Go Go Gadget Hammer: Flipping Nested Pointers for Arbitrary Data Leakage

Youssef Tobah
University of Michigan
ytobah@umich.edu

Andrew Kwong
UNC Chapel Hill
andrew@cs.unc.edu

Ingab Kang
University of Michigan
igkang@umich.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Kang G. Shin
University of Michigan
kgshin@umich.edu

Abstract

Rowhammer is an increasingly threatening vulnerability that grants an attacker the ability to flip bits in memory without directly accessing them. Despite efforts to mitigate Rowhammer via software and defenses built directly into DRAM modules, more recent generations of DRAM are actually *more* susceptible to malicious bit-flips than their predecessors. This phenomenon has spawned numerous exploits, showing how Rowhammer acts as the basis for various vulnerabilities that target sensitive structures, such as Page Table Entries (PTEs) or opcodes, to grant control over a victim machine.

However, in this paper, we consider Rowhammer as a more *general* vulnerability, presenting a novel exploit vector for Rowhammer that targets particular *code patterns*. We show that if victim code is designed to return benign data to an unprivileged user, and uses nested pointer dereferences, Rowhammer can flip these pointers to gain arbitrary read access in the victim's address space. Furthermore, we identify gadgets present in the Linux kernel, and demonstrate an end-to-end attack that precisely flips a targeted pointer. To do so we developed a number of improved Rowhammer primitives, including kernel memory massaging, Rowhammer synchronization, and testing for kernel flips, which may be of broader interest to the Rowhammer community. Compared to prior works' leakage rate of .3 bits/s, we show that such gadgets can be used to read out kernel data at a rate of 82.6 bits/s.

By targeting code gadgets, this work expands the scope and attack surface exposed by Rowhammer. It is no longer sufficient for software defenses to selectively pad previously exploited memory structures in flip-safe memory, as any victim code that follows the pattern in question must be protected.

1 Introduction

In recent decades, the field of computer architecture has made great strides in boosting performance while reducing power and area costs. Such aggressive optimization has reaped considerable benefit for use in the common case, but has also given rise to a plethora of security vulnerabilities. Of particular interest is the advancement of DRAM, packing more information into denser areas while neglecting security risks.

Consequently, the Rowhammer bug [22] has shown how attackers can take advantage of the tightly-packed capacitors in DRAM to flip bits in memory without directly accessing them. By rapidly accessing a row of memory, an attacker can induce disturbance effects on adjacent rows, causing their capacitors to leak charge and flip their values from 1 to 0, or vice versa. This newfound ability to flip bits led to a wealth of follow-up work, demonstrating both how to flip bits on newer generation DIMMs [13, 19, 25] and how to exploit the flips to escape sandboxes [40], gain root privilege [14, 40, 43, 44], and leak secret keys [26], among other attacks [2, 10, 12, 15, 28, 31, 34, 36, 41, 42, 51].

However, a majority of these attacks focus on targeting *specific sensitive targets* [14, 14, 40, 43, 44, 51]. These prior works consider the dangers of bit flips in important structures such as PTEs [40, 43, 44, 51] and security-critical code (e.g. sudo password checks [14]). This led to various mitigation proposals that protect PTEs from bit-flips. [49, 53].

In contrast, few have considered more general targets Rowhammer can exploit to leak data. RAMBleed [26] demonstrated that attackers can hammer their own memory to leak individual bit-values of adjacent rows, and SpecHammer [42] showed how Rowhammer can be used to leak data via Spectre gadgets. However, RAMBleed only allows for leaking one bit of information per flip, and SpecHammer is restricted to leaking information while in the speculative state. Moreover, to the best of our knowledge, no work beyond these has explored how Rowhammer can be used to read out victim data without relying on flipping PTE bits or otherwise gaining root. While defenses addressing the PTE vulnerability already exist [49, 53], it is still unknown if the scope of these mitigations is sufficient. Thus, we pose the following questions:

Do there exist additional, hitherto unknown, code sequences that yield an arbitrary confidentiality break under a Rowhammer attack? If so, what would the implications be of such a vulnerability?

1.1 Our Contributions

In this paper, we present a new type of Rowhammer *gadget* that offers answers to these questions. This gadget, consisting of nested pointer dereferences, shows that by flipping victim pointers, Rowhammer can be used to gain arbitrary read access to a victim's address space. Furthermore, we found that such gadgets are quite common, and discovered 29 unique instances in the Linux kernel's filesystem handler code alone. We additionally developed an end-to-end exploit targeting one such kernel gadget to gain arbitrary read access to a victim's address space. Unlike prior work targeting bit-flips in the kernel, we target kernel-stack variables, bypassing any defenses that protect PTEs against flips [49, 53]. To the best of our knowledge, this is the first exploit using kernel stack flips without relying on deduplication or speculative execution.

```
1     func(initial_pointer){
2       pointerA = initial_pointer
3       pointerB = *pointerA
4       return_value = *pointerB
5       return return_value
6   }
```

Listing 1: GadgetHammer toy gadget.

Rowhammer Gadget. Our core contribution is the observation that a common code behavior serves as an exploitable Rowhammer gadget. A simple exemplary gadget is shown in Listing 1. In its most general form, the gadget has two key requirements: 1) a nested pointer dereference and 2) the return of data to a calling attacker. That is, a pointer (e.g. `PointerA` in Listing 1) must first be dereferenced (Line 3) to retrieve a second pointer value (`PointerB`) that is subsequently dereferenced (Line 4). Then, the data from the second dereference should be sent back to the attacker (Line 5).

```
1     func(struct* init_pnt){
2       struct* strct_pntA = init_pnt->mbr_pnt;
3       struct* strct_pntB = strct_pntA->mbr_pnt;
4       ret_val = strct_pntB->mbr_val;
5       return ret_val
6   }
```

Listing 2: GadgetHammer example kernel gadget.

If we assume the attacker has the ability to flip a bit in the first pointer, she can redirect it to *attacker-controlled data*, allowing an attacker to set an arbitrary value for the second pointer. Consequently, this pointer can be set to point to any arbitrary address in the victim's address space, leading the second pointer dereference to read out arbitrary values that get passed back to the attacker.

Gadget Presence. While the prior listing demonstrated a toy example, Listing 2 shows an example closer to the real-world gadgets present in the Linux kernel. The danger of this gadget comes from the fact that the Linux kernel relies heavily on the use of struct and function pointers. Instead of passing numerous arguments to function calls, custom structs can be designed to carry all necessary information (represented by `init_pnt` in Listing 2). While this style of programming provides the convenience of passing a single struct pointer to a function, it also leads to many nested pointer dereferences, making kernel code ripe for exploitation. Furthermore, gaining control of a struct pointer via bit-flipping gives the attacker control over every member value of the struct as well, allowing the attacker to inject her own data into numerous variables, granting control over the victim function.

1.2 Challenges

Performing an end-to-end attack against the Linux kernel has several key requirements that form the following core challenges:

- **Challenge 1:** We must precisely flip pointer values in a particular thread in the kernel's address space.
- **Challenge 2:** We must run Rowhammer in parallel with the victim gadget, flipping values in "real-time" synchronously with the victim process.
- **Challenge 3:** We must point to data that we control within the victim address space.
- **Challenge 4:** Finally, we must demonstrate an end-to-end attack on an example gadget to show the practicality of exploiting such gadgets.

Primitive 1: Flipping Kernel Stack Bits. The attack requires flipping a pointer located within a victim process. Furthermore, in the case of the Linux kernel, this means flipping a pointer residing on the kernel stack. Typically, Rowhammer attacks "massage" flip-vulnerable physical pages into the victim address space to subsequently flip victim data at will. However, in the case of the Linux kernel, prior work either flipped page table entries (PTEs) or relied on other vulnerabilities such as Spectre [2, 40, 42]. In particular, kernel stack massaging is a probabilistic process, involving allocating numerous kernel stacks across many threads. The low accuracy of this technique, along with the challenge of pin-pointing which thread contains the flip-vulnerable page, has prevented its use in real-world attacks.

To overcome this challenge, we improved upon existing kernel memory massaging techniques and devised a primitive for accurately identifying which thread contains the flip-vulnerable page in a time-efficient manner. For massaging, we observe that we can identify numerous flip-vulnerable pages and massage them all into the kernel at once, improving the chance that a flippable page will be used by the kernel stack. For identifying flippable threads, we observe that we can run, hammer, and check all victim threads simultaneously, requiring us to perform Rowhammer only once to identify which thread contains the flip-vulnerable page. By employing these primitives, we can flip target kernel bits.

