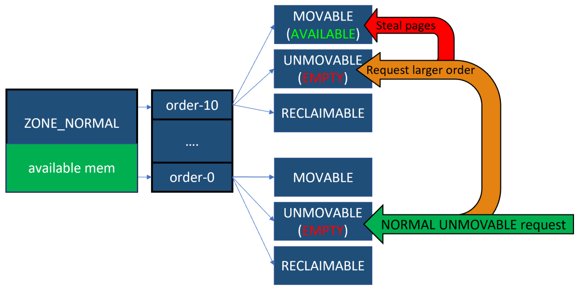


Fig. 4: **Physical Page Stealing**

Allocator Under Presure. As mentioned above, when the zone's total number of free pages falls below the watermark, the next most preferred zone is used. However, as zones include multiple migrate-types, it is possible for the freelist of the requested migrate-type to be empty, yet have enough total zone memory to be above the watermark. In this case, the allocator calls a *stealing function* that steals pages from given "fallback" migrate-types and converts them to the type originally requested. As shown in Fig. 4, this function attempts to steal the largest available block from the fallback type. For UNMOVABLE memory, the first fallback is RECLAIMABLE memory, and the second is MOVABLE memory.

Kernel Massaging Steps. The steps required for kernel stack massaging are similar to those of user space stack massaging. The key difference is that the attacker must first apply memory pressure to force the kernel into using user pages.

Step 1: Draining Kernel Pages. As non-privileged attackers, we cannot directly allocate UNMOVABLE pages. However, each time an allocation is made via `mmap` a page table entry (PTE) is needed to map the virtual and physical pages. Since PTEs use kernel memory, each mmap call uses both user and kernel memory. However, multiple PTEs can fit within a single page, and the address of a PTE depends on its corresponding virtual address. We need to efficiently make allocations large enough such that each PTE needs a new page, but small enough such that the process is not killed for allocating too much memory. Mapping pages at 2MB aligned addresses provides the smallest allocation size such that each PTE allocates a new page. Such allocations are made until no MOVABLE pages remain, using the `pagetypeinfo` file to monitor the amount of remaining pages. Subsequent mappings will use RECLAIMABLE pages for PTEs. Once the necessary pages have been depleted, the next kernel allocation will use the largest available MOVABLE block.

On machines without access to `pagetypeinfo`, we instead use `buddyinfo` (which is world readable for all kernel versions) and monitor the draining of MOVABLE and UNMOVABLE blocks together (performing Step 1 and Step 2 at the same time), only draining order 4 or higher UNMOVABLE blocks. (See Appendix A for a more detailed explanation of `buddyinfo` compared to `pagetypeinfo`.

Step 2: Draining User Pages. Memory is now in a state that will force the kernel to use the largest available MOVABLE block. However, we need the kernel to use a specific single page (the page containing a bit-flip). We, therefore, need to ensure the target page resides in this block. It is advantageous to make the largest available block as small as possible to improve the chance that the kernel uses the target page for its stack allocation. Thus, the next step is to drain as many high-order free blocks as possible, without dropping the total number of free-pages below the watermark. In our machine, we were able to drain all blocks of order 4 or higher.

Step 3: Freeing Target Page. The goal is to free the target page such that it resides in the largest available block. However, freeing this page will send it to the PCP rather than the buddy allocator freelist. Even when it is free from the PCP, if it does not have any free buddies, it will remain in the order-0 freelist. The freed target page needs to coalesce into an order-4 block, such that the single largest remaining free block contains the flip-vulnerable target page. Fortunately, as explained in Section IV, we have already guaranteed the target page is part of an order-4 (or larger) block (i.e. our target page is part of a 2MiB, order-10 block). Therefore, we can free the target page and all of its buddies to ensure it will coalesce into the largest available block.

The last obstacle is the PCP list, since even when unmapping a contiguous high-order block, all pages are placed on the appropriate PCP list. However, the `zoneinfo` file shows how many pages reside in each PCP list, and the maximum length of each list, at any given time. Thus, additional pages can be unmapped until the number of pages in the PCP list reaches the maximum length (186 pages on our machines according to `zoneinfo`). This forces pages to be evicted from the PCP list and sent to the buddy allocator freelist, placing the target page in the largest free block of MOVABLE memory.

Step 4: Allocating Kernel Stack. Having freed the target page, and knowing the next kernel stack allocation will use user memory, we can now force a kernel stack allocation. However, freeing pages to force the target page out of the PCP will have slightly alleviated memory pressure, meaning some UNMOVABLE pages will be free. Kernel stack allocations will consume these pages, and subsequent allocations

Experimental Configurations	SMAP	pagetypeinfo	THP	Leakage
i7-4770,DDR3,Linux 4.17.3	OFF	Readable	N/A	20b/s
i7-7700, DDR4, Linux 5.4.1	ON	Restricted	madvise	6b/m
i9-9900K, DDR4, Linux 5.4.1	ON	Restricted	madvise	6b/m
i7-10700K,DDR4,Linux 5.4.0	ON	Restricted	madvise	6b/m

TABLE II: List of configurations used for our experiment. All mitigations are in their default configurations.

will convert the block containing the target page into an UNMOVABLE block. Additionally, because of the kernel's buddy system, the block will be split in half, with one half being used for the kernel stack, and the other half moved to the lower order UNMOVABLE freelist. The target page may be in either half, and allocations must continue to be made to ensure the target page is used for a kernel stack.

Therefore, we use a *kernel stack spray*, allocating many kernel stacks until the UNMOVABLE pages are all depleted again. We perform the kernel stack spray by spawning many threads. Each thread can spin in an empty loop until the spraying is done, and then be tested one-by-one by having the thread make the victim syscall and hammering the target variable until we observe a leak. Once the thread with the target page is found, the other threads are released. We can now flip a stack variable residing in the kernel.

Results. This technique has approximately 66% accuracy with the `pagetypeinfo` technique (60% accuracy with `buddyinfo`). We expect it to succeed within two attempts.

VI. Gadget Exploitation

At this point, we have forced victim stacks in both user space and kernel space to use flip-vulnerable addresses. We can now flip array offset values, force a misspeculation, and leak target values. As a proof of concept, we demonstrate end-to-end double and triple gadget attacks on example victims in user and kernel spaces. These examples serve to verify the attack's ability to leak data. The double gadget attack is demonstrated in user-space, and the triple gadget in the kernel.

Setup. For the double gadget attack, we use a Haswell i7-4770 CPU with Ubuntu 18.04 and Linux kernel version 4.17.3, the default version shipped on our machine. The DRAM used consists of a pair of Samsung DDR3 4GiB DIMMs. For the triple gadget attacks, we use the same machine in addition to machines with Kaby Lake i7-7700, Coffee Lake Refresh i9-9900K, and Comet Lake i7-10700K processors. The latter three machines each use a DDR4 8GB DIMM and run Linux Kernel version 5.4.1, 5.4.1, and 5.4.0, respectively. These configurations are shown in Table II. Note that the two newer processors have additional defenses (i.e., SMAP and restricted `pagetypeinfo` access) not supported by Haswell. We demonstrate our attack even in the presence of such defenses. KASLR is enabled on all machines. Additionally, transparent hugepages (THPs) are set to their default setting of being user-allocatable via an madvise syscall.

A. Double Gadget – Stack Canary Leak

In this section we demonstrate how stack canaries can be stolen using a double gadget residing in user space code.

Stack Canaries. A stack canary is a value placed on the stack, adjacent to the return pointer, as a defense mechanism against buffer overflow attacks. An attacker attempting to overflow a buffer and write to a return pointer will overwrite the canary, which causes the program to halt. Due to their low-cost and effectiveness at preventing buffer overflow attacks, canaries have long been widely deployed as effective, light-weight stack overflow defense mechanisms [10].

Even though they are randomly generated, stack canaries of a child process belonging to a parent process will always have the same stack canary. Thus, if a child process's canary is leaked, it is possible to perform a buffer overflow attack on any child belonging to the same parent, assuming that the code suffers from the memory corruption vulnerability. For example, OpenSSH handles encryption through child processes spawned by a single daemon. Leaking the canary of any one of these child processes allows for circumventing this defense on any other child to leak secret keys.

```
uint16_t array1, array2;
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

Listing 4: Double gadget

Example Victim. The victim for this example attack lives within a thread spawned by the attacker, and the victim consists of a double gadget like the one shown in Listing 4, where each array is of type `uint16_t` (Line 1). The arrays live in memory shared by the victim and attacker, but attacks without this requirement are possible by using a Prime+Probe side channel [35]. The code is compiled such that stacks include secret canaries and cease execution if a canary is modified. Having the victim reside in an attacker-spawned thread allows for user space stack massaging, but extends to any process that can be forcibly spawned, such as OpenSSH [27].

Stealing Canaries. Due to their location at the end of the victim stack, just past the end of target arrays, stack canaries act as a prime target for the double gadget attack. Reading the canary requires flipping lower-order bits of the array offset, such that the corresponding array access points just past the end of the array to the stack canary.

A stack canary is typically 32 to 64-bits long and stored at the address just below the return pointer. Spectre v1 attacks steal a single "word" of data per malicious offset value, where a word corresponds to the innermost array's data type. In our victim, `array1` is a `uint16_t` array. Each malicious value of `x` points to and steals an 16 bit value, meaning the gadget must be used four times, each with a different malicious value to steal a 64-bit stack canary.

Target Flip. The Rowhammer bit-flip needs to push the offset past the end of the victim array and point to the stack canary. Since the stack canary is separated into multiple words, we may either find a victim row with multiple bit-flips, or allow the victim to naturally cycle through values and hammer with the necessary timing to push the offset to different words of the canary. We use the latter approach, since we observe few rows that contain multiple flips on our machine.

Memory Templating and Massaging. We perform memory templating as described in Section IV to find useful bit-flips. The victim offset resides at a particular page offset within the stack, meaning the required flip must occur at the same offset. Memory was templated for approximately 2.5 hours to find this specific flip. The page containing this flip is unmapped and the victim thread is spawned, forcing the offset variable within the victim thread to use the flip-vulnerable page.

Triggering Spectre. The victim is left to run with legal values used for its offset, which trains the branch predictor. We wait for the victim to set the offset to the appropriate value corresponding to the given target word of the canary. For this example, the victim and attack code run synchronously, but FLUSH+RELOAD can be used to accurately monitor the execution of victim code to provide attacker synchronization [52]. We then evict the offset from the cache, forcing the gadget to use the flipped value in a state of misspeculation. One word of the canary is accessed and used as an offset to load data into the cache, allowing us to use FLUSH+RELOAD to retrieve the target. The victim value is left to change, and the hammering is repeated to retrieve the rest of the canary.

Leakage Rate. As mentioned before, the array accesses 16 bits at a time, meaning 16 bits are leaked per flip and instance of FLUSH+RELOAD. We observed a leakage rate of approximately 8b/s, meaning the entire canary is leaked in about 8 seconds with 100% accuracy.

B. Triple Gadget - Arbitrary Kernel Reads

This second example demonstrates how the triple gadget within a kernel syscall can be used to achieve arbitrary reads of kernel memory. This is particularly dangerous since kernel memory is shared across all processes, meaning an attacker with access to kernel memory can observe values handled by the kernel for any process running on the same machine.

```
  if(x < array1_size){
    attacker_offset = array0[x]
    victim_data = array1[attacker_offset]
    y = array2[victim_data*512];
}
```

Listing 5: Triple Gadget

Example Victim. The example victim for this attack is a syscall in which we inserted a triple gadget, as shown in Listing 5. Since syscalls execute with kernel privilege, any data within the kernel can be leaked. For this example, we target a 10-character string within the syscall's code that is out of bounds from the target arrays. Additionally, the attacker and victim share the arrays used in the triple gadget.

Memory Templating. As done in the double gadget attack, we begin by finding a useful bit-flip. The purpose of the flip here is to force the victim array (in the kernel) to point to the attacker-controlled data. Thus, a specific high order bit-flip is needed to point from the victim to region of data we control. To reduce the time required to find the bit-flip, we configure the victim such that it can use an array offset at any position in the stack, by including victim variables at every offset position. Therefore, there is no need to find a flip at a specific offset; we only need to change a specific bit at any aligned 64-bit word within the page.

Attacker Controlled Data. One method of controlling data in the victim's address space would be to simply allocate a large memory chunk on the user space heap and fill this chunk with the desired value. The bit-flip would then cause the victim to point from kernel memory to our data in user memory. However, this requires breaking Kernel Address Space Layout Randomization (KASLR) in order to precisely know the difference between the target kernel address and the controlled user space address. Furthermore, Supervisor Mode Access Prevention (SMAP) blocks the kernel form reading user memory, and is enabled by default on the last several generations of Intel processors [2]. Therefore, we instead inject our data into the kernel at sets of addresses that differ from the target-flip-address by a single bit.

SMAP Bypass. We borrow from kernel heap-spray attacks [12], [21], which demonstrate methods of filling the kernel heap with attacker controlled data. These techniques take advantage of syscalls such as `sendmsg` or `msgsnd`, which allocate kernel heap memory using `kmalloc` and then move user data into these kernel addresses. To prevent these syscalls from freeing the data before returning, attackers use the `userfaultfd` syscall to stall the kernel. This syscall allows users to define their own thread that will handle any page faults on specified pages. When the attackers call a data-inserting syscall (such as `sendmsg`) they pass arguments with `N` pages worth of data, but only allocate `N - 1` physical pages. When `sendmsg` attempts to copy the data from user to kernel space, it will encounter a page fault on the final page. The thread fault handler, assigned by `userfaultfd`, is configured to spin in an endless loop, leaving `sendmsg` stuck, after having copied `N - 1` pages of user data into kernel memory.

Stack Data Insertion. While the above method is useful for inserting attacker-controlled data into the kernel's *heap*, heap-insertion is not useful for SpecHammer since kernel heap addresses will never have only one bit of difference from kernel stack addresses. However, numerous syscalls, including `sendmsg`, take a user defined *message header* which is placed on the kernel stack. To ensure that this inserted value will land on an address that is one bit-flip away from the flip-target, we spawn many threads that all use `sendmsg` to insert kernel stack data, giving high probability (87%) of an address match.

Controlling Page Offsets. The only remaining issue is the offset within the page. Stack offsets for kernel syscalls are always fixed and we need to insert data into an address with a page offset that matches that of our flip-target. Fortunately for the attacker, there are numerous syscalls (e.g. `sendmsg`, `recvmsg`, `setxattr`, `getxattr`, `msgsnd`) that allow for writing up to 256 bytes of the kernel stack, giving a range of offset options. Additionally, these syscalls are called from other syscalls as well, (e.g.`socket`, `send`, `sendto`, `recv`, `sendmmsg`, `recvmmsg`) and each of these use a varying amount of stack space before calling the previously listed syscalls, essentially allowing the attacker to "slide" the position of the inserted data up and down the stack.

As an example, we find that the target-variable of the example gadget presented in Section VII-B has a page offset of `0xd20` (when it is called during the spawning of a new thread) and `sendmmsg` can be used to control data on the kernel stack from `0xcf0` to `0xd70`. Thus, the triple gadget attack can work by pointing from a victim kernel address to an attacker controlled kernel address, allowing the attack to work in the presence of SMAP. Since KASLR only randomizes the kernel's base address, the difference between these addresses remains constant, thereby neutralizing KASLR.

Kernel Stack Massaging. Next, we run the kernel stack massaging technique from Section V-B, forcing the syscall to use the flip-vulnerable page for its array offset. We allocate numerous threads as part of the stack spray, and there is a possibility none of the kernel stacks contain the flip-vulnerable page. Therefore, we check each thread for the target page, and if the page is not found, we repeat the templating and massaging steps until a target page lands within a kernel stack.

Triggering Spectre. Finally, the thread containing the target page makes the syscall containing the victim gadget, which runs repeatedly with a loop of of legal offset values in order to train the branch predictor. The offset value is occasionally hammered and evicted from the cache, causing the inner most array to point to user data in a state of misspeculation. The FLUSH+RELOAD side channel is used to confirm the target secret (in this case, the value of the victim's string) has been correctly leaked. We then modify the attacker-controlled data to point to any secret value within the attacker's address space, and the hammering is repeated to leak the next target value.

Offline Phase Performance When running on the Haswell machine, in which SMAP is disabled and `pagetypeinfo` is unrestricted, the time taken to find pages with useful flips and land a such a page in the kernel is 34 minutes. While our new `buddyinfo` and SMAP bypass techniques present slightly reduced accuracies, they conversely *reduce* the time needed to find flips and land a useful page. The `buddyinfo` technique relaxes the requirements on draining user pages (to only draining order 4 or larger blocks, rather than draining all blocks), meaning each massaging attempt takes less time.

Furthermore, the SMAP technique allows a *range* of bits to be useful, since we need any flip that points from (victim) kernel stack to (our controlled) kernel stack. These two regions of memory are much closer together the case of a kernel stack victim and controlled user space region, meaning we can make a selection among many lower order bits (bits 5 through 28) rather than being forced to flip the only high-order bit that points from kernel space to user space (bit 45). Thus, while this technique introduces another probabilistic element (with 87% accuracy) the time needed to find a single useful flip to perform the attack is reduced. Consequently, the attack requires an average 9 minutes on average to find a useful flip and land it in the kernel across all machines.

Leakage Rate. `array1` is of type `uint8_t`, meaning each misspeculation leaks 8 bits of data. After performing the prerequisite templating and massaging steps, the leakage occurs at a rate of 16 to 24b/s on DDR3. We leaked the target string with 100% accuracy. When running on DDR4, multi-sided hammering is required, which requires more time per hammering round, consequently reducing the leakage rate to about 4 to 19b/min (6b/min on average), also with 100% accuracy on the three DDR4 machines listed in Table II.

VII. Gadgets in the Linux Kernel

To understand the impact the SpecHammer relaxation has on the number of exploitable gadgets in real-world code, we run a gadget search tool, Smatch [29], on the Linux kernel.

A. Gadget Search.

Smatch. Smatch was initially designed for finding bugs in the Linux kernel. However, after Spectre was discovered, a `check-spectre` function was added, which searches for gadgets. It searches for segments of code in which a nested array access occurs after a conditional statement, and the offset into the array is controlled by an unprivileged user. It additionally checks if the nested accesses occur within the maximum possible speculation window, and if the accesses use an array_index _nospec macro, which sanitizes array offsets by bounding them to a specified size.

Tool Modification. We modified the tool to remove the condition of an attacker controlled offset, and searched only for gadgets in which the attacker *does not* control the offset. In addition, we added a function to search for triple gadgets as well, which checks if the value of a nested array access is used as an offset for a third array access.

Results. When running the unmodified `check-spectre` function on the Linux kernel 5.6, we find about 100 double gadgets, and only 2 triple gadgets. Modifying the function to search for SpecHammer gadgets leads it to report about 20,000 double gadgets, and about 170 triple gadgets.

Bypassing Taint Tracking. Such a large number of potential gadgets exposes more holes for Spectre attacks on sensitive, real-world code. Furthermore, oo7 [51], which is the only defense that can efficiently mitigate all forms of Spectre [4], does not work against SpecHammer gadgets. This defense identifies nested array access that use an *untrusted* array offset value (i.e. a value coming from an unprivileged user). Any gadgets using such an offset are considered "tainted," and are prevented from performing out of bounds memory accesses. However, since the newly discovered gadgets use variables that cannot be directly modified by attackers, they are considered trustworthy, and would go unmitigated by oo7.

Additional Gadgets. Even after making the modification to smatch to include gadgets without attacker-controlled offsets, we observed that smatch was still unable to detect all potential SpecHammer gadgets, demonstrating that existing gadget detection tools are not sufficient for finding all exploitable code.

B. Kernel Gadget Exploit

To understand the nature of gadgets that remained undetected by smatch, we chose to explore the kernel source code by hand to identify potential gadgets that may be newly exploitable with the flexibility granted by Rowhammer. For example, in addition to manipulating array offsets, Rowhammer bit-flips allow for the indirect modification of pointers as well. Modifying a single `struct` pointer can lead to a chain of pointer dereferences ending with secret-dependent cache accesses. This points to a new type of gadget compared to those presented in Spectre [25], as it relies on pointer deferences rather than nested array accesses. One particular example of this lies in the kernel's `page_alloc.c` file.

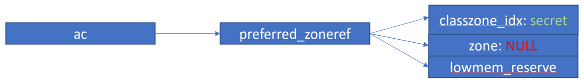


Fig. 5: **alloc_context struct pointer**

page_alloc.c This file contains the code used for all physical page allocation. The `get_page_from_freelist` function in particular contains the SpecHammer gadget; a simplified version with only the relevant code lines is presented in Listing 6. Note that the gadget does not contain cosecutive array accesses, but rather dereferences consecutive struct pointers, and uses the result for an array access. The `allocation_context` (`ac`) struct pointer, shown in Figure 5, is particularly important, as many variables used in the function are obtained from this pointer.

