Exploring the Effects of Memory Vulnerabilities Across the Computer Architecture Stack

by

Youssef Tobah

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2025

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Todd Austin
Associate Professor Jiasi Chen
Professor J. Alex Halderman

Youssef Tobah
ytobah@umich.edu
ORCID iD: 0009-0009-6987-8851

DEDICATION

To my family for their unconditional love and support.

ACKNOWLEDGEMENTS

The contents of this thesis were only possible thanks to the contributions of many helpful, knowledgeable, and kind individuals.

Firstly, I would like to thank my advisor, Prof. Shin. Completing each project included in this thesis was a long and challenging process, and through all that Prof. Shin always remained patient, letting each project take the time it needed until it was ready. Additionally, his encouragement and guidance in how to identify problems and lead research project has been invaluable and will undoubtedly shape the rest of my career as a researcher and engineer.

Additionally, I would like to thank my colleagues in RTCL, both past and present. In particular, Tim Trippel, Chun-Yu Chen, Duc Bui, Mert D. Pese, and Haichuan Ding were always happy to give helpful advice from the perspective of PhD candidate further along in the process, and their words and insights not only helped me tackle hard problems, but also gave me strong motivation, helping me believe I could reach the same points they did as a PhD student.

I would also like to thank Hsun-Wei Cho, my colleague who started graduate school alongside me, as well as Noah Curran, Brian Tang, Mingke Wang, Wei-Lun Huang, Long Huang, Kailai Cui, and Liangkai Liu for their help and feedback on research projects over the years.

My research collaborators and coauthors have also made many invaluable contributions to my research. This includes Daniel Genkin, who's insights into side-channel security helped grow my knowledge in the area in leaps and bounds from nothing, as well as his students Andrew Kwong and Ingab Kang for their technical advice and expertise. Additionally, I would like to thank Kyle Ingols and Kevin Bush for hosting me at Lincoln Lab and providing their own technical insights during my time there as well.

Finally, I want to thank my family for their support over the years, as everything I am today is thanks to them.

This research was supported in part by the National Science Foundation (Grant No. 2245223) and the Office of Naval Research (Grant No. N00014-22-1-2622).

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

In the pursuit of more performant, power- and area-efficient systems, computers have been exposed to a plethora of security risks and vulnerabilities at various levels of the computer architecture stack. Oftentimes, the effects of a vulnerability at one level propagates into other levels, making it challenging to build defenses that encompass all the effects of a particular exploit. One such vulnerability, known as *Rowhammer*, allows attackers to flip bits in memory without ever accessing them by rapidly accessing adjacent addresses. Since its discovery, researchers have shown how bit-flips in memory can propagate through a victim's machine to form the basis for numerous exploits, including leaking cryptographic keys, denial of service, and privilege escalation.

This thesis explores Rowhammer at various levels of the computer architecture stack. Rowhammer attacks typically require an understanding of multiple components of the stack, including memory, architecture, and operating systems, in addition to other potential systems depending on the target. This thesis contributes to knowledge in the space at multiple levels, further demonstrating the threat Rowhammer poses beyond prior work.

More specifically, this thesis presents techniques for reverse-engineering a memory controller's address mapping, how Rowhammer can be combined with microarchitectural exploits, the threat Rowhammer poses to particular software patterns, and lastly, an exploration of the efficiency of end-to-end Rowhammer attacks.

By studying Rowhammer as a systems problem involving multiple layers of the computer architecture stack, the goal of this thesis is to demonstrate the need for defenses that solve the Rowhammer issue at its root. More specifically, my goal is to motivate and inform defenses that prevent bit-flips from occurring or propagating through the victim machine, rather than continuing to build defenses that patch specific exploits or points of vulnerability. By rooting out bit-flips at their source, computers can be made more secure against Rowhammer regardless of what new variety of bit-flip-based exploits attackers are able to craft in the future.

CHAPTER 1

Introduction

The trend in computer architecture has long been to push toward developing more power- and area-efficient systems while neglecting potential security risks, spawning a plethora of microarchitectural and side-channel vulnerabilities. For example, architectural features such as caching and speculative execution have brought indispensable performance boosts, but can leak micro-architectural information by following execution paths dependent on "secret" input data. These unintended side effects are demonstrated in works, such as Spectre [25], Flush+Reload [87], and Meltdown [49] among other powerful exploits.

Meanwhile, advancements in memory have allowed for packing increasing amounts of storage in smaller spaces, but simultaneously allow attackers to take advantage of "disturbance effects" between these tightly packed memory cells. More specifically, researchers have shown that thanks to these tightly packed cells, attackers can indirectly modify rows of memory without ever directly accessing them. By rapidly accessing (or "hammering") controlled rows of memory, attackers can accelerate the leakage rate of adjacent rows, flipping victim addresses and effectively gaining write access to otherwise in accessible memory. This phenomenon is known as the *Rowhammer* bug and is the focus of this thesis.

Since its initial discovery, Rowhammer has spawned a plethora of research efforts further exploring the bug. Some of these works focus on the phenomenon of the flips themselves at the memory controller and DRAM levels, analyzing new patterns for hammering memory to more efficiently flip bits [31, 20, 42]. Other efforts developed techniques for weaponizing bit-flips, discovering how flips in data and code can grant attackers access to secret information [23, 67]. Additional works have pushed the bounds of these attacks, presenting Rowhammer attacks via browsers and remote network connections [50, 69]. Lastly, several defenses have emerged in attempts to mitigate bit-flips and their side-effects at each of these levels, each inevitably trumped by follow-up attacks [53, 1].

Prior work demonstrates it is not enough to attempt protecting against Rowhammer at a single level of the computer architecture stack. The ultimate goal of this thesis is to motivate

and inform defenses that can better protect against Rowhammer in the future. Many prior defenses attempt to patch out specific vulnerabilities or points of exploit, by, for example, placing buffers around sensitive values, or repositioning potential known Rowhammer targets in memory to keep them safe from bit-flips. However, all too often, such defenses are shortly followed up by new exploits showing either ways to directly bypass those defenses or how new, alternative targets can be exploited to achieve the same results. To protext against Rowhammer, defenses are needed which can prevent these bit-flips from occurring altogether. Ideally, bit-flips should be avoided from occurring in DRAM in the first place, meaning the best defense would be more resilient DRAM. However, since this is not helpful for existing DRAM, at the very least, future defenses should seek to prevent bit-flips from propagating through software so that they may not be used for dangerous exploits. Simply patching specific exploits and allowing bit-flips to otherwise propagate throughout the victim's machine creates a high likelihood that new exploits will be developed and abused in the future. The work demonstrated in this thesis makes this point clear by presenting multiple of such exploits.

Thus, this thesis explores Rowhammer at various stages of the computer architecture abstraction layers, extending Rowhammer's reach at each level. In particular, this thesis develops new techniques for reverse engineering DRAM addressing on newer generations of CPU and DRAM architectures (Chapter 2), as well as new exploits targeting microarchitectural (Chapter 3) and software vulnerabilities (Chapter 4). Chapter 5 studies and improves various existing Rowhammer techniques to better understand the most efficient methods for attacking modern architectures.

Below is an overview of each subject covered in this thesis.

1.1 DRAM Addressing

Rowhammer attacks center around targeting a particular victim address by repeatedly accessing adjacent aggressor addresses. Thus, all Rowhammer attacks require the attacker to access specific targeted addresses in DRAM. However, unprivileged attackers only have access to virtual addresses, which are indirectly mapped to physical addresses, which, in turn, are mapped to DRAM addresses. Thus, a crucial first step is to reverse-engineer these mappings. A prior work called *DRAMA* [58] developed a technique that became the basis for DRAM address reverse-engineering in numerous follow-up works. The mappings were found to be consistent across most DDR3 compatible memory controllers as well as DDR4 compatible memory controllers.

However, beginning with 12th generation Intel processors, the DRAMA technique no

longer seemed to work for producing the physical-to-DRAM mapping, and attempting existing Rowhammer techniques using prior mappings produced no bit-flips. Chapter 2 discusses my approach for reverse engineering the mapping of newer generation processors. Physical probes are used to confirm the mapping has indeed changed from prior generations, and analyzing the new pattern reveals a mapping scheme carefully designed to trick DRAMA into reporting an incorrect mapping with full confidence. Using these mapping functions, we achieve the first Rowhammer bit-flips on 12th generation machines. Additionally, we identify the address mapping for DDR5 DIMMs, discovering a modification made to these functions that no longer allows for timing side-channel reverse engineering, and proposing a new technique that can be used despite this limitation.

1.2 Microarchitecture

In addition to developing new ways to flip bits on various architecture and memory generations, Rowhammer researchers have also explored how Rowhammer-induced bit-flips can be utilized for malicious exploits [67, 23]. This includes flipping page table entries (PTEs) to grant access to sensitive addresses, flipping opcodes to bypass password checks, and flipping cryptographic keys to produce outputs leaking those keys' values.

In parallel with Rowhammer, offensive cybersecurity research has recently introduced a new class of microarchitectural, transient execution side-channel attacks. These attacks, such as Spectre [41] and Meltdown [49], demonstrate that microarchitectural optimizations, such as branch prediction or out-of-order processing, can leave behind sensitive or secret data observable via leaky side-channels. For example, Spectre demonstrates that when branch predictors allow out-of-bounds accesses and eventually roll back the access to prevent users from reading out-of-bounds data, the accessed data is left behind in the data cache. From there, attackers can utilize a cache side-channel to read out the secret data.

In Chapter 3, I present how Rowhammer can be combined with Spectre to produce an enhanced microarchitectural side-channel attack, focusing on my IEEE S&P 2022 paper, SpecHammer [71]. We make the observation that, while presenting a powerful new class of attack, Spectre has a detrimental weakness in requiring attacker controlled variables in the victim's address space. More specifically, the basis for the first presented Spectre variant consists of manipulating an array index variable until an out-of-bounds array access occurs under a state of misspeculation. In order to both mis-train the branch predictor, as well as steer the array index to point to specific target data, the attacker must have full control over the array index variable—a condition rarely seen in real-world code.

SpecHammer, however, demonstrates that it is possible to perform this variant of Spec-

tre without any attacker-controlled variables in the victim's address space. By observing that Rowhammer is an attack that allows precisely for modifying values without directly controlling their variables, we show how a Rowhammer bit-flip can be used to manipulate array index values and induce a Spectre attack on uncontrolled victim, overcoming several key challenges along the way.

Of particular note is the limitation of Rowhammer bit-flips compared to fully controlling a victim variable. Rowhammer can be expected to flip a single bit of this variables' value at best, making it difficult to steer the victim to point to precise addresses. To overcome this challenge, we introduce the concept of a *Spectre triple gadget* as shown in Listing 1.1. This consists of a triple nested array access with a conditional statement. By flipping the first array index variable ("x" in the case of Listing 1.1), we can point the first array access to *attacker controlled data* and have full control over the subsequent array access.