Primitive 2: Real-time Flips. In order for the target bit-flip to be useful, we need the ability to flip the bit in synchronization with the victim code's execution. In the case of a kernel target, the victim is a pointer variable residing on the kernel stack. Thus, whenever the victim function is called, the target variable will first be initialized with pointer data before the pointer is used. Our bit-flip needs to occur *between initialization and pointer-use, while the victim function is running*. If the flip happens before initialization, the flipped data will simply be overwritten by initialization data. If the flip occurs after the pointer is used, the flip has no affect on the victim function. Furthermore, Rowhammer bit-flips occur in DRAM, but the pointer's initialization will cause its value to be cached. If we manage to flip the pointer's value in DRAM, but subsequent use of the pointer reads cached data, our flip will effectively be "masked" by the cache and rendered useless. Therefore, we must also ensure victim data is evicted from the cache *between pointer initialization and pointer use*.

To overcome these timing issues, we utilize kernel stalling techniques to delay kernel execution long enough to flip bits. We show such stalling can be achieved either by running parallel threads that contend for resources required by the gadget, or by using the Filesystems in USErspace (FUSE) interface to create a file system handler that can stall such shared resources indefinitely. Additionally, we show that forming an eviction set for the flip-vulnerable page allows us to efficiently evict cache lines and prevent the cache from masking flips.

Primitive 3: Pointing to Attacker Controlled Data. The goal is to read out arbitrary data from the victim address space. However, Rowhammer can realistically flip one or two bits at best, making it difficult to overwrite a pointer to point to arbitrary addresses. We therefore flip a pointer to point to data that we control in order to populate a second pointer with our data. This means we must control data that is one bit-flip away from the original, unflipped address. For a kernel attack, this means populating the kernel with attacker-controlled values.

To this end, we devised a technique utilizing pipes that allows the attacker to fill the kernel heap with arbitrary values. By sending data into a pipe and not reading it out, we can indefinitely store data in the heap. This allows us to fill the kernel with "artificial" malicious structs. Thus, when the victim struct pointer is flipped to point to our kernel data, the struct will populate its member variables with our malicious values, granting us control of the syscall's variables. From here, we can direct the syscall to read out any address in memory.

Primitive 4: End-to-end Attack. Finally, in order to demonstrate the practicality of this attack compared to prior work, we demonstrate an end-to-end attack on code identified in a Linux kernel-syscall. We demonstrate a maximum leakage rate of 82.6 bits/s, improving the 0.3 bit/s leakage reported in prior confidentiality-based Rowhammer work [26, 42].

Contributions. We make the following contributions:

- Identifying a new class of code patterns that can be exploited as a Rowhammer gadget. (Section 4.1)
- Improving memory massaging techniques for flipping kernel bits (Section 4.2).
- A novel technique for testing for kernel flips (Section 4.3).
- New synchronization techniques (Section 4.3).
- Performing an end-to-end attack on a Linux Kernel syscall, demonstrating a leakage rate of 82.6 bits/s (Section 5).

1.3 Disclosures

We sent a copy of our paper to the Linux kernel security team on April 26, 2023 and ran all experiments on our machines.

2 Background

2.1 Gadget-based Attacks

Orthogonal to Rowhammer is a class of attacks that takes advantage of exploitable patterns in victim code. These exploitable code snippets are referred to as *gadgets*. The central idea of such attacks is to identify gadgets in victim code and use them to lead the victim process to do malicious work for the attacker. For example, return oriented programming (ROP) attacks work by redirecting control flow to sequences of instructions that end in the `return` instruction. Attackers scan memory for these "ROP *gadgets*" (i.e., series of instructions that can be chained together for malicious purposes), and overwrite return addresses to point to them.

A more recent class of gadget-based attacks has spawned from Spectre [42] and subsequent work in the speculative domain [4]. These attacks are based on finding victim gadgets that can trigger states of speculative execution. With Spectre, attackers train branch predictors to predict a particular execution path, then force the opposite execution path to occur. This results in a misprediction, meaning any speculatively executed code will eventually be undone. While in the speculative state, however, attackers can use gadgets to access out of bounds data [1, 24, 39] or perform arbitrary code execution [29], and retrieve speculatively read data via a covert-channel. Spectre is difficult to mitigate [4] as it not only leverages performance-critical processor features (branch prediction), but can target any victim containing a gadget.

2.2 Pipes

Pipes are channels in the kernel used for passing data between processes. The convenience of pipes is that transmitters and receivers do not need to synchronize. Transmitters can send data into a pipe, and data will be stored in the kernel indefinitely until it is read out of the same pipe.

2.3 Caching

Cache Interference. One important consideration with Rowhammer is cache interference. Rowhammer requires re-

peatedly accessing rows of DRAM to induce flips in adjacent DRAM rows. Therefore, between each aggressor access, the attacker must flush these addresses from the cache. This ensures each access activates a DRAM row and does not simply interact with the cache instead. Additionally, the attacker must ensure the target victim address has been flushed from the cache as well before hammering begins. Otherwise, even if a bit is successfully flipped in DRAM, the victim may subsequently read data from the cache, "masking" the bit-flip.

Cache Eviction. If the attacker has access to a `clflush` instruction and target addresses, she can directly flush addresses from the cache to prevent interference. However, in scenarios where there is no `clflush` [15] the attacker may need to rely on cache eviction. This is a technique in which an attacker identifies a group of addresses (called the *eviction set*) that all belong to the same cache set as a flush target. Since the cache can only hold a limited number of entries from a particular set, accessing all of the addresses in the eviction set causes the flush target to be evicted from the cache, providing a substitute to direct flushing. Prior work has demonstrated techniques for efficiently generating minimal eviction sets [45].

Cache Side-channels. Caches are also useful as a source of side-channel leakage. Accessing cached data is faster than accessing data from DRAM, meaning timing memory accesses can reveal information about victim access patterns and physical memory. For example, the PRIME+PROBE [33] technique begins by filling (or "priming") the cache with attacker controlled addresses. Then, the victim is left to run. Next, the attacker attempts to access the same set of addresses that initially filled the cache, timing (or "probing") how long each access takes. If the accesses are all fast, the attacker knows the victim never accessed memory occupying the same cache set. If any accesses are slow, however, another process must have accessed a conflicting address, revealing information about victim memory accesses.

2.4 Rowhammer

Exploits. Rowhammer [22] demonstrated how attackers can flip bits in memory without accessing them, spawning a plethora of new attacks taking advantage of these bit-flips [2, 7, 8, 12, 14, 15, 26, 28, 36, 40–43, 52]. Most notably, the first exploit [40] showed how Rowhammer can flip values in page table entries (PTEs), which could in turn be used to gain root access. This was followed up by numerous attacks focused on achieving the same goal of flipping PTEs to gain root under new threat models, such as attacks targeting mobile devices [43, 44], and attacks that flipped PTEs through the browser [15] among others [31].

Defenses. To counter the threat posed by Rowhammer, various defenses have been developed to either prevent bit-flips or protect sensitive data. The only widely deployed defense in the former category is Target Row Refresh (TRR), which can be easily bypassed with more advanced hammering techniques [13, 19, 25]. Next-generation DDR5 DIMMs use a new technique called refresh management (RFM), but even this has shown to be inadequate [30]. Software defenses have sought to protect sensitive structures, by, for example, placing buffers between user space and kernel space [3] or placing PTEs in flip-safe memory [49].

DRAM Organization. At its core, Rowhammer takes advantage of DRAM design and its reliance on capacitors. DRAM organizes memory into channels, ranks, banks, rows, and cells. The lowest level of DRAM is the cell, which stores a single bit of information using a capacitor. A fully charged capacitor represents a 1 and discharged capacitor a 0 (or vice versa).

Upon accessing a DRAM address, an entire row of cells is activated, meaning the charge from each cell in the row is pulled into the corresponding row buffer, where the bit values can be passed to the processor. Once the memory access is complete, the charges are restored to their original cells.

Flipping Bits. Rowhammer made the observation that capacitors are being packed more and more tightly together in newer generations of DIMMs, and can thus have *disturbance effects* on adjacent capacitors. In particular, accessing a row of memory and temporarily discharging and recharging the corresponding cells can slightly *accelerate the leakage rate* of adjacent capacitors. Repeatedly accessing a row of memory can thus pull an adjacent capacitor's charge below their threshold value, flipping the capacitor's value from 1 to 0 (or vice versa), before it has the chance to be refreshed. Rowhammer therefore enables bit-flips in addresses attackers should not be able to modify by repeatedly accessing (or "hammering") their own accessible rows.