```
1    get_page_from_freelist{
2      struct alloc_context *ac;
3      struct zoneref *z = ac->preferred_zoneref;
4      struct zone *zone;
5
6      for(zone=z->zone;zone;z=find_next_zone(z,ac->
     zone_highidx);
7      zone=z->zone){
8        ....
9        preferred_zone = ac->preferred_zoneref;
10       idx = preferred_zone->classzone_idx;
11       ....
12       z->lowmem_reserve[idx];
13     }
14 }
```

Listing 6: Code Gadget for the double gadget attack

Forcing Misspeculation By manipulating the value of `ac` to point to a region of attacker-controlled code, it is possible to control all variables obtained from an `ac` dereference, and control the victim's execution flow. More specifically, an attacker run the function normally, teaching the predictor that the `for` loop at Listing 6, Line 6 will be entered. Then, `ac` can be modified by hammering such that the dereferences at Lines 3 (`z = ac->preferred_zoneref`) and 6 (`zone = z->zone`) set zone equal to `NULL`. This triggers a misspeculation, since the `for` loop should terminate immediately, but will actually begin its first iteration due to the prior training. Furthermore, `ac` has been set such that during this misspeculation, the chain of dereferences at Listing 6 Lines 9 and 10 causes `idx` to equal secret data, causing a secret-dependent access at Line 12 (`lowmem_reserve[idx]`), recoverable by cache side channel.

Results. To empirically verify this behavior, we instrumented page_alloc.c file to flip bits as needed, and found it is possible to manipulate the function's control flow and cause a misspeculation that leaks kernel data. We recovered an 8-bit character inserted in the kernel code that is normally out of range of the manipulated array, by inserting code that uses a FLUSH+RELOAD channel. This can be replaced with PRIME+PROBE to retrieve secrets without modifying `page_alloc`.

VIII. MITIGATIONS

Spectre. Developing a defense focused on the Spectre aspects is likely the more difficult option. While other variants of Spectre received effective and efficient mitigations [4], [30], [46], Spectre v1 was seen as more as an inherent security flaw caused by branch prediction with no simple solution.

Taint tracking, the only defense previously known to protect against all forms of Spectre v1 [4], [51], is thwarted by the new combined attack, as it relies on a Spectre limitation not present in the combined attack. Other defenses [5], [38], [39] designed to protect against Spectre v1 provide incomplete protection, working only in specific cases, and often come at a prohibitively high performance cost [4].

Rowhammer. For Rowhammer, on the other hand, numerous hardware and software defenses have been developed to prevent or detect bit-flips, beginning with PARA [23]. PARA randomly refreshes rows, giving more weight to rows with repeated accesses. However, this does not guarantee protecting rows that are about to flip, but only grants a high probability of refresh. For our triple-gadget attack that requires a single bit-flip, PARA does not guarantee protection.

A defense similar to PARA, target row refresh (TRR) *does guarantee* a refresh whenever two aggressor rows pass a certain activation threshold. However, TRResspass [14] has recently shown how bit-flips can be obtained despite TRR by performing scattered aggressor row accesses. Furthermore, by applying this technique, DDR4 was found to be even more susceptible than DDR3 to bit-flips [22].

Another common hardware defense against bit-flips is error correcting codes (ECC). Initially designed to catch bit-flips induced by natural errors, these functions are able to correct single flips, and detect up to two flips, within a given row. However, ECCploit [9] demonstrated a timing side-channel produced by single-flip corrections, that allows attackers to find rows containing multiple flips. By simulatenously flipping multiple bits, Rowhammer attacks can go undetected by ECC, making ECC an ineffective defense.

IX. CONCLUSION

We have demonstrated how Spectre and Rowhammer can be combined to circumvent the only defense believed to work against all forms of Spectre v1. Furthermore, we found the number of potential gadgets in the Linux kernel increases drastically with this new attack. Proof-of-concept gadgets show the attack's ability to leak data from user and kernel space victims. In future work we seek to understand the effect of relaxing gadget requirements on other sensitive code.

References

[1] 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,” *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[2] A. Baumann, “Hardware is the new software,” in *16th Workshop on Hot Topics in Operating Systems*. ACM, 2017, pp. 132–137.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, “Dedup est machina: Memory deduplication as an advanced exploitation vector,” in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 987–1004.

[4] C. Canella, J. Van 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 *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 249–266.

[5] C. Carruth. (2018) Rfc: Speculative load haredning (a spectre variant #1 mitigation.

[6] A. Chakraborty, S. Bhattacharya, S. Saha, and D. Mukhopadhyay, “Explframe: exploiting page frame cache for fault analysis of block ciphers,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1303–1306.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[8] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, “Are we susceptible to rowhammer? an end-to-end methodology for cloud providers,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 712–728.

[9] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, “Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[10] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, “Buffer overflows: Attacks and defenses for the vulnerability of the decade,” in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00*, vol. 2. IEEE, 2000, pp. 119–129.

[11] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, “SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript,” in *USENIX Sec*, Aug. 2021. [Online]. Available: Paper=https://download.vusec.net/papers/smash_sec21.pdfWeb=https://www.vusec.net/projects/smashCode=https://github.com/vusec/smash

[12] L. Dixon, “Using userfaultfd,” 2016. [Online]. Available: https://blog.lizzie.io/using-userfaultfd.html

[13] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, “Grand pwning unit: Accelerating microarchitectural attacks with the gpu,” in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 195–210.

[14] P. Frigo, E. Vannacc, 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 *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.

[15] M. Gorman, “Understanding the linux virtual memory manager,” *IEEE Transactions on Software Engineering*, 2004.

[16] D. Gruss, “Program for testing for the dram ”rowhammer” problem using eviction,” May 2017. [Online]. Available: https://github.com/IAIK/rowhammerjs

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O’Connell, W. Schoechl, and Y. Yarom, “Another flip in the wall of rowhammer defenses,” in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[18] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer. js: A remote software-induced fault attack in javascript,” in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2016, pp. 300–321.

[19] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, “Spectector: Principled detection of speculative information flows,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[20] J. Horn. (2018) speculative execution, variant 4: speculative store bypass. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[21] invictus, “Linux kernel heap spraying / uaf,” 2017. [Online]. Available: https://invictus-security.blog/2017/06/15/linux-kernel-heap-spraying-uaf/

[22] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, “Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 638–651.

[23] 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,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[24] V. Kiriansky and C. Waldspurger, “Speculative buffer overflows: Attacks and defenses,” *arXiv preprint arXiv:1807.03757*, 2018.

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[26] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[27] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, “Rambleed: Reading bits in memory without accessing them,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[28] C. Lameter and M. Kim, “Page migration,” 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/page_migration

[29] J. LCorbet, “Finding spectre vulnerabilities with smatch,” 2018. [Online]. Available: https://lwn.net/Articles/752408/

[30] M. Linton and P. Parseghian, “More details about mitigations for the cpu speculative execution issue,” 2018. [Online]. Available: https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, “Meltdown: Reading kernel memory from user space,” in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[32] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, “Nethammer: Inducing rowhammer faults through network requests,” in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 710–719.

[33] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[34] O. Mutlu and J. S. Kim, “Rowhammer: A retrospective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[35] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *Cryptographers’ track at the RSA conference*. Springer, 2006, pp. 1–20.

[36] C. Percival, “Cache missing for fun and profit,” 2005.

[37] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, “{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks,” in *25th {USENIX} security symposium ({USENIX} security 16)*, 2016, pp. 565–581.

[38] F. Pizlo, “What spectre and meltdown mean for webkit,” 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[39] T. C. Projects, “Site isolation,” 2018. [Online]. Available: https://www.chromium.org/Home/chromium-security/site-isolation

[40] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, “Flip feng shui: Hammering a needle in the software stack,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1–18.

[41] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “Zombieload: Cross-privilege-boundary data sampling,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[42] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, “Netspectre: Read arbitrary memory over network,” in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 279–299.

[43] M. Seaborn and T. Dullien, “Exploiting the dram rowhammer bug to gain kernel privileges,” *Black Hat*, vol. 15, p. 71, 2015.

[44] M. Seaborne, "Program for testing for the dram "rowhammer" problem," Aug 2015. [Online]. Available: https://github.com/google/rowhammer-test
[45] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 213–226.
[46] P. Turner, "Retpoline: a software contruct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886
[47] 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.
[48] VandySec, "rowhammer-armv8," Apr 2019. [Online]. Available: https://github.com/VandySec/rowhammer-armv8
[49] K. Viswanathan, "Disclosure of hardware prefetcher control on some intel processors," 2014. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html
[50] vusec, "trresspass," Mar 2020. [Online]. Available: https://github.com/vusec/trrespass
[51] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2020.
[52] Y. Yarom. (2016) Mastik: A micro-architectural side-channel toolkit. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf
[53] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise l3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014.
[54] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering rowhammer hardware faults on arm: A revisit," in *ASHES*, 2018.
[55] Z. Zhang, Y. Cheng, Y. Zhang, and N. Surya, "Ghostknight: Breaching data integrity via speculative execution," in *arXiv*, 2020.

APPENDIX

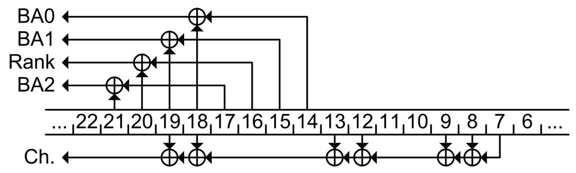


Fig. 6: **Physical to DRAM map for Ivy Bridge/Haswell (taken from [37]).**

A. Reverse Engineering Virtual to DRAM Address Mapping

The following section explains the techniques used to obtain the virtual to DRAM address mapping needed for double-sided Rowhammer. These techniques manipulate the Linux buddy allocator to first obtain a virtual to physical address mapping [27]. Then, they use a timing side-channel to determine which physical addresses correspond to rows in the same bank [37], reverse engineering the physical to DRAM address mapping. However, these techniques relied on the use of the `pagetypeinfo` file for memory manipulation, which has since been restricted to high privileged users. We therefore develop a new technique using the world-readable `buddyinfo` file.

Buddy Allocator. The buddy allocator is Linux's system for handling physical page allocation. It consists of lists of free pages organized by *order* and *migratetype*. The order is essentially the size of a free block of memory. Typically, requests for pages from user space (for example, via `mmap`) are served from order-0 pages. Even if the user requests many pages, she will likely be served with a non-contiguous block of fragmented pages. If there are no free blocks of the requested size, the smallest available free block is split into two halves, called *buddies*, and one buddy is used to serve the request, while the other is placed in the free list of the order one less than its original order. When pages return to the freelist, if their corresponding buddy is also in the freelist, the two pages are merged and moved to a higher-order freelist. Migratetypes essentially determine whether a page is meant to be used in user space (MOVABLE pages) or kernel space (UNMOVEABLE pages) [15].

pagetypeinfo & buddyinfo files. The `pagetypeinfo` file shows how many free blocks are available for each order and migratetype. While previous techniques [27], [47] used this file to track the state of free memory, `pagetypeinfo` has since been made unreadable for low-privilege users. However, a similar file, called `buddyinfo` shows how many *total* free blocks are available for each order, combining the number of kernel and user pages. Since `pagetypeinfo` has been restricted from attacker access, we present a new technique that uses `buddyinfo` for obtaining contiguous blocks of memory.

Obtaining Contiguous Memory Blocks. In order to control sets of contiguous DRAM rows, we must first obtain a large chunk of contiguous physical memory. For the eventual memory massaging step, described in Section V, the bit-flip needs to reside in a contiguous block of memory at least 16 pages long. Additionally, as we will see in the following paragraph, a 2MiB block will be helpful in obtaining physical addresses. However, if we request a 2MiB block via `mmap`, the allocator will service this request via fragmented, rather than contiguous, memory. Therefore, to obtain a 2MiB contiguous block, we first allocate enough memory to drain all smaller sized (1MiB or smaller) user blocks, forcing the allocator to supply us with a contiguous 2MiB block.

Using the buddyinfo file. However, with `buddyinfo` we can only see the combined total of user *and* kernel blocks remaining, but need to know when the number of 1MiB (and lower) *user* blocks is worth less than 2MiB of memory. To bypass this issue, we allocate blocks while monitoring the remaining total amount via `buddyinfo`. By placing our allocations at consecutive virtual addresses, we ensure our allocations will mostly use user blocks, since kernel blocks for new page table allocations will rarely be needed. Therefore we can continue to drain blocks and watch the *total* 1MiB block count decrease until it hits a *minimum value* and increases again. This behavior signifies there were no remaining user blocks to fulfill the request, requiring the 1MiB user block free list to be refilled. The observed *minimum value* is therefore the number of free 1MiB *kernel pages*, allowing us to subtract this value from the total value at any given moment to obtain the number of free 1MiB user pages.

We run the drain process again, subtracting the number of kernel pages, until the remaining 1MiB user pages equals 0. We can use the same process to drain the smaller blocks until they consist of less than 2MiB worth of memory. Finally, we request two 2MiB chunks of memory via `mmap`. Since the allocator does not have enough smaller order blocks to fulfill this request with fragmented pages, it is forced to supply a contiguous 2MiB chunk. Our approach is able to produce 2MiB pages with the same 100% accuracy of `pagetypeinfo` . Since the additional step of calculating the number of kernel blocks needs to be performed only once during the entire attack (*not* once per massaging attempt), using the *buddyinfo* technique incurs a negligible time cost.

Physical Addresses. To obtain the virtual to physical memory mapping, we use technique presented in [27]. Having already obtained a 2MiB block, we can learn the lowest 21 bits of a physical address by finding the block's offset from an aligned address. We obtain this offset by timing accesses of multiple addresses to learn the distances between addresses on the same bank. By identifying the distance for each page within the the block, we can retrieve the offset. With the mapping from virtual to physical to DRAM addresses, we can sort virtual addresses into aggressor and victim addresses corresponding to three consecutive DRAM rows.

DRAM Addresses. Next, we require the physical to DRAM address mapping. We can obtain it using Pessl's timing side-channel [37]. This technique takes advantage of DRAM banks' rowbuffer. Upon accessing memory, charges are pulled from the accessed row into a rowbuffer. Subsequent accesses read from this buffer, reducing access latency. All rows that are part of the same bank share a single rowbuffer. Therefore, consecutive accesses to different rows within the *same bank* will have increased latency, since each access needs to overwrite the rowbuffer. By accessing pairs of physical addresses and categorizing them into fast and slow accesses, an attacker can learn whether pairs lie in the same bank. Attackers can compare the bits of enough addresses that lie in the same bank to retrieve the mapping from physical addresses to DRAM.

Pessl et. al. [37] present the mapping function for numerous processors, such as the Haswell mapping (shown in Figure 6). Therefore, for attacks on Haswell, we can use this mapping as is. For newer processors, we run Pessl's attack (as provided in [50]) on several machines, and obtain the mapping for Kaby Lake, Coffee Lake, and Comet Lake processors.

Contiguous Blocks on DDR4. We previously explained the need for 2MiB blocks when hammering on a Haswell machine, since the physical to DRAM mapping uses the lower 21 bits. Newer processors use up to bit 24 for their mapping when a machine uses two channels with two DIMMs on each channel (4-DIMM configurations). Up to bit 22 is used for two-DIMM configurations and up to bit 21 for one-DIMM configurations [11]. These newer processors are designed designed to use DDR4. DDR4 Rowhammer techniques such as TRRespass [14], use hugepages to obtain 2MB blocks which are sufficient for one-DIMM configurations. For two-DIMM configurations, memory massaging techniques can be used to obtain 4MB contiguous blocks [11]. For 24-bit configurations, accuracy is reduced by the number of unknown bits, meaning 1/4 reduction of flips in the worst case of 24 bits.

Rowhammer.js	Without cache flushes	With cache flushes
hits	105,530,250	1
misses	377,915	107,347,967
%flips on misses	100%	100%
flips	12	2806

TABLE III: The effect of flushing victim addresses on Rowhammer.js

B. Modifications Made to Rowhammer Code

Rowhammer.js Modifications The code listings in this section show the changes we made to existing Rowhammer repositories to prevent the cache from masking bit-flips. Listing 7 shows the changes made to Rowhammer.js's native code. The first change on lines 530 and 531 fix a simple error regarding virtual and physical addresses. The original code passes virtual addresses into the `get_dram_mapping` function, while this function is designed to use physical addresses. The second modification occurs in lines 560 to 573. In these additional lines of code, we flush any victim rows immediately after they are initialized with test values. This ensures that when we later read these rows to check for flips, we will read directly from DRAM and not the cache.

TRRespass Modifications Listing 8 shows the modifications made to TRRespass. We found that cache flushes needed to be added to multiple regions of code to minimize the number of hits that occur when checking for flips. Data is first initialized in the `init_stripe` function starting at line 386. This function is called once during a TRRespass session to initialize the entire region of victim data. While many rows are naturally evicted from the cache due to initialization over a region too large to fit in the cache all at once, many initialized values do still remain in the cache in the original code. We therefore added flushes after every write to memory.

TRRespass then checks for flips using the `scan_stripe` function starting in line 571. When finding a flip (if `res` is non-zero), the flipped data is reinitialized to its initial value. However, there may still be some data within the same cache line that has not yet been checked. We therefore flush the cache to ensure checked data is pulled from DRAM and not the cache.

After completing a hammering session, TRRespass calls `fill_stripe` which refills victim rows with initial data. Similar to the `init_stripe` function, we must flush this initial data from the cache.

Finally, while the `hPatt_2_str` function (starting at line 134) does not directly interact with victim data, we found that its `memset` call does pull victim data into the cache. This is likely due to the processor's buddy cache system, which pulls adjacent cache-lines together, even when only one is accessed, to improve performance. We therefore flush this memset data as well.

C. Verifying the Effects of Caching.

In order to confirm that the reads were in fact reading cached data, we modified the existing code to measure the number of cache hits and misses that occur per victim address check. We

```
.....
if(OFFSET2 > =0)
second_row_page = pages_per_row[row_index+2].at(OFFSET2);
if (
  //*****fixed bug**********
  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,first_row_page)*0x1000))
  !=
  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,second_row_page)*0x1000))
  //***********************
 )
{

....

  #ifdef FIND_EXPLOITABLE_BITFLIPS
  for(size_t tries = 0; tries < 2; ++tries)
  #endif
  {
    //*****cache flush victim**********
    int32_t offset = 1;
    for (; offset < 2; offset += 1)
    for (const uint8_t* target_page8 :
    pages_per_row[row_index+offset])
    {
      const uint64_t* target_page = (const uint64_t*)
      target_page8;
        for (uint32_t index = 0; index < (512);
        ++index) {
            uint64_t* victim_va = (uint64_t*)
            &target_page[index];
            asmvolatile("clflush(%0)"::"r"(victim_va):%"memory");
          }
        }
    //*********************************
    hammer(first_page_range, second_page_range, number_of_reads);
....
}
```

Listing 7: Rowhammer.js Modifications

TRRespass	Without cache flushes	With cache flushes
hits	23,914,118	14,078
misses	2,081,626,490	2,105,526,350
%flips on misses	100%	100%
flips	431	4795

TABLE IV: The effect of flushing victim addresses on TRRespass

do so using by timing each access and marking fast accesses (less than 100 cycles) as cache hits and all slower accesses as cache misses. Since accesses pull entire *cache lines* into the cache, and each line is 64B, we only measure the first access per cache line, and all other accesses within the same set are labeled according to the timing of their first address. Additionally, we measured the number of hits and misses observed when extra cache flushes were added to ensure we read victim data from DRAM rather than the cache. Finally, we disabled the cache prefetcher [49], since otherwise, accessing a single set would pull additional sets into the cache and make subsequent accesses appear to be cache hits even if they had been flushed prior to hammering. For the DDR3 tests, we used a Haswell i7-4770 processor running Linux kernel 4.17.3, and Samsung DDR3 4GB DIMM. For DDR4 we used a Coffee Lake i7-8700K processor running Linux kernel 5.4.0 and Samsung DDR4 8GB DIMM. The experiments were run for 2 hours each. The data was initialized with a 0-1-0 stripe pattern for all experiments.

The results are shown in Table III (DDR3) and Table IV (DDR4). The DDR3 test was based on Rowhammer.js [16] and DDR4 on TRRespass [50] as they are the latest Rowhammer repositories for their respective type of DIMM. For both tests 100% of the flips were observed on cache miss accesses, supporting our observation that the cache masks bit-flips. With the DDR3 tests, neglecting to use victim cache flushes results in a large majority (99.64%) of the flip-checks reading cached data. A non-negligible 377,915 accesses *do occur* on cache misses, which is likely why the original code was able to observe any flips at all. However, once the cache flushes are added, nearly all the accesses directly read from DRAM, revealing a drastic number of flips that had been previously masked by cache, resulting in a 233x increase in flips.

As for the DDR4 results, the unmodified code already had a large number of misses. The reason is that a larger region of data is initialized all at once before being hammered, which results in much of the data being evicted from the cache due to the cache's limited size. However, the additional flushes were able to reduce the number of hits by 99.94%, drastically reducing the amount of bit flips masked by the cache.