We demonstrate a proof of concept SpecHammer attack on a triple gadget injected into the Linux kernel, observing a leakage rate of up to 24 bits/s on DDR3 and 19 bits/min on DDR4. Additionally, we run an automatic gadget search tool to better understand the effects of relaxing Spectre variant 1's requirement of controlling a victim variable. When running the search tool with the original requirements on the latest Linux kernel at the time, we observed about 100 Spectre v1 gadgets and 2 triple gadgets. Upon removing the requirement of an attacker-controlled variable, these numbers inflate to 20,000 double gadgets and 170 triple gadgets.

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

Listing 1.1: Pseudocode triple gadget

1.3 Operating Systems and Software

Going beyond transient execution attacks, this thesis also explores the concept of *Rowhammer gadgets* that function on their own without relying on techniques from Spectre or other transient execution microarchitectural attacks. Chapter 4 covers my USENIX Security 2024 paper, GadgetHammer [72], which presents the first "Rowhammer gadget."

This gadget demonstrates that any victim code containing a particular code pattern (or code "gadget"), is susceptible to arbitrary data leakage via a single bit-flip. Since prior Rowhammer exploits tended to target specific data or code, such as PTEs or the `sudo`

binary [23], numerous defenses have been developed to protect the specific victims from bit-flips [1, 5]. However, GadgetHammer demonstrates that it is not enough to defend these specific targets, as general code patterns exist that can fall victim to Rowhammer as well.

More specifically, we present the GadgetHammer triple gadget, a code pattern similar to that of SpecHammer's triple gadget, albeit with two key differences. First, the code does not need to be contained within a conditional statement, as there is no need to rely on speculative execution or a misprediction. Second, the gadget must send data back to a calling attacker, either via a return variable or pointer dereference: a requirement found to be common among function calls to privileged processes that return requested data to a calling user.

Going beyond the results demonstrated in SpecHammer, we successfully ran an end-to-end attack on a GadgetHammer gadget found in an unmodified Linux kernel, demonstrating a real world example in which sensitive software containing this gadget was at risk. In order to execute this attack, we additionally developed an improved memory massaging procedure to ensure the target victim variable would utilize a flip-vulnerable DRAM address, demonstrating the first end-to-end Rowhammer attack on a Linux kernel stack variable.

1.4 End-to-End Efficiency

Rowhammer presents numerous potential threats in the form of bit-flips. In addition to exploit-focused work similar to SpecHammer and GadgetHammer, several works have sought to explore new hammering patterns that can more efficiently flip bits. For example, TRRespass [20] presented a novel technique for flipping bits on DDR4, and several follow up works have shown how to increase the rate of flips several times over with higher-precision techniques [31, 42, 53, 35].

However, a Rowhammer bit-flip is only useful to an attacker if it can be utilized for an exploit. Yet a majority of these works that explore more efficient ways to flip bits do not test their techniques in end-to-end attacks. Indeed, a common approach has been to report only the time needed to *find* the first bit-flip that could be used for an exploit, but now actually employ that flip in an end-to-end attack.

What this approach neglects to consider is how these new more efficient techniques affect the success rate of the actual bit-flip-based exploit. For example, a crucial component of most Rowhammer attacks is "massaging" memory to ensure the target victim variables use flip-vulnerable addresses. However, newer Rowhammer techniques, such as TRRespass, require controlling numerous aggressor addresses simultaneously and do not consider how this might affect the success rate of memory massaging.

Thus, in this chapter Chapter 5 we explore such past Rowhammer techniques and extend them to function for *end-to-end* exploits. We do so by developing novel memory massaging techniques that provide enhanced accuracy compared to prior work, and that allow for the first PTE attack on DDR4 in conjunction with existing DDR4 hammering techniques such as TRRespass [20] and SledgeHammer [35]. Additionally, we run an end-to-end comparison between newer and older Rowhammer techniques, and find that while newer techniques can flip bits more efficiently, older techniques give a faster end-to-end attack time due to their reduced impact on memory massaging accuracy. With this result, we demonstrate the need for future Rowhammer techniques to consider how their methods work in conjunction with memory massaging and when used in end-to-end attacks. Since the goal for any hammering technique is to flip bits that may be used in an exploit, it is crucial to consider the practicality of such flips on not merely the efficiency with which they can be produced.

CHAPTER 2

Reverse Engineering DRAM Addressing for Rowhammer Attacks

The last several decades have seen tremendous advancements in computer science and engineering, bringing more efficient and performant machines to the world while using smaller are and power costs. However, along with these efficiencies come new exploits that take advantage of optimizations for the common case. The Rowhammer bug is one such exploit that takes advantage of the tight packing of capacitors on recent memory modules, allowing attackers to rapidly access memory addresses they control in order to induce disturbance effects on nearby rows of memory and flip bits in those uncontrolled rows without ever needing to access them directly.

Shortly after Rowhammer's discovery followed a back-and-forth between attacks and defenses attempting to patch out Rowhammer on newer machines and exploits demonstrating how those defenses can be broken. Indeed, on the latest generation of Intel processors at the time of writing (generation 12), existing Rowhammer techniques are no longer able to flip bits on DDR4 memory. The goal of this work, then, is to understand why existing techniques no longer work and to prove that once the issue is understood, it is indeed possible to flip bits on a newer machine despite the changes.

We find that the main difference blocking prior techniques is a change in the CPUs memory controller, which now uses a new set of addressing functions for mapping physical addresses to DRAM addresses. This mapping uses an increased number of physical address bits for its mapping function, preventing prior techniques from working in the current form. We find that by making simple adjustments to these techniques, we are able to obtain the mapping functions and flip bits even when using newer processors. Furthermore, since the latest processors are also compatible with DDR5, we reverse engineer the DDR5 mapping functions as well.

2.1 Introduction

The last several decades have seen progress and advancements in leaps and bounds in the field of computer science and engineering. New developments have improved efficiency and performance at every level, from physical design, to architecture, to software. However, as systems became more complex to handle more use cases and larger data sets, while also maintaining a shrinking power and area cost, new vulnerabilities emerged, taking advantage of all the new and additional features.

Indeed, the last decade has seen advanced cyber attacks that either target or utilize various components and features across the computer architecture stack that were originally intended only to boost performance or efficiency [6, 25, 71, 86, 35, 38, 67, 65, 56]. These various exploits target features and structures such as speculative execution, caches, out-of-order execution, and numerous other performance-saving components of a victim's machine. By optimizing for the common-case, these advancements do not consider the potential security risks of fringe cases that may occur when used in ways not originally intended.

As a particular example, and of interest to his paper, we consider advancements made at the memory level in DRAM. With each new generation of DRAM, designers seek to pack more and more memory into tighter areas, using smaller capacitors that each utilize less voltage. However, this trend directly led to a still-unsolved memory exploit known as Rowhammer, which shows that attackers can take advantage of disturbance effects between these tightly-packed capacitors to induce charge-leakage in rows of memory without ever needing to access them directly. Thus, thanks to the efficiencies of newer DRAM modules, attackers can flip bits in victim memory simply by repeatedly accessing adjacent memory addresses repeatedly.

While this vulnerability was first discovered over a decade ago [38], it has yet to be solved even on following generations of DRAM and CPU memory controllers. Indeed, executing a Rowhammer exploit involves not only the victim DRAM dual in-line memory module (DIMM), but also manipulating the CPUs memory controller to access and activate precise targetted rows. However, the same techniques the worked in the original Rowhammer paper have been successfully demonstrated on follow-up memory controllers that did not patch the issue. Similarly, when DDR4 became available to the public, researchers quickly found ways to execute a DDR4 Rowhammer exploit, and the discovered techniques worked on numerous following generations of CPUs.

However, we find via our own experimental observations, that Rowhammer techniques that worked up to 10th generation Intel CPUs are no longer effective on machines using 11th generation or newer CPUs. Indeed, while attempting Rowhammer on a particular DIMM

may induce bit-flips on a 10th generation Intel CPU, attaching the same DIMM to a newer CPU does not induce flips. Rowhammer is largely dependent on the nature of the DIMM itself, but we find in this work that updates added to newer CPUs were enough to prevent attackers from inducing bit-flips.

Thus, in order to better understand the current and future state of Rowhammer research on newer victims, we seek to answer the following questions in this work:

Can we induce Rowhammer bit-flips on victim machines using newer generation CPUs? What changes were added to newer generation CPUs that prevent older techniques from working?

2.1.1 Our Contributions

We find answers for both of the above questions, both learning that it is possible to induce flips on newer generation CPUs as well as understanding exactly what obstacles prevented existing techniques from working. We find the root cause of the issue is tied to the CPU's memory controller and develop techniques to overcome the recent modifications, allowing for the first Rowhammer attack on newer generation CPUs and uncovering the address mapping functions for DDR5 as well. Our contributions are outlined in more detail below.

Understanding New Memory Mapping Functions Rowhammer attacks are centered around activating particular rows of memory that surround targetted victim rows. Thus, in order to perform a Rowhammer attack, one needs to understand exactly which rows in DRAM she reads or writes to with any given access. However, DRAM addresses are not directly exposed to users, and instead, users can only observe virtual addresses, which map to physical addresses that in turn map to DRAM addresses. Existing techniques developed for 4th generation CPUs have worked reliably up through 10th generation CPUs [58]. However, these techniques no longer seem to work on newer machines.

Thus, in this work, we manually reverse engineered the DRAM mapping functions to understand what has changed from prior work and learn why old techniques no longer work. We find that the newer mapping functions use a larger amount of physical address bits, which older techniques did not account for. Additionally, this use of higher order physical bits means existing techniques for mapping virtual addresses to physical addresses will also no longer work, since they can only obtain the 21 lowest bits, while the newer mapping functions use up to bit 36.

Developing Techniques for Obtaining Newer Mapping Functions Since older techniques no longer work on newer generation memory controllers, in this work, we develop techniques for successfully obtaining the mapping from physical memory to DRAM. We find

that modifying prior techniques to account for the larger number of bits allows for the same base technique, based on timing side-channels, to be used even for newer CPUs using DDR4 DIMMs.

Flipping Bits on Newer Generation DDR4 Victims Armed with the newfound ability to reverse engineer the physical to DRAM mapping on newer generation CPUs, we are able to execute an end-to-end Rowhammer attack on a newer generation victim machine, achieving the first bit-flips on a 12th generation Intel CPU. This result demonstrates the Rowhammer issue has indeed not been solved even on newer machines, and more advanced, holistic defenses will be needed to truly prevent future exploits.