DDR4. Rowhammer was first demonstrated on DDR3 DIMMs. In response to the attack, a defense called Target Row Refresh (TRR) was added to DDR4 [20]. TRR tracks row activations, and if the number of activations on a particular row crosses a set threshold, adjacent rows are refreshed immediately, preventing bit-flips in these targeted victim rows.

However, TRRespass [13] demonstrated that Rowhammer accesses can be scattered to multiple addresses across a bank, preventing the TRR counter from properly tracking activations while still inducing leakage in the row at the center of all these accesses; a technique called multi-sided Rowhammer. Thus, TRRespass demonstrated that even DDR4 is vulnerable to Rowhammer. This was followed up by numerous works [8, 19, 25] that all demonstrated new techniques for flipping bits on DDR4, revealing that these new DIMMs were even more vulnerable to bit-flips underneath TRR.

2.5 Memory Massaging

With Rowhammer, an attacker can trigger bit-flips on a target physical page. However, for a flip to be exploitable, the attacker must force a victim process to use the flip-vulnerable (or "flippy") page. This is typically done via a primitive referred to as *memory massaging*, which manipulates (or "mas-

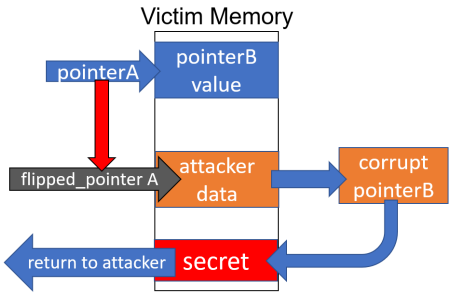

Figure 1: Flipping a pointer to return secrets to an attacker.

sages") a physical memory allocator into a state that is likely to serve its next allocation request using the flip-vulnerable page [6, 26, 36, 42, 43]. The attacker can then force the victim to use a flippy page and flip victim variables at will.

For user-space attacks, the attacker can simply deallocate the flip-vulnerable page, placing it in a page frame cache (PFC) [6]. Subsequent victim allocations on the same processor core will pull from the flip vulnerable page from the PFC, allocating its variables on a flip-vulnerable page. Kernel attacks are more complex since kernel and user-space memory use different pools of physical pages. The general idea is to drain kernel memory to the point where the kernel is forced to pull from the pool of free user-space pages. For a PTE attack, the attacker can force PTE allocations by mapping virtual addresses to physical memory [40]. For an attack on kernel stack variables, kernel stacks can be allocated by creating new threads, which each allocate their own kernel stack [42].

3 Threat Model

We assume the attacker can use unprivileged software on the victim machine. We also assume the victim machine uses an uncompromised operating system. Additionally, we assume the victim machine uses a DIMM vulnerable to Rowhammer.

4 GadgetHammer

4.1 Attack Overview

Example Gadget. The central idea of the GadgetHammer attack is to target gadgets which grant arbitrary read access upon flipping a victim pointer. An example gadget is shown in Listing 1. The requirements for a gadget are that a pointer (`pointerA`, Line 2) is dereferenced to obtain a second pointer value (`pointerB`, Line 2), and the second pointer is subsequently dereferenced (Line 3). The value of this second dereference should be returned to the user calling the gadget.

Exploiting the Gadget. As shown in Figure 1 the gadget can be exploited if we assume the attacker has the capability of flipping `pointerA` (Line2, Listing 1) to lead `pointerA` to point to attacker controlled data. Having `pointerA` point to an address we control effectively gives us control the value of `pointerB` (due to the dereference in Line 3). With full control over the value of `pointerB`, we can effectively read any data from the victim's address space into `return_value` (Line 4) which gets returned to the attacker (Line 5).

Technical Challenges. Exploiting the gadget as described requires the use of several key primitives. We must first identify DRAM addresses containing bits that can be flipped as needed for the attack. Then, we need to force the victim gadget to use a physical page corresponding to this flip-vulnerable address. Additionally, we must control data at an address that is one bit-flip off from the address the victim would normally point to. Lastly, we must flip the victim while the gadget is running and efficiently flush victim data from the cache to ensure the victim directly reads the flipped data from DRAM.

The following sections explain how we overcome each of these challenges, beginning with the "offline stage" where we identify flip-vulnerable addresses and force the victim to use them, followed by the "online stages" where we confirm the presence of our bit-flip and finally use the gadget to leak data.

4.2 Offline Stage

Memory Templating. The first step of any Rowhammer attack is to find which physical addresses in victim memory are subject to "useful" bit-flips (i.e., bit-flips at the same page offset as our target victim), a step commonly referred to as "memory templating". We follow the same steps as prior work [13, 26, 40], allocating transparent huge pages, and hammering groups of addresses until finding useful flips, relying on TRRespass [46] to induce flips in DDR4.

Massaging Physical Memory We now control physical pages containing useful bit-flips, as well as the corresponding aggressor rows that we can use to induce these flips. However, to run the exploit, we need these flips to occur in target victim values, not pages that we control. We must therefore "massage memory" to force the victim into using the flippy page.

In the case of targeting a kernel variable, we must massage our physical page onto the kernel stack. To this end, we can use a kernel memory massaging primitive from [42]. We drain kernel memory such that it is forced to steal user space pages for subsequent allocations. We can then free the flip-vulnerable page and spawn numerous threads, which each allocate memory for their own kernel stack, relying on one of these threads to allocate the flippy page for its stack.

Improving Memory Massaging Probability. A weakness of memory massaging techniques is their probabilistic nature. A low success rate technique requires the attacker to repeat the time-consuming step of finding bit-flips and checking if they landed in the kernel (see Section 4.3). In order to better ensure that we can land a flip in the kernel once we find it, we template to find *many useful flips*, instead of just a single flip, before attempting massaging. As shown in Figure 2, finding numerous flippy pages, freeing them all into the page allocator,

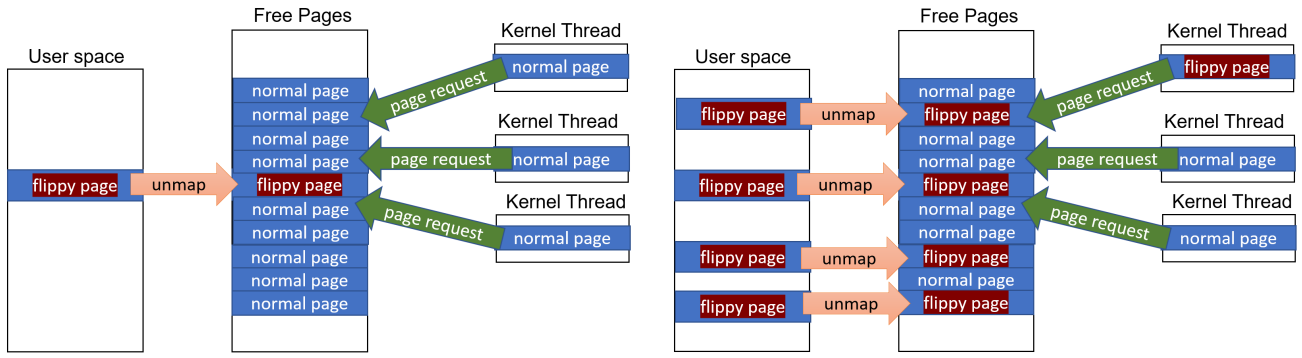

Figure 2: : Improving memory massage probability. The left side shows the lower chance that a kernel stack will allocate a flip-vulnerable page if there is only one flip-vulnerable page in memory. The right side shows how these odds can be improved if more flip-vulnerable pages are freed before forcing kernel stack allocations.

and then forcing kernel stack allocations, can greatly increase the probability that a flippy page can land in the kernel.

4.3 Online Stage: Testing Flips

At this point we control numerous threads and hope at least one thread has allocated this flippy page for its kernel stack. In the steps that follow, we must call the victim syscall and trigger bit-flips *while the syscall is running* in order to first identify which thread contains the flippy page and then subsequently begin leaking data from the kernel.

Checking for kernel flips. For the memory massage step, we allocated many threads to drain any remaining kernel memory and then steal pages from userspace until our recently-freed flippy page was used for a kernel stack allocation. To continue the attack, we must first identify which thread contains the flippy page. At this stage, we cannot simply attempt the attack to verify whether a flip landed, since additional uncertainties remain in upcoming stages of the attack. Thus, to reduce one unknown at a time, we need a method to test for flips that is isolated from the rest of the attack.