```
.....
char *hPatt_2_str(HammerPatter * h_patt, int fields)
{
  static char patt_str[256];
  char *dAddr_str;

  memset(patt_str, 0x00, 256);
  //******new cache flushes******
  clflush(patt_str);
  clflush(patt_str + 64);
  clflush(patt_str + 128);
  clflush(patt_str + 192);
  clflush(patt_str + 256);
  //*****************************

.....

void fill_stripe(DRAMAddr d)addr, uint8_t val, ADDRMapper *
mapper)
{
  for (size_t col = 0; col < ROW_SIZE; col += (1 << 6)) {
    d_addr.col = col;
    DRAM_pte d_pte = get_dram_pte(mapper, &d_addr);
    memset(d_pte.v_addr, val, CL_SIZE);
    //******new cache flushes******
    clflush(d_pte.v_addr);
    clflush((d_pte.v_addr) + CL_SIZE);
    //*****************************
  }
}

.....

void init_stripe(HammerSuite * suite, uint8_t val){
.....
      for (size_t col = 0; col < ROW_SIZE; col += (1 <<
      6)) {
        d_tmp.col = col;
        DRAM_pte d_pte = get_dram_pte(mapper, &d_tmp);
        memset(d_pte.vaddr, val, CL_SIZE);
        //******new cache flushes******
        clflush(d_pte.v_addr);
        clflush((d_pte.vaddr) + 64);
        //*****************************
      }
    }
  }
}
.....
void scan_stripe(HammerSuite * suite, HammerPattern * h_patt,
size_t adj_rows, uint8_t val){
.....
  if(res){
    for (int off = 0; off < CL_SIZE; off++){
      if (!((res >> off) & 1))
        continue;
      d_tmp.col += off;