Obtaining DDR5 Mappings Lastly, newer generation machines are also compatible with the latest model of DRAM, DDR5. Thus, we additionally explored the physical to DRAM address mapping on DDR5 as well. We find that when using DDR5 memory, the CPU's memory controller will utilize a different mapping function than that used with DDR4. We physically reverse engineered the mapping function for DDR5, and learn that even when accounting for additional bits, prior techniques no longer work due to the nature of the new mapping functions. We find that new techniques must be developed if an attacker wishes to obtain the mapping functions on a DDR5 machine without physical probing. However, since the mapping functions will be the same for any machine using a particular generation of CPU and DDR5, simply probing the functions on an attacker-controlled machine using the same components as the target victim is enough to learn the victim's mapping function. Thus, we provide this result for generation 12 CPUs, as well as the techniques used to obtain these mappings.

2.1.2 Challenges

Before we can achieve the results of obtaining new address mappings and achieving bit-flips on newer machines, we must overcome multiple key challenges tied both to the process of reverse engineering the mappings and to the nature of the new mappings and how they interfere with existing processes.

Reverse Engineering Address Mappings The only known technique for obtaining the physical to DRAM mapping is the timing side-channel technique presented in DRAMA [58]. This no longer works on newer hardware, making us, as attackers, blind to what the new mappings could be. We must therefore physically probe the victim machine in order to obtain a ground truth mapping and devise a new method from there. This requires understanding DDR4 protocols and working around DDR4's high transfer rate despite utilizing hardware that does not possess a high enough sampling rate. We are able to overcome these obstacles

by studying the DDR4 standards and observing how the various pins' signals synchronize together, eliminate the need for full signals obtained with a proper sampling rate.

Working with New Mappings Once the ground truth mappings are obtained, we must then devise a new technique that can obtain the mappings via software. We find that modifying the existing code to support and extended set of bits allows for the timing side-channel to accurately reveal the physical to DRAM mappings on DDR4. However, the DDR5 mappings are configured in a way such that it will always lead a timing-channel analysis based technique to report an incorrect result with 100% confidence. Additionally, on both DDR4 and DDR5, the mappings are too large to perform Rowhammer without the use of 1GB hugepages, meaning page coloring techniques are needed to perform Rowhammer, making prior works' techniques of obtaining 2MB contiguous blocks via memory manipulation no longer useful.

Summary of Contributions. This paper makes the following contributions:

- Obtaining the memory mapping functions from physical to DRAM addresses on newer generation Intel CPUs (Section 2.3).
- Extending prior techniques to obtain such mappings via software (Section 2.3.4).
- Achieving the first bit-flips on generation 12 (Section 2.3.5).
- Obtaining the memory mapping functions for DDR5 (Section 2.4).

2.2 Background

The following sections provide background on Rowhammer, DRAM mapping functions, and reverse engineering techniques useful for understanding the remaining technical content of the paper.

2.2.1 DRAM Organization

The Rowhammer bug is a byproduct of the nature of DRAM. Thus it is useful to understand the structure and layout of DRAM. A single bit of memory is stored within a *cell* of DRAM, which uses a capacitor maintain a bit-value. For example, a fully charged capacitor may signify that cell is storing a bit-value of `1`, while a discharged capacitor stores a `0`. Cells are arranged in *rows*, and anytime memory is accessed, an entire row's bit-values are passed along to the CPU. Upon an access, the row's charged is passed to a *row buffer*, from which the CPU can read the row's data until the access is completed.

Rows are organized into groups known as *banks*, with each bank having its own row buffer. Banks are then categorized into *ranks*, which refer to the front and back sides of the DIMM.

I.e., Rank 0 may represent the group of banks of the front side of the DIMM, while Rank 1 represents the group of banks on the back side. The highest level of organization is the *channel*, which groups entire DIMMs (also called DIMM sticks) based on the slot they are attached to on the motherboard.

Additionally, DDR4 (the fourth generation standard for DRAM) introduced a new level of organization called the *bank group*, which groups together subsets of banks within a rank. Bank groups share resources that allow for faster consecutive accesses on addresses within different bank groups.

2.2.2 Rowhammer

The Rowhammer vulnerability allows attackers to flip bits in memory without ever directly accessing them [38]. The bug is a result of DRAM's reliance on capacitors and the close packing of those capacitors in DDR3 and beyond. In particular, capacitors, by nature, leak charge over time. If left on their own for long enough, any values stored in DRAM would be lost as the capacitors lose their charge. Therefore, DRAM periodically refreshes each of its cells, restoring any capacitors storing a `1` with full charge.

What Rowhammer discovered is that each time a row of memory is accessed, and the row's charges are temporarily discharged into the rowbuffer, *disturbance effects* are induced on neighboring capacitors. These disturbance effects can accelerate the leakage rate of adjacent capacitors, causing their charge to drop below the threshold value that signifies whether the capacitor stores a 1 or 0. Thus, by rapidly and repeatedly accessing rows of memory and draining adjacent capacitors below their threshold value before DRAM has a chance to refresh those capacitors, attackers can induce bit-flips in rows without ever needing to access them directly. Since some capacitors are configured to represent 1 when discharged and 0 when charged, this can be used to induce both 1 to 0 and 0 to 1 flips.

2.2.3 Double-Sided Rowhammer

When the technique was first discovered, it was also shown that bit-flips could be induced more efficiently if attackers not only access a single row repeatedly, but rather, access a pair of rows that surround a target victim row. This technique is known as *double-sided Rowhammer*, and was shown to drastically increase the rate at which an attacker can induce bit-flips in a given victim.

2.2.4 DDR4 Rowhammer

With the discovery of Rowhammer, and the threat it posed to the integrity of DRAM data, newer generations of DRAM incorporated defenses to prevent attacker-induced bit-flips. In particular, DDR4 was given a built-in defense known as Target Row Refresh (TRR) [33], which tracks activations of pairs of rows. If any pair of rows crosses a fixed threshold of repeated activations, the rows adjacent to the potential aggressors are granted an early refresh. Thus, any work done by an attacker toward draining a capacitor of its charge is undone.

While TRR did put a stop to double-sided and single-sided Rowhammer as performed on DDR3, it was not long before researchers discovered new ways to induce bit-flips on DDR4 even in the presence of TRR [20]. Indeed, it was observed that TRR's design contains a key flaw in the fact that TRR is only capable of tracking a single pair of aggressors at a time. Thus, if an attacker strikes a pair of aggressors repeatedly, and then switches to a different set of aggressors, TRR's tracker will reset as it begins to count the activations on the second pair of aggressors. TRRespass [20] used this observation to develop the *n-sided* or *multi-sided* Rowhammer technique, which scatters accesses across a bank to prevent TRR from inducing an early refresh, while still draining the capacitors of a target victim row. Using TRRespass, it was shown that DDR4 is even *more vulnerable* to bit-flips once TRR is bypassed, likely due to capacitors being packed more closely together on newer DIMMs.

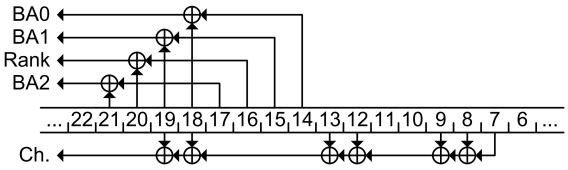

Figure 2.1: **Physical to DRAM map for Ivy Bridge/Haswell (taken from [58]).**

2.2.5 DRAMA

Considering the efficacy of both the double-sided and multi-sided hammering techniques, it is clear Rowhammer requires attackers to target particular rows in DRAM contained within the same bank and surrounding the same target victim row. However, attackers are not able to observe DRAM addresses directly. Instead, unprivileged attackers can interact with *virtual addresses*, which map to *physical addresses*, which in turn map to DRAM addresses. The virtual to physical mapping uses a page table, controlled entirely via software and indirect mappings. The physical to DRAM mapping, conversely, uses a direct mapping contained entirely in the hardware of the CPU's memory controller. This mapping typically consists of

sets of physical address bits XORd together to produce single DRAM bank, rank, or channel bits as shown in Figure 2.1.

Privileged users may view the virtual to physical mapping via a restricted file called the `pagemap`, but even privileged users have no way of viewing the physical to DRAM mapping. It is up to the CPU manufacturors to include this mapping function within their documentation, and in the case of Intel, the physical to DRAM mapping function is never publicly documented.

Furthermore, attackers cannot assume adjacent addresses are in the same bank since the mapping functions are usually designed to scatter consecutive accesses across banks. This is done so that accesses on consecutive memory addresses (such as consecutive values from an array) do not cause row-buffer conflicts that would slow down each access. Thus, to perform effective Rowhammer attacks, attackers must reverse engineer the physical to DRAM mapping to learn which groups of addresses belong to the same bank and can be used for multi-sided attack patterns.

DRAMA [58] demonstrated how a timing-side channel tied to row buffer conflicts can be used to reverse engineer this mapping via software. More specifically, DRAMA made the observation that consecutive accesses on pairs of addresses within the same bank are observably faster than consecutive accesses on pairs of addresses contained in separate banks. An attacker can therefore access pair of addresses and change a single bit a time within a single address of the set, while observing if that bit change had an affect on the speed of the consecutive accesses. If the bit-change had an effect, it can be concluded that the changed bit was part of the DRAM mapping function. By extending this process to test sets of multiple simultaneous bit-changes, one can obtain all the sets of XOR functions used for the mapping.

2.2.6 Older Mapping Functions

On older CPUs, physical to DRAM mapping functions used the lower 21 bits of physical address to determine rank, bank, and channel [58]. This meant an attacker only needed to know the lower 21 bits of physical address in order to have full control over which DRAM addresses she can access. Numerous techniques have been developed to obtain the lower 21 bits of physical address [45, 14]. These typically centered around obtaining a large block of contiguous memory, and ensuring the block was aligned, meaning the lower physical address bits of the first address in the block would have a value of 0, along with the corresponding virtual address. If the block is contiguous, and the lower order bits have the same value as the corresponding virtual address, then the attacker knows that consecutive virtual addresses

will correspond to consecutive physical address, meaning the lower bits of the virtual and physical address will be the same for the stretch of the contiguous block of physical memory. This allows attacker to have control over the lower order bits of physical address and target specific addresses in DRAM.

Prior work has demonstrated various methods for obtaining contiguous blocks of memory [80, 45, 14] such as using mounted hugepages, transparent hugepages, or draining smaller blocks of memory so that the memory allocator is forced to grant a large contiguous block upon a subsequent request. If a block of contiguous memory is 2MB and aligned, an attacker can discern the lower 21 bits of any address within that block, giving enough information for targetting particular DRAM addresses uses older generation mapping functions.

2.3 Reverse Engineering DDR4 Mappings

The primary goal of this work is to obtain the physical to DRAM mapping functions on a 12th generation Intel processor. These mappings are needed to facilitate Rowhammer attacks on such hardware and confirm that the vulnerability is still an open issue that needs a proper solution. In this section we detail the techniques used to obtain the mapping via physical probing, as well as what modifications are needed to existing software techniques for obtaining these mappings and performing Rowhammer attacks without physical access to the victim machine.