Flip-check Syscall. In order to check which thread holds our flippy page, we rely on a second syscall, separate from our target gadget, that returns a binary result dependent on the presence of the flip. This syscall served a similar purpose to the Spectre-based oracle used in prior work [25], however, it does not rely on the use of a speculation-based exploit. Such a syscall, which we will call the *tester syscall* should meet two requirements. The first, is that is must contain a local variable at the same page offset as the target gadget syscall. Second, the syscall returns a value that is dependent on said local variable. We find that meeting both requirements is feasible.

Requirement 1: Variable Offset. In the case of the first requirement (matching page offset), the page offset depends entirely on the local variable's position on the stack. This, in turn, is a direct result of the total number of *local variables allocated* and *nested function calls made* over the course of the entire tester syscall.

See Figure 3 for a simplified example. If we call our tester syscall from userspace, and the highest-level function of this syscall, `FunctionA` allocates three local variables, those variables might occupy page offsets 0x100 through 0x110. If `FunctionA` calls another function, `FunctionB`, which allocates three additional variables, those variables may reside further down the stack at offsets 0x120 through 0x130. To find a suitable target for a tester syscall, we thus only need to find a local variable that resides a similar number of function calls deep among the myriad of syscall options in the kernel.

Furthermore, many variants of similar syscalls exist (e.g. `write` and `writev` or `setxattr` and `getxattr`) which will have paths similar to each other, with slight differences in number of function calls made and local variables allocated. This effectively allows for "sliding" the position of target local variables up and down the stack until finding a suitable target.

Requirement 2: Flip Dependent Result. We can meet the second requirement thanks to the good programming practices followed by the kernel. Syscalls commonly have many checks throughout their functions to catch errors, preventing bad values from propagating through kernel code and causing crashes. Thus, if a local variable is flipped to an incorrect value, the syscall may detect something is wrong and gracefully return an error code as opposed to crashing, creating a flip-dependant result. Otherwise, if the syscall behaves properly, we can move on to check the next thread. As we will see in Section 5.1, we find `filesystem` syscalls work quite well as tester syscalls.

Working Around Cache Flushes. Challenges still remain in flipping syscall bits. The first is the issue of cache flushing. Upon calling our victim gadget, our target flip variable will first be initialized before we can induce our flip. This will likely cause the victim variable to be cached, thus "masking" any potential flips with cached data.

Therefore, we take advantage of cache eviction techniques [45]. Instead of directly flushing our victim from the cache, we will fill the same cache set with arbitrary data, forcibly

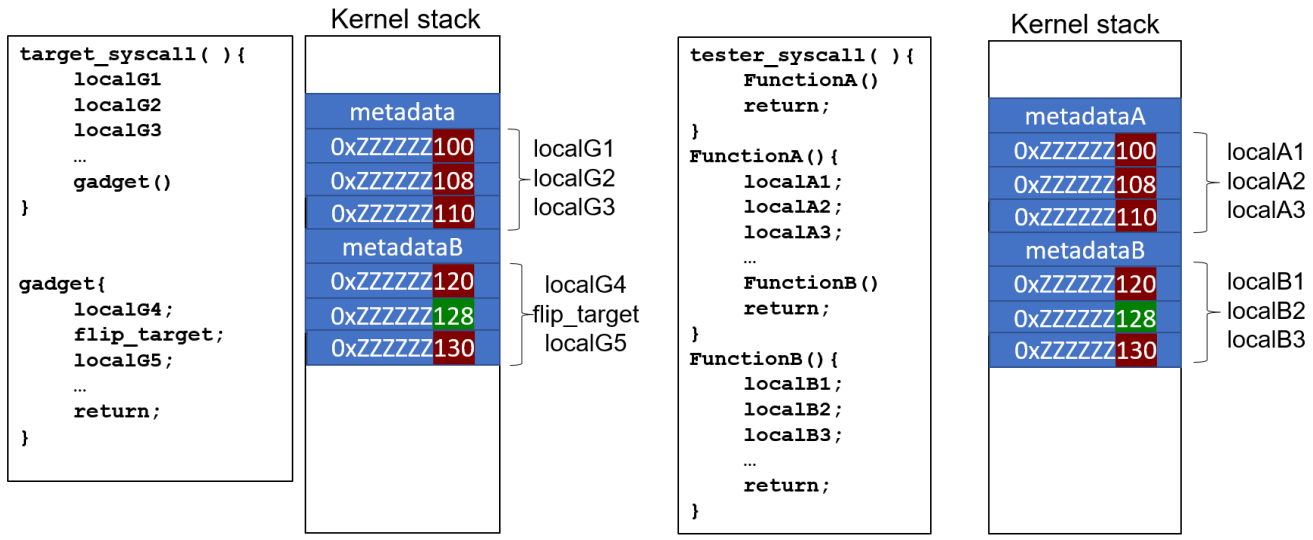

Figure 3: Function calls placing data on the stack. The left side shows the target gadget's stack, and that the variable we seek to flip happens to reside at page offset 0x128. The right side shows the tester gadget, which stores variables at a similar stack depth, also storing a local variable at page offset 0x128.

evicting our target victim, exposing it Rowhammer. We observe that with physically-indexed caches, we can easily form eviction sets for our victim during the templating phase, since during that phase, we control the victim physical page and know the lower physical address bits. When we release the victim page and force the syscall to use it, we can use the same eviction set made of userspace pages, giving us the ability to flush kernel data from the cache at will.

Stalling Kernel Execution. The last remaining challenge for our tester syscall is flipping the victim variable in parallel with syscall execution. We must wait for the victim variable to be first initialized with its pointer value, and then flip the pointer before it gets dereferenced. If the syscall is left to run normally, this window will likely be too tight to induce a bit-flip, especially in the case of DDR4 where we require multi-sided hammering. Furthermore, since our victim runs in the kernel, we have no direct way of knowing which line of code the victim is executing at any given time and cannot precisely synchronize our hammering to begin when needed.

FUSE. Prior work relied on the `userfaultfd` syscall, which can be used to indefinitely stall the kernel [9, 42]. However, this syscall has recently been restricted to superuser privilege. Alternatively, Linux's filesystem in userspace (FUSE), can be used to achieve a similar effect [17]. With FUSE, we can map files to an attacker-defined filesystem, force the victim syscall to interact with such files, and define the filesystem handlers to stall indefinitely. This consequently stalls the victim syscall, giving us room to flip bits.

Additionally, even if the target syscall does not directly trigger any of our handlers, so long as the syscall requests a lock, semaphore, or mutex, we can call other syscalls that use the same resources, wait for them to hold the lock, and then stall them indefinitely. Without access to the lock, our target syscall will not be able to run until our handler completes. Moreover, since the kernel forbids mixing declarations and code, and since optimal code holds locks for as briefly as possible, lock requests tend to be conveniently located between victim variable initialization and use.

Thread Contention. For distributions of Linux that may not have FUSE installed, we devise an alternate technique that can delay the tester syscall. We simply allocate numerous threads that each request the same lock as our tester syscall and have the syscall run in parallel with all threads. With numerous threads contending for the same resource, the tester syscall's execution becomes delayed, giving us enough time to hammer and flip bits. This technique is less reliable than FUSE, as the tester syscall may get the resource ahead of its contenders, before we have time to flip the victim. We show that despite this disadvantage, this technique can work against the tester syscall in practice (see Section 5.1).

Simultaneous Thread Hammering. We can now hammer a syscall to identify which thread contains the flip-vulnerable page. However, consider that a single "round" of Rowhammering typically requires tens to hundreds of thousands of accesses to flip a bit. Additionally, its helpful to perform multiple repeated rounds of hammering to ensure bit-flips. This means our hammering operations will require time on the order of milliseconds. Furthermore, as explained in Section 4.2, we have identified multiple flip-vulnerable pages, each with their own aggressor set, that could each have been successfully massaged into our target victim. This means we need to hammer every aggressor set for each test, and we need to repeat this process for every thread that potentially contains a flip-vulnerable page. Multiplying the time needed for a sin-

gle hammer procedure by the number of aggressor sets and number of threads results in a process that takes several hours.