      flip.d_vict = d_tmp;
      flip.f_og = (uint8_t) t_val;
      flip.f_new = *(uint8_t *) (pte.v_addr + off);
      flip.h_patt = h_patt;
      export +flip(&flip);
      memset(pte.v_addr + off, t_vall, 1);
      //******new cache flushes******
      clflush(pte.v_addr + off);
      //*****************************
    }
    memset((char *)(pte.v_addr), t_val, CL_SIZE);
    //******new cache flushes******
    clflush(pte.v_addr);
    clflush((pte.v_addr) + CL_SIZE);
    //*****************************
  }
```

Listing 8: TRRespass Modifications

Dear Program Committee Members,

Enclosed is an updated version of our manuscript titled "SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks", which was previously submitted to IEEE S&P 2021 Winter as submission 243. The program committee's detailed comments were very helpful in improving the paper.

Our responses to the revision comments are found below, followed by our responses to each of the reviewers' requests.

Revision Comment

1) Be more explicit about the requirements of the proposed attack, in particular with respect to OS mitigations (SMAP, KASLR, pagetypeinfo) and newer hardware (i.e., DDR4).

[Our Response]: We have added Table II which details the CPU, memory, and OS version for each tested machine. Additionally, the table shows which mitigations were active during the experiments. Note that these are the default settings for each machine, as shipped by the OS vendor. In particular, SMAP is not supported on the older Haswell architecture, and is enabled on newer Kaby Lake, Coffee Lake and Comet Lake architectures by default. Likewise, pagetypeinfo is unrestricted on older kernel versions and is not available from user space on more modern kernel versions. Finally, KASLR is enabled on all machines used in this paper.

2) Demonstrate (with a working PoC) that the described attack can be used to defeat the aforementioned software and hardware countermeasures. If the presence of additional vulnerabilities and/or prerequisites is needed to defeat such countermeasures, please list them clearly.

[Our Response]: We developed a PoC to successfully leak data in the simultaneous presence of these mitigations:

- **Pagetypeinfo**: Instead of relying on the pagetypeinfo file, we use a similar, unrestricted file called buddyinfo to obtain contiguous blocks of memory (See Appendix. A). We then perform kernel memory massaging (Section V. B, under "**Step1: Draining Kernel Pages**").
- **SMAP:** To bypass SMAP, we no longer require the attacker to read user space data from the kernel. Instead, the unprivileged attacker uses a syscall to insert attacker controlled data onto the kernel stack and point our target variable to this data. See the Section VI.B, under "**SMAP Bypass**", "**Stack Data Insertion**", and **"Controlling Page Offsets."**
- **KASLR:** Since we can bypass SMAP, we no longer need to perform an access from the kernel stack to userspace. Thus, we performed our attack with KASLR enabled. The distance between the targeted array access and the injected data is unaffected since only the base address of the kernel is randomized (as discussed in Section VI.B, under "**Controlling Page Offsets**").

- **DDR4:** We use TRRespass to obtain bit-flips on DDR4, as discussed in Section IV.A.

In Section VI, we present experiments that successfully leak data on our PoC gadgets with all of these defenses enabled at the same time, using DDR4 hardware.

3) Measure how the aforementioned countermeasures affect the rate at which the proposed attack can exfiltrate bits.

[Our Response]: The techniques we develop to bypass the above countermeasures actually improves the performance of our attack, reducing the time required for the attack's offline phases of memory templating and memory massaging. (See the discussion in Section VI.B, under **"Offline Phase Performance"**).

- **Pagetypeinfo**: By using buddyinfo instead of pagetypeinfo, we no longer fully drain user pages, and perform the kernel and user page drain simultaneously (See Section V.B, under "**Step1: Draining Kernel Pages**"). While this results in massaging attempts having a slightly lower accuracy (from 66% to 60%), each attempt completes in about half of the time, resulting in an overall shorter process.
- **SMAP:** Previously, for kernel attacks, we required flipping a single bit that would point a victim from kernel space to user space. With our new SMAP bypass technique, we now only need to point to one of many kernel addresses whose contents we control. This allows us to use a range of bit-flips to perform the attack. While this technique introduces another probabilistic element to the offline phase (which works with 87% accuracy), it also reduces the time required to find a useful flip, since the attack can proceed immediately upon finding any flip within this range, as discussed in Section VI.B, under **"Offline Phase Performance"**.

Thus, with our new techniques developed to defeat these countermeasures, the offline phase for the triple gadget attack now completes with 9 minutes on average, where it previously required 34 minutes. (See Section VI.B under **"Kernel Stack Massaging"**)

The double gadget attack sees negligible difference for its offline phase, since the only difference is that buddyinfo is used to obtain 2MiB pages, rather than using pagetypeinfo, which incurs negligible additional latency. (See Appendix A, under **"Using the buddyinfo file"**).

For the online phase, the rate of leakage is reduced. Since DDR4 is equipped with Target Row Refresh (TRR) we must use multi-sided hammering, requiring more time to flip bits that trigger leakage. Because of the time needed for multi-sided hammering, the leakage rate drops from 24b/s at best to about 19b/min as best (See Section VI.B under "**Leakage Rate**"). As with DDR3, we leak data with 100% accuracy.

4) As mentioned in ReviewC, the claim about an oversight from the previous work on Rowhammer should be better clarified and evaluated. ReviewC suggests possible additional experiments to evaluate this claim.

[Our Response]: We clarify the changes we made to existing Rowhammer code in Appendix B., showing that the addition of cache flushes is enough to drastically increase the reported number of flips per DIMM. In Appendix C., we present additional experiments which show that with prior approaches, the vast majority of bit-flips are in fact masked by the cache. We measured the number of cache hits and misses that occur each time the attacker attempts to check a victim row for a flip. Table III and Table IV show that when using prior Rowhammer code, checking the victim row results in a cache *hit* the vast majority of the time, and yet 100% of the observed bit flips occur during a cache *miss*. This is strong evidence that previous code repositories fail to observe the great majority of bit flips due to the cache masking flips in the victim row.

Furthermore, we confirm that with our code modifications (which flush the victim row before hammering) the number of cache hits upon checking the victim row is drastically reduced. This consequently allows the attacker to check DRAM directly and observe flips that would have otherwise been masked by the cache. Therefore, by implementing the additional experiments from ReviewC, we find that the lack of cache flushes in previous work did indeed mask many bit-flips, and adding cache flushed drastically increases the number of observed flips.

We address specific reviewers' comments below.

Thank you again for the opportunity to improve our work through the process of a major revision!

Sincerely,
The Authors.

Review #243A
===
Weaknesses
----
* When taking into account all countermeasures that are disabled in this paper, it does require an impressive chain of exploits.

[Our Response]: We have implemented new techniques for bypassing each of these countermeasures. Please refer to our response to revision point # 2 above. We demonstrate that the attack can be performed with SMAP and KASLR enabled and pagetypeinfo restricted, which is the default configuration on modern systems.

Detailed comments for author

Let me start by saying that even though I was not a fan of combining attacks just for the sake of combining them, I actually liked the paper because precisely it is not for the sake of combining them, but it offers a real advantage for the attacker.

The thing I am most concerned about is that a lot of known and implemented countermeasures have been disabled to avoid difficulties (no SMAP, no KASLR, an old kernel version that does not restrict the pagetypeinfo, also having an older CPU for simpler DRAM functions). I think it is fair enough for the PoC, since it involves a lot of engineering that has mostly been solved by other papers. But I am afraid that it would misrepresent the risk posed by the attack if it is not sufficiently taken into account (actually it might be in both directions: "this paper did not have SMAP or KASLR, therefore this is irrelevant in a properly configured system", or "there are quick fixes for bypassing these countermeasures so it works the same way" when it might not). I appreciate that there are some insights on how to bypass them. I would appreciate more details on what bypassing each countermeasure involves in terms of time and probability of success. The least clear for me is the SMAP bypass, but each could benefit from a few sentences on their impact.

[Our Response]: Please see our response at the top to revision point #2. In particular, the new version now contains an attack working on newer DDR4 hardware, and newer kernel versions in their default configuration, with all countermeasures enabled (including SMAP, KASLR, and restricted pagetypeinfo). Additionally, regarding the time and probability of success of defeating mitigations, please see our response at the top to revision point #3. The end result of bypassing these mitigations is that the offline phase now requires 9 minutes to complete on average while it previously required 34 minutes. However, due to hammering DDR4 requiring more time than hammering DDR3, the leakage rate has reduced to 19b/min at best compared to the 24b/s reported in the previous version.

Mitigations. On mitigating Spectre, you only point out that current mitigations are thwarted by the novel gadgets exploitable with SpecHammer. However, you also find the gadgets by modifying the Smatch tool. Does it mean that when we find these gadgets we cannot patch them, or would it be possible to adapt current approaches to include the new gadgets?

[Our Response]: The new gadgets can be patched out by modifying the code to prevent exploitable behavior. However, as explained in Section VII, SpecHammer's lifts the requirement of having a user controlled input to the speculative execution gadget. Thus, we expect there are many new gadgets that have gone unreported by smatch. Moreover, Spechammer's lifting of the requirement of having a user controlled input means that more advanced taint-tracking based tools such as oo7 cannot be used to find SpecHammer gadgets. Thus, we require a new tool that can more accurately detect gadgets with the newly relaxed requirements. Developing a new tool, however, is non-trivial and we leave this for future work.

I find it hard to believe that mitigating Rowhammer is more straightforward than mitigating Spectre, and as you rightfully point out, most (if not all) defenses against Rowhammer, such as ECC and TRR, have been bypassed for now and still lead to bit flips.

[Our Response]: Indeed, both are difficult issues to solve. The main reason for stating Spectre is more difficult is that Spectre v1 attacks are inherent to the *intended feature* of CPUs performing computation before the branch's body is resolved and all safety/hazard checks are completed. Rowhammer is a side-effect of packing DRAM cells too close to each other. Furthermore, while numerous defenses have been developed (but eventually thwarted) for Rowhammer, oo7 is the only Spectre defense that had coverage over all gadget types and did not require hardware modifications. Thus, as a consequence to its more recent discovery, Spectre is the least understood attack between the two, with fewer feasible prevention approaches.

A few questions or remarks:
* Section IV.A: "we begin by obtaining a large chunk of contiguous physical memory via mmap" seems to contradict what was written in the previous page ("Even if the user requests many pages, she will likely be served with a non-contiguous block of fragmented pages.")

[Our Response]: This has been reworded to be clearer. We must first obtain a large block of physically contiguous memory, but mmap will only service our requests with fragmented pages. Therefore, we must perform a series of steps to force the allocator into servicing our request with a physically contiguous block. Note this section has been moved to Appendix A as it largely discusses techniques implemented in prior work.

* Section IV.A. In the example given in Figure 3 for Ivy Bridge and Haswell CPUs, the DRAM functions do not use bits higher than bit 21, which is useful with 2MB pages. Different CPUs and DRAM organization can however use bits higher than bit 21, which results in unknown physical address bits. What is the impact on your attack?

[Our Response]: We have modified the paragraph to address this question. Since the state-of-the-art for hammering on DDR4 and such newer CPUs relies on hugepage support, we do the same. When using the latest Linux kernel in its default settings, 2MB hugepages are enabled by default and provide contiguous memory up to 2MB. In this version, we demonstrate that this is enough to induce bit-flips on Kaby Lake (i7-7700), Coffee Lake Refresh (i9-9900K), and Comet Lake (i7-10700K) machines with a one-DIMM configuration. Additionally, it is possible to use memory massaging techniques to obtain up to 4MB pages, making it possible to obtain up to bit 22. Newer CPUs only go up to bit 22 for two-DIMM configurations, and up to bit 24 for four-DIMM configurations. While having two unknown bits this does result in a 75% reduction in flips, the attack still remains possible.

* Section V.A, on user space stack massaging. The technique works with 63% accuracy but can be done again if it failed. How do you know it failed?

[Our Response]: We modified this section to answer this question. To check if the massaging failed, we attempt the attack and check for data on the cache side-channel. If the massaging fails, there will be no flip and thus no leaked data on the cache side-channel. We use this as an indication to re-attempt the attack.

* I would appreciate more details in terms of differences between this work and other memory massaging techniques since this is an important contribution of this paper. For example, you write that "they either used probabilistic methods [36]", but the result of your massaging techniques also do not work with 100% success.

[Our Response]: We have added to the third paragraph of Section V. to clarify this point. It is true that our method is also somewhat probabilistic. However, the older methods required a PTE to use a flip-vulnerable page while we require a kernel stack to use a flip-vulnerable page. This means prior work could fill most of physical memory (3GB out of 4GB) with PTEs, giving acceptable odds that a PTE will use the target page. Spawning kernel stacks, however, requires spawning new threads, which requires much more resources than allocating new PTEs. Therefore our kernel thread spray is quite limited (about 16MB out of 4GB on our system), meaning if we simply unmapped a page and made many allocations as done in prior work, the odds of successfully forcing the victim to use our target page are extremely low. Thus, we must first exhaust the correct free lists to place the memory allocator in a state that maximizes the odds of a kernel stack using a target page even with our limited spray.

Minor comments
* p2: "attack-controlled" -> attacker-controlled
* p3: "if a victim access" -> accesses
* p3: "must have accesses" -> accessed
* p3: "attackers can accurately discern when addresses a victim interacts with" -> problem with the sentence
* Table I should be clear that the rowhammerjs code that was tested here is the native one, for people unfamiliar with the repository (else it looks like you had bit flips in JavaScript, which is not correct and not the point here).
* p9: "MOVAVABLE" -> MOVABLE
* p11: "the the"
* p12: "of of"
* p12: "an nested" -> a nested
* p12: Figure 7 is positionned on the text, probably due to an unfortunate negative vspace
* Throughout the text it is written "mis-speculation", "misspeculation", and "missspeculation". The last one looks like a typo, but consider having a single writing for this word.
* Ref [13] has been accepted to S&P '20
* Ref [25] has been accepted to the SLIM workshop @ EuroS&P '20
* Ref [35] has been accepted to ESORICS '19
* Ref [31] has a typo in its title "Exploiing" -> "Exploiting"
* Not sure what reference [38] is, it looks like a bad duplicate of reference [12] with the wrong year for the conference, the wrong authors, and additional typos in the author list ("Shwarz" is

spelled Schwarz, "L. Mortiz" should be "M. Lipp", "D. Mohimi" should be "D. Moghimi" although again he is not an author of this paper)
* There are many issues in the author list of [12] as well, please use:
https://dblp.org/rec/conf/sp/GrussLSGJOSY18.html?view=bibtex

[Our Response]: We have corrected these typos and errors.

Requested changes

I would appreciate more details on what bypassing each countermeasure involves in terms of time and probability of success.

[Our Response]: Please refer to revision point #3 above. In short, in this version we show an attack with SMAP, KASLR enabled and with the pagetypeinfo file being restricted. While the success probability per attempt does decrease a bit, our techniques also shorten the time for a single attack attempt, resulting in an overall reduction of the attack's expected running time.

Review #243B
===

Weaknesses

- Unclear how much the combined attack is feasible in a different variety of environments and hardware configurations.

[Our Response]: In Section VI we demonstrate our attack on four different machines, with different CPUs (Haswell (i7-4770), Kaby Lake (i7-7700), Coffee Lake Refresh (i9-9900K), and Comet Lake (i7-10700K)), Linux kernel versions (4.17.3, 5.4.1, 5.4.0), and DIMMs (DDR3, DDR4), showing that the attack can leak data on various configurations. These configurations are presented in Table II.

- The authors make some simplifying assumptions (e.g., no KASLR).

Our Response: Please refer to our response to revision point # 2 at the top. In particular we remove our assumptions regarding SMAP, KASLR and pagetypeinfo files, leaving the system in its default configuration.

Detailed comments for author

I enjoyed reading this paper. The authors manage to combine two techniques to achieve a powerful primitive. This required designing new memory massaging approaches and exploring new ways to use the original Rowhammer and Spectre attacks.

I think the evaluation of this paper could be strengthened. In fact, the authors only try their attacks in specific hardware and software configurations, while clearly different configurations can affect both RowHammer and Spectre feasibility significantly.

[Our Response]: We have performed additional experiments on newer CPUs, DRAM DIMMs, and OS versions and present the results in Section VI. Table II details the configuration of each machine. The attack is still feasible on new machines, albeit with a reduced leakage rate (19b/min at best versus 24b/s at best) due to DDR4 hammering requiring more time than DDR3 hammering. We confirm the attack is able to leak data with 100% accuracy on newer machines.

The paper also makes some assumptions (both about the hardware and the software), which are not well listed nor well motivated. For instance, SMAP is disabled. What do the authors mean in "SMAP is disabled by default on our Haswell system"? As far as I know, SMAP is enabled by default in all modern Linux kernels. Also (from what I can read here: https://lwn.net/Articles/517475/), it should be available on Haswell. (if not available on the tested system, authors should consider using a more modern CPU).
In addition, in Section VI.B, the authors assume no KASLR, and in their hardware configuration, they use DDR3, while DDR4 implements additional protection against RowHammer (which, however, seems to be bypassable).

[Our Response]: Regarding SMAP on Haswell, we believe SMAP was first supported on Broadwell processors, and is thus not present on Haswell. We empirically confirmed that we could read user data from kernel space under the default settings on this CPU. Attempting to turn on SMAP manually, we wrote to bit 21 of the CR4 register, which should normally enable SMAP, but we were still able to read user data from kernel space. While the LWN article mentions SMAP was set to arrive on Haswell, it was likely delayed to Broadwell. Additionally, this paper
https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/baumann-hotos17.pdf
shows in Table 1 that SMAP support was added in 2014, while Haswell launched in 2013 and Broadwell in 2014
.
Next, as stated in our response to revision point # 2 above, we have implemented and tested a method to bypass SMAP (discussed in Section VI.B under "**SMAP Bypass**") on newer systems where SMAP is enabled. The technique essentially consists of inserting data into another region of the kernel, and accessing that data instead of userspace data. We tested this technique on newer processors (Kaby Lake, Coffee Lake Refresh, and Comet Lake) and were able to perform our attack with SMAP enabled.
Regarding KASLR, since our new technique points from kernel stack data to kernel stack data, we no longer need to derandomize KASLR. As for DDR4, the machines used for the new experiments all contained DDR4 DIMMs.

I totally understand that an attack paper does not have to include ways to defeat all possible countermeasures. However, I think the paper will benefit from having a Table detailing all the hardware and software assumptions in a single place, together with an explanatory "discussion" section.

[Our Response]: We have added Table II which details the configuration for each tested machine. Note that since we implemented techniques to defeat each mitigation, the configurations shown in the table are the default for each machine. This means we no longer need to disable any defenses to leak data with our attack and our attack is feasible when all countermeasures are simultaneously enabled.

Finally, I found it very surprising what the authors explain in terms of "Under-reported Flip-rate in Prior Work" (page 7). Have the authors contacted the original studies' authors (mentioned in Table I) about this issue? What was their answer? This topic seems quite orthogonal with respect to the main contribution of the paper. The authors should better clarify if and how significantly the discovery of this higher bit-rate is important for the success of their proposed combined attack.

[Our Response]: We have added to Section IV., under "**The Need for Useful Flips"** to better explain why the increased bit-rate is required to perform our attack. At a high level, our attack requires bit-flips at specific page offsets, so that when flipped, the victim variable points to the targeted secret data. This requires hammering many addresses until encountering such a flip and, thus the need for an increased flipping rate to reduce the attack's end-to-end time. With the original number of (underreported) flips, the attack would take an infeasibly long amount of time. (E.g., since we found a 525x increase in flips on some DIMMs, instead of requiring 34 minutes to complete the offline phase using our new Rowhammer technique, we could expect the attack to take up to 300 hours before completing).

Typos/Formatting:
- Figure 7 overlaps the text
- "missspeculation"

[Our Response]: We have corrected these issues.

Requested changes

- Evaluate what proposed in different hardware and software configuration.

[Our Response]: We have performed experiments on new machines, as discussed in Section VI, showing the configuration for each machine in Table II.

- Better explain what are the hardware and software "prerequisites" of the proposed combined attack, and how they affect its feasibility.

[Our Response]: Table II shows what defenses are enabled for each test machine. Note that each of these settings is the default for the respective machine. The new techniques we use to bypass each mitigation (discussed in Revision Comment #2) allow us to perform our attack on the machines' default configurations, removing any "prerequisites" present in our original submission.

Review #243C
===

Weaknesses

There is no real PoC of the proposed attack.

[Our Response]: Please refer to revision point # 2 at the top. In particular we developed a PoC to successfully leak data on DDR4 hardware in the simultaneous presence of SMAP, KASLR and an inaccessible pagetypeinfo file.

The claim on the oversight of previous Rowhammer work is not clear.

[Our Response]: We have added Appendix B. which details the modifications made to existing code to increase the number of flips. We also performed additional experiments to verify that the additional flips were truly an effect of caching . We discuss the techniques and results of these experiments in Appendix C. Also, please see our response to revision point # 2 at the top.

Detailed comments for author

This paper is well written. When I was reading it, I expected to see a real attack against the Linux kernel that's not possible before in the evaluation section. However, the only attack described in the paper requires instrumenting the code to flip the bits. Given all the previous work on Rowhammer and Spectre, I think a working PoC is required for a new attack paper in this category to be accepted by S&P.

[Our Response]: In our evaluation in Section VI, we demonstrate that our attack can be used to leak kernel data in the presence of a SpecHammer gadget in the kernel. In particular we are able to demonstrate attacks using both double and triple gadgets, on DDR4 hardware, while SMAP and KASLR enabled, and the pagetypeinfo file is inaccessible.

Next, tackling the task of gadget finding, the purpose of Section VII-A is to show the prevalence of SpecHammer gadgets. We wanted to understand and demonstrate whether there were gadgets that may have gone unreported by smatch. As we show, Spechammer increases the number of speculative execution gadgets by several orders of magnitude (from 100 to 20,000 for double gadgets).

The situation is compounded by the fact that gadget finding tools tend to miss more complex gadgets. In particular, the gadget discussed in Section VII is one we found by sifting through kernel code by hand, and was completely missed during an automatic search due to its complexity. Seeking to assess exploitability of such complex gadgets, we then instrumented the code to prove the feasibility of leaking data via this gadget. Having been able to find a gadget by hand, shows that there are likely many more gadgets not reported by smatch. However, having a systematic method to find Spectre (and SpecHammer) gadgets (without false negatives) is a non-trivial, open problem which we leave to future work.

The paper claims that previous work on Rowhammer attacks has an oversight that they observe "cached data". However, I found it hard to understand this claim. A hammering test usually has three phases:

(1) Init (which includes initializing the victim rows)
(2) Hammer
(3) Check for bit flips (which means read the victim rows)

If the victim rows remain in the cache throughout all these three phases, doesn't this mean that the victim rows will have no bits flipped?

[Our Response]: This is correct, and this is why neglecting to flush the victim row before checking it is a detrimental oversight. We do note that some of the flips will remain visible due to the victim being naturally evicted from the cache by other system activity, allowing for some hammering attempts to succeed. Expectedly however, flushing the victim from the cache (as opposed to relying on natural evictions) significantly improves the amount of flips found.

How could previous work observe any bit flips if they would check "cached data when checking for flips"?

[Our Response]: It is likely that in some cases the cached data is naturally evicted from the cache in the process of initializing other addresses. The cache can only store a limited amount of data at a given time, meaning it is likely that while initializing a large amount of addresses, some of the cached initial data is evicted from the cache to make room for new cached data. In the new experiments (added in Section IV under **Verifying the Effects of Caching**) we show that the previous work did in fact incur some cache misses, and that all of the observed flips occurred on such cache misses. However, the number of misses drastically increases when we add in explicit victim address flushes. This correlates to the increase in flips, supporting the idea that previously, flips were masked by cached data.

The paper claims that the SpecHammer attack leads to many more gadgets in the Linux kernel than the regular Spectre attack. However, it would be much more convincing if the paper can exploit any of the new gadgets to launch a real attack.

[Our Response]: In Section VII we show that the gadget-finding tools are inadequate for finding SpecHammer gadgets, as we find a gadget unreported by smatch even when it was configured to use the relaxed requirements of SpecHammer gadgets. We demonstrate the feasibility of exploiting these unreported gadgets by exploiting the gadget presented in Section VII, albeit with an instrumented flip. This suggests the need for better (and more systematic) gadget finding approaches, a task we leave for future works.

There are two typos:

1. page 8: "remove and"

2. page 10: "in the the PCP"

[Our Response]: We have corrected these issues.

More detailed comments about the oversight claim:

The paper does not have enough detail for me to understand the importance of their contribution on RowHammer. The authors modified the code of two previous projects and these modifications led to more bit flips in their tests. It is difficult to evaluate whether these modifications just fixed a bug in the code (that led to some victims being cached) or they introduced new access patterns that bypass RowHammer mitigations or alter the results in some other way. Ideally, to evaluate the importance of their finding, I would've liked to see the details of the code changes the authors made.

[Our Response]: Please see our response to revision point # 4 at the top. We have added an explanation of the changes made to existing code in Appendix B.

The authors could have made additional experiments to validate that the lower bit rates were due to cached victim addresses rather than to some other factor. For example, they could've designed experiments to identify the cached victim addresses and show the whenever victims are not present in the cache, the new and old codebases have similar bit flip rates. Writing high-quality RowHammer code is very tricky, and it's easy to be fooled. It's difficult to assess that the authors' cache flush claim is the one solely responsible to the difference in bit rates they measured.

[Our Response]: We understand why this may have been unclear in the original paper. We show the code changes in the Appendix, which shows that the only changes we made consist of adding cache flushes, supporting the idea that these flushes are solely responsible for the difference in bit rates.

Additionally, we demonstrate a new set of experiments based on these suggestions. The new experiments (added in Section IV under **Verifying the Effects of Caching**) measure whether each access of a victim address (to check for flips) is a cache miss or cache hit. We find that bit flips are only observed upon cache misses. Our additional cache flushes increase the amount of cache misses, causing more access to read directly from DRAM, preventing the cache from masking bit-flips. Please see our response to revision point # 4 at the top for more details.

One more point about cached victim rows. Previous code uses memset to initialize victim rows. Nowadays, memset increasingly is implemented using non-temporal store instructions. This is deliberate because it is assumed the memset is used for zero-ing pages, and it makes little sense to cache zero pages. It is possible that the authors of TRRespass knew about this optimization and deliberately didn't bother to flush victim rows because using memset is enough to effectively "flush" them.

[Our Response]: Indeed, TRRespass does use memset to initialize victim rows. However, we have verified on the machines used in our experiments that memset does, in fact, write data to the cache. In an isolated program, we set values using memset, and subsequently accessed them, timing each access. We then set values using memset, flushed them from the cache, and then accessed the values while timing each access. Without cache flushes, each access was on the order of 30 cycles. With flushes, each access was on the order of 200 cycles, confirming that memset values are in the cache.
Additionally, it can be seen in Section IV.B and the Appendix that adding cache flushes to TRRespass did provide an increased flip rate.

Review #243D
===

Weaknesses

The complete end-to-end attack still appears to need a lot of background characterization and does not compare favorably to the power of software (memory safety) attacks.

[Our Response]: We have added new techniques for relaxing the background characterization required for the attack. More specifically, we demonstrate how to bypass SMAP and KASLR in Section V. B., and use the unrestricted buddyinfo file rather than pagetypeinfo, as discussed in Appendix A. Furthermore, we demonstrate that the chain of exploits is feasible on newer DDR4 machines in Section VI, as shown in Table II. Please see our response to revision point # 2 at the top for more details.

Detailed comments for author

At a high level, using Rowhammer with Spectre to enhance Spectre seems obvious; after all Rowhammer is a "arbitrary" write primitive, and it should be obvious that writing to arbitrary locations can create new powerful capabilities. The power of spechammer comes from manipulating inputs to branches that are not directly (or easily) reachable by the inputs. What the paper really shines is in detailed steps necessary to accomplish this, and in doing so describes some novel attack tricks and improves the precision of attacks. Overall an interesting read.

The comments below are aimed at improving the paper.

Rowhammer enhancements

The authors show that Rowhammer can be enhanced further by flushing the victim row before hammering. Prior papers appear to have relied on the victim row to be removed from the cache due to the conflict miss during the rowhammering process. This oversight results in additional rowhammer flips.

As the authors mention in the paper, the TRR paper used a custom FPGA setup to stress test for Rowhammer, and I'd expect that setup to be more aggressive than the one presented in the paper, yet this paper reports 5.1x more flips. It might be worth digging a bit more into this and explain this data.

[Our Response]: The 5.1x increase refers to running the original TRRespass code on our machine compared to running our improved version also on our machine. When running on an FPGA, prior work was indeed able to find many more flips due to having direct control over DRAM accesses, which is not possible on commodity CPUs.

Second a large number of systems today use DDR4. Having more DDR4 would have added more weight to the results here.

[Our Response]: We have provided the results for experiments done on an additional 2 DDR4 DIMMs (to have an equal number of DDR3 and DDR4 results). We find similar improvements on all three DIMMs (approximately 8x, 7x, and 6x increases in flips). Also note that previously, we reported a 5.1x increase in DDR4. Thanks to the comments of Reviewer C, we more thoroughly understood where to place cache flushes in TRRespass and were able to achieve a higher improvement (about 8x) on the same DIMM.

Third, in the TRR paper one of the vendors with "UL" (unlimited) capability is not impacted at all by Rowhammer. What would be interesting if you take one of those chips/DIMMs and determine if/how the flushing bug impacts the error rate.

[Our Response]: The TRRespass authors did not list which vendor or DIMM model featured unaffected UL DIMMs. Nonetheless, we confirmed that all of our DDR4 DIMMs are UL, and were able to get flips on all of them.

Fourth, for the sake of reproducibility, you should present more details on full system configuration on which you observed these flips.

[Our Response]: We have added details on the system configurations to Section IV. B. under **"Comparison of Rowhammer Techniques."**

Five, compared to the other observations, this part on enhancing rowhammer does not seem central to SpecHammer and seems like an add-on. Anything to emphasize the centrality of this observation would make the paper more coherent instead of a potpourri of optimizations.

[Our Response]: We have added an explanation to Section IV. B. under **"The Need for Useful Flips"** which explains why the increased flip rate is important for our attack. Our attack requires specific bit-flips, and so to find specific flips within a reasonable amount of time, we need a high flip rate. The Rowhammer enhancement allows us to achieve a high flip rate on the DIMMs in our possession which we initially believed to have low flip rates due to the issues with existing code. With the previous low flip rates, finding useful flips for our attack would take an infeasibly long amount of time.

Six, Section IV on memory templating appears to be using previously published results, or speculation on how things can happen. It might be good to get rid of this section or move it to a background section.

[Our Response]: In the original submission, the memory templating section did use techniques from previous work. However, now that we can no longer rely on pagetypeinfo, we have implemented new techniques that take advantage of buddyinfo. We have thus modified this section to give a high-level explanation of the memory templating step (removing the details found in prior work) and explain the use of the buddyinfo file in Appendix A.

Stack massaging:

The core contribution of the paper appears to be stack massaging.

* The success rate for stack massaging is reported to be 63% and 66% for user and kernel stacks respectively. Yet, the paper dings 2015 Black Hat paper on PTE manipulation as being a probabilistic process. This seems unfair.

[Our Response]: We have removed the comment about the 2015 paper being probabilistic and better explain how our technique improves on this prior work in the third paragraph of Section V.

* The kernel stack messaging appears to build on Seaborn and relies on `pagetypeinfo` access. In the absence of `pagetypeinfo` the entire process has to rely on a timing side channel which would again make the process probabilistic and unreliable. Further it would significantly

increase the time for such massaging to be completed. It would be great for the authors to provide this number for the timing attack.

[Our Response]: While our initial idea was to use a timing attack, we found there is a file called buddyinfo which is completely unrestricted and provides very similar information to pagetypeinfo. We explain the use of this file in Appendix A, and find it has comparable accuracy (60% for kernel stack massaging).

* The experimental demos appear to be a fairly mature microarchitecture (>5 years), and on DDR3 memory. It might be better to provide the above results on newer microarchitectures and memories.

[Our Response]: We perform additional experiments on newer processors (Coffee Lake Refresh, Kaby Lake, and Comet Lake) all using DDR4 DIMMs, and present the results in Section VI.B. The configurations of the machines we used are listed in Table II.

Kernel Triple Gadget:

`attacks that work even through SMAP may also be possible. For example, the attacker could utilize a syscall that allows for controlling data within the kernel. Alternatively, she can scan the kernel for a string of bits with value matching the target address, and have the target array offset point to those bits`

I agree that the ideas described in the paper for carrying out the triple gadget attack in the presence of SMAP appear to be promising. However, the paper could really be more convincing if the ideas were actually demonstrated in the presence of SMAP.

[Our Response]: We developed a technique that uses a syscall to insert data into the kernel and allows us to access attacker-controlled data from a kernel gadget even in the presence of SMAP. We describe this technique in detail in Section V. B under **"SMAP Bypass."** Also, please see our response to revision point # 2 at the top.

Requested changes

Relax assumptions regarding SMAP and `pagetypeinto`, and report measurements on newer architectures and memories.

[Our Response]: As discussed in Revision Comment #2 and the in-line comments above, we have implemented techniques to bypass SMAP and pagetypeinfo. For SMAP, we insert attacker-controlled data onto the kernel stack (See Section V. B "SMAP Bypass"). For pagetypeinfo, we instead use the similar but unrestricted, buddyinfo (See Appendix A.).

The offline phase of templating and massaging runs faster (9 minutes on average instead of 34 minutes) but the online phase has a slower leakage rate (19b/min at best instead of 24b/s) (See Section VI.B).

SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks

Paper #~~243~~312, 13 pages + references

***Abstract*—The recent *Spectre attacks* have ~~recently~~ revealed how the performance gains from branch prediction come at the cost of weakened security. Spectre Variant 1 (v1) shows how an attacker-controlled variable passed to speculatively executed lines of code can leak secret information to an attacker. Numerous defenses have since been proposed to prevent Spectre attacks, each attempting to block all or some of the Spectre variants. In particular, defenses using taint-tracking are claimed to be the only way to protect against all forms of Spectre v1. However, we show that the defenses proposed thus far can be bypassed by combining Spectre with the well-known Rowhammer vulnerability. By using Rowhammer to modify victim values, we relax the requirement that the attacker needs to share a variable with the victim. Thus, defenses that rely on this requirement, such as taint-tracking, are no longer effective. Furthermore, without this crucial requirement, the number of gadgets that can potentially be used to launch a Spectre attack increases dramatically; those present in Linux kernel version 5.6 increases from about 100 to about 20,000 via Rowhammer bit-flips. Attackers can use these gadgets to steal sensitive information such as stack cookies or canaries, or use new *triple gadgets* to read any address in memory. We demonstrate two versions of the combined attack on example victims in both user and kernel spaces, showing the attack's ability to leak sensitive data.**

I. Introduction

Computer architecture development has long put emphasis on optimizing for performance in the common case, often at the cost of security. Speculative execution is one feature following this trend, as it provides significant performance gains at a detrimental security cost. This feature attempts to predict a program's execution flow before determining the correct path to take, saving time on a correct prediction, and simply ~~rolling~~ rolls back any code executed in the case of a misprediction. However, such predictions may mistakenly speculate that malicious code or values are safe, allowing for attackers to temporarily bypass safeguards and run malicious code within misspeculation windows.

The potential of such speculative and out-of-order exploits was first demonstrated by ~~the recent~~ Spectre [25] and Meltdown [31], which revealed a new class of vulnerabilities rooted in transient execution. These attacks have shaken the world of computer architecture and security, leading to a large body of work in transient execution attacks [4], [5], [24], [33], [42] and defenses [5], [38], [39], [46], [51].

~~While transient execution attacks focus on information leakageacross security domains~~Moving away from information leakage, Rowhammer [23] is a complimentary vulnerability that breaks the integrity of data and code stored in ~~the machine~~a machine's main memory. ~~At a high level~~More specifically, the tight packing of transistors in DRAM DIMMs allows attackers to induce bit-flips in inaccessible memory addresses, by rapidly accessing physically-adjacent memory rows. ~~Similar~~ Similarly to Spectre, Rowhammer has spawned numerous exploits ~~[17], [27], [32], [34], [37], [40], [43], [45], [47]~~ [3], [11], [13], [17], [18], [27], [32], [34], [37], [40], [43], [45], [47] , including the recent bypass of dedicated defenses, such as Targeted Row Refresh (TRR) [50] and Error Correcting Codes (ECC-RAM) [9].

While both Spectre and Rowhammer ~~are~~ have been extensively studied individually, much less is known, however, about the combination of both vulnerabilities. Indeed, only one prior work, GhostKnight [55], has considered the new exploit potential resulting from combining both techniques. At a high level, GhostKnight demonstrates that despite their transient nature, speculative memory accesses can cause bit-flips in addresses that Rowhammer could not reach alone, resulting in bit-flips at those memory locations. However, GhostKnight only shows how Spectre can be used to enhance Rowhammer, and neglects to consider the complimentary question of how Rowhammer may be used to enhance Spectre. Noting that most modern machines are vulnerable to both Spectre and Rowhammer, in this paper we ask the following questions~~.~~:

Can the Rowhammer vulnerability be used to strengthen Spectre attacks? In particular, can an attacker somehow leverage Rowhammer to alleviate Spectre's main limitation of having a gadget inside the victim's code with attacker controlled inputs? Finally, what implications do combined attacks have on existing Spectre mitigations?

A. Our Contributions

We demonstrate that Rowhammer and Spectre can, in fact, be combined to evade the proposed defenses ~~, increasing~~ and increase the number of exploitable gadgets in widely-used code. In what follows, we provide a high-level overview of this combined attack~~(called), our contributions, as well as the ,~~, called SpecHammer, and discuss our discovery of newly exploitable gadgets in the kernel.

Attack Methods. The core idea of SpecHammer is to trigger a Spectre v1 attack ~~via~~ by using Rowhammer bit-flips to insert malicious values into victim gadgets. We present two forms of SpecHammer: the first relaxes the restrictions on ordinary Spectre gadgets (which will henceforth be called *double gadgets*), and the second uses new *triple gadgets* to provide arbitrary reads with just a single bit-flip.

Double Gadget Exploit. Ordinarily, Spectre v1 allows an attacker to send any malicious value to a Spectre gadget and read memory arbitrarily within the victim's address space. The main weakness of Spectre v1 is that it requires a gadget within the victim's code that uses an attacker controlled offset variable, limiting Spectre v1's attack surface. The target for the first version of SpecHammer, however, is a portion of code that meets all the requirements of a Spectre gadget, but *does not provide the attacker any direct way to control the victim offset*. By using Rowhammer, it is possible to modify the offset and trigger a Spectre attack on such victims to leak sensitive data. This attack eliminates Spectre v1's main weakness, allowing for exploits on a wider range of code.

~~It is well-known that Rowhammer can only~~ Unfortunately, Rowhammer can be used to flip, at ~~most,~~ best, only a few bits for a given ~~target~~ word of memory, ~~thus~~ limiting control the attacker has over the victim offset. ~~However~~Nonetheless, we demonstrate how the attacker, even with ~~such~~ limited control, is still able to leak sensitive data. For example, it is feasible to flip bits in the offset such that it points to just past the bounds of an array. This allows for leaking secret stack data, such as stack ~~cookies and~~ canaries designed to protect against buffer-overflow attacks [10]. That is, we show how the double gadget exploit can be used to leak such secrets, ~~incapacitating~~ bypassing stack protection mechanisms.

Triple Gadget Exploit. While the first exploit poses a threat to a common defense against ~~the powerful~~ buffer-overflow ~~attack~~attacks, its scope is more limited than the original Spectre attack which leaked arbitrary memory in the victim's address space. The second type of SpecHammer attack, however, can be used to dump the data of any address in memory. This method relies on a *triple gadget*, which has similar behavior to the Spectre v1 gadget, except that it features a triple nested access. Using this, the attacker can modify an offset to point to attacker-controlled data. This data can be set to point to secret data, which leads to the use of secret data in a nested array access, just as is done in Spectre v1. The ~~attack-controlled~~ attacker-controlled data can be modified to point to any secret within the attacker's address space, including kernel memory ~~for gadgets~~ when exploiting a triple gadget residing in the kernel. Thus, a single bit-flip allows for arbitrary memory reads, as opposed to the double gadget which is more restricted in what addresses it can leak.

Challenges. Implementing these SpecHammer attacks presents several key challenges~~.~~ :

1) We must find addresses containing useful Rowhammer bit-flips that can force a victim to access secret data under misspeculation.
2) We need to massage memory ~~so as~~ to force victims to allocate their array offset variables at addresses that contain these useful flips. For targets residing in the kernel, this means massaging kernel stack memory.
3) We must demonstrate that flipping an array offset value in a Spectre v1 gadget can ~~, in fact,~~ leak data under misspeculation.
4) Finally, we need to find gadgets in sensitive real-world code ~~in order~~ to understand the impact of relaxing gadget requirements.

Challenge 1: Producing Sufficient Rowhammer Flips. ~~The attack relies on finding specific flips that allow for leaking~~ SpecHammer requires bit-flips at specific page offsets in order to leak secret data. ~~Code~~ To that aim, we used the code repositories attached to prior work [16], [44], [48], [50] ~~are publicly available for testing how susceptible a given DRAM DIMM is to Rowhammer bit-flips. The~~ in order to test the susceptibility of DRAM DIMMs to Rowhammer attacks. Unfortunately, the amount of flips produced by these repositories suggests it is ~~rare~~ hard to find a DIMM with enough bit-flips to practically ~~run the proposed attack.~~ execute SpecHammer ~~requires bit-flips at specific offsets, and the flips reported by the existing repositories occur at few addresses, making it unlikely for an attacker to find the required flips~~.

However, as we show in Section IV, we observe that all of these repositories make a key oversight regarding cached data: they first initialize victim rows, ~~cause~~ and then induce bit-flips *in DRAM* (not caches), ~~and neglect flushing~~ but neglect to flush the victim cache line before checking for flips. This leads them to observe *cached data* when checking for flips, leaving many flips in the DRAM arrays unobserved. By correcting these oversights, we are able to increase the number of bit flips by 248x in the worst case and 525x in the best case on DDR3, and 16x in the best case on DDR4, demonstrating bit-flips are much more common than previous work would suggest~~, not only allowing~~ . Not only does this allow us to run SpecHammer ~~but also making~~ , but it also makes Rowhammer attacks more practical than previously thought.

Challenge 2: Stack Massaging. For the SpecHammer attack, the target for Rowhammer bit-flips is a variable used as an index into an array. Such offsets are most often allocated as local variables, meaning they are located on the stack. Rowhammer attacks rely on massaging targets onto physical addresses that are vulnerable to bit-flips. However, to the best of our knowledge, only one prior work [40] has demonstrated hammering stack variables~~and relied~~ , relying on memory deduplication ~~[40], which is now disabled by default, to massage the~~ to massage stack data as needed. ~~The~~ With deduplication now disabled by default, SpecHammer ~~attack~~ thus requires a new way of massaging a victim stack into place. Furthermore, the most attractive targets for this attack are gadgets residing in the kernel, as they can be used to leak kernel data, and hence a *kernel stack massaging* primitive is highly desirable.

Yet, the prior examples of kernel massaging focused on PTEs, ~~and did so probabilistically~~ rather than the stack [43], or were performed on mobile devices, taking advantage of features exclusive to Android [47]. Thus, we develop new primitives for massaging both user and kernel stacks, in order to allow for stack hammering without the use of deduplication (Section V).

Challenge 3: Proof-of-Concept (PoC) Demonstration. As a proof of concept~~(PoC)~~, we demonstrate (in Section VI) the variations of the attack on example artificial victims in both ~~the~~ user and kernel spaces. We demonstrate the double gadget attack in user space and the triple gadget attack in kernel space due to each attack's applicability in its respective space. These PoC attacks act as the basis for eventual attacks on the gadgets already found in widely-used code, and show that the attack is capable of leaking data at a rate of up to 24 bits/s ~~.~~ on DDR3 and 19 bits/min on DDR4.

Challenge 4: Kernel Gadgets. In order to better understand the effects of relaxing gadget requirements, we ~~have searched for~~ found the number of gadgets present in the Linux kernel, with the original Spectre v1 restrictions compared to the amount of SpecHammer gadgets. As shown in Section VII, we find that with the original requirements, there are about 100 ordinary, double gadgets, and only 2 triple gadgets. Modifying the function to search for gadgets vulnerable to our SpecHammer attack leads it to report about 20,000 double gadgets, and about 170 triple gadgets. ~~Therefore,~~ Thus, we show the number of potential gadgets ~~within~~ in the kernel is ~~shown to be much larger~~ greater than previously understood.

Summary of Contributions. This paper makes the following contributions~~.~~ :

- Combining Rowhammer and Spectre to relax the crucial requirement of an attacker-controlled offset for Spectre gadgets, discovering more than 20,000 additional gadgets in the Linux kernel (Section III & Section VII).
- Development of new methods for precisely massaging a victim stack in user space, and for massaging kernel memory, allowing an attacker to exploit the numerous gadgets present in the Linux kernel (Section V).
- Correcting oversights made by prior Rowhammer techniques to improve ~~the rate of bit-flips~~ bit-flip rate by 525x in the best case (Section IV).
- Demonstrating how SpecHammer gadgets can be used to obtain stack canaries for buffer-overflow attacks and how triple gadgets can be used to provide arbitrary reads from any memory address on example user and kernel space victims, respectively (Section VI).

II. Background

We present the necessary background information on Spectre and Rowhammer needed to understand the new combined attack, SpecHammer. Since Spectre relies on previous cache side-channels, relevant cache attacks are explained as well. ~~Since one form of the new attack is used to bypass buffer-overflow defenses, we also discuss buffer overflow attacks.~~

A. Cache Side-Channel Attacks

The cache was initially designed to bridge the gap between processor speeds and memory latency, but inadvertently led to a powerful side-channel exploited for numerous attacks [25], [35], [36], [52], [53]. By timing memory accesses, an attacker can tell whether data is being pulled from the cache (a fast access) or DRAM (a slow access), and can therefore observe a victim's memory access patterns.

Most relevant to SpecHammer is the Flush+Reload technique [53]. The goal is to use the cache to observe a victim's access patterns on memory shared by the victim and attacker. For example, if a victim ~~access~~ accesses particular addresses dependent on a secret value (e.g., using bits of a secret key as an array index), understanding which addresses the victim accesses can leak valuable secret information.

The technique first prepares the cache by flushing any cache lines the victim may potentially access using the `clflush` instruction. Then the victim is allowed to run, and will only access particular addresses dependent on secret data, loading *only* the corresponding blocks into the cache. Next, the attacker accesses all blocks of memory the victim *may have* accessed, while timing each access. If the access is slow, it implies data needs to be moved from DRAM to the cache, meaning the victim did not access any addresses within the block. However, if the access is fast, data is being pulled from the cache, meaning the victim must have ~~accesses~~ accessed an address corresponding to the same cache line. Thus, by taking advantage of the drastic timing difference in latency between a cache hit versus a cache miss, attackers can accurately discern ~~when~~ which addresses a victim interacts with and~~consequently~~ , consequently, any secret data used to control which addresses were accessed.

B. Spectre

Speculative and Out-of-Order Execution. In order to improve performance, modern processors utilize out of order execution to avoid necessarily waiting for instructions to complete when subsequent instructions are ready to be run. In the case of linear execution flow, processors utilize *out of order* (OoO) execution, running instructions out of program order, and only committing instructions once all preceding instructions have been committed as well. When a program has branching execution paths that depend on the result of certain instructions, the processor uses *speculative execution*, predicting which path the branch will take. If the prediction is incorrect, ~~and~~ any code run in the speculation window is simply undone, causing negligible performance overhead compared to not speculating at all.

Transient Execution Attacks. Running instructions before prior instructions have committed, due to OoO or speculative execution, creates a period of transient execution. Such transient execution windows have long been considered benign, as any code that should not have run is rolled back, and only proper code is committed. However, through the Meltdown [31] and Spectre [25] attacks, researches have recently demonstrated how OoO and speculative execution, can be used by ~~attacker~~ attackers to force programs to run using malicious values, uninhibited by safe guards that only take effect *after* the transient execution is complete. By the time the code is rolled back, the malicious values have left architectural side effects (e.g. placed data in the cache) that can be used to leak data even through transient execution. SpecHammer focuses on Spectre and the domain of speculative execution.