2.3.1 Ground Truth Probing

Before attempting to develop software techniques to automatically obtain the victim mapping functions, we can obtain the mapping on our own, attacker-controlled machine via physical probing. On prior generations of Intel CPUs, the mapping function was always the same for any two CPUs of a given model. As we will see in Section 2.4, this behavior holds true for 12th generation Intel CPUs as well. Therefore, a viable technique for obtaining the mapping is to obtain a CPU of the same model as the victim and use full physical access and control with full permissions to reverse engineer the mapping via physical probing. The same mapping can then be used to launch an attack on a victim machine via native code execution, remote execution, or other various attack vectors that do not require physical access or elevated permissions.

Figure 2.2: **Physical probing experimental setup.**

2.3.2 DDR4 Protocol and Pin Layout

Before we can physically probe DDR4, we must first understand DDR4's protocol for memory access, as well as its pin layout, so that we may know which pins to probe and the meaning of their signals. We are primarily interested in learning the mapping functions for the bank bits in the DRAM address, since with control over the bank values, we can ensure that all our accesses stay within the same bank for a single-rank DIMM in a single-channel configuration. On DDR4, four pins are dedicated specifically to controlling the bank and bank group values, meaning these are the pins which we must probe. DDR4 breaks its memory accesses into multiple stages, with the relevant stage for our purposes being *row activation*. It is during this stage that the bank pins are activated.

2.3.3 Methods

Our goal is to understand the physical to DRAM mapping functions for each of the four bank bits (corresponding to the four bank pins). We therefore begin by attaching a probe to one of the four bank pins (shown in Figure 2.2) followed by performing repeated memory accesses on a specific address. For our access, we mount a 1GB hugepage, of which the first physical address is 0x180000000. After recording our sample of the bank pin, we can then change our memory access to access a physical address with only a single physical address

Bank Address Bit	Physical Address Mapping Function
BG0	8 XOR 12 XOR 19
BG1	14 XOR 18 XOR 23 XOR 27 XOR 31 XOR 34
BG2	15 XOR 20 XOR 24 XOR 28 XOR 34
BA0	16 XOR 21 XOR 25 XOR 29 XOR 33
BG2	17 XOR 22 XOR 26 XOR 30
CH	9 XOR 12 XOR 13 XOR 18 XOR 19

Table 2.1: Physical to DRAM mapping function on DDR5.

bit difference from the first and observe if the signal on the probed bank bit changed. If so, we know that the modified physical address bit is used for that bank pin's mapping function. We then repeat this procedure until we've tested every bit we can change. That is, we access address 0x180000001, followed by 0x180000002, then, 0x180000004, etc. Once this procedure is completed for a single bank pin, it is then repeated for the remaining bank pins.

Once we complete these steps for all four bank pins, we have a full table showing which bank pins are affected by which physical address bits. For example, if changing bit 9, bit 11, and bit 13 each had an effect on the value of the pin corresponding to bank group 0, then bits 9, 11, and 13 are part of a set used in the mapping function for that pin. We then modify multiple pins within each set and confirm that mapping functions consist of XORing all the physical address bits within a set, as was done in older CPUs' mapping functions.

2.3.4 Observed Mapping Functions

The observed mapping functions are laid out in Table 2.1. As shown in the table, the mapping functions consist of linear XOR patterns, as used on older CPUs. However, a key difference is that the newer mapping functions no longer use only the lowest 21 bits of physical address. The newer functions use all the way up to bit 33.

While the core concept of a timing side-channel based on Rowbuffer conflicts might still work for these new mapping functions, DRAMA in its current form does not support reverse engineering for these higher order bits. We modified DRAMA, extending functionality for up to bit 33, allowing it to give accurate results even for newer generation CPUs.

2.3.5 Rowhammer on Generation 12

With the new mapping functions obtained, we are now ready to perform Rowhammer on newer generation CPUs. However, since the mapping functions use up to bit 33, using a contiguous 2MB block of memory is no longer enough to ensure our aggressor addresses will all be within the same bank. Thus, for the sake of verifying Rowhammer's presence on a newer generation machine, we use a mounted 1GB hugepage to have a stretch of contiguous memory large enough for a multi-sided attack. Another viable technique could bypass the

need for contiguous memory altogether via *page coloring*. With this technique, attackers can allocate memory and compare timing conflicts between controlled addresses to determine if they are in the same bank. Sets of addresses within the same bank can then be used for multi-sided Rowhammer. Page coloring has been implemented in existing Rowhammer attacks, making it a viable technique for Rowhammer without contiguous memory [35], albeit, while introducing additional complexity.

2.4 Reverse-Engineering DDR5 Mappings

Since 12th generation Intel CPUs are also compatible with DDR5, and following generations are *only* compatible with DDR5 and not DDR4, we also sought to reverse-engineer the mapping on DDR4. DDR5 uses a new protocol and pin layout, giving rise to additional challenges we had to overcome to reverse-engineer the physical to DRAM mapping.

2.4.1 DDR5 Protocol and Pin Layout

DDR5 no longer uses dedicated pins for controlling which bank or bank group is accessed. Instead, general *command addressing* pins are used, which serve various functions depending on the command the DIMM is issued. For example, certain command address pins may serve as bank pins during an "activate" command, or as opcode pins during a "mode register write" command. Furthermore, commands now each consist of two steps, with each step performed on a separate clock cycle. For example, an activate consists of two cycles in which the first cycle is used to determine the bank, bank group, and lower order row bits, followed by a second cycle to determine the remaining row bits. This means that the command addressing pins associated with the bank and bank group only represent bank numbers for half the command's duration. During the second half, they act as row pins.

DDR5 DIMMs also have multiple channels built-in. Half of the pins on the DIMM correspond to Channel A, and the other half Channel B. Thus, any individual memory access will utilize, at most, half of the pins as it accesses either Channel A or Channel B. subsequent accesses made to different channels are slightly faster than those done within the same channel.

2.4.2 Methods

One challenge here is that the probe we possessed for our experiments did have a high enough sampling rate to accurately present the signal of the bank pin. Furthermore, due to DDR5's method of breaking memory accesses into multiple stages, and breaking commands

Bank Address Bit	Physical Address Mapping Function
BG0	8 XOR 12 XOR 19
BG1	14 XOR 18 XOR 23 XOR 27 XOR 31 XOR 34
BG2	15 XOR 20 XOR 24 XOR 28 XOR 34
BA0	16 XOR 21 XOR 25 XOR 29 XOR 33
BG2	17 XOR 22 XOR 26 XOR 30
CH	9 XOR 12 XOR 13 XOR 18 XOR 19

Table 2.2: Physical to DRAM mapping function on DDR5.

into multiple cycles, the bank pins would not always represent bank numbers when they were sampled.

To work around this issue, we simultaneously probed signals from other pins that could act as control signals. In particular, when command addressing pins CA1 and CS are set low, and regular reads to memory are performed repeatedly, we know that the pins used for determining bank numbers will represent bank numbers. It is only when CS or CA1 are set high that the pins of interest may represent other values.

Thus, we attached probes to these additional control pins and synchronized our samples to correspond with these pins. Even though we could not produce a fully accurate signal due to the low sampling rate, having a snapshot synchronized to the moment these pins were both pulled low was enough to be sure that the probed bank pin did indeed represent a bank number.

With this configuration, we are ready to repeat the same procedure as before, consisting of accessing values within a 1GB hugepage and changing only one bit at a time. There is one additional step we must take with DDR5 to account for its multiple channels. Similar to bank addressing functions, the channel will also have its own addressing function to determine whether Channel A or B is being accessed. While Channel A is being accessed, all Channel B pins are held low, and vice versa. Therefore, if the probes are attached to a Channel A bank pin, and any bits are changed that affect the channel, the Channel A bank pin will be held low regardless of the rest of the physical address bits' values and will result in an incorrect signal.

Therefore, we first begin by determining which pins affect the channel. We do so by following the procedure of modifying only a single bit at a time, but instead of observing bank pins specifically, we observe whether multiple pins on a particular channel are active or inactive during the access. If a physical bit change affects the status of being active or inactive, then we know that pin is tied to the channel addressing function. We continue this process until we find all physical address bits that affect the channel. We then hold those bits at a steady value while modifying all other bits one by one. We once again group physical address bits into sets depending on which bank bits they affect.

In order to account for any channel pins that may also affect bank bits, we modify

channel bits in pairs, ensuring that our accesses remain in the same channel, since mapping functions are a result of XORing physical addressing bits. But changing all possible pairs, we can identify which channel bits may also affect bank addressing functions. The results are presented in Table 2.2.

2.4.3 Complications with DDR5

DDR5's built-in channels, and sharing of channel bits with bank bits creates a unique issue regarding software reverse-engineering. In particular, using the DRAMA technique will always produce an incorrect result with full confidence. This is due to the reuse of bits 8, 12, and 19 in both channel and bank addressing functions. DRAMA operates by first finding pairs of addresses that do not produce timing conflicts and masks out which bits are different between each address in the pair. Any sets of masked bits have some relationship that affects bank addressing functions. DRAMA will then identify relationship within these sets of bits to determine the addressing functions.

Suppose, for example, DRAMA considers a pair of addresses in which bit 8 is toggled. In order to stay within the same bank, bit 12 or bit 19 must also be toggled to keep BG0 the same. However, if bit 12 or 19 is toggled, the channel will be affected, meaning bit 9, 13, or 18 must also be toggled to stay within the same bank. Therefore, DRAMA will identify a pattern between bits 8, 9, 13, and 18. The relationship between these bits does exist, but it is not due to a singular addressing function. Additionally, a pattern is found between 8, 9, 13, 18, and 19, and normally this superset would overtake the previously found subset of 8, 9, 13, and 18. However, this superset also happens to be the XOR of two previously disovered patterns, 8, 9, 13, and 18 and 8, 12, and 19, meaning the superset is discarded and any XOR of smaller relationships is typically just an XOR of existing mapping functions. Therefore, DRAMA is incompatible with DDR5's mapping, with no way to filter out the relationship of overlapping bits.

2.4.4 DDR5 Workarounds

Since software techniques can no longer be used to reverse-engineer DDR5's mapping, alternative solutions will be needed for Rowhammer on DDR5. One approach is to follow the physical probing procedures outlined in this thesis. As previously mentioned, multiple CPUs of the same model will typically use the same addressing functions. We additionally confirmed via probing on multiple generation-12 CPUs that the addressing functions are indeed the same from CPU to CPU. Therefore, the mappings provided here are sufficient for Rowhammer attacks on generation-12 CPUs. For future CPUs, if the mappings change

from generation 12, a valid approach, as mentioned previously, would be to acquire the same hardware as the victim and reverse-engineer the mapping via probing using the steps outlined in this work. Alternatively, the page coloring technique mentioned above for DDR4 would also be a viable technique for DDR5.