To reduce the time required, we instead hammer all threads in parallel. In the case of using FUSE for stalling, we create a single file shared by all threads, and run every thread until it is forced to stall by our file system handler. This causes every thread to load its local variable data in DRAM and keep it there until the filesystem allows the threads to continue. Now, running our eviction sets and hammering our aggressors for one round will simultaneously hammer all threads, flipping any variables that use a flip-vulnerable page. We therefore only need to perform our hammering rounds once, reducing the required time from hours to seconds.

Reducing Risk of Bad Flips. Simultaneously massaging numerous flippy pages into kernel memory introduces a new risk. We may potentially cause an important kernel structure (other than our target) to use a flip-vulnerable page and inadvertently flip a critical value. In the worst case, this can cause a kernel panic and crash the victim system, which is undesired since our goal is to target confidentiality, not availability.

To help alleviate the risk of a crash, we can take advantage of a PRIME+PROBE side-channel. Note that we have already formed an eviction set of addresses that occupy the same cache set as our flip-vulnerable victim. Besides directly using these addresses for eviction, we can also use them for PRIME+PROBE testing. Before we perform any hammering, we first access every address in our eviction set to ensure the cache is occupied by our data. We then repeatedly access our eviction set, timing how long each access takes, and run a candidate victim thread in parallel. If the victim thread uses the flip-vulnerable victim page, it will affect the time required to access our eviction set addresses, revealing which threads may use the massaged flip-vulnerable pages.

PRIME+PROBE tends to be a noisy measurement, and repeating measurements enough times to eliminate noise would take impractically long. We therefore use the noisy measurement with conservative thresholds to filter which threads should be hammer tested. The hammer testing then shows which thread uses the flippy page with 100% accuracy.

4.4 Online Stage: Leaking Data

The remaining step is to flip the target syscall pointer, leading it to point to data we control, granting arbitrary kernel reads.

Targetting the Kernel Heap. The first challenge is how to point this victim to our own controlled data. An obvious choice might be to flip a high order bit and force a point to userspace, where we could control a large region of memory. However, recent processors come with supervisor mode access prevention (SMAP). This prevents the kernel from accessing any data in userspace. We must therefore inject our own data directly into the Linux kernel. Even though our target pointer resides on the stack, the pointer value may point either to kernel stack *or* heap data. Since we can realistically

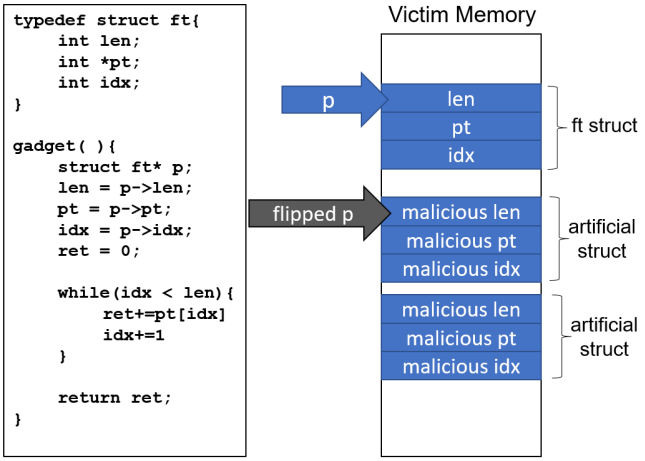

Figure 4: Filling victim memory with artificial structs that contain attacker data. Flipping a struct pointer to point to our structs gives us control of the true struct's member variables.

flip only a single bit, it is most practical to attempt changing this pointer to point to an alternate address also within the same memory region. Since the gadget chosen for our example end-to-end attack uses a heap pointer (Section 5), we focus on injecting our data into the kernel heap, noting that injecting data into the stack is possible as well [9, 42].

Spraying the Heap. The first question is how to inject heap data. Prior works have explored different techniques for heap spraying [50], however, here we have an advantage in that we have none of the usual constraints characteristic of heap spray exploits, such as reusing dangling pointers.

We find that pipes act as a convenient mechanism for stuffing the kernel with data that we control, as previously demonstrated on non-Linux operating systems [18]. Pipes are designed to act as files that users can read and write from, but instead of storing values in an actual file, they are stored in the kernel heap. Pipes can store 16 pages of data, and multiple pipes can be allocated up to the operating system's hard limit of about 1 million files. Furthermore, we can swap data in and out of the pipe at any time by simply writing to our pipe of choice, which will prove useful for the final step of the attack.

Spray Contents. We need to fill the heap with values such that if the victim pointer points to our heap data, we can lead subsequent instructions in the syscall to point to secret data. We thus consider the structure of the victim struct and form *artificial structs* to populate the kernel heap. When the victim is flipped to point to an artificial struct, every subsequent dereference will pull from our artificial struct, allowing us to fill the victim syscall with our own values and take over the execution path. As an example, suppose the target gadget behaves as show in Figure 4. The key to this gadget is the pointer `p` pointing to a struct of type `ft`. We can flip this pointer to point to a location of memory we control, containing an artificial `ft` struct. From here, malicious, `len`, `pt`, and

idx values can be set to control the number of addresses the syscall will read , and what each of those addresses will be.

Positioning Heap Contents One challenge is positioning our data so that its address is one flip away from the victim pointer. Like any address, the victim address consists of two components, the higher order bits signifying the page, and the lower order bits signifying the page offset. Our injected heap data will reside on a separate page from the victim, meaning our bit-flip must occur in the *page* field of the address, and the *page offset* of our injected data must match that of the victim variable's original pointer.

Since this victim value is pointing to the heap, the page and offset values are randomized, as the heap values are allocated at random. To counter the random nature of the page value, we spray the heap with as many artificial structs as possible, maximizing the odds that we control a page one bit-flip away.

The page offset value is also random, but, since we target struct pointers, the lower order bits of the page offset are restricted based on the size of the struct being pointed to, as the struct must be aligned in memory. For example, we find that pointer value in our target in Section 5.1 is always a multiple of 0x40. Therefore, within the pages we control in the heap, we can position our artificial struct at every multiple of this value over the space of the entire page. This guarantees that if we point to the *page* containing our data, the victim struct will point to an *offset* containing our data as well.

Checking Heap Spray Using Flip. At this point, we have filled the heap with data we control. However, we need to be sure we control data that is one flip away from our victim (See Figure 5). Therefore, we must first set our attacker-controlled heap values such that they point to fixed *control* data in the kernel, before we attempt to leak secrets. This way, we can spray the heap, flip our bit, and check if the gadget leaks out control data to confirm a successful heap spray. For the control data, we use tables used for Linux's AES encryption libraries, as they contain 16KB of contiguous, constant, unique data. If we're able to successfully read out bytes from this table, we know our heap data landed in a useful position. Otherwise, we simply deallocate our heap data and attempt a reallocation. We can point to control data in presence of KASLR by relying on existing techniques to derandomize kernel addresses [38].

It is worth noting here that the repeated attempts at landing heap data at a useful address is precisely why the tester syscall is needed. Without the tester syscall, we would be unsure whether we are unable to flip bits due to not controlling a flippy page in the kernel or due to an unsuccessful heap-spray attempt, and would have to run numerous heap-spray attempts on *each thread* to be confident the issue is due to the lack of a flippy bit. Fortunately, the tester syscall guarantees the presence of a flip in a given thread, allowing us to repeatedly heap-spray and hammer a single thread, while being fully confident in the presence of a bit-flip.

Pointing to Leak Target. At last, we are ready to arbitrarily read data from the kernel. For this final step, we simply write to the pipes containing our heap data, filling the pipes with pointers to whichever address we wish to leak. We then run the syscall again, flip the gadget pointer, and read out data from the chosen address. For example, we can point to the physmap, and read all physical memory on the victim machine.

Results. The leakage rate depends on the data returned by the victim syscall. Each time we hammer and call the syscall, we get one return value with leaked data. Since we stall for about 0.5 seconds to give time for hammering, the maximum theoretical leakage rate is 128 bits/s (when the return value is 64 bits). See Section 5.2 for an empirical evaluation on a chosen example gadget present in the kernel.

5 Attacking the Linux Kernel

As a proof-of-concept of the risk posed by GadgetHammer we demonstrate an end to end attack on the Linux kernel, successfully mounting our attack on an existing syscall.