```
if(x < array1_size){
  y = array1[x]
  z = array2[y * 4096];
}
```

Listing 1: Spectre v1 Gadget

Spectre Attacks. Spectre [25] presents multiple ways in which an attacker can exploit speculative execution. We focus on Spectre v1, which is illustrated with the following example. Assume the victim contains the lines of code shown in Listing 1 and `x` is an attacker-controlled variable. The attack requires first training the branch predictor to predict that the *if* statement will be entered. The attacker can then change `x` such that reading `array1[x]` accesses a secret value beyond the end of `array1`. Even though `x` may be out of bounds, the secret value will still be accessed thanks to speculative execution, as the branch predictor has been trained accordingly. While the data read from `array2` is never committed to `z`, speculative execution still causes `array2` to use the secret value `y` as an index and load data at ("secret" * 4096) + `array2 base address` into the cache.

The attacker then uses FLUSH+RELOAD [53] to check what cache line was pulled, to reveal the `array2` index, exposing the secret value. One key assumption this attack makes is that the attacker controls `x`, as she needs to change `x` to the malicious value used to access secret data via `array1`.

Prevalence of Gadgets. Since Spectre attacks rely on the presence of a *gadget* in the victim code, the prevalence of gadgets in sensitive code becomes a crucial question. Researchers have developed tools [19], [29], [51] to automate the process of finding gadgets within target code. For example, smatch [29], a kernel debugging tool, was extended with the capability to report Spectre v1 gadgets within the Linux kernel. On kernel version 5.6, smatch reports about 100 gadgets~~across all kernel code.~~.

Followup Attacks. Upon Spectre's discovery, numerous papers emerged detailing how alternate variants could be used for new attack vectors ~~[?], [4], [20], [24], [26], [33], [41], [42]~~ [4], [7], [20], [24], [26], [33], [41], [42]. These included performing speculative writes [24], running a Spectre attack over a network [42], and combining Spectre with other side-channels to ~~perform reads on~~ exploit "half gadgets" that ~~simply~~ require a single array access within ~~an~~ a conditional statement [41].

C. Rowhammer

The Rowhammer bug [23] presents a way of modifying values an attacker does not have direct access to. The exploit takes advantage of the fact that DRAM arrays use capacitors to store bits of data, where a fully-charged capacitor indicates a 1 and a discharged capacitor indicates a 0. As transistors became smaller, DRAM became more dense, packing the capacitors closer together. ~~Kim~~ [23] found that by rapidly accessing values in DRAM, causing them to be quickly discharged and restored to their original values, disturbance effects can increase the leakage rate of capacitors in neighboring rows. Thus, by rapidly accessing (or "hammering") an aggressor row, an attacker can discharge neighboring capacitors ~~(or charge empty neighboring capacitors)~~ flipping 1s to 0s (or 0s to 1s) in neighboring memory locations.

DRAM Organization & Double-Sided Rowhammer. A DRAM array ~~typically~~ consists of multiple channels, each of which corresponds to a set of ranks, ~~each holding~~ where each ranks holds numerous banks. Each bank ~~, in turn,~~ consists of an array of rows made ~~up of individual capacitors that contain~~ of capacitors containing the individual bits of data. While it is possible to cause flips by rapidly accessing single DRAM rows [17], it is much more efficient to use double-sided Rowhammer ~~: a technique that alternates between hammer~~ (i.e alternating between hammering two aggressor rows ~~that surround~~ surrounding a single victim row). By increasing the number of adjacent accesses, the capacitor's leakage rate increases, drastically improving the efficiency of inducing flips. ~~Performing double-sided~~ Double-sided Rowhammer requires hammering adjacent DRAM rows within the same bank. However, ~~programmers~~ attackers cannot directly see the DRAM addresses of values they interact with. Instead, they can only see the virtual addresses~~, which~~. These are mapped to physical address, ~~and then~~ which are mapped to DRAM addresses.

Exploits. As with Spectre, Rowhammer inspired numerous exploits taking advantage of the ability to modify inaccessible memory. This began with Seaborn and Dullien [43] demonstrating how a flip can be used both to perform a sandbox escape, as well overwrite page table entries. Many exploits followed ~~[1], [17], [27], [32], [34], [37], [40], [45], [47]~~ [1], [3], [17], [27], [32], [34], [37], [40], [45], [47], demonstrating how Rowhammer can be used for privilege escalation on mobile devices [47], flipping bits through a web browser using JavaScript [16], as well as remotely attacking a victim over a network [32], [45]. Gruss et al. [17] additionally showed how ~~the many proposed defenses against Rowhammer could~~ many Rowhammer defenses can be defeated.

GhostKnight. To the best of our knowledge, only one prior work, GhostKnight [55], has demonstrated how Spectre and Rowhammer can be combined for a more powerful attack. Since Spectre allows for accessing arbitrary memory within a given address space, GhostKnight made the observation that rapidly accessing a pair of aggressor addresses can cause flips in the speculative domain. This effectively increases Rowhammer's attack surface by allowing for bit-flips at addresses only reachable under speculative execution.

III. SPECHAMMER

Our combined SpecHammer attack shows how Rowhammer can be used ~~enhance Spectre v1~~to enhance Spectre, bypassing proposed defenses and relaxing the requirements for a Spectre v1 gadget. We present two versions~~,~~: a double gadget attack and triple gadget attack, each striking a different trade-off between the attack's capabilities and the assumptions made regarding the availability of gadgets in the victim's code.

A. Double Gadget Attack: Removing Attacker Control

As discussed in Section II, a key limitation of Spectre ~~is the requirement~~ v1 is that the attacker ~~controls~~ must control a variable used as ~~an index into~~ a victim array index. We relax this ~~restrictive requirement~~ restriction by using Rowhammer to modify the ~~array offset variable without its~~ index variable without direct access.

```
if(x < array1_size){
  victim_data = array1[x]
  z = array2[victim_data * 512];
}
```

Listing 2: Pseudocode double gadget

Attack Overview. At a high level, the goal of the double gadget exploit is to mount ~~a~~ Spectre v1 ~~attack even if we do~~ attacks even if the attacker does not have direct control over the array offset. We use Rowhammer to modify this offset value, causing an array to access secret data ~~, and leaking~~ and leak it via a cache side-channel.

Listing 2 presents a gadget exploited by the first version of our attack, which uses the same gadgets as Spectre v1. In addition to assuming the presence of such code gadgets in the victim's code, we also assume that the victim's address space contains some secret data~~, which the attacker wants to leak~~. Finally, unlike the Spectre v1 attack, we do not assume any adversarial control over the values of x. Rather than controlling x directly, the attacker instead exploits Rowhammer to trigger a bit-flip in the value of x, such that array1[x] accesses the secret data.

Step 1: Memory Templating. The first step in any Rowhammer-based attack is to template memory in order to find victim physical addresses that contain useful bit-flips, i.e., a flip that will cause x to point to the desired data. As described in Section II, templating essentially consists of hammering many physical addresses until finding a pair of aggressors that correspond to a victim row with a useful flip. After finding a physical address with a suitable flip, our memory massaging technique (see Section V) is used to ensure that the value of x resides in this physical address, making it susceptible to Rowhammer-induced bit-flips.

Step 2: Branch Predictor Training. After placing the victim's code in a Rowhammer-susceptible location, the attacker trains the victim's branch predictor by executing the victim code normally. As we are executing the victim's code with legal values of x, it is the case where x < array1_size, which results in the CPU's branch predictor being trained to predict that the if in the first line of Listing 2 is taken. See ~~??~~ Figure 1(left) for an illustration.

Step 3: Hammering and Misspeculation. Next, the attacker hammers x, leading to the state in Figure ~~??~~1(right), where a bit-flip (marked in red) increases the value of x such that it points to the secret data past the end of array1. It is also necessary for the attacker to evict the value of x from the cache beforehand, ~~thus~~ ensuring the next time it is read, the flipped value in DRAM is used, as opposed to the previously cached value. After evicting array1_size, the attacker triggers the victim's code. As array1_size is not cached, the CPU ~~falls back to~~ uses the branch predictor, and speculates forward assuming that the ~~branch~~ if in Line 1 of Listing 2 is taken. Next, due to the bit-flip affecting ~~the value of~~ x, the access to array1 uses a malicious offset, resulting in secret being used as ~~an index to~~ array2's index, thereby causing a secret-dependent memory block to be loaded into the cache. Finally, the CPU eventually detects and attempts to undo the results of the incorrect speculation, returning the victim to the correct execution according to program order. However, as discovered by Spectre [25], the state of the CPU's cache is not reverted, resulting in a secret-dependent element of array2 being cached. See ~~??~~Figure 1(right).

Step 4: Flush+Reload. ~~Finally~~To recover the leaked data from the speculative domain, the attacker uses a FLUSH+RELOAD side channel [53] in order to retrieve the secret. More specifically, the attacker accesses each value of array2 while timing the duration of each memory accesses. Since all values of array2 were previously flushed from the cache, the attacker's timed access should be slow if no accesses happened between the eviction and this stage of the attack. However, if a timed access is fast, that memory block must have been recently accessed. In this case, due to the access to array2[secret*512] during speculation, the attacker should observe a fast access when measuring the offset secret*512, thereby learning the value of secret.

B. Triple Gadget Attack: Enabling Arbitrary Memory Reads

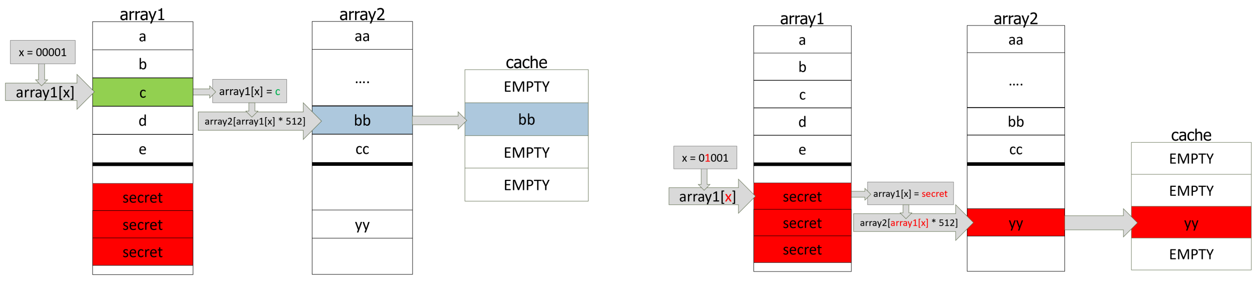
The attack presented in Section III-B assumes that the attacker can use Rowhammer to flip arbitrary bits in the victim's physical memory. In practice, however, Rowhammer-induced bit-flips are not sufficiently common to flip the number of bits required for leaking arbitrary addresses. An attacker can flip, at most, a few bits of the array offset, limiting the addresses she can reach. In order to provide for arbitrary reads even with the limited control provided by Rowhammer, we develop another variation that utilizes "triple gadgets". With just a single bit-flip, an attacker can use a triple gadget to point an array offset to attacker controlled data. This data can then be set to point to any value in memory, allowing an attacker to leak arbitrary data with a single flip, as detailed below.