2.5 Mitigations

Even if future hardware changes the mapping functions from those used in Generation-12 memory controllers, if they have the same implementation just with different sets of bits or different XOR functions, then the same techniques can be used to obtain the mappings and perform Rowhammer attacks on such machines. Therefore, more advanced mitigations would be required to avoid this issue on future machines.

2.5.1 Hardware Mitigations

One approach would simply be to not use the same mapping function across all CPUs, but to have varied functions even within multiple units of the same CPU model. This way, even if an attacker physically probes one CPU, they will not know the mapping functions for a victim she cannot physically access. A potential issue with this approach, however, is that there may be a limited number of possible mapping functions the memory controller could possibly use while maintaining efficiency for common memory access patterns. Additionally, even without physical probing, attackers can use the row buffer timing side channel to reverse-engineer the mappings via software. Even on DDR5 where DRAMA cannot be used to obtain exact mappings, just knowing that that two addresses are within the same bank is enough to leak information about victim activity [58], or to perform Rowhammer via page coloring [35]. Furthermore, hardware mitigations cannot be applied retroactively to old machines, meaning software mitigations should also be considered.

2.5.2 Software Mitigations

Software mitigations can take advantage of the multilayered structure of memory address translation. In particular, since users only have access to virtual addresses, which have to be translated to physical address, that map directly to DRAM addresses, one approach for a defense could be to dynamically alter virtual to physical page mappings at runtime. Therefore, if an attacker uses a timing side-channel or page coloring approach to build sets of addresses located within the same bank, swapping physical pages would allow code to

continue execution wile preventing those sets from remaining within the same bank. Additionally, systems already exist for swapping physical pages to swap space when a machine is under high memory pressure. The drawback to such an approach could be a potential performance loss each time pages are swapped out. However, since Rowhammer is merely one class of attacks that spawned as a result of computer architects boosting performance at the cost of security, future mitigations should consider potentially allowing for performance loss if it means building more secure, reliable, trustworthy systems.

2.6 Conclusion

In this work, we presented techniques for reverse-engineering physical to DRAM addressing on 12th generation Intel CPUs for both DDR4 and DDR5. We additionally achieved the first bit-flips using a Rowhammer attack on a 12th generation machine, and identified mapping functions that prevent the use of pure timing side-channel techniques for reverse-engineering such mappings, while posing solutions that have provably allowed for Rowhammer attacks under such circumstances.

CHAPTER 3

SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks

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.

3.1 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 [41] and Meltdown [49], 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 [6, 7, 52, 66, 39] and defenses [61, 60, 81, 7, 74].

Moving away from information leakage, Rowhammer [38] 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 [50, 75, 62, 45, 67, 58, 54, 69, 23, 4, 24, 19, 15], including the recent bypass of dedicated defenses, such as Targeted Row Refresh (TRR) [80] and Error Correcting Codes (ECC-RAM) [12].

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 [91], 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?

3.1.1 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 [13]. 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 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 [22, 68, 80, 77] 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 Section 3.4, 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 [62] 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 [67], or were performed on mobile devices, taking advantage of features exclusive to Android [75]. 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 Section 3.5).

Challenge 3: Proof-of-Concept (PoC) Demonstration. As a proof of concept, we demonstrate (in Section Section 3.6) 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. We demonstrate a leakage 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 Section 3.7, 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 4.4 & Section 3.7).
- Development of new methods for massaging a victim stack in user and kernel space, allowing an attacker to exploit the numerous gadgets present in the Linux kernel (Section 3.5).
- Correcting oversights made by prior Rowhammer techniques to improve bit-flip rate by 525x in the best case (Section 3.4).
- 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 3.6).

3.2 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.

3.2.1 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 [56, 87, 85, 57, 41]. 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 [87]. 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, 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.

3.2.2 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.

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 [49] and Spectre [41] 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.

```
1 if(x < array1_size){
2   y = array1[x]
3   z = array2[y * 4096];
4 }
```

Listing 3.1: Spectre v1 Gadget

Spectre Attacks. Spectre [41] 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 Listing 3.1 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 [87] 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 [81, 47, 25] to automate the process of finding gadgets within target code. For example, smatch [47], 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 [6, 39, 10, 44, 52, 65, 66, 27]. These included performing speculative writes [39], running a Spectre attack over a network [66], and combining Spectre with other side-channels to exploit "half gadgets" that require a

single array access within a conditional statement [65].

3.2.3 Rowhammer

The Rowhammer bug [38] 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. [38] 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 [23], 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 [67] demonstrating how a flip can be used both to perform a sandbox escape, as well overwrite page table entries. Many exploits followed [45, 75, 23, 1, 58, 62, 69, 50, 54, 4], demonstrating how Rowhammer can be used for privilege escalation on mobile devices [75], flipping bits through a web browser using JavaScript [22], as well as remotely attacking a victim over a network [50, 69]. Gruss et al. [23] additionally showed how many Rowhammer defenses can be defeated.

GhostKnight. To the best of our knowledge, only one prior work, GhostKnight [91], 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

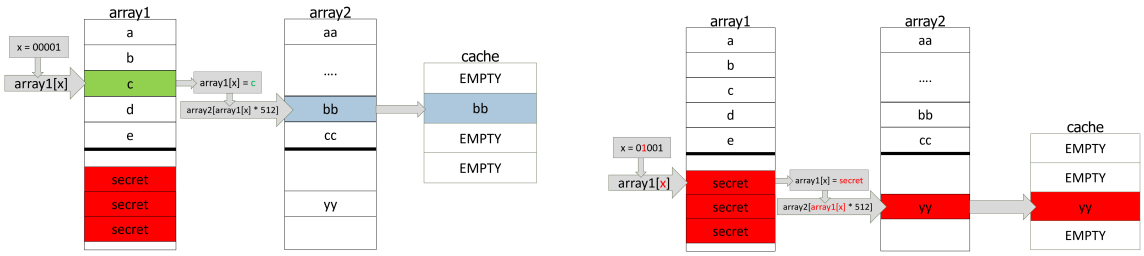

Figure 3.1: Example attack scenario. (left) Training phase with legal value. (right) Attack phase with malicious value.

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.

3.3 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.

3.3.1 Double Gadget Attack: Removing Attacker Control

As discussed in Section Section 5.2, 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.

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

Listing 3.2: 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 3.2 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 5.2, 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 3.5) 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 3.2 is taken. See Figure 3.1(left) for an illustration.

Step 3: Hammering and Misspeculation. Next, the attacker hammers `x`, leading to the state in Figure Figure 3.1(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 3.2 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 [41], the state of the CPU's cache is not reverted, resulting in a `secret`-dependent element of `array2` being cached. See Figure 3.1(right).

Step 4: Flush+Reload. To recover the leaked data from the speculative domain, the attacker uses a Flush+Reload side channel [87] 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`.

3.3.2 Triple Gadget Attack: Enabling Arbitrary Memory Reads

The attack presented in Section 3.3.2 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 3.3: Pseudocode triple gadget

Attack Overview. For the triple gadget attack, we utilize a new type of code gadget; see Listing 3.3 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 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 3.3.1, 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 3.3 is typically

taken.

Step 3: Hammering and Misspeculation. Next the attacker hammers `x`, leading to the state in Fig. Figure 3.2, 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 3.3.1, 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 3.3 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.

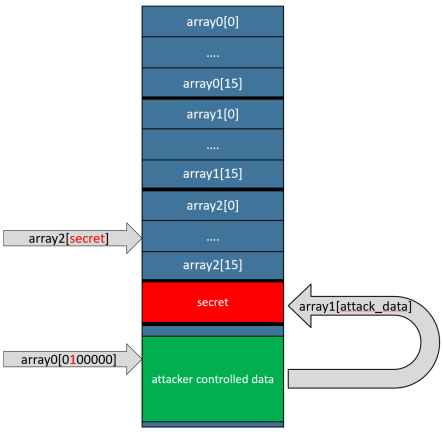

Figure 3.2: **Triple gadget example**

Step 4: Flush+Reload. Finally, as in the case of Section Section 3.3.1, 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 3.3. However, in Section Section 3.6.2 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.

3.4 Memory Templating

The high level description provided in Section Section 4.4 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 Section 3.5.

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 [45, 58] and giving a more detailed description in Appendix A.1.

3.4.1 Obtaining DRAM row indices from virtual addresses

As explained in Section Section 5.2, Rowhammer is drastically more effective when two aggressor rows that pinch a victim row are hammered in succession, a technique called double sided hammering [38]. 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 [58]. 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 [45], 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.1) 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 [20], 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. For two-DIMM configurations, it is possible to use memory massaging techniques to obtain 4MB of contiguous memory.

3.4.2 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 [45, 38], 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 [22, 89, 68, 80], 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 [22] 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.

Model	Samsung (DDR3)	Axiom (DDR3)	Hynix (DDR3)
rowhammer-test [68]	1	0	0
rowhammerjs [22]	4	9	2
rowhammerjs (corrected addresses)	15	38	32
rohammerjs with victim flushes	7,883	11,005	7,943

Table 3.1: Comparison between prior Rowhammer techniques and our new cache-flushing technique on DDR3. Note that rowhammerjs refers to the code in its "native" directory.

Model	Samsung (DDR4)	Samsung (DDR4)	Samsung (DDR4)
TRResspass [80]	947	2,976	2,134
TRResspass with victim flushes	7,916	17,958	15,611

Table 3.2: Comparison between prior Rowhammer techniques and our new cache-flushing technique on DDR4.

Under-reported Flip-rate in Prior Work. Upon inspection of numerous public Rowhammer repositories [22, 89, 68, 80] 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 A.3 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 3.1 and Table 3.2. 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 [22] 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 [20], 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 A.2.

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. The DDR4 DIMMs all have model number M378A1K43BB2-CRC.

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 [11, 37], they did so using techniques unavailable on general purpose machines. In the case of [37], 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, [11] 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 testing one's own machines for Rowhammer bugs, and show how flushing the victim row 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 A.3.

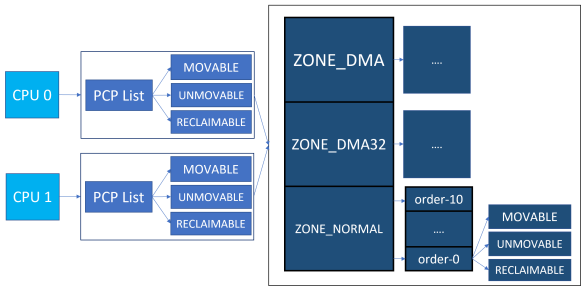

Figure 3.3: **Linux memory organization**

3.5 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 [62], 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 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[67]. 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, 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 are Android specific [75].

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.1), 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 Figure 3.3.

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 [21].

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 [21].

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[46].

PCP Lists. Finally, the PCP list (also referred to as the *Page Frame Cache*) [9] 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.