5.1 Target Victims

Target Gadget. We identify a suitable GadgetHammer gadget in the ioctl syscall. In particular, within the pipe_ioctl function located in fs/pipe.c. A simplified version of this gadget is shown in Listing 3. The usual use case of this function is for the user to pass in a file descriptor referring to a pipe (filp on Line 1), and receive the number of unread bytes contained in that pipe. The syscall begins by extracting the private_data pointer from filp and passing the pointer value to the local pipe variable (Line 2). The pipe variable now points to a struct containing all corresponding metadata.

```
1    pipe_ioctl(struct file *filp, unsigned int
   cmd, unsigned long arg){
2        struct pipe_inode_info *pipe = filp->
   private_data;
3        int count, head, tail, mask;
4        ...
5        __pipe_lock(pipe);
6        count = 0;
7        head = pipe->head;
8        tail = pipe->tail;
9        mask = pipe->ring_size-1;
10       while(tail != head){
11         count += pipe->bufs[tail & mask].len;
12         tail++;
13       }
14       __pipe_unlock(pipe);
15       return put_user(count, (int _user *)arg);
16   }
```

Listing 3: GadgetHammer example kernel gadget

The syscall then locks a semaphore (Line 5) to ensure the pipe will not be modified while values are read out. On Lines 7-9 meta data is extracted giving the starting address (head), ending address (tail) and maximum size of the pipe data

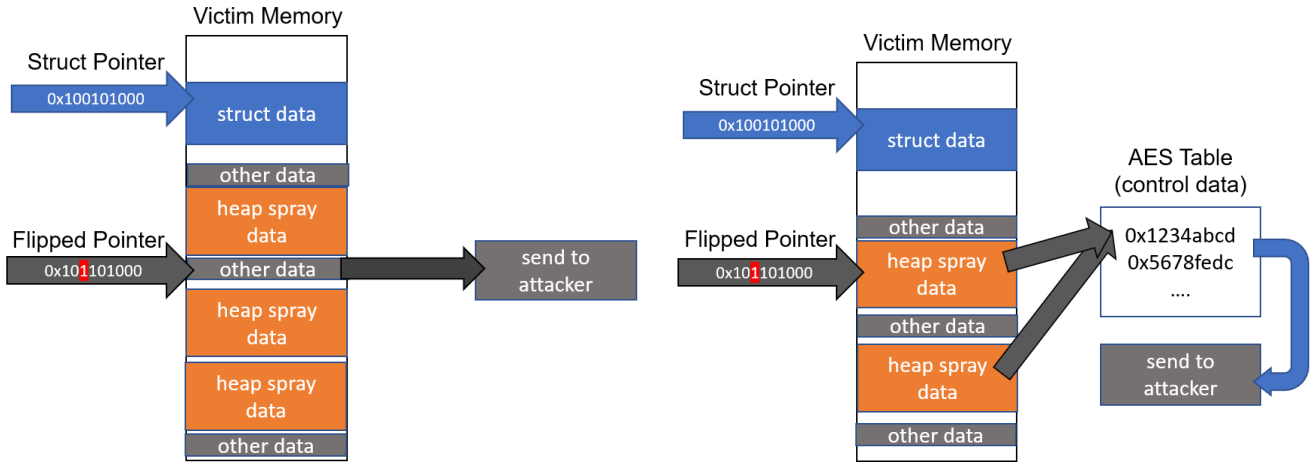

Figure 5: Confirming heap spray results. The left side shows that if our heap sprayed data does not land at an address one bit-flip away from the victim, the gadget will return junk values. We repeat the heap spray until we can read out our "control" data from the kernel, as shown on the right side.

(`ring_size`). This data is used to iterate through the pipe's buffers, starting from the tail and ending at the head (Lines 10 - 12), using `count` to keep a running sum of the number of bytes in each buffer. Finaly, the pipe semaphore is unlocked (Line 14) and the number of bytes is passed back (as `count`) to the calling function via `put_user` (Line 15).

Flipping the Target Gadget. The target victim for bit-flipping is the `pipe` pointer variable declared and initialized on Line 2. Note that this gadget follows the pattern of heavily relying on this pointer for passing arguments, as the struct pointer is repeatedly dereferenced such that its members populate the function's local variables.

The goal, then, is to flip the value of `pipe` such that it points to our data. This data is then subsequently used as the base address for the `bufs` array (Line 11). Therefore, we can set this base address of `bufs` to point to any kernel address and read out its contents as an array access. From here, the `tail` variable is masked by a `mask` variable, and used as an index into our base array address to read out leaked data. This leaked data then gets added to the `count` variable (Line 11). After this, the `tail` index variable is incremented and additional data will be read out from the next entry in `bufs` and added to `count`. This process repeats as many times as specified by the `loop` and `head` variables, due to the loop in Line 10.

In a hypothetical scenario of controlling only the `bufs` array, the target data becomes partially scrambled by the addition of subsequent values. Even then, it would be possible to leak sums of secret data at various starting addresses and then filter out the noisy additions from the desired target data. However, here lies the strength of controlling the `pipe` struct pointer itself, as we can control all the variables of this function and consequently control the number of loop iterations and the array index. By setting our injected data to populate `tail` with `0` and `head` with `1`, we can ensure the victim will loop only once, writing the desired leak target to `count` and then immediately returning this value to the calling attacker.

Tester Gadget. As explained in Section 4.3, it is useful to have a *tester gadget* that we can use to confidently check for bit-flips in the kernel. We identify the `removexattr` function, located in the `fs/xattr.c` file as such a gadget, shown in Listing 4. We can reach this gadget via the `fremovexattr` syscall. This syscall is part of the `xattr` family of syscalls, which are used to interface with "extended attributes," essentially adding additional properties to files for security and file management. As the name suggests, `removexattr` removes an attribute from a specified file. To specify which attribute to remove, users pass the name of the attribute (as a string) as one of the syscalls arguments.

The function starts by copying an attribute name passed by the user via `name` into the local kernel variable `kname` (Line 4). Lines 5-8 simply check if a proper attribute name with an appropriate length was used before calling `vfs_removeattr` to remove the specified attribute (Line 9).

```
1    removexattr(....,const char __user *name){
2      int error;
3      char kname[XATTR_NAME_MAX + 1];
4      error = strncpy_from_user(kname, name,
   sizeof(kname));
5      if(error == 0 || error == sizeof(kname))
6      error = -ERANGE;
7      if(error < 0)
8      return error;
9      return vfs_removexattr(..., kname);
10   }
```

Listing 4: GadgetHammer example tester gadget

Flipping Tester Gadget. The key to this syscall is the `kname` variable shown at Line 3. This is a 256 byte long array

that stores the name of the attribute, passed in as a string from userspace. To use the syscall as a tester gadget, we first open any arbitrary file and add an attribute with a name consisting of 256 characters. We then call `fremovexattr` to remove the attribute we just added, meaning `kname` (Line 3) will be occupied by 256 characters of our choice. Normally, `removexattr` would then continue execution and remove our attribute via `vfs_removexattr` (Line 9).

However, here we use Rowhammer to flip one of the characters stored in `kname`. If we successfully flip one of these characters, `kname` will no longer have a string that properly specifies an attribute, meaning `vfs_removexattr` will be unable to remove our attribute and will return an error instead (Line 9). Thus, by calling `fremovexattr`, hammering, and checking if our attribute is still on the specified file, (or simply checking if an error was returned) we can test for flips in the kernel. Furthermore, the 256 addresses the `kname` array occupies in the kernel stack overlaps with the address of the `pipe` variable, our flip target in our main leakage gadget (Listing 3, Line 2). This allows us to use `fremovexattr` (via `kname`) to test for bit-flips at the required offset in the kernel stack.

Simultaneous Hammering In order to avoid the impractically long time required to hammer each thread one by one, we run all threads simultaneously until they have all loaded their syscall stack variables into memory and are stalled on our filesystem. This allows us to perform our hammering rounds once and simultaneously test all threads for flips. However, this also requires the `fremovexattr` call in every thread to use the same file. Furthermore, we test each thread by attempting to remove an attribute, where unsuccessful removal means no flip occurred. Since all threads use the same file, each thread needs to add a unique attribute, otherwise removal of the attributes by any thread would cause subequent threads to have unsuccessful removals (regardless of flips) since the file no longer holds the attribute.

However, a file can store only a limited number of attributes, preventing us from adding thousands of unique attributes that would be required for the thousands of threads. Therefore, we organize threads into groups, with each group sharing a single, unique file. Since each group contains a limited number of threads, each thread is free to add a unique attribute to its group's file without hitting the limit. We can then stall all of these "group files" simultaneously and test all threads, each with a unique attribute. Any thread that returns an error contains a flip-vulnerable page.