```
if(x < array1_size){
  attacker_offset = array0[x]
  victim_data = array1[attacker_offset]
  y = array2[victim_data*512];
}
```

Listing 3: Pseudocode triple gadget

Attack Overview. For the triple gadget attack, we utilize a new type of code gadget; see Listing 3 for an example. At a high level, while the original Spectre v1 assumed that an attacker controlled variable x is used by the victim for a nested access into two arrays (e.g., array2[array1[x]]), here we assume that the victim performs a triple nested access using x, namely, array2[array1[array0[x]]].

By using such gadgets, the attacker can modify the innermost array offset (x) such that array0[x] points to attacker controlled data. This, in turn, allows ~~him~~ her to send arbitrary offsets to array2[array1[ ]], resulting in the ability to recover arbitrary information from the victim's address space. More specifically, our attacks proceeds as follows.



~~Training phase with legal value~~ ~~Attack phase with malicious value~~

Fig. 1: ~~Example attack scenario~~Example attack scenario. (left) Training phase with legal value. (right) Attack phase with malicious value.

Steps 1+2: Memory Profiling and Branch Predictor Training. As in Section III-A, the attacker starts by profiling the machine's physical memory, aiming to find physical addresses that contain useful bit-flips. The attacker then executes the victim's code normally, thus training the branch predictor to observe that the `if` in Line 1 of Listing 3 is typically taken.

Step 3: Hammering and Misspeculation. Next the attacker hammers `x`, leading to the state in Fig. 2, in which a bit-flip (marked in red) increases the value of `x` such that it points past the end of `array0`, into attacker controlled data. As in the case of Section III-A, ~~after evicting `array1_size`,~~ the attacker triggers the victim's code ~~thus forcing~~ after evicting `array1_size`, which causes the CPU to fall back onto the branch predictor, speculatively executing the branch in Line 1 of Listing 3 as if it was taken. The attacker controls the value in address `array0+x`, which results in an attacker-controlled value being loaded as the output of `array0[x]` in Line 2. Proceeding with incorrect speculation, the CPU executes `array1[array0[x]]` (Lines 2 and 3), resulting in the attacker controlling (through `array0[x]`) which address the victim loads from memory. The value of `array1[array0[x]]` is then leaked through the cache side channel, following the access to `array2` in Line 4.

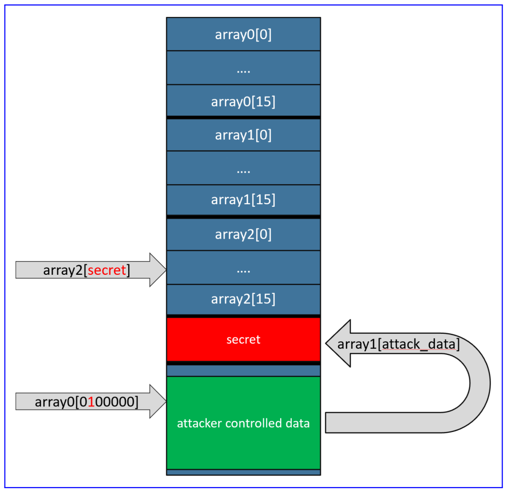


Fig. 2: **Triple gadget example**

Step 4: Flush+Reload. Finally, as in the case of Section III-A, the attacker uses a FLUSH+RELOAD side channel in order to leak the value accessed during speculation.

Comparison to Double Gadgets. While the triple gadgets ~~assume~~ require a triple-nested array access inside the victim's code, they also offer the advantage that multiple precise bit-flips are no longer needed ~~in order to read~~ for reading the victim's data. In particular, as only one bit-flip is used to point `array0[x]` into attacker-controlled data, multiple values can be read using the same bit-flip value. ~~In particular, by~~ By varying the value of `array0[x]` and launching the attack repeatedly, the attacker can dump the entire victim address space using a single carefully controlled bit-flip.

Kernel Attacks. This attack is particularly dangerous when performed on a gadget residing in the kernel~~. On a system without~~, as a single bit-flip can be used to read the entire kernel space. At first blush, it may seem that Supervisor Mode Access Prevention (SMAP)~~enabled, the attacker can simply allocate numerous pages containing the desired target value, and have the victim offset point to these allocated pages.~~ , which prevents *kernel-to-user* accesses, will prevent the attack by disallowing the kernel from accessing the user-controlled data on line 2 of Listing 3. However, in Section VI-B we show how to bypass this mitigation, demonstrating how an attacker can use syscalls to inject data into the kernel, and afterwards use a single bit-flip to point from the gadget to this controlled kernel data. Since SMAP does not block *kernel-to-kernel* reads, this technique allows for performing the triple gadget attack even with SMAP enabled.

~~While SMAP is disabled by default on our Haswell system, attacks that work even through SMAP may also be possible. For example, the attacker could utilize a syscall that allows for controlling data within the kernel. Alternatively, she can scan the kernel for a string of bits with value matching the target address, and have the target array offset point to those bits.~~

IV. Memory Templating

~~While the previous section provided general descriptions of the core portion of the attack, two key prerequisite steps were assumed to have already been taken:~~ The high level description provided in Section III assumes two key prerequisites. First, the *memory templating* step ~~to find a~~ is used to find useful flip-vulnerable address~~, and~~. Next, the *memory massaging* step is used to force the target victim variable to use this address. In ~~what follows~~this section, we describe the memory templating process~~. (Section V will detail the steps for stack massaging both in user and kernel space.)~~ , deferring stack massaging to Section V.

The goal of templating is to obtain ”useful” bit-flips, meaning they can be used to flip an array offset variable and trigger a SpecHammer attack. Vulnerability to bit-flips depends on the nature of an individual DIMM, requiring hammering many addresses to learn which ones ~~contains~~ contain useful flips. The techniques used for templating borrow largely from existing work, and we therefore keep the descriptions high-level, referring readers to the appropriate prior work [27], [37] and giving a more detailed description in Appendix A.

A. Obtaining DRAM row indices from virtual addresses

As explained in Section II, Rowhammer is drastically more effective when two aggressor rows that pinch a victim row are hammered in succession, a technique called double sided hammering [23]. Finding flips via double sided Rowhammer requires controlling three consecutive DRAM rows. However, as unprivileged attackers, we have no direct way of determining how our virtual pages map to DRAM rows, preventing us from performing double sided hammering. We ~~therefore take advantage of existing techniques [27], [37] to obtain this mapping . These techniques manipulate the Linux buddy allocator to first obtain a virtual to physical address mapping [27]. Then, they use a timing side-channel to determine which physical addressescorrespond to rows in the same bank [37], reverse engineering the physical to DRAM address mapping.~~

~~The buddy allocator is Linux's system for handling physical page allocation. It consists of lists of free pages organized by~~ must therefore reverse engineer this mapping before we can begin hammering. Since virtual address map to physical addresses, which in turn map to DRAM rows, we must obtain both the ~~*order*~~*virtual to physical* and ~~*migrate type*~~*physical to DRAM* ~~. The order is essentially the size of a free block of memory. The smallest size is order 0, which consists of a single page. An order x sized block consists of 2^x contiguous pages. Typically, requests for pages from user space (for example, via `mmap`) are served from order-0 pages. Even if the user requests many pages, she will likely be served with a non-contiguous block of fragmented pages. If there are no free blocks of the requested size, the smallest available free block is split into two halves, called *buddies*, and one buddy is used to serve the request, while the other is placed in the free list of the order one less than its original order. When pages return to the freelist, if their corresponding buddy is also in the freelist, the two pages are merged and moved to a higher-order freelist [15]. The `pagetypeinfo` file shows how many free blocks are available for each order.~~

mappings.

~~In order to control sets of contiguous DRAM rows, we begin by obtaining a large chunk of contiguous physical memory via `mmap`. For the eventual memory massaging step, described in Section V, the bit-flip needs to reside in a contiguous block of memory at least 16 pages long. Additionally, as we will see in the following paragraph, a 2MiB block will be helpful in obtaining physical addresses. To obtain a~~ For the latter, we use Pessl's DRAMA technique [37]. For the former, we only need the physical address bits used to determine the corresponding rank, bank, and channel. For a Haswell processor using DDR3, these are the lowest 21 bits. Thus, we can use the techniques presented in RAMBleed [27], to obtain a conitguous 2MiB ~~contiguous block, we first allocate enough memory to drain all smaller sized blocks, tracking the amount left with the~~ page, giving us the lower 21 physical address bits. Since this technique relies on the recently restricted `pagetypeinfo` file~~. Then, we allocate a large chunk of memory via `mmap`, guaranteeing that all pages belong to contiguous chunks of the needed size.~~

~~Note that while our machine's Linux kernel (version 4.17.3) allows for use of the `pagetypeinfo` file, more recent kernel versions restrict the use of this file to privileged users. However, since it is used exclusively for knowing when certain blocks of memory have been drained, attacks without this file are possible by using a timing side channel to determine when requested blocks were not available, forcing the allocator to take a longer execution path [47]~~, we use a new technique that relies on the world-readable `buddyinfo` file instead (see Appendix A) The time required for this step is unaffected by using the new `buddyinfo` technique.

~~**Physical to DRAM map for Ivy Bridge/Haswell (taken from [37]).**~~

~~To obtain the virtual to physical memory mapping, we use technique presented in [27]. Having already obtained a 2MiB block, we can learn the lowest~~ For newer architectures that use DDR4 memory, we follow the methodology of TRRespass [14], using transparent hugepages which are enabled by default in Linux kernel version 5.14, the latest version at the time of writing. Note that for a one-DIMM configuration, only up to bit 21 ~~bits of a physical address by finding the block's offset from an aligned address. We obtain this offset by timing accesses of multiple addresses to learn the distances between addresses on the same bank. By identifying the distance for each page within the the block, we can retrieve the offset. With the mapping from virtual to physical to DRAM addresses, we can sort virtual addresses into aggressor and victim addresses corresponding to three consecutive DRAM rows.~~ is needed, even on newer architectures. For two-DIMM configurations, it is possible to use memory massaging techniques to obtain up to 4MB of contiguous memory.

~~Next, we require the physical to DRAM address mapping. We can obtain it using Pessl's timing side-channel [37]. This technique takes advantage of DRAM banks' rowbuffer. Upon accessing memory, charges are pulled from the accessed row into a rowbuffer. Subsequent accesses read from this buffer, reducing access latency. All rows that are part of the same bank share a single rowbuffer. Therefore, consecutive accesses to different rows within the *same bank* will have increased latency, since each access needs to overwrite the rowbuffer. By accessing pairs of physical addresses and categorizing them into fast and slow access, an attacker can learn whether pairs lie in the same bank. Attackers can compare the bits of enough~~

~~addresses that lie in the same bank to retrieve the mapping from physical addresses to DRAM.~~

~~Since Pessl et. al. [37] obtained the mapping function for multiple memorycontrollers, including the one used for our experiments, we can simply use the mapping function given in Figure 6.~~

B. Hammering Memory~~.~~

With all the obtained memory sorted into rows, we initialize the aggressors and victims with values reflective of our desired flips. In our case, we seek to increase an array offset value to point to secret data, meaning we want to flip a particular victim bit from 0 to 1. We therefore initialize potential victim rows to contain all 0s. Since double sided hammering is most effective when the victim bit is pinched between two bits of the opposite value [23], [27], we set aggressor rows to all 1s, giving a 1-0-1, aggressor-victim-aggressor stripe configuration.

Inducing Flips. As done in prior work and existing Rowhammer templating code [16], [44], [50], [54], we repeatedly read and flush aggressor rows from the cache to ensure each read directly accesses DRAM and causes disturbance effects on neighboring rows. After doing a fixed number of reads, we read the victim row to check for any bit-flips, which in this case would mean a bit set to 1 anywhere in the victim row's value. We save addresses containing useful flips (i.e., a bit-flip that would cause an array offset to point to a secret), and move onto the memory massaging phase. Note that the above steps *neglect to flush the victim address cache lines*. Consequently, when we try to read the victim to check if we induced a flip, we will likely be reading cached initial data.

The Need for Useful Flips. Upon running existing Rowhammer code [16] on numerous DDR3 DIMMs, we experienced a somewhat low flip-rate of approximately 2 to 5 flips per hour. However, for our SpecHammer attack, we require specific bit-flips (a single bit position out of a 4KiB page), to point from an array to a secret, meaning it would take an infeasibly long amount of time to find the required bit in the average case. One option to overcome this would be to test many DIMMs until finding one particularly susceptible to Rowhammer, limiting the attack only to such susceptible DIMMs. However, we observed an oversight in existing Rowhammer repositories pertaining to the issue of cached victim data, which causes a susceptible DIMM to *appear* sturdy against flips, when, in fact, a vast majority of flips are simply being *masked* by cached data. By modifying these existing repositories, we found that the same DIMMs are vulnerable to thousands of flips per hour, allowing us to perform our attack on DIMMs that were previously thought to be safe.

Under-reported Flip-rate in Prior Work. Upon inspection of numerous public Rowhammer repositories [16], [44], [50], [54] ~~,~~ designed to test a ~~DIMMs vulnerability to rowhammer~~DIMM's vulnerability to Rowhammer, we observed that they all made the victim row cache oversight mentioned in the previous paragraph. By performing the above steps, reading a victim row to check for a bit-flip will likely result in reading the cached initialization data, leading to severe under-reporting of the actual number of flips obtainable on any tested DIMMS. Any flips that are reported are likely due to victim data being unintentionally evicted from the cache due to other memory accesses replacing those cache lines. In Appendix C we describe experiments we conducted to prove that cache effects are indeed responsible for masking bit flips.

Comparison of Rowhammer Techniques. In order to fully understand the effect this oversight had on finding bit-flips, we compared prior work with ~~enhanced versions that includes the~~ our victim cache flush ~~. The results~~ modification.

The results are presented in Table I~~compare the amount of bit-flips obtained using existing, public Rowhammer repositories with our code that includes victim cache flushes.~~ . We ran each program using double sided hammering over a two hour period with a 1-0-1 stripe configuration, then for 2 hours testing for using 0-1-0. The total flips over both runs are shown in the table.

Note that the repository for Rowhammer.js [16] contains an error that uses *virtual addresses* rather than *physical addresses* when determining which addresses reside on the same bank, and is thus split into 2 entries: one for the unmodified Rowhammer.js and the other for the same code with the error removed excluding the cache flush oversight. Finally, we used TRResspass [14], the latest Rowhammer templating repository, exclusively for DDR4, since it uses techniques designed to bypass DDR4 exclusive defenses. The changes we made to these repositories are detailed in Appendix B.

We perform our DDR3 experiments on a Haswell i7-4770 CPU with Ubuntu 18.04 and Linux kernel version 4.17.3. For the DDR4 experiments, we use a Coffee Lake i7-8700K CPU with Ubuntu 20.04 and Linux kernel version 5.8.0.

Results. For DDR3, ~~in the worst case~~ when compared to Rowhammer.js with the addressing error removed, our code improved the flip rate by 248x in the worst case, and by a factor of x525 in the best case~~a factor of 525.~~. As for TRRespass, we found that modifying the the code to include victim cache flushes resulted in ~~5.1x~~ 6x to 8x flips on DDR4 DIMMs. While prior ~~,~~ Rowhammer surveys have found larger numbers of flips [8], [22], they did so using techniques unavailable on general purpose machines. In the case of [22], the goal was to understand DIMMs vulnerability to Rowhammer at the *circuit* level, and thus DIMMs were tested via FPGAs to remove ~~and~~ higher-level sources of interference that may have reduced the number of flips. Similarly, [8] sought to achieve flips on servers, and their techniques can only work on multi-socket systems. In contrast, we use code that is designed for users to test their own machines for Rowhammer bugs, and show how ~~a simple modification~~ ensuring that the victim row is flushed before it is checked can drastically increase the number of flips.

In order to verify these additional flips were a result of cache flushing, we performed additional experiments to verify that data was in fact being pulled from memory and not the cache for each flip. These experiments are detailed in Appendix C.

height **Model**	Samsung (DDR3~~, 4GB~~)	Axiom (DDR3~~, 4GB~~)	Hynix (DDR3~~, 4GB~~)	Samsung (DDR4~~, 4GB~~)	Samsung (DDR4)	Samsung (DDR4)
rowhammer-test [44]	1	0	0	-	-	-
rowhammerjs [16]	4	9	2	-	-	-
~~rowhammerjs (using correct~~ rowhammerjs (corrected addresses)	15	38	32	-	-	-
rohammerjs with victim flushes	7,883	11,005	7,943	-	-	-
TRResspass [50]	-	-	-	~~1,694~~ 947	2,976	2,134
TRResspass with victim flushes	-	-	-	~~8,752~~ 7,916	17,958	15,611

TABLE I: Comparison between prior Rowhammer techniques and our new cache-flushing technique~~which includes flushing the victim row after initialization~~. Since the techniques listed in the top 4 rows are designed for DDR3, we did not run them on DDR4 DIMMs~~, signified by a "-"~~. Similarly, TRResspass is designed for DDR4 and was not run on DDR3~~DIMMs~~. Note that rowhammerjs refers to the code in its "native" directory.

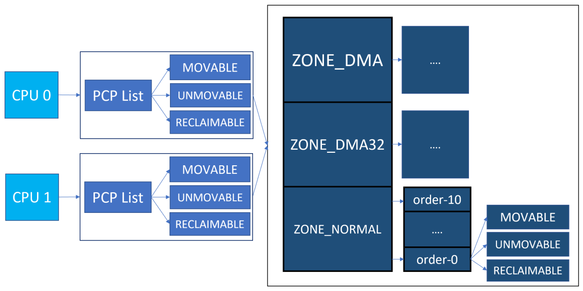


Fig. 3: **Linux memory organization**

V. Memory (Stack) Massaging

With possession of a useful, flip-vulnerable address, the next step is to force the victim variable into this address. The target victim is a variable used as an offset into an array. Such variables are most often allocated as local variables, and hence reside on the victim's stack. Therefore, in order to flip such variables and trigger the attack, we need to place the victim's stack on the flip-vulnerable page obtained from the templating step. Only one prior work has demonstrated stack massaging [40], and used (the now-disabled) page deduplication to do so~~, which has since been disabled by default~~.

Note that bit-flips correspond to particular DRAM addresses, which are fixed to specific *physical address*. Physical addresses, however, can be mapped to various different virtual addresses through a page table mapping. Thus, the goal is to force the victim to use a particular *physical page*.

Furthermore, if the victim resides in kernel code, the attacker needs ~~the ability~~ to massage *kernel stacks* which adds an additional layer of complexity compared to massaging user space stacks, since an unprivileged attacker cannot directly manipulate (allocate and deallocate) kernel pages. While prior work has demonstrated kernel massaging by forcing PTEs to use certain pages, they ~~either used probabilistic methods [43], which are~~ use methods too imprecise for ~~stack manipulation, or use~~ kernel stack massaging [43]. This existing technique simply unmaps the flip-vulnerable page and fills physical memory with PTEs until one uses the recently unmapped page. For kernel stack massaging, new threads need to be spawned to allocate kernel stacks. Since spawning new threads is resource-intensive (relative to PTE allocation), we cannot spray a majority of memory with stack threads and must manipulate memory into a state that that maximizes the odds of a *limited* spray using the target page. Other prior work has demonstrated more deterministic techniques, but uses methods exclusive to Android ~~devices~~ [47].

In this section we develop a novel technique for massaging *kernel* memory by taking advantage of ~~the Linuxbuddy allocator (explained in Section A)~~Linux's physical page allocator, the "buddy allocator" (see Appendix A), and its per-CPU (PCP) list system. Before describing our technique, we provide background on the memory structures we manipulated to achieve our result. An overview ~~of these systems~~ is shown in Figure 3.

Memory Zones. Within the buddy allocator, pages of memory are organized Within the buddy allocator, in addition to being sorted by order, free pages are also sorted by their *zone*. Zones represent ranges of physical addresses. Each zone has a particular *watermark level* of free pages~~, where if~~. If the zone's total free memory ever drops below the watermark level, requests are handled by the next most preferred zone. For example, a process may request pages from ZONE_NORMAL, but, if the number of free NORMAL pages is too low, the allocator will attempt to service the request from ZONE_DMA32 [15].

Page Order. Within each zone, pages are sorted into blocks by *size*, also called their *order*, where an order-x block contains 2^x contigous pages. The allocator always attempts to fulfill requests from the smallest order possible, but if no small order blocks are available, a larger block will be broken in half, and one half is used to fulfill the request [15].

Migrate-types. Pages are further organized by *migrate-type*. Migrate-types determine whether the virtual-to-physical address mapping can be changed while the page is in use. For example, if a process controls virtual pages that map to physical pages with the migrate-type `MOVABLE`, it is possible ~~for~~ to replace the physical page~~to be replaced with a different one, while keeping the virtual address the same by simply changing the mapping~~, by mapping the same virtual address to a different physical address [28].

PCP Lists. Finally, the PCP list (also referred to as the *Page Frame Cache*) [6] is essentially a cache to store recently freed order-0 pages. Each CPU corresponds to a set of first-in-last-out lists organized by zone and migrate-type. Whenever an order-0 request is made, the allocator will first attempt to pull a page from the appropriate PCP list. If the list is empty, pages are pulled from the ~~order 0~~ order-0 freelist of the buddy allocator. When pages are freed, they are always placed in the appropriate PCP list. Even if a contiguous higher-order block is freed, each individual page is placed on a PCP list, and they are merged only when they are returned from the PCP list to the buddy allocator freelist. Thus, the system serves to quickly fetch pages that were recently freed on the same CPU, rather than needing direct access to the buddy allocator.

A. *User Space Stack Massaging*

Building on existing user space massaging techniques [6], [27], the main goal is to free the flip-vulnerable page currently in the attacker's possession, and then force a victim allocation that will use the recently freed page. In the case of stack massaging, this means forcing a new stack allocation. The techniques presented here follow similar steps as those done in prior work [6], [27]. While prior works use this process to massage pages allocated via `mmap`, we massage victim stacks.

Stack allocation. User space stacks are allocated upon spawning a new process or thread, and use ZONE_NORMAL, migratetype MOVABLE memory. Additionally, even though they typically use more than one page, the request is handled as multiple order-0 requests, meaning pages are pulled from a PCP List. Pages obtained from `mmap` calls in user space also use NORMAL, MOVABLE memory, meaning stack pages and the controlled flip-vulnerable page are of the same type. Therefore, freeing the flip-vulnerable page via `unmap` will place the page in the same PCP list used for stack allocation.

Massaging Steps. Now understanding Linux stack allocation, stack massaging is performed using the following steps:

Step 1: Fodder Allocations. First, we make "fodder" allocations to account for any allocations made by the victim before allocating the stack. It is possible the target variable does not reside on the first page of the victim's stack. Therefore, we must first calculate how many pages will be used by the victim before the victim allocates the stack page containing the target, and allocate such number of fodder pages.

Step 2: Unmapping Pages. We then free the flip-vulnerable page, placing it in the PCP List, and then free the fodder pages, placing them in the same list above the flip-vulnerable page.

Step 3: Victim Allocation. Finally, we spawn the victim process, forcing it to perform the predicted allocations, and target stack allocation. Any allocations that occur prior to the target allocation will remove the fodder pages from the PCP List, forcing the stack to use the target page.

Results. This technique works with about 63% accuracy, which is acceptable since it only needs to be done once to mount the attack. If this step fails, we can attempt massaging again, and expect it to succeed within two tries. We can check for a massage failure by running the subsequent steps of the attack (i.e. calling the victim containing the gadget and hammering our aggressors) and checking for data on the cache side-channel. If no data is observed, we re-attempt massaging.

B. *Kernel-Space Stack Massaging*

Targeting gadgets in the kernel similarly requires forcing stack variables to use specific, flip-vulnerable pages. Like with user-space stack allocation, a kernel stack is allocated upon creation of a new thread or process, and that stack is used for all syscalls made by that thread or process. However, unlike user-space stacks, kernel stacks use UNMOVABLE memory, meaning they pull pages from PCP list different from that used by user space `mmap` and `unmap` calls. Therefore, the attacker needs a method to force the kernel to use "user pages" (MOVAVABLE pages) instead of "kernel pages" (UNMOVABLE pages). We observe from Seaborn [43] that the kernel does use user pages when memory is under pressure, and build on Seaborn's techniques to allow for a more precise memory massaging technique that allows for massaging kernel stacks.

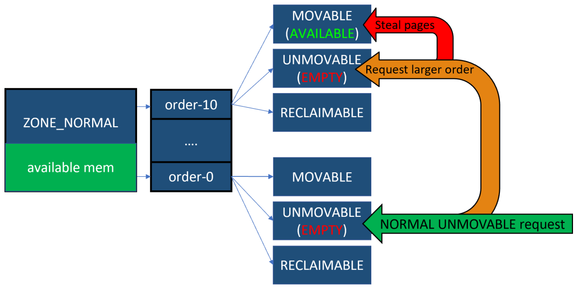


Fig. 4: **Physical Page Stealing**

Allocator Under Presure. As mentioned above, when the zone's total number of free pages falls below the watermark, the next most preferred zone is used. However, as zones include multiple migrate-types, it is possible for the freelist of the requested migrate-type to be empty, yet have enough total zone memory to be above the watermark. In this case, the allocator calls a *stealing function* that steals pages from given "fallback" migrate-types and converts them to the type originally requested. As shown in Fig. 4, this function attempts to steal the largest available block from the fallback type. For UNMOVABLE memory, the first fallback is RECLAIMABLE memory, and the second is MOVABLE memory.

Kernel Massaging Steps. The steps required for kernel stack massaging are similar to those of user space stack massaging. The key difference is that the attacker must first apply memory pressure to force the kernel into using user pages.

Step 1: Draining Kernel Pages. As non-privileged attackers, we cannot directly allocate UNMOVABLE pages. However, each time an allocation is made via `mmap` a page table entry (PTE) is needed to map the virtual and physical pages. Since PTEs use kernel memory, each mmap call uses both user and kernel memory. However, multiple PTEs can fit within a single page, and the address of a PTE depends on its corresponding virtual address. We need to efficiently make allocations large enough such that each PTE needs a new page, but small enough such that the process is not killed for allocating too much memory. Mapping pages at 2MB aligned addresses provides the smallest allocation size such that each PTE allocates a new page. Such allocations are made until no

MOVABLE pages remain, using the `pagetypeinfo` file to monitor the amount of remaining pages. Subsequent mappings will use RECLAIMABLE pages for PTEs. Once the necessary pages have been depleted, the next kernel allocation will use the largest available MOVABLE block.

On machines without access to `pagetypeinfo`, we instead use `buddyinfo` (which is world readable for all kernel versions) and monitor the draining of MOVABLE and UNMOVABLE blocks together (performing Step 1 and Step 2 at the same time), only draining order 4 or higher UNMOVABLE blocks. (See Appendix A for a more detailed explanation of `buddyinfo` compared to `pagetypeinfo`.

Step 2: Draining User Pages. Memory is now in a state that will force the kernel to use the largest available MOVABLE block. However, we need the kernel to use a specific single page (the page containing a bit-flip). We, therefore, need to ensure the target page resides in this block. It is advantageous to make the largest available block as small as possible to improve the chance that the kernel uses the target page for its stack allocation. Thus, the next step is to drain as many high-order free blocks as possible, without dropping the total number of free-pages below the ~~water mark~~watermark. In our machine, we were able to drain all blocks of order 4 or higher.

Step 3: Freeing Target Page. The goal is to free the target page such that ~~is~~ it resides in the largest available block. However, freeing this page will send it to the PCP rather than the buddy allocator freelist. Even when it is free from the PCP, if it does not have any free buddies, it will remain in the order-0 freelist. The freed target page needs to coalesce into an order-4 block, such that the single largest remaining free block contains the flip-vulnerable target page. Fortunately, as explained in Section IV, we have already guaranteed the target page is part of an order-4 (or larger) block (i.e. our target page is part of a 2MiB, order-10 block). Therefore, we can free the target page and all of its buddies to ensure it will coalesce into the largest available block.

The ~~only remaining~~ last obstacle is the PCP list, since even when unmapping a contiguous high-order block, all pages are placed on the appropriate PCP list. However, the ~~Linux~~ `zoneinfo` file shows how many pages reside in each PCP list, and the maximum length of each list, at any given time. Thus, additional pages can be unmapped until the number of pages in the ~~the~~ PCP list reaches the maximum length (186 pages on our ~~machine~~ machines according to `zoneinfo`). This forces pages to be evicted from the PCP list and sent to the buddy allocator freelist, placing the target page in the largest free block of MOVABLE memory.

Step 4: Allocating Kernel Stack. Having freed the target page, and knowing the next kernel stack allocation will use user memory, we can now force a kernel stack allocation. However, freeing pages to force the target page out of the PCP will have slightly alleviated memory pressure, meaning some UNMOVABLE pages will be free. Kernel stack allocations will consume these pages, and subsequent allocations will convert the block containing the target page into an UNMOVABLE block. Additionally, because of the kernel's

Experimental Configurations	SMAP	pagetypeinfo	THP	Leakage
i7-4770,DDR3,Linux 4.17.3	OFF	Readable	N/A	20b/s
i7-7700, DDR4, Linux 5.4.1	ON	Restricted	madvise	6b/m
i9-9900K, DDR4, Linux 5.4.1	ON	Restricted	madvise	6b/m
i7-10700K,DDR4,Linux 5.4.0	ON	Restricted	madvise	6b/m

TABLE II: List of configurations used for our experiment. All mitigations are in their default configurations.

buddy system, the block will be split in half, with one half being used for the kernel stack, and the other half moved to the lower order UNMOVABLE freelist. The target page may be in either half, and allocations must continue to be made to ensure the target page is used for a kernel stack.

Therefore, we use a *kernel stack spray*, allocating many kernel stacks until the UNMOVABLE pages are all depleted again. We perform the kernel stack spray ~~simply~~ by spawning many threads. Each thread can spin in an empty loop until the spraying is done~~. Then, each thread can~~ , and then be tested one-by-one by having the thread make the victim syscall and hammering the target variable until ~~a data leakis observed~~we observe a leak. Once the thread with the target page is found, the other threads ~~can all be released. With this final step, we~~ are released. We can now flip a stack variable residing ~~within~~ in the kernel.

Results. This technique has approximately 66% accuracy ~~. Similar to the user space massaging, we~~ with the `pagetypeinfo` technique (60% accuracy with `buddyinfo`). We expect it to succeed within two attempts.

VI. Gadget Exploitation

At this point, we have forced victim stacks in both user space and kernel space to use flip-vulnerable addresses~~, making us ready to~~ . We can now flip array offset values, force a misspeculation, and leak target values. As a proof of concept, we demonstrate end-to-end double and triple gadget attacks on example victims in user and kernel spaces. These examples serve to verify the attack's ability to leak data. The double gadget attack is demonstrated ~~on a~~ in user-space~~victim, using our user stack massaging technique~~, and the triple gadget ~~attack in kernel space, using our kernel stack massaging technique~~in the kernel.

Setup. ~~The machine used for both attacks uses~~ For the double gadget attack, we use a Haswell i7-4770 CPU with Ubuntu 18.04 and Linux kernel version 4.17.3, the default version shipped on our machine. The DRAM used consists of a pair of Samsung DDR3 4GiB ~~1333MHz DIMMs.~~ DIMMs. For the triple gadget attacks, we use the same machine in addition to machines with Kaby Lake i7-7700, Coffee Lake Refresh i9-9900K, and Comet Lake i7-10700K processors. The latter three machines each use a DDR4 8GB DIMM and run Linux Kernel version 5.4.1, 5.4.1, and 5.4.0, respectively. These configurations are shown in Table II. Note that the two newer processors have additional defenses (i.e., SMAP and restricted `pagetypeinfo` access) not supported by Haswell. We demonstrate our attack even in the presence of such defenses. KASLR is enabled on all machines. Additionally, transparent hugepages (THPs) are set to their default setting of being user-allocatable via an madvise syscall.

~~**Stealing stack canary example.**~~

A. Double Gadget – Stack Canary Leak

In this section we demonstrate how stack canaries can be stolen using a double gadget residing in user space code.

Stack Canaries. A stack canary is a value placed on the stack, adjacent to the return pointer, as a defense mechanism against buffer overflow attacks. An attacker attempting to overflow a buffer and write to a return pointer will ~~inadvertently~~ overwrite the canary~~. Upon jumping to an addresses pointed to by a stack return pointer, the stack canary can be compared to its original value. If the canary has been overwritten, the victim is alerted that the return pointer has been tampered with, and the program ceases execution.~~ , which causes the program to halt. Due to their low-cost ~~, but~~ and effectiveness at preventing buffer overflow attacks, canaries have long been widely deployed as effective, light-weight stack overflow defense mechanisms [10].

Even though they are randomly generated, stack canaries of a child process belonging to a parent process will always have the same stack canary. Thus, if a child process's canary is leaked, it is possible to perform a buffer overflow attack on any child belonging to the same parent, assuming that the code suffers from the memory corruption vulnerability. For example, OpenSSH handles encryption through child processes spawned by a single daemon. Leaking the canary of any one of these child processes allows for circumventing this defense on any other child to leak secret keys.