3.5.1 User Space Stack Massaging

Building on existing user space massaging techniques [45, 9], 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 [45, 9]. 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.

3.5.2 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 [67] 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.

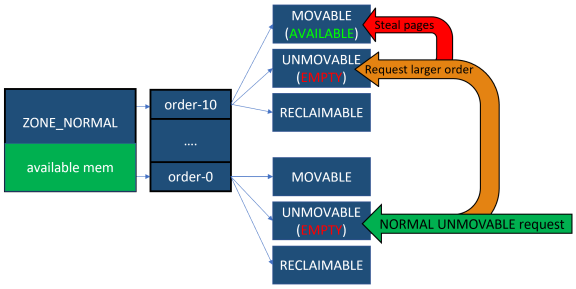

Figure 3.4: **Physical Page Stealing**

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. Figure 3.4, 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.1 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 Section 3.4, we have already guaranteed the target page is part of an order-4 (or larger) block. Therefore, we can free the target page and all of its buddies to ensure it will coalesce into the largest available block.

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

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 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.

3.6 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, respectively. These examples serve to verify the attack's ability to leak data.

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 5.1. Note that the two newer processors have additional defenses 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.

3.6.1 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 [13].

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 3.4: 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 3.4, 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 [56]. 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 [45].

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.

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 Section 3.4 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 [85]. 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.

3.6.2 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.

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

Listing 3.5: Triple Gadget

Example Victim. The example victim for this attack is a syscall in which we inserted a triple gadget, as shown in Listing 3.5. 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 [29, 16, 30], 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 Section 3.7.2 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 Section 3.5.2, 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 5.1.

3.7 Gadgets in the Linux Kernel

3.7.1 Gadget Search

Smatch. Smatch [47] 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[81], which is the only defense that can efficiently mitigate all forms of Spectre [6], 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.

3.7.2 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 [41], 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.

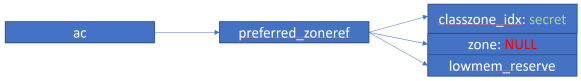

Figure 3.5: **alloc_context struct pointer**

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 3.6. 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 3.5, 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 3.6: 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 3.6, 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 3.6 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`.

3.8 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 [48, 74, 6], 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 [81, 6], is thwarted by the new combined attack, as it relies on a Spectre limitation not present in the combined attack. Other defenses [7, 61, 60] designed to protect against Spectre v1 provide incomplete protection, working only in specific cases, and often come at a prohibitively high performance cost [6].

Rowhammer. For Rowhammer, on the other hand, numerous hardware and software defenses have been developed to prevent or detect bit-flips, beginning with PARA [38]. 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 [20] 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 [37].

Another common hardware defense against bit-flips is error correcting codes (ECC). Ini-

tially 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 [12] 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.

CHAPTER 4

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

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.

4.1 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 [38] 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 [42, 31, 20] and how to exploit the flips to escape sandboxes [67], gain root privilege [67, 75, 76, 23], and leak secret keys [45], among other attacks [50, 69, 4, 62, 71, 19, 24, 90, 54, 58, 17].

However, a majority of these attacks focus on targeting *specific sensitive targets* [67, 23, 75, 76, 90, 23]. These prior works consider the dangers of bit flips in important structures such as PTEs [67, 76, 75, 90] and security-critical code (e.g. sudo password checks[23]). This led to various mitigation proposals that protect PTEs from bit-flips.[84, 92].

In contrast, few have considered more general targets Rowhammer can exploit to leak data. RAMBleed [45] demonstrated that attackers can hammer their own memory to leak individual bit-values of adjacent rows, and SpecHammer [71] 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 [84, 92], 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?

4.1.1 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 [84, 92]. 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 4.1: 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 4.1. 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 4.1) 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 4.2: 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 4.2 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 4.2). 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.

4.1.2 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 [4, 67, 71]. 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[45, 71].
Contributions. We make the following contributions:

- Identifying a new class of code patterns that can be exploited as a Rowhammer gadget. (Section 4.4.1)
- Improving memory massaging techniques for flipping kernel bits (Section 4.4.2).
- A novel technique for testing for kernel flips (Section 4.4.3).
- New synchronization techniques (Section 4.4.3).
- Performing an end-to-end attack on a Linux Kernel syscall, demonstrating a leakage rate of 82.6 bits/s (Section 4.5).

4.1.3 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.

4.2 Background

4.2.1 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 [71] and subsequent work in the speculative domain [6]. 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 [41, 3, 65] or perform arbitrary code execution [52], and retrieve

speculatively read data via a covert-channel. Spectre is difficult to mitigate [6] as it not only leverages performance-critical processor features (branch prediction), but can target any victim containing a gadget.

4.2.2 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.

4.2.3 Caching

Cache Interference. One important consideration with Rowhammer is cache interference. Rowhammer requires repeatedly 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` [24] 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[78].

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 [56] 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.

4.2.4 Rowhammer

Exploits. Rowhammer [38] demonstrated how attackers can flip bits in memory without accessing them, spawning a plethora of new attacks taking advantage of these bit-flips [67, 23, 62, 45, 71, 91, 69, 19, 50, 75, 12, 24, 14, 4]. Most notably, the first exploit [67] 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 [75, 76], and attacks that flipped PTEs through the browser [24] among others [54].

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 [20, 42, 31]. Next-generation DDR5 DIMMs use a new technique called refresh management (RFM), but even this has shown to be inadequate [53]. Software defenses have sought to protect sensitive structures, by, for example, placing buffers between user space and kernel space [5] or placing PTEs in flip-safe memory [84].

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 [33]. 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 [20] 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 [42, 31, 14] that all demonstrated new techniques for flipping bits on DDR4, revealing that these new DIMMs were even more vulnerable to bit-flips underneath TRR.

4.2.5 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 "massages") a physical memory allocator into a state that is likely to serve its next allocation request using the flip-vulnerable page [62, 9, 75, 45, 71]. 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) [9]. 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 [67]. For an attack on kernel stack variables, kernel stacks can be allocated by creating new threads, which each allocate their own kernel stack [71].

4.3 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.

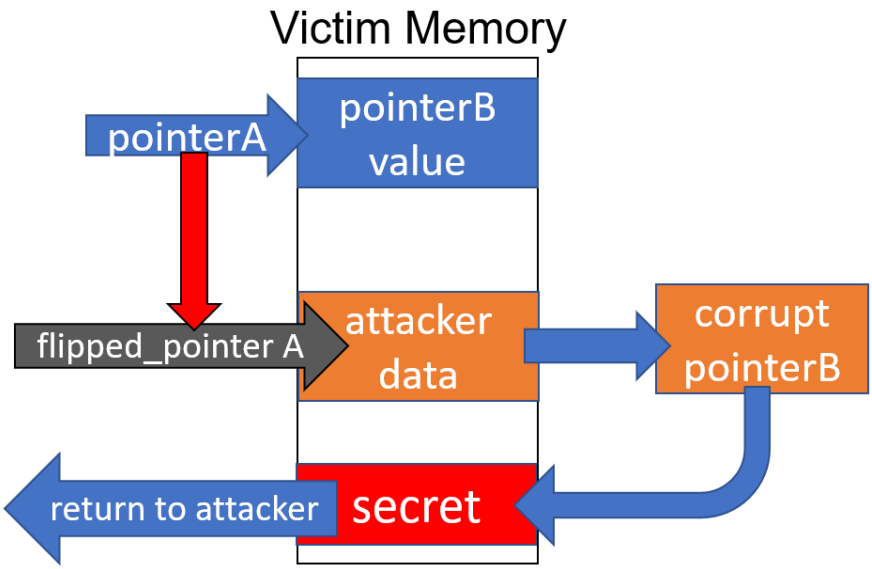

Figure 4.1: **Flipping a pointer to return secrets to an attacker.**

4.4 GadgetHammer

4.4.1 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 4.1. 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 4.1 the gadget can be exploited if we assume the attacker has the capability of flipping `pointerA` (Line2, Listing 4.1) 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

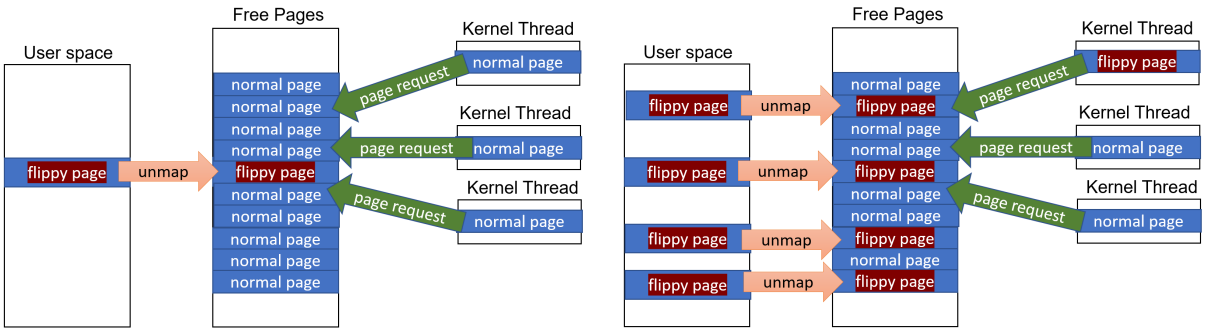

Figure 4.2: **Improving memory massage probability.** The left side shows the lower chance that a kernel stack will allocate a flip-vulnerable page if there is only one flip-vulnerable page in memory. The right side shows how these odds can be improved if more flip-vulnerable pages are freed before forcing kernel stack allocations.

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.

4.4.2 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 [45, 67, 20], allocating transparent huge pages, and hammering groups of addresses until finding useful flips, relying on TRRespass [80] 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 [71]. 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 4.4.3). 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 4.2, finding numerous flippy pages, freeing them all into the page allocator, and then forcing kernel stack allocations, can greatly increase the probability that a flippy page can land in the kernel.