Stalling. We use FUSE to stall execution for a guaranteed 0.5 seconds to allow a comfortable hammering window. This is accomplished by first using FUSE to create our own filesystem, including the handlers for any files mapped to our system. We define these handlers such that any basic interactions with corresponding files (e.g., writing, reading) will stall indefinitely. Next, we map a pipe to our filesystem and attempt to write to this pipe via `pipe_write`. Writing to a pipe requires holding a mutex via `pipe_lock`, and since this write interaction is defined by our filesystem, the write to the pipe will stall and the mutex will be held indefinitely. Thus, any subsequent calls to `pipe_ioctl` will stall indefinitely upon hitting `pipe_lock` (Listing 4, line 5), as the mutex is held by our `pipe_write` call, thereby allowing us to create a stalling window for hammering.

While FUSE provides the advantage of creating an indefinite stalling window, the FUSE package is not installed by default on all distributions of Linux. Thus, we have also successfully flipped tester gadget bits without FUSE by relying only on thread contention. We spawn 1000 threads, each simultaneously attempting to write to a pipe, causing them to contend over the `pipe_lock` mutex. This leads to a delay in `pipe_ioctl` as well, as it must contend with 1000 other threads for the mutex before the syscall can complete. Since we do not control the order in which mutex requests are served, the delay can vary from completely negligible to above the required 0.5 second needed for hammering, depending entirely on the order in which threads are given the mutex. We observe that repeating this approach 100 times per gadget-flip-attempt is sufficient to guarantee at least one attempt will encounter a stalling window large enough to allow for bit-flipping.

Thus, while FUSE grants a guaranteed stalling window, and guaranteed bit-flips within a single hammering round, thread contention does not rely on a potentially unavailable package, but requires more attempts per bit-flip.

Deallocating the Tester Gadget. Lastly, now that we have successfully flipped a bit in our tester gadget, we need to use the same flip in our target gadget. This can be achieved simply by calling the target gadget within the same thread as the tester gadget. That is, when we return from the tester gadget syscall, all the tester gadget variables will be popped off the kernel stack, making room for the target gadget variables.

5.2 Attack Execution

Experimental Setup. Having identified our targets, we conducted the attack following the steps laid out in Section 4. We ran our evaluation on an Intel i9-9900K processor, using a Samsung M378A1K43BB1-CPB DDR4 DIMM, running an unmodified Linux kernel version 5.16.2 (Ubuntu 18.05.5 LTS) as well as an Intel i7-7700 CPU, using a Samsung M378A1K43BB2-CRC DDR4 DIMM running an unmodified Linux kernel version 5.16.2 (Ubuntu 20.04.6 LTS). To better understand the performance and success rate of each stage of the attack, we ran each stage in isolation repeatedly for 24 to 72 hours before running all stages together in a complete end-to-end attack.

Memory Templating We began by searching for 5 bit-flips during the templating stage, as well as an evictions set for each flip. We restricted our search only to flips at the page offset of our flip-target, `pipe` (`0xecb` through `0xec9`). Within a 48 hour period of repeated hammering, we observed 7682 bit-flips within pages at these offsets. Of these flips, 370 were reliably

reproducible. On average, it took 5.8 minutes to obtain 1 reproducible flip. Searching for flips at 5 different addresses strikes a reasonable balance between the time needed for bit-flip search and success rate of memory massaging.

Memory Massaging We unmapped our 5 reproducible flippy pages and allocated 8192 threads, hoping at least 1 flippy page would be successfully massaged into the kernel stack. Additionally, to reduce the risk of hammering a bit in a dangerous position, we used our PRIME+PROBE side-channel to check if our victim threads contained one of our flippy pages in the required position on the stack. Note that the stack allocated multiple pages, but we only consider an attempt successful if the flippy page is allocated to the stack page containing our flip-target. We observed that an unmapped flippy page landed at the required position in the kernel stack 3 times out of 37 attempts, yielding an accuracy of 8.1%. While the accuracy is quite low, massaging attempts can be repeated until the page lands, as done in prior work [40].

Additionally, to better understand how increasing the number of simultaneously massaged flips affects massaging, we ran tests unmapping a single flippy page and unmapping 50 flippy pages. The 1 flip test showed a success rate of 0.65%, landing 1 out of 153 attempts, while the 50-flip test showed a 33% accuracy, landing 4 out of 12 attempts.

Prime+Probe. Finally, we tested the ability of PRIME+PROBE to detect cases in which a flippy page landed in the kernel stack. Massaging attempts for these experiments were made using 5-flippy pages at a time. In this case, the flippy page was considered to have landed if it resided anywhere among the 4 pages of kernel stack (rather than the specific target page) since that is the extent to which PRIME+PROBE can detect. Among 13 cases of the victim kernel stack containing a flippy page, 8 were correctly identified via PRIME+PROBE, yielding an accuracy of 61.5%. Any of the remaining false negatives would result in another massaging attempt, giving a total average time of 25 hours for the offline stage.

It is also worth noting that of all the massaging attempts, the PRIME+PROBE side-channel reported 5 false positives out of 34 instances (14.7%) of a flip not landing in the victim stack, allowing hammering to occur when a bit-flip is in a potentially dangerous position in the kernel. However, we additionally ran a test consisting of 60 5-flip massage tests *without* the PRIME+PROBE side-channel and observed only 2 crashes. Thus, even when allowing a small percentage of false negatives, we did not observe any dangerous kernel flips while the side-channel was active, making PRIME+PROBE a useful filter for preventing kernel panics.

Checking for Flips. Next, we checked which threads contain flippy bits. For each thread, we called our tester syscall, and stall operation for 0.5 second, while performing simultaneous hammering on all threads. Thanks to the simultaneous hammer, this step completed within a single stalling window of 0.5 second. Since our bit-flips are reliably reproducible, this step completed with 100% accuracy in one attempt.

Heap Spraying. Finally, we sprayed the heap with artificial structs. We then called `ioctl` within the thread containing the flippy page, and hammered in parallel. If we did not leak the expected data, we assumed the heap spray did not land our target data at a location one bit-flip away and sprayed again.

For our evaluation, we made 3485 attempts to land our attacker-controlled data into the heap at a position 1 bit-flip away from our target gadget. On average, each attempt required 129 spray attempts, averaging 38.9 seconds to land data at a useful position in the heap. Once the spray had successfully landed, we could swap the sprayed data without changing its position in the heap, allowing us to point to any address in the kernel without needing to heap-spray again.

Leakage Rate. After completing the previous stages end-to-end on an unmodified kernel, we left the attack to leak data over a 48-hour period. This process consisted of repeatedly running the victim gadget, hammering our aggressors, reading out arbitrary kernel data, swapping the sprayed data to point to a new arbitrary kernel address, and repeating. We observed an average leakage rate of 82.6 bits/second over this period.

5.3 Effects of Noise

To evaluate the performance of our attack under noisy conditions, we ran our attack code in parallel with benchmarks from the Phoronix Test Suite [35] running the default `CPU` and `memory` (RAM) benchmarking suites.

The average time needed to find a reproducible bit-flip increased from 5.8 minutes without noise to 25.15 minutes under a CPU-heavy load and 309 minutes under a memory heavy load. The drastic effect of the memory benchmark is likely due to DDR4 Rowhammer relying on striking memory in particular patterns to trigger bit-flips. Interference with this careful pattern of accesses likely triggers TRR and causes early refreshes before the bits have a chance to flip.

In contrast, the heap spray is not as sensitive to noise. The CPU-intensive benchmarks increased the average time to land heap data from 38 seconds to 52.4 seconds, and the memory-intensive benchmarks increased the time to 55 seconds. This is likely due to the heap spray relying on controlling large contiguous regions of the heape, and that ordinary user processes are unlikely to allocate large amounts of kernel heap.

Finally, the leakage rate under a CPU-heavy load decreased from 82.6 bits/s to 41.9 bits/s and under a memory-heavy load to 45.75 bits/s. The reduced rate is largely due to noise slowing down the process of swapping out all the heap-sprayed values when switching to a new address to leak.

5.4 Additional Gadgets

Read Gadget Search Tool. To explore the prevalence of gadgets in the kernel, we modified an existing gadget search tool. We extended smatch [27] to report any instance in which a nested pointer dereference sends data to a user via `put_user`,

`copy_to_user`, or `copy_to_iter`, which all have the property of passing kernel data to a calling unprivileged user. We classify such patterns as *read gadgets*.