```
1  uint16_t array1, array2;
2  if(x < array1_size){
3    victim_data = array1[x]
4    z = array2[victim_data * 512];
5  }
```

Listing 4: Double gadget

Example Victim. The victim for this example attack lives within a thread spawned by the attacker, and the victim consists of a double gadget like the one shown in Listing 4, where each array is of type `uint16_t` (Line 1). The arrays live in memory shared by the victim and attacker, but ~~as previously mentioned,~~ attacks without this requirement are possible ~~through the use of~~ by using a PRIME+PROBE side channel [35]. The code is compiled such that stacks include secret canaries and cease execution if a canary is ~~ever modified.~~

modified. Having the victim reside in an attacker-spawned thread allows for user space stack massaging, but extends to any process that can be forcibly spawned~~by the attacker, as can done to~~, such as OpenSSH [27].

Stealing Canaries. Due to their location at the end of the victim stack, just past the end of target arrays, stack canaries act as a prime target for the double gadget attack. Reading the canary requires flipping lower-order bits of the array offset, such that the corresponding array access points just past the end of the array to the stack canary.

A stack canary is typically 32 to 64-bits long and stored ~~just before~~ at the address just below the return pointer. Spectre v1 attacks steal a single "word" of data per malicious offset value, where a word corresponds to the innermost array's data type. In our victim, `array1` is a `uint16_t` array. Each malicious value of `x` points to and steals an 16 bit value, meaning the gadget must be used four times, each with a different malicious value to steal a 64-bit stack canary. ~~An example is pictured in Fig. ??. Here, a 64 bit stack-canary is split up across four addresses, each containing two bytes. The left side shows a legal access used in the training phase, and the right side shows bit-flips used to retrieve portions of the canary.~~

Target Flip. The Rowhammer ~~flip~~ bit-flip needs to push the ~~the~~ offset past the end of the victim array and point to the stack canary. Since the stack canary is separated into multiple words, we may either find a victim row with multiple bit-flips, or allow the victim to naturally cycle through values and hammer with the necessary timing to push the offset to different words of the canary. We use the latter approach, since we observe few rows that contain multiple flips on our machine.

Memory Templating and Massaging. We perform memory templating as described in Section IV to find useful bit-flips. The victim offset resides at a particular page offset within the stack, meaning the required flip must occur at the same offset. Memory was templated for approximately 2.5 hours to find this specific flip. The page containing this flip is unmapped and the victim thread is spawned, forcing the offset variable within the victim thread to use the flip-vulnerable page.

Triggering Spectre. The victim is left to run with legal values used for its offset, ~~training~~ which trains the branch predictor. We wait for the victim to set the offset to the appropriate value corresponding to the given target word of the canary. For this example, the victim and attack code run synchronously, but FLUSH+RELOAD can be used to accurately monitor the execution of victim code to provide attacker synchronization [52]. We then evict the offset from the cache, forcing the gadget to use the flipped value in a state of ~~mis-speculation~~misspeculation. One word of the canary is accessed and used as an offset to load data into the cache, allowing us to use FLUSH+RELOAD to retrieve the target. The victim value is left to change, and the hammering is repeated to retrieve the rest of the canary.

Leakage Rate. As mentioned before, the array accesses 16 bits at a time, meaning 16 bits are leaked per flip and instance of FLUSH+RELOAD. We observed a leakage rate of approximately 8b/s, meaning the entire canary is leaked in about 8 seconds with 100% accuracy.

B. Triple Gadget - Arbitrary Kernel Reads

This second example demonstrates how the triple gadget within a kernel syscall can be used to achieve arbitrary reads of kernel memory. This is particularly dangerous since kernel memory is shared across all processes, meaning an attacker with access to kernel memory can observe values handled by the kernel for any process running on the same machine. ~~This may include sensitive data such as user passwords. Such data is typically only accessible with sufficient privilege, but arbitrary~~

~~reads using this attack gives unprivileged attackers access to anything reachable by the kernel.~~

```
if(x < array1_size){
  attacker_offset = array0[x]
  victim_data = array1[attacker_offset]
  y = array2[victim_data*512];
}
```

Listing 5: Triple Gadget

Example Victim. The example victim for this attack is a syscall in which we inserted a triple gadget, as shown in Listing 5. Since syscalls execute with kernel privilege, any data within the kernel can be leaked. For this example, we target a 10-character string within the syscall's code that is out of bounds from the target arrays. Additionally, ~~as with the previous example,~~ the attacker and victim share the arrays used in the triple gadget.

Memory Templating. As done in the double gadget attack, we begin by finding a useful bit-flip. The purpose of the flip here is to force the victim array (in the kernel) to point to the attacker-controlled data~~in the user space heap~~. Thus, a specific high order bit-flip is needed to point from ~~kernel space to user space.~~ the victim to region of data we control. To reduce the time required to find the bit-flip, we configure the victim such that it can use an array offset at any position in the stack, by including victim variables at every offset position. Therefore, there is no need to find a flip at a specific offset; we only need to change a specific ~~(bit 45 in our case)~~ bit at any aligned 64-bit word within the page.

Attacker Controlled Data. ~~We allocate many addresses within user space using the MMAP_FIXED flag to fix the addresses such that the single flip will cause the kernel array to point to the~~ One method of controlling data in the victim's address space would be to simply allocate a large memory chunk on the user space heap and fill this chunk with the desired value. The bit-flip would then cause the victim to point from kernel memory to our data in user memory. However, this requires breaking Kernel Address Space Layout Randomization (KASLR) in order to precisely know the difference between the target kernel address and the controlled user space address. Furthermore, Supervisor Mode Access Prevention (SMAP) blocks the kernel form reading user memory, and is enabled by default on the last several generations of Intel processors [2]. Therefore, we instead inject our data into the kernel at sets of addresses that differ from the target-flip-address by a single bit.

SMAP Bypass. We borrow from kernel heap-spray attacks [12], [21], which demonstrate methods of filling the kernel heap with attacker controlled data. ~~Additionally, the user-controlled data needs to point to the virtual address containing the desired secret data . In this case, the~~ These techniques take advantage of syscalls such as `sendmsg` or `msgsnd`, which allocate kernel heap memory using `kmalloc` and then move user data into these kernel addresses. To prevent these syscalls from freeing the data before returning, attackers use the `userfaultfd` syscall to stall the kernel. This syscall allows users to define their own thread that will handle any page faults on specified pages. When the attackers call a data-inserting syscall (such as `sendmsg`) they pass arguments with `N` pages worth of data, but only allocate `N - 1` physical pages. When `sendmsg` attempts to copy the data from user to kernel space, it will encounter a page fault on the final page. The thread fault handler, assigned by `userfaultfd`, is configured to spin in an endless loop, leaving `sendmsg` stuck, after having copied `N - 1` pages of user data into kernel memory.

Stack Data Insertion. While the above method is useful for inserting attacker-controlled data ~~points to a string within the kernel syscall that is outside the range of all arrays. This example attack assumes the victim does not have kernel address space layout randomization (KASLR) enabled, simplifying the process of obtaining kernel addresses. Past work has shown how kernel addresses can be obtained even through KASLR [?], meaning such techniques can be applied to attack systems with KASLRenabled.~~ into the kernel's *heap*, heap-insertion is not useful for SpecHammersince kernel heap addresses will never have only one bit of difference from kernel stack addresses. However, numerous syscalls, including `sendmsg`, take a user defined *message header* which is placed on the kernel stack. To ensure that this inserted value will land on an address that is one bit-flip away from the flip-target, we spawn many threads that all use `sendmsg` to insert kernel stack data, giving high probability (87%) of an address match.

Controlling Page Offsets. The only remaining issue is the offset within the page. Stack offsets for kernel syscalls are always fixed and we need to insert data into an address with a page offset that matches that of our flip-target. Fortunately for the attacker, there are numerous syscalls (e.g. `sendmsg`, `recvmsg`, `setxattr`, `getxattr`, `msgsnd`) that allow for writing up to 256 bytes of the kernel stack, giving a range of offset options. Additionally, these syscalls are called from other syscalls as well, (e.g.`socket`, `send`, `sendto`, `recv`, `sendmmsg`, `recvmmsg`) and each of these use a varying amount of stack space before calling the previously listed syscalls, essentially allowing the attacker to "slide" the position of the inserted data up and down the stack.

As an example, we find that the target-variable of the example gadget presented in Section VII-B has a page offset of `0xd20` (when it is called during the spawning of a new thread) and `sendmmsg` can be used to control data on the kernel stack from `0xcf0` to `0xd70`. Thus, the triple gadget attack can work by pointing from a victim kernel address to an attacker controlled kernel address, allowing the attack to work in the presence of SMAP. Since KASLR only randomizes the kernel's base address, the difference between these addresses remains constant, thereby neutralizing KASLR.

Kernel Stack Massaging. Next, we run the kernel stack massaging technique from Section V-B, forcing the syscall to use the flip-vulnerable page for its array offset. ~~Numerous threads are allocated~~ We allocate numerous threads as part of the stack spray, and there is a possibility none of the kernel stacks contain the flip-vulnerable page. Therefore, ~~each thread is checked~~ we check each thread for the target page, and if the page is not found, we repeat the templating and massaging steps ~~are run again~~ until a target page lands within a kernel stack. ~~Overall, combined with the time taken to find useful flips, it takes a total of 34 min on average to land a useful flip within the kernel.~~

Triggering Spectre. Finally, the thread containing the target page makes the syscall containing the victim gadget, which runs repeatedly with a loop of of legal offset values in order to train the branch predictor. The offset value is occasionally hammered and evicted from the cache, causing the inner most array to point to user data in a state of ~~mis-speculation~~missspeculation. The FLUSH+RELOAD side channel is used to confirm the target secret (in this case, the value of the victim's string) has been correctly leaked. We then modify the attacker-controlled data to point to any secret value within the attacker's address space, and the hammering is repeated to leak the next target value.

Offline Phase Performance When running on the Haswell machine, in which SMAP is disabled and `pagetypeinfo` is unrestricted, the time taken to find pages with useful flips and land a such a page in the kernel is 34 minutes. While our new `buddyinfo` and SMAP bypass techniques present slightly reduced accuracies, they conversely *reduce* the time needed to find flips and land a useful page. The `buddyinfo` technique relaxes the requirements on draining user pages (to only draining order 4 or larger blocks, rather than draining all blocks), meaning each massaging attempt takes less time.

Furthermore, the SMAP technique allows a *range* of bits to be useful, since we need any flip that points from (victim) kernel stack to (our controlled) kernel stack. These two regions of memory are much closer together the case of a kernel stack victim and controlled user space region, meaning we can make a selection among many lower order bits (bits 5 through 28) rather than being forced to flip the only high-order bit that points from kernel space to user space (bit 45). Thus, while this technique introduces another probabilistic element (with 87% accuracy) the time needed to find a single useful flip to perform the attack is reduced. Consequently, the attack requires an average 9 minutes on average to find a useful flip and land it in the kernel across all machines.

Leakage Rate. `array1` is of type `uint8_t`, meaning each misspeculation leaks 8 bits of data. After performing the prerequisite templating and massaging steps, the leakage occurs at a rate of ~~about 2 to 3 words per second, meaning~~ 16 to 24b/s on DDR3. We leaked the target string with 100% accuracy. When running on DDR4, multi-sided hammering is required, which requires more time per hammering round, consequently reducing the leakage rate to about 4 to 19b/min (6b/min on average), also with 100% accuracy on the three DDR4 machines listed in Table II.

VII. Gadgets in the Linux Kernel

~~The threat the combined attack poses is two-fold. First, it can bypass taint tracking, which is the only proposed defense that would block all types of Spectre gadgets [4], [51]. Second, the restriction for what qualifies as a Spectre gadget is relaxed such that the attacker no longer needs to control the offset variable.~~ To understand the impact ~~this~~ the SpecHammer relaxation has on the number of exploitable gadgets in real-world code, we run a gadget search tool, Smatch [29], on the ~~latest~~ Linux kernel.

A. Gadget Search.

Smatch. Smatch was initially designed for finding bugs in the Linux kernel. However, after Spectre was discovered, a `check-spectre` function was added, which searches for gadgets. It searches for segments of code in which ~~an~~ a nested array access occurs after a conditional statement, and the offset into the array is controlled by an unprivileged user. It additionally checks if the nested accesses occur within the maximum possible speculation window, and if the accesses use an array_index _nospec macro, which sanitizes array offsets by bounding them to a specified size.

Tool Modification. We modified the tool to remove the condition of an attacker controlled offset, and searched only for gadgets in which the attacker *does not* control the offset. In addition, we added a function to search for triple gadgets as well, which checks if the value of a nested array access is used as an offset for a third array access.

Results. When running the ~~original,~~ unmodified `check-spectre` function on the Linux kernel 5.6, we find about 100 double gadgets, and only 2 triple gadgets. Modifying the function to search for ~~gadgets vulnerable to the combined attack~~ SpecHammergadgets leads it to report about 20,000 double gadgets, and about 170 triple gadgets.

Bypassing Taint Tracking. Such a large number of potential gadgets exposes more holes for Spectre attacks on sensitive, real-world code. Furthermore, oo7 [51], which is the only defense that can efficiently mitigate all forms ~~Spectre [4]~~ of Spectre [4], does not work against SpecHammer gadgets. This defense identifies nested array access that use an *untrusted* array offset value (i.e. a value coming from an unprivileged user). Any gadgets using such an offset are considered ~~"~~"tainted," and are prevented from performing out of bounds memory accesses. However, since the newly discovered gadgets use variables that cannot be directly modified by attackers, they are considered trustworthy, and would go unmitigated by oo7.

Additional Gadgets. Even after making the modification to smatch to include gadgets without attacker-controlled offsets, we observed that smatch was still unable to detect all ~~gadgets that were potentially exploitable by~~ potential SpecHammer ~~-~~gadgets, demonstrating that existing gadget detection tools are not sufficient for finding all exploitable code.

B. Kernel Gadget Exploit

To understand the nature of gadgets that remained undetected by smatch, we chose to explore the kernel source code by hand to identify potential gadgets that may be newly exploitable with the flexibility granted by Rowhammer. For example, in addition to manipulating array offsets, Rowhammer bit-flips allow for the indirect modification of pointers as well. Modifying a single `struct` pointer can lead to a chain of pointer dereferences ending with secret-dependent cache accesses. This points to a new type of gadget compared to those presented in Spectre [25], as it relies on pointer deferences rather than nested array accesses. One particular example of this lies in the kernel's `page_alloc.c` file.

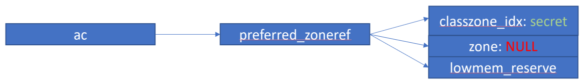


Fig. 5: **alloc_context struct pointer**

page_alloc.c This file contains the code used for all physical page allocation. The `get_page_from_freelist` function in particular contains the SpecHammer gadget~~, with~~; a simplified version with only the relevant code lines is presented in Listing 6. Note that the gadget does not contain cosecutive array accesses, but rather dereferences consecutive struct pointers, and uses the result for an array access. The `allocation_context` (`ac`) struct pointer, shown in Figure 5, is particularly important, as many variables used in the function are obtained from this pointer.