4.4.3 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 [42], 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 4.3 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 vari-

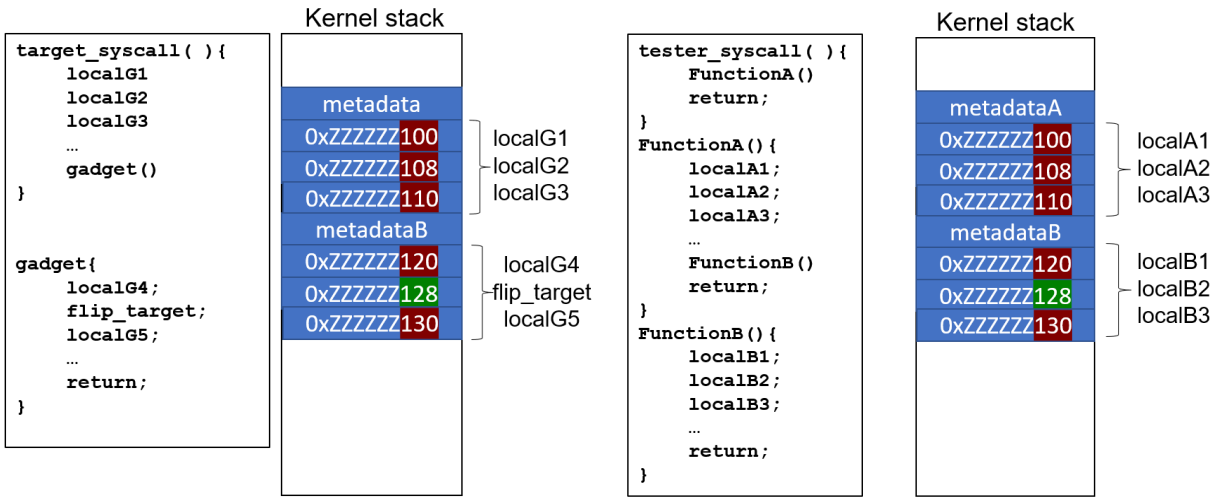

Figure 4.3: **Function calls placing data on the stack.** The left side shows the target gadget's stack, and that the variable we seek to flip happens to reside at page offset 0x128. The right side shows the tester gadget, which stores variables at a similar stack depth, also storing a local variable at page offset 0x128.

ables 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 4.5.1, 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 [78]. Instead of directly flushing our victim from the cache, we will fill the same cache set with arbitrary data, forcibly 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 [71, 16]. 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 [28]. 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 4.5.1).

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 4.4.2, 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 single 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.

4.4.4 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 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 4.5), we focus on injecting our data into the kernel heap, noting that injecting data into the stack is possible as well [71, 16].

Spraying the Heap. The first question is how to inject heap data. Prior works have explored different techniques for heap spraying [88], 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 [30]. 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 4.4. 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

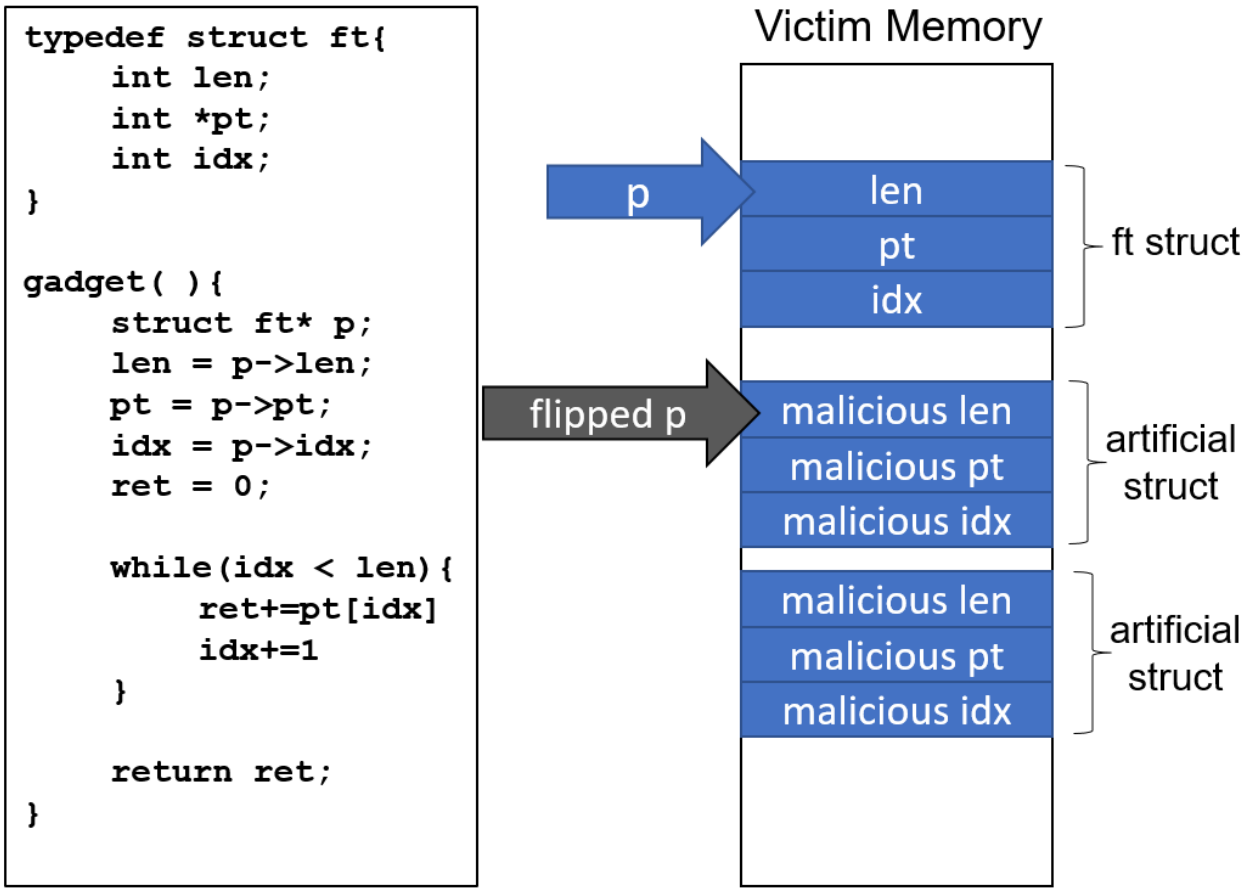

Figure 4.4: **Filling victim memory with artificial structs that contain attacker data.** Flipping a struct pointer to point to our structs gives us control of the true struct's member variables.

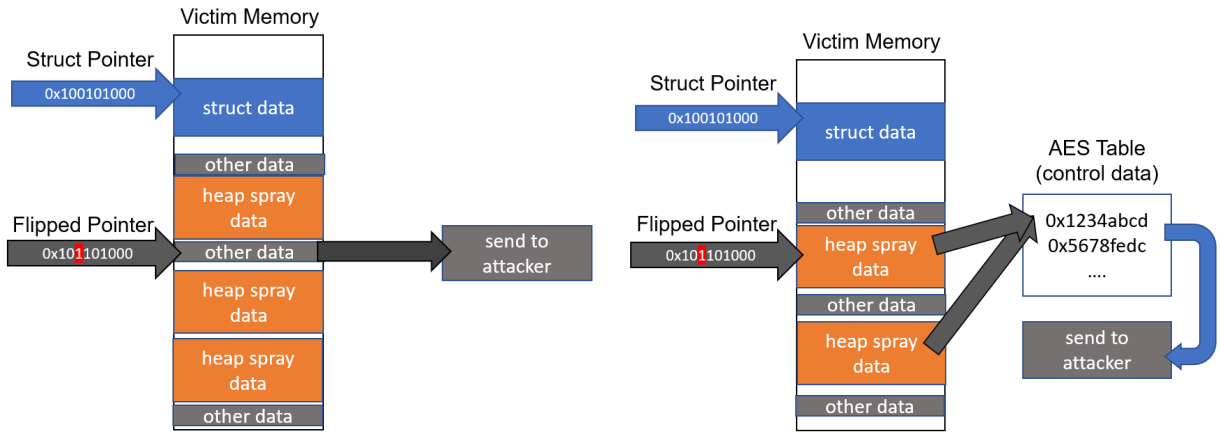

Figure 4.5: **Confirming heap spray results.** The left side shows that if our heap sprayed data does not land at an address one bit-flip away from the victim, the gadget will return junk values. We repeat the heap spray until we can read out our "control" data from the kernel, as shown on the right side.

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 4.5.1 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 4.5). 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 [64].

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 4.5.2 for an empirical evaluation on a chosen example gadget present in the kernel.

4.5 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.

4.5.1 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 4.3. 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 4.3: 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 (`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 4.4.3, 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 4.4. 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 4.4: 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 4.3, 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 4.4, 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.

4.5.2 Attack Execution

Experimental Setup. Having identified our targets, we conducted the attack following the steps laid out in Section 4.4. 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 [67].

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.

4.5.3 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 [59] 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.

4.5.4 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 [47] 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 4.5. We note smatch's known deficiencies in reporting both false positives [6] and false negatives [71], but believe it is the best option for detecting gadgets on the Linux kernel [8]. We leave the development of a more precise tool for future work.

4.6 Prior Attacks, Mitigations, and Conclusion

4.6.1 Prior Attacks

The first Rowhammer exploit demonstrated how the phenomenon could be used to flip PTE bits and give an attacker root privilege [67], and numerous follow-up studies have

demonstrated new techniques for reproducing PTE flips under various attack scenarios [75, 76, 24, 54]. 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 [20] or suggest alternate structures to target [23]. GadgetHammer is, to the best of our knowledge, the first work to consider targeting gadgets.

4.6.2 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 [6] 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 [25], or lack full coverage of all possible code paths, relying on approximate techniques such as fuzzing [55]. Second, defenses designed to protect against specific patterns can be bypassed by exploiting slight variations [40].

Software Mitigations for Rowhammer. Numerous attempts have been made at preventing exploits via software. For example, CATT [5] 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 [23]. Other defenses use a more direct approach, storing only key kernel structures, specifically PTEs, in flip-safe memory [84]. 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 5.2, both DDR4's Target Row Refresh [33], and DDR5's Refresh Management [34], have proven to be inadequate [53, 20].

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 [63, 82, 18, 83]. However, such mitigations require area and performance overheads unlikely to be relinquished by manufacturers. Additionally, RAMBleed [45] and ECCsploit [12] 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.

CHAPTER 5

Exploration of End-to-End Attacks

Since its discovery in 2014, the Rowhammer bug has remained a largely unmitigated problem on modern computers. More recent generations of DRAM (i.e., DDR4 and DDR5) have designed and included built-in defenses in an attempt to prevent bit-flips—but to no avail, as numerous works have demonstrated various ways of effectively flipping bits even in the face of these new defenses. However, while all DDR4-hammering works demonstrate the ability to find exploitable flips, few demonstrate the ability to execute end-to-end attacks. Indeed, finding vulnerable bit-flips is only the first of multiple steps in the attack chain required for Rowhammer exploits, raising an important question: "how dangerous or threatening are these bit-flips on machines equipped with DDR4?" The key component missing from prior DDR4-hammering is the ability to *massage memory* and ensure that a sensitive target victim would actually use a memory address that is vulnerable to bit-flips. Without forcing a victim to reside on a flip-vulnerable address, there is no way to trigger malicious exploits. We find that achieving successful memory massaging on DDR4 is a non-trivial effort—one that has been neglected in favor of works focused only on generating bit-flips more efficiently on DDR4.