Write Gadgets. We also observed that a similar pattern can be exploited to form a potential *write gadget*. Complimentary to `copy_to_user` is `copy_from_user`, which writes user space data to a specified, safe kernel address. Furthermore, with the read gadget, we require a *nested* pointer dereference. First, to point to data we control, which in turn points to a target address to leak. With a single dereference, we would simply read out data we already control back to ourselves. However, with the write gadget, we only require a *single* derefrence. If we flip a pointer to point to data we control, and the kernel *writes to that address* we can arbitrarily write to any address in memory. In other words, `copy_from_user` itself includes a guaranteed pointer dereference, giving us the same exploitable behavior as the read gadget without an explicit nested dereference occurring prior to the function call. We thus extend smatch further to report any instance of a pointer dereference in which the result is used as the target of `get_user`, `copy_from_user`, or `copy_from_iter`.

Results. We ran our modified smatch on Linux kernel version 5.16.2 and observed 192 read gadgets, and 28 write gadgets. We demonstrate an end-to-end attack on one such gadget in Section 5. We note smatch's known deficiencies in reporting both false positives [4] and false negatives [42], but believe it is the best option for detecting gadgets on the Linux kernel [5]. We leave the development of a more precise tool for future work.

6 Prior Attacks, Mitigations, and Conclusion

6.1 Prior Attacks

The first Rowhammer exploit demonstrated how the phenomenon could be used to flip PTE bits and give an attacker root privilege [40], and numerous follow-up studies have demonstrated new techniques for reproducing PTE flips under various attack scenarios [15, 31, 43, 44]. GadgetHammer is complimentary to PTE flipping in that it demonstrates how Rowhammer can be used to exploit *general code patterns* in the Linux kernel. In contrast to prior work focused on PTE flipping, GadgetHammer cannot be neutralized by protecting a specific kernel structure (e.g., page tables) but requires more holistic defenses to be considered. Similarly, other Rowhammer works present new ways to flip bits [13] or suggest alternate structures to target [14]. GadgetHammer is, to the best of our knowledge, the first work to consider targeting gadgets.

6.2 Mitigations

Code Patches. One defense option is to patch victim code to remove any gadgets. However, creating such a defense is non-trivial. As seen from attempts at patching Spectre gadgets [4] the challenge is two-fold. First, it is difficult to design tools that can automatically detect gadgets. They either use methods too slow to scale to large code bases, such as the kernel [16], or lack full coverage of all possible code paths, relying on approximate techniques such as fuzzing [32]. Second, defenses designed to protect against specific patterns can be bypassed by exploiting slight variations [23].

Software Mitigations for Rowhammer. Numerous attempts have been made at preventing exploits via software. For example, CATT [3] proposed placing buffers between user and kernel memory. That way, hammering user-space aggressor rows can cause flips only in the buffer addresses. However, prior work has demonstrated that it is possible to flip sensitive data despite these buffers [14]. Other defenses use a more direct approach, storing only key kernel structures, specifically PTEs, in flip-safe memory [49]. However, we have shown that Rowhammer attacks do not need to target specific structures, but rather can target general code patterns (nested pointers).

Hardware Mitigations for Rowhammer. The root of the Rowhammer problem is the ability to flip bits in DRAM. Therefore, numerous mitigations have attempted to prevent Rowhammer by adding defenses directly to DIMMs to detect and prevent flips. However, as mentioned in Section 2, both DDR4's Target Row Refresh [20], and DDR5's Refresh Management [21], have proven to be inadequate [13, 30].

Additional prior works have proposed ways to allow for bit-flips, but prevent the use of flipped data by the CPU or reallocating hammered rows of memory [11, 37, 47, 48]. However, such mitigations require area and performance overheads unlikely to be relinquished by manufacturers. Additionally, RAMBleed [26] and ECCsploit [7] have shown how Rowhammer can be effective even in the presence of integrity checks.

Ultimately, past attempts at mitigations have shown that allowing flips to occur in DRAM will inevitably lead to new vulnerabilities. The surest way to protect against Rowhammer would be to reconsider the fundamental causes of vulnerable DIMMs, such as how much voltage is supplied to capacitors, or the length of the slowest allowable refresh rate, to prevent bit-flips from occurring in the first place.

Conclusions. In this paper, we have identified a new type of Rowhammer gadget, demonstrating a particular pattern that makes code vulnerable to confidentiality exploits via Rowhammer. In future work, we hope to consider additional code patterns that may be susceptible to bit-flips.

7 Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Defense Advanced Research Projects Agency (DARPA) under contracts HR00112390029 and W912CG-23-C-0022, the Office of Navel Research (ONR) under Grant No. N00014-22-1-2622; the National Science Foundation under grant CNS-1954712; and gifts by Cisco and Qualcomm.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*, pages 987–1004. IEEE, 2016.

[3] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 117–130, 2017.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 249–266, 2019.

[5] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 666–680. IEEE, 2022.

[6] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopadhyay. Explframe: exploiting page frame cache for fault analysis of block ciphers. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1303–1306. IEEE, 2020.

[7] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[8] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Smash: Synchronized many-sided rowhammer attacks from javascript. In *USENIX Security Symposium*, pages 1001–1018, 2021.

[9] Lizzie Dixon. Using userfaultfd. 2016. URL:https://blog.lizzie.io/using-userfaultfd.html.

[10] Michael Fahr Jr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray Perlner, Arkady Yerukhimovich, et al. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, 2022.

[11] Ali Fakhrzadehgan, Yale N Patt, Prashant J Nair, and Moinuddin K Qureshi. Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 373–386. IEEE, 2022.

[12] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 195–210. IEEE, 2018.

[13] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762. IEEE, 2020.

[14] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261. IEEE, 2018.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.

[16] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2020.

[17] Jann Horn. How a simple linux kernel memory corruption bug can lead to complete system compromise, 2021. URL:https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html.

[18] Alex Ionescu. Sheep year kernel heap fengshui: Spraying in the big kids' pool. 2014. URL:https://www.alex-ionescu.com/kernel-heap-spraying-like-its-2015-swimming/-in-the-big-kids-pool/.

[19] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE, 2022.

[20] JEDEC. Jesd209-4d lpddr4, 2017. URL:https://www.jedec.org/standards-documents/docs/jesd209-4b.

[21] JEDEC. Jesd79-5b ddr5 sdram, 2022. URL:https://www.jedec.org/standards-documents/docs/jesd79-5b.

[22] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[23] Paul Kocher. Spectre mitigations in microsoft's c/c++ compiler, 2018. URL:https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[25] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. {Half-Double}: Hammering from the next row over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, 2022.

[26] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[27] Jonathan LCorbet. Finding spectre vulnerabilities with smatch, 2018. URL:https://lwn.net/Articles/752408/.

[28] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.

[29] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[30] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 735–753. IEEE, 2022.

[31] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[32] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498, 2020.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[34] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for cross-cpu attacks. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 565–581, 2016.

[35] phoronix-test suite. Phoronix test suite 10.8.4, Jun 2023. URL:https://github.com/phoronix-test-suite/phoronix-test-suite.

[36] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 1–18, 2016.

[37] Anish Saxena, Gururaj Saileshwar, Prashant J Nair, and Moinuddin Qureshi. Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 108–123. IEEE, 2022.

[38] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: leaking data on meltdown-resistant cpus (updated and extended version). *arXiv preprint arXiv:1905.05725*, 2019.

[39] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[40] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15:71, 2015.

[41] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 213–226, 2018.

[42] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Spechammer: Combining spectre and rowhammer for new speculative attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698. IEEE, 2022.

[43] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *CCS*, 2016.

[44] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardion: Practical mitigation of dma-based rowhammer attacks on arm. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, pages 92–113. Springer, 2018.

[45] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54. IEEE, 2019.

[46] vusec. trresspass, Mar 2020.

[47] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. Shadow: Preventing row hammer in dram with intra-subarray row shuffling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 333–346. IEEE, 2023.

[48] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J Nair. Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 374–389. IEEE, 2023.

[49] Xin-Chuan Wu, Timothy Sherwood, Frederic T Chong, and Yanjing Li. Protecting page tables from rowhammer attacks using monotonic pointers in dram true-cells. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 645–657, 2019.

[50] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.

[51] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41. IEEE, 2020.

[52] Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Ghostknight: Breaching data integrity via speculative execution. *arXiv preprint arXiv:2002.00524*, 2020.

[53] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Surya Nepal, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. {SoftTRR}: Protect page tables against rowhammer attacks using software-only target row refresh. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 399–414, 2022.