```
1   get_page_from_freelist{
2     struct alloc_context *ac;
3     struct zoneref *z = ac->preferred_zoneref;
4     struct zone *zone;
5
6     for(zone=z->zone;zone;z=find_next_zone(z,ac->
    zone_highidx);
7     zone=z->zone){
8       ....
9       preferred_zone = ac->preferred_zoneref;
10      idx = preferred_zone->classzone_idx;
11      ....
12      z->lowmem_reserve[idx];
13    }
14  }
```

Listing 6: Code Gadget for the double gadget attack

Forcing Misspeculation By manipulating the value of `ac` to point to a region of attacker-controlled code, it is possible to control all variables obtained from an `ac` dereference, and control the ~~execution flow of the allocator function~~victim's execution flow. More specifically, an attacker ~~can allow the function to run normallyto teach~~ run the function normally, teaching the predictor that the `for` loop at Listing 6, Line 6 will be entered. Then, `ac` can be modified by hammering such that the dereferences at Lines 3 (`z = ac->preferred_zoneref`) and 6 (`zone = z->zone`) set zone equal to ~~0 or~~ `NULL`. This triggers a ~~missspeculation~~misspeculation, since the `for` loop should terminate immediately, but will actually begin its first iteration due to the prior training. Furthermore, `ac` has been set such that during this misspeculation, the chain of dereferences at Listing 6 Lines 9 and 10 causes `idx` to equal secret data, causing a secret-dependent access at Line 12 (`lowmem_reserve[idx]`)~~which can be recovered by a~~, recoverable by cache side channel.

Results. To empirically verify this behavior, we instrumented page_alloc.c file to flip bits as needed, ~~we~~ and found it is possible to manipulate the function's control flow and cause a misspeculation that leaks kernel data. We recovered an ~~8bit~~ 8-bit character inserted in the kernel code that is normally out of range of the manipulated array, by inserting code that uses a Flush+Reload channel. This can be replaced with Prime+Probe ~~,~~to retrieve secrets without modifying `page_alloc`.

VIII. Mitigations

~~Since the presented attacks rely on combining Spectre and Rowhammer, defenses can protect against the new attacks by either mitigating Spectre or Rowhammer.~~

Spectre. Developing a defense focused on the Spectre aspects is likely the more difficult option. While other variants of Spectre received effective and efficient mitigations [4], [30], [46], Spectre v1 was seen as more as an inherent security flaw caused by branch prediction with no simple solution.

Taint tracking, the only defense previously known to protect against all forms of Spectre v1 [4], [51], is thwarted by the new combined attack, as it relies on a Spectre limitation not present in the combined attack. Other defenses [5], [38], [39] designed to protect against Spectre v1 provide incomplete protection, working only in specific cases, and often come at a prohibitively high performance cost [4].

Rowhammer. For Rowhammer, on the other hand, numerous hardware and software defenses have been developed to prevent or detect bit-flips, beginning with PARA [23]. PARA ~~initially~~ randomly refreshes rows~~with a low probability for each row. As pairs of rows are repeatedly accessed, the refresh probability for the potential victim rows is increased~~, giving more weight to rows with repeated accesses. However, ~~PARA's main weakness is that it~~ this does not guarantee ~~a refresh for~~ protecting rows that are about to flip, but ~~rather~~ only grants a high probability of refresh. For ~~example, the triple gadget attack presented here needs just~~ our triple-gadget attack that requires a single bit-flip, ~~and PARA cannot provide guaranteed protectionagainst this flip~~PARA does not guarantee protection.

A defense similar to PARA, target row refresh (TRR) *does guarantee* a refresh whenever two aggressor rows pass a certain activation threshold. ~~TRR is already widely implemented in DDR4, and was long believed to be the strongest defense against Rowhammer.~~ However, TRResspass [14] has recently shown how bit-flips can be obtained despite TRR by performing scattered aggressor row accesses. Furthermore, ~~it was shown that~~ by applying this technique, DDR4 ~~is~~ was found to be even more susceptible than DDR3 to bit-flips [22]~~, and we have presented how to obtain an even greater number of DDR4 flips with modifications to TRResspass.~~.

Another common hardware defense against bit-flips is error correcting codes (ECC). Initially designed to catch bit-flips induced by natural errors, these functions are able to correct single flips, and detect up to two flips, within a given row. ~~Commonly used in high-end servers, ECC were believed to be an effective, albeit expensive, defense against~~

~~Rowhammer.~~ However, ECCploit [9] demonstrated a timing side-channel produced by single-flip corrections~~. This side channel can be used to detect flips, one at a time, even under ECC, allowing~~ , that allows attackers to find rows ~~with more than two flips. By flipping three or more bitssimulatenously~~containing multiple flips. By simulatenously flipping multiple bits, Rowhammer attacks can go undetected by ECC~~. Additionally, RAMBleed [27], demonstrated how bit-flips can be used to *read* unaccessible memory requiring the attacker to know *where* flips have occurred rather that relying on the effects of a flip propagating through a victim machine. The timing side-channel can be used to tell if a flip occured, even if it was corrected by ECC, making it~~ , making ECC an ineffective defense.

IX. Conclusion

We have demonstrated how Spectre and Rowhammer can be combined to circumvent the only defense believed to work against all forms of Spectre v1. Furthermore, we found the number of potential gadgets in the Linux kernel increases drastically with this new attack. Proof-of-concept gadgets show the attack's ability to leak data from ~~both~~ user and kernel space victims. In future work we ~~would like~~ seek to understand the effect of relaxing gadget requirements on other sensitive code~~besides the Linux kernel.~~.

References

[1] 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," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[2] A. Baumann, "Hardware is the new software," in *16th Workshop on Hot Topics in Operating Systems*.1em plus 0.5em minus 0.4emACM, 2017, pp. 132–137.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *2016 IEEE symposium on security and privacy (SP)*.1em plus 0.5em minus 0.4emIEEE, 2016, pp. 987–1004.

[4] C. Canella, J. Van 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 *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 249–266.

[5] C. Carruth. (2018) Rfc: Speculative load haredning (a spectre variant #1 mitigation.

[6] A. Chakraborty, S. Bhattacharya, S. Saha, and D. Mukhopadhyay, "Explframe: exploiting page frame cache for fault analysis of block ciphers," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1303–1306.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*.1em plus 0.5em minus 0.4emIEEE, 2019, pp. 142–157.

[8] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 712–728.

[9] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 55–71.

[10] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 119–129.

[11] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript," in *USENIX Sec*, Aug. 2021. [Online]. Available: Paper=https://download.vusec.net/papers/smash_sec21.pdfWeb=https://www.vusec.net/projects/smashCode=https://github.com/vusec/smash

[12] L. Dixon, "Using userfaultfd," 2016. [Online]. Available: https://blog.lizzie.io/using-userfaultfd.html

[13] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," in *2018 IEEE Symposium on Security and Privacy (SP)*.1em plus 0.5em minus 0.4emIEEE, 2018, pp. 195–210.

[14] P. Frigo, E. Vannacc, 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 *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 747–762.

[15] M. Gorman, "Understanding the linux virtual memory manager," *IEEE Transactions on Software Engineering*, 2004.

[16] D. Gruss, "Program for testing for the dram "rowhammer" problem using eviction," May 2017. [Online]. Available: https://github.com/IAIK/rowhammerjs

[17] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[18] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International conference on detection of intrusions and malware, and vulnerability assessment*. 1em plus 0.5em minus 0.4emSpringer, 2016, pp. 300–321.

[19] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[20] J. Horn. (2018) speculative execution, variant 4: speculative store bypass. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

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

[21] invictus, "Linux kernel heap spraying / uaf," 2017. [Online]. Available: https://invictus-security.blog/2017/06/15/linux-kernel-heap-spraying-uaf/

[22] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 638–651.

[23] 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," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[24] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757*, 2018.

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[26] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[27] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 695–711.

[28] C. Lameter and M. Kim, "Page migration," 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/page_migration

[29] J. LCorbet, "Finding spectre vulnerabilities with smatch," 2018. [Online]. Available: https://lwn.net/Articles/752408/

[30] M. Linton and P. Parseghian, "More details about mitigations for the cpu speculative execution issue," 2018. [Online]. Available: https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[32] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing rowhammer faults through network requests," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 710–719.

[33] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[34] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[35] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers track at the RSA conference*. Springer, 2006, pp. 1–20.

[36] C. Percival, "Cache missing for fun and profit," 2005.

[37] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks," in *25th {USENIX} security symposium ({USENIX} security 16)*, 2016, pp. 565–581.

[38] F. Pizlo, "What spectre and meltdown mean for webkit," 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[39] T. C. Projects, "Site isolation," 2018. [Online]. Available: https://www.chromium.org/Home/chromium-security/site-isolation

[40] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 1–18.

[41] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[42] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 279–299.

[43] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[44] M. Seaborne, "Program for testing for the dram "rowhammer" problem," Aug 2015. [Online]. Available: https://github.com/google/rowhammer-test

[45] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 213–226.

[46] P. Turner, "Retpoline: a software contruct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[47] 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.

[48] VandySec, "rowhammer-armv8," Apr 2019. [Online]. Available: https://github.com/VandySec/rowhammer-armv8

[49] K. Viswanathan, "Disclosure of hardware prefetcher control on some intel processors," 2014. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html

[50] vusec, "trresspass," Mar 2020. [Online]. Available: https://github.com/vusec/trrespass

[51] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2020.

[52] Y. Yarom. (2016) Mastik: A micro-architectural side-channel toolkit. [Online]. Available: https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

[53] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise l3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014.

[54] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering rowhammer hardware faults on arm: A revisit," in *ASHES*, 2018.

[55] Z. Zhang, Y. Cheng, Y. Zhang, and N. Surya, "Ghostknight: Breaching data integrity via speculative execution," in *arXiv*, 2020.

APPENDIX

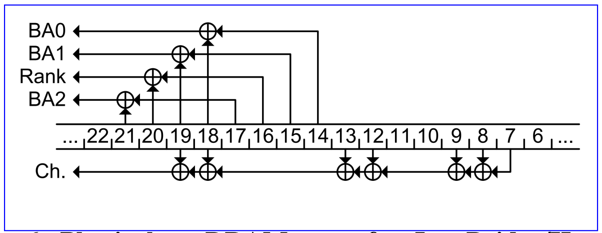


Fig. 6: **Physical to DRAM map for Ivy Bridge/Haswell (taken from [37]).**

A. Reverse Engineering Virtual to DRAM Address Mapping

The following section explains the techniques used to obtain the virtual to DRAM address mapping needed for double-sided Rowhammer. These techniques manipulate the Linux buddy allocator to first obtain a virtual to physical address mapping [27]. Then, they use a timing side-channel to determine which physical addresses correspond to rows in the same bank [37], reverse engineering the physical to DRAM address mapping. However, these techniques relied on the use of the `pagetypeinfo` file for memory manipulation, which has since been restricted to high privileged users. We therefore develop a new technique using the world-readable `buddyinfo` file.

Buddy Allocator. The buddy allocator is Linux's system for handling physical page allocation. It consists of lists of free pages organized by *order* and *migratetype*. The order is essentially the size of a free block of memory. Typically, requests for pages from user space (for example, via `mmap`) are served from order-0 pages. Even if the user requests many pages, she will likely be served with a non-contiguous block of fragmented pages. If there are no free blocks of the requested size, the smallest available free block is split into two halves, called *buddies*, and one buddy is used to serve the request, while the other is placed in the free list of the order one less than its original order. When pages return to the freelist, if their corresponding buddy is also in the freelist, the two pages are merged and moved to a higher-order freelist. Migratetypes essentially determine whether a page is meant to be used in user space (MOVABLE pages) or kernel space (UNMOVEABLE pages) [15].

pagetypeinfo & buddyinfo files. The `pagetypeinfo` file shows how many free blocks are available for each order and migratetype. While previous techniques [27], [47] used this file to track the state of free memory, `pagetypeinfo` has since been made unreadable for low-privilege users. However, a similar file, called `buddyinfo` shows how many *total* free blocks are available for each order, combining the number of kernel and user pages. Since `pagetypeinfo` has been restricted from attacker access, we present a new technique that uses `buddyinfo` for obtaining contiguous blocks of memory.

Obtaining Contiguous Memory Blocks. In order to control sets of contiguous DRAM rows, we must first obtain a large chunk of contiguous physical memory. For the eventual memory massaging step, described in Section V, the bit-flip needs to reside in a contiguous block of memory at least 16 pages long. Additionally, as we will see in the following paragraph, a 2MiB block will be helpful in obtaining physical addresses. However, if we request a 2MiB block via `mmap`, the allocator will service this request via fragmented, rather than contiguous, memory. Therefore, to obtain a 2MiB contiguous block, we first allocate enough memory to drain all smaller sized (1MiB or smaller) user blocks, forcing the allocator to supply us with a contiguous 2MiB block.

Using the buddyinfo file. However, with `buddyinfo` we can only see the combined total of user *and* kernel blocks remaining, but need to know when the number of 1MiB (and lower) *user* blocks is worth less than 2MiB of memory. To bypass this issue, we allocate blocks while monitoring the remaining total amount via `buddyinfo`. By placing our allocations at consecutive virtual addresses, we ensure our allocations will mostly use user blocks, since kernel blocks for new page table allocations will rarely be needed. Therefore we can continue to drain blocks and watch the *total* 1MiB block count decrease until it hits a *minimum value* and increases again. This behavior signifies there were no remaining user blocks to fulfill the request, requiring the 1MiB user block free list to be refilled. The observed *minimum value* is therefore the number of free 1MiB *kernel pages*, allowing us to subtract this value from the total value at any given moment to obtain the number of free 1MiB user pages.

We run the drain process again, subtracting the number of kernel pages, until the remaining 1MiB user pages equals 0. We can use the same process to drain the smaller blocks until they consist of less than 2MiB worth of memory. Finally, we request two 2MiB chunks of memory via `mmap`. Since the allocator does not have enough smaller order blocks to fulfill this request with fragmented pages, it is forced to supply a contiguous 2MiB chunk. Our approach is able to produce 2MiB pages with the same 100% accuracy of `pagetypeinfo` . Since the additional step of calculating the number of kernel blocks needs to be performed only once during the entire attack (*not* once per massaging attempt), using the *buddyinfo* technique incurs a negligible time cost.

Physical Addresses. To obtain the virtual to physical memory mapping, we use technique presented in [27]. Having already obtained a 2MiB block, we can learn the lowest 21 bits of a physical address by finding the block's offset from an aligned address. We obtain this offset by timing accesses of multiple addresses to learn the distances between addresses on the same bank. By identifying the distance for each page within the the block, we can retrieve the offset. With the mapping from virtual to physical to DRAM addresses, we can sort virtual addresses into aggressor and victim addresses corresponding to three consecutive DRAM rows.

DRAM Addresses. Next, we require the physical to DRAM address mapping. We can obtain it using Pessl's timing side-channel [37]. This technique takes advantage of DRAM banks' rowbuffer. Upon accessing memory, charges are pulled from the accessed row into a rowbuffer. Subsequent accesses read from this buffer, reducing access latency. All rows that are part of the same bank share a single rowbuffer. Therefore, consecutive accesses to different rows within the *same bank* will have increased latency, since each access needs to over-write the rowbuffer. By accessing pairs of physical addresses and categorizing them into fast and slow accesses, an attacker can learn whether pairs lie in the same bank. Attackers can compare the bits of enough addresses that lie in the same bank to retrieve the mapping from physical addresses to DRAM.

Pessl et. al. [37] present the mapping function for numerous processors, such as the Haswell mapping (shown in Figure 6). Therefore, for attacks on Haswell, we can use this mapping as is. For newer processors, we run Pessl's attack (as provided in [50]) on several machines, and obtain the mapping for Kaby Lake, Coffee Lake, and Comet Lake processors.

Contiguous Blocks on DDR4. We previously explained the need for 2MiB blocks when hammering on a Haswell machine, since the physical to DRAM mapping uses the lower 21 bits. Newer processors use up to bit 24 for their mapping when a machine uses two channels with two DIMMs on each channel (4-DIMM configurations). Up to bit 22 is used for two-DIMM configurations and up to bit 21 for one-DIMM configurations [11]. These newer processors are designed designed to use DDR4. DDR4 Rowhammer techniques such as TRRespass [14], use hugepages to obtain 2MB blocks which are sufficient for one-DIMM configurations. For two-DIMM configurations, memory massaging techniques can be used to obtain 4MB contiguous blocks [11]. For 24-bit configurations, accuracy is reduced by the number of unknown bits, meaning 1/4 reduction of flips in the worst case of 24 bits.

B. Modifications Made to Rowhammer Code

Rowhammer.js Modifications The code listings in this section show the changes we made to existing Rowhammer repositories to prevent the cache from masking bit-flips. Listing 7 shows the changes made to Rowhammer.js's native code. The first change on lines 530 and 531 fix a simple error regarding virtual and physical addresses. The original code passes virtual addresses into the `get_dram_mapping` function, while this function is designed to use physical addresses. The second modification occurs in lines 560 to 573. In these additional lines of code, we flush any victim rows immediately after they are initialized with test values. This ensures that when we later read these rows to check for flips, we will read directly from DRAM and not the cache.

TRRespass Modifications Listing 8 shows the modifications made to TRRespass. We found that cache flushes needed to be added to multiple regions of code to minimize the number of hits that occur when checking for flips. Data is first initialized in the `init_stripe` function starting at line 386. This function is called once during a TRRespass session to initialize the entire region of victim data. While many rows are naturally evicted from the cache due to initialization over a region too large to fit in the cache all at once, many initialized values do still remain in the cache in the original code. We therefore added flushes after every write to memory.

```
.....
if(OFFSET2 > =0)
second_row_page = pages_per_row[row_index+2].at(OFFSET2);
if (
  //*****fixed bug**********
  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,first_row_page)*0x1000))
  !=
  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,second_row_page)*0x1000))
  //***********************
 )
{

....

  #ifdef FIND_EXPLOITABLE_BITFLIPS
  for(size_t tries = 0; tries < 2; ++tries)
  #endif
  {
    //*****cache flush victim**********
    int32_t offset = 1;
    for (; offset < 2; offset += 1)
    for (const uint8_t* target_page8 :
    pages_per_row[row_index+offset])
    {
      const uint64_t* target_page = (const uint64_t*)
      target_page8;
        for (uint32_t index = 0; index < (512);
        ++index) {
            uint64_t* victim_va = (uint64_t*)
            &target_page[index];
            asmvolatile("clflush(%0)"::"r"(victim_va):%"memory");
          }
        }
    //********************************
    hammer(first_page_range, second_page_range, number_of_reads);
....
}
```

Listing 7: Rowhammer.js Modifications

TRRespass then checks for flips using the `scan_stripe` function starting in line 571. When finding a flip (if `res` is non-zero), the flipped data is reinitialized to its initial value. However, there may still be some data within the same cache line that has not yet been checked. We therefore flush the cache to ensure checked data is pulled from DRAM and not the cache.

After completing a hammering session, TRRespass calls `fill_stripe` which refills victim rows with initial data. Similar to the `init_stripe` function, we must flush this initial data from the cache.

Finally, while the `hPatt_2_str` function (starting at line 134) does not directly interact with victim data, we found that its `memset` call does pull victim data into the cache. This is likely due to the processor's buddy cache system, which pulls adjacent cache-lines together, even when only one is accessed, to improve performance. We therefore flush this memset data as well.

C. *Verifying the Effects of Caching.*

In order to confirm that the reads were in fact reading cached data, we modified the existing code to measure the number of cache hits and misses that occur per victim address check. We do so using by timing each access and marking fast accesses (less than 100 cycles) as cache hits and all slower accesses as cache misses. Since accesses pull entire *cache lines* into the cache, and each line is 64B, we only measure the first access per cache line, and all other accesses within the same set are labeled according to the timing of their first address. Additionally, we measured the number of hits and misses observed when extra cache flushes were added to ensure we read victim data from DRAM rather than the cache. Finally, we

Rowhammer.js	Without cache flushes	With cache flushes
hits	105,530,250	1
misses	377,915	107,347,967
%flips on misses	100%	100%
flips	12	2806

TABLE III: The effect of flushing victim addresses on Rowhammer.js

TRRespass	Without cache flushes	With cache flushes
hits	23,914,118	14,078
misses	2,081,626,490	2,105,526,350
%flips on misses	100%	100%
flips	431	4795

TABLE IV: The effect of flushing victim addresses on TRRespass

```
.....
char *hPatt_2_str(HammerPatter * h_patt, int fields)
{
  static char patt_str[256];
  char *dAddr_str;

  memset(patt_str, 0x00, 256);
  //******new cache flushes******
  clflush(patt_str);
  clflush(patt_str + 64);
  clflush(patt_str + 128);
  clflush(patt_str + 192);
  clflush(patt_str + 256);
  //*****************************

.....

void fill_stripe(DRAMAddr d)addr, uint8_t val, ADDRMapper *
mapper)
{
  for (size_t col = 0; col < ROW_SIZE; col += (1 << 6)) {
    d_addr.col = col;
    DRAM_pte d_pte = get_dram_pte(mapper, &d_addr);
    memset(d_pte.v_addr, val, CL_SIZE);
    //******new cache flushes******
    clflush(d_pte.v_addr);
    clflush((d_pte.v_addr) + CL_SIZE);
    //*****************************
  }
}

.....

void init_stripe(HammerSuite * suite, uint8_t val){
.....
      for (size_t col = 0; col < ROW_SIZE; col += (1 <<
      6)) {
        d_tmp.col = col;
        DRAM_pte d_pte = get_dram_pte(mapper, &d_tmp);
        memset(d_pte.vaddr, val, CL_SIZE);
        //******new cache flushes******
        clflush(d_pte.v_addr);
        clflush((d_pte.vaddr) + 64);
        //*****************************
      }
    }
  }
}
.....
void scan_stripe(HammerSuite * suite, HammerPattern * h_patt,
size_t adj_rows, uint8_t val){
.....
  if(res){
    for (int off = 0; off < CL_SIZE; off++){
      if (!((res >> off) & 1))
        continue;
      d_tmp.col += off;

      flip.d_vict = d_tmp;
      flip.f_og = (uint8_t) t_val;
      flip.f_new = *(uint8_t *) (pte.v_addr + off);
      flip.h_patt = h_patt;
      export +flip(&flip);
      memset(pte.v_addr + off, t_vall, 1);
      //******new cache flushes******
      clflush(pte.v_addr + off);
      //*****************************
    }
    memset((char *)(pte.v_addr), t_val, CL_SIZE);
    //******new cache flushes******
    clflush(pte.v_addr);
    clflush((pte.v_addr) + CL_SIZE);
    //*****************************
  }
```

Listing 8: TRRespass Modifications

disabled the cache prefetcher [49], since otherwise, accessing a single set would pull additional sets into the cache and make subsequent accesses appear to be cache hits even if they had been flushed prior to hammering. For the DDR3 tests, we used a Haswell i7-4770 processor running Linux kernel 4.17.3, and Samsung DDR3 4GB DIMM. For DDR4 we used a Coffee Lake i7-8700K processor running Linux kernel 5.4.0 and Samsung DDR4 8GB DIMM. The experiments were run for 2 hours each. The data was initialized with a 0-1-0 stripe pattern for all experiments.

The results are shown in Table III (DDR3) and Table IV (DDR4). The DDR3 test was based on Rowhammer.js [16] and DDR4 on TRRespass [50] as they are the latest Rowhammer repositories for their respective type of DIMM. For both tests 100% of the flips were observed on cache miss accesses, supporting our observation that the cache masks bit-flips. With the DDR3 tests, neglecting to use victim cache flushes results in a large majority (99.64%) of the flip-checks reading cached data. A non-negligible 377,915 accesses *do occur* on cache misses, which is likely why the original code was able to observe any flips at all. However, once the cache flushes are added, nearly all the accesses directly read from DRAM, revealing a drastic number of flips that had been previously masked by cache, resulting in a 233x increase in flips.

As for the DDR4 results, the unmodified code already had a large number of misses. The reason is that a larger region of data is initialized all at once before being hammered, which results in much of the data being evicted from the cache due to the cache's limited size. However, the additional flushes were able to reduce the number of hits by 99.94%, drastically reducing the amount of bit flips masked by the cache.