Our work thus aims to bridge the evident gap between flipping bits on DDR4 and Rowhammer's ultimate purpose of functioning exploits. We achieve this goal by developing advanced memory-massaging techniques that allow DDR4 bit-flips to be useful for striking sensitive targets in memory, making kernel-level, root-granting Rohwmmaer exploits on DDR4 machines possible. Using these new techniques, we are able to perform the first page-table-flipping exploits on Intel CPUs using DDR4, and boost kernel massaging accuracy on DDR4 from about 0.65% to about 23% for an approximately 35x improvement. With this new memory-massaging technique, our goals are twofold. First, we hope to increase the value of the hammering techniques presented in prior work by showing DDR4 bit-flips can be used for real-world exploits. Second, we aim to emphasize the need for future Rowhammer research to consider the compatibility of their methods with memory-massaging techniques

and kernel-level exploits since if it turns out a technique is incompatible with memory massaging, the technique is incapable of exploiting real-world exploits and hence poses no real threat.

5.1 Introduction

Recent trends in computer hardware technology have pushed for improving performance in the common case at the risk of introducing potential security vulnerabilities in fringe cases. As the last several years of security research has shown, reducing area, power, and time costs for common operations has introduced unintended side-effects that allow attackers to breach security with relative ease [54, 3, 41, 62, 75, 58, 38, 20, 32, 56, 6]. One such case in particular has introduced a vulnerability in DRAM that has gone unsolved since its discovery over a decade ago.

In recent years, DRAM modules, or "DIMMs" (Dual In-line Memory Modules) have packed an increasing number of capacitors within smaller and tighter areas, leading these capacitors to induce disturbance effects on each other, making them ripe for exploits via unintended side-effects. In particular, Rowhammer [38] has shown that an attacker can rapidly access rows of memory she controls in order to induce disturbance effects that can flip bits in addresses she normally would not be able to access at all.

This newfound ability to flip target bits at will via Rowhammer has spawned a plethora of follow-up works exploring new ways to induce bit-flips and exploit them for malicious attacks. For example, while the first Rowhammer paper demonstrated bit-flips on DDR3 on a general-purpose computer, the following years showed it is possible to flip bits on mobile devices [75, 76], through the browser [14, 24, 35], via remote connections [69], and even on DDR4 [31, 20, 42, 35] and DDR5 DIMMs [32]. Additionally, novel exploits have shown various ways an attacker can leverage a bit-flip to gain root access or leak sensitive data, by, for example, flipping page table entries (PTEs) [67], bits in the sudo binary [23], pointers and array indices, [71, 73] or bits adjacent to secret key bits [45].

Among these various exploits, the most popular and common exploit used to demonstrate the dangers of bit-flips is the original PTE attack demonstrated by Seaborne and Dullien [67]. This requires flipping a bit at a particular offset such that a PTE points to an incorrect address, enabling an attacker to read and write from arbitrary addresses in memory. Works that successfully achieve bit-flips on new hardware, such as DDR4, will often demonstrate the effectiveness of their work by reporting how long it takes to find a bit-flip that can be used for a PTE attack [20, 31, 42]. However, to the best of our knowledge, of these various techniques used for flipping bits on DDR4, there have been no successful PTE attacks on a

DDR4 machine when using an Intel processor. Other attacks have demonstrated alternate ways of targeting the kernel on DDR4 besides flipping PTEs, such as GadgetHammer [71], which flips kernel stack bits. However, such techniques demonstrate detrimentally low-accuracy memory massaging (0.65%) leading to unreliable attacks that must be repeated for dozens of hours before fully executing their exploits.

Indeed, with the exception of ZenHammer [32], which reports an end-to-end PTE attack on AMD Zen CPUs, prior works demonstrating the ability to achieve a PTE flip, have not yet demonstrated an end-to-end PTE attack. The focus of Rowhammer research of this nature has largely been on generating bit-flips on new hardware, either providing the first DDR4 flips [20], flipping bits on previously unflippable DIMMs [31], or generating flips more efficiently [35]. While these works of generating new flips are valuable, they only represent the first stage of a Rowhammer attack. By contrast, little has been done to demonstrate the use of these flips in an end-to-end PTE exploit.

Yet, when it comes to Rowhammer attacks, the work of positioning a flip-vulnerable address in an exploitable location in memory is of equal importance to inducing those bit-flips in the first place. Indeed, a bit-flip is only dangerous if it can be used to flip an sensitive victim. For example, numerous Rowhammer defenses are built on the very idea that flips can occur in the machine if memory is organized in a way such that those flips will never reach sensitive code or data [5, 84]. However, it is currently unclear whether many of the existing DDR4 Rowhammer techniques are capable of positioning flip-vulnerable addresses as needed for these attacks, thus making it unclear whether these bit-flips actually pose a threat in the realm of DDR4.

These observations thus raise the following important questions we seek to answer in this paper: *Can we practically execute PTE exploits on DDR4 DIMMs? How effective are the various DDR4 Rowhammer techniques for end-to-end Rowhammer attacks?*

5.1.1 Our Contributions

Our goal is to fill the gap in current Rowhammer research that lacks the development of end-to-end exploits for DDR4. We find memory massaging on DDR4 to be a non-trivial challenge, understandably not included in prior DDR4 work, and develop techniques to overcome the associated barriers. In particular, we take on the task of bridging the gap between inducing bit-flips on a DDR4 machine and gaining root access on said victim machine, practically demonstrating the threat posed by Rowhammer on DDR4.

PTE Massaging. We find the main challenge in achieving end-to-end exploits is tied to *memory massaging*. While prior works have successfully induced flips at the bit-offsets

needed for PTE attacks, the essential work still remains to "massage memory" such that PTEs are allocated on these pages with the correct flip-vulnerable offsets. As demonstrated in prior works, [62, 71], memory massaging is non-trivial, and comes with its own challenges when attempted on DDR4 machines. We achieve the first successful PTE massaging on a DDR4 machine, aside from ZenHammer [32] which has only been demonstrated on AMD Zen CPUs.

Improved Massage Accuracy. In addition to achieving PTE massaging, we also demonstrate an improved massaging accuracy for other techniques over prior works. In particular, our techniques are shown to achieve up to 23% accurate kernel stack massaging as compared to the 0.65% accuracy reported in prior work.

Re-evaluation of Prior Works. Lastly, since the focus of prior works was to generate bit-flips, there is currently a lack of understanding on how compatible the various existing DDR4 techniques are with PTE massaging. Memory massaging, particularly in the kernel, is a complex process that requires fine-grained control over the victim machine's memory allocator. Therefore, it is very sensitive to the state of the victim machine's physical pages. Compared to DDR3 techniques, DDR4 techniques require controlling a larger number of aggressor pages to induce bit-flips. More recent works have shown that techniques that utilize larger numbers of aggressors can induce an increased number of bit-flips [35].

However, we find that additional requirement of physical memory, i.e., the increased number of aggressor addresses, results in a worse memory massaging performance. We observe that the *oldest* DDR4 Rowhammer technique [20] outperforms the latest technique [35] when evaluated by massaging accuracy, with a 23% accuracy for the former, and 11% accuracy for the latter. Considering memory massaging is often the slowest (or most time-consuming) stage of a Rowhammer attack [71, 73], for such attacks, the end-to-end attack time needed for older techniques can drastically outperform those of newer techniques. The memory massaging stage leads some attacks to require up to 24 hours to complete, giving large windows of opportunities for defenses to detect suspicious activity and half attacker processes before they can cause any harm. Thus, by drastically increasing the efficiency via faster memory massaging, we can better understand which techniques pose a real-world threat know which techniques to use as a basis for future offensive Rowhammer research. With this result, we stress the need for future work to consider techniques compatible with end-to-end attacks, rather than *only* considering how to induce as many bit-flips as possible.

5.1.2 Challenges

Achieving practical end-to-end results with DDR4 hammering techniques requires overcoming several key challenges regarding memory massaging:

- **Challenge 1:** We must thoroughly understand any sources of interference that might affect memory massaging.
- **Challenge 2:** We must work under the restriction of transparent hugepages.
- **Challenge 3:** We must suppress allocations from background processes that attempt to reclaim memory when under high pressure.

Challenge 1: Identifying All Sources of Interference. The first challenge is to develop a thorough understanding of why prior memory-massaging techniques no longer work on DDR4 machines. Existing memory-massaging techniques have worked reliably for DDR3 Rowhammer exploits and demonstrated end-to-end results on older victim machines [67, 62]. However, the same techniques no longer work on newer hardware. Intuitively, this should not be the case, since memory massaging is a process of manipulating an operating system's physical page allocator. In the case of Linux and its allocator (known as the *buddy allocator*), the same operating system version and same allocator can be used on both DDR3 and DDR4 machines. Thus, the same massaging technique can be used on a DDR3 machine and a DDR4 machine with the same physical page allocator, but will work on the DDR3 machine and not the DDR4 machine.

Thus, if we hope to achieve accurate and reliable memory massaging, we must achieve a better understanding of what other components of the victim machine can affect the memory-massaging process and mitigate their interference. We find that the allocation policy of the CPU itself can affect the chances of success of a particular massaging technique, leading newer machines (with newer CPUs) to require different techniques than older ones. With this newfound understanding of the CPUs interference, we have developed a massaging technique that works around the newer CPUs "lazy allocation" policy, and achieve successful massaging on newer machines, as discussed in Section 5.4.1. Additionally, we find the hammering techniques used for DDR4 can also induce interference in the massaging process due to the increased aggressor address requirements. We present a more accurate massaging to overcome these requirements in Section 5.4.

Challenge 2: Overcoming Hugepages. DDR4 hammering techniques rely on controlling numerous aggressor addresses. As explained in Section 5.2, these addresses typically must be within the same bank and must be contiguous. By default, the buddy allocator does not fulfill requests for contiguous chunks of memory with contiguous chunks of physical pages. Instead, it returns fragmented physical pages that cannot be reliably used for hammering.

To work around this, DDR4 techniques have relied on the use of hugepages [80, 14], which guarantee large blocks of contiguous memory. However, we find that within the buddy allocator, hugepages are managed differently than normal pages, which, in turn, hinders prior works' massaging techniques from working. We thus develop techniques that can successfully massage victim memory even when aggressor and victim addresses are contained within hugepages, allowing for memory massaging under the hugepage restriction of DDR4 Rowhammer.

Challenge 3: Reducing Allocation Noise. Kernel memory massaging techniques typically require incurring high pressure on victim memory. Such techniques allocate many pages from victim memory in order to decrease the pool of remaining free pages and control the state of this pool of free pages more easily. However, when in this state of high memory pressure and low page-availability, many processes will begin to give up their own physical pages temporarily into swap space. This means many processes running on the victim machine will have all their virtual memory temporarily not backed by any physical pages. The moment any of these processes need to access their own memory, they will trigger a new allocation. This causes high traffic on the pool of remaining free pages, which creates interference with the memory-massaging process that needs to delicately control the state of the allocator, resulting in worse memory-massaging performance. We introduce techniques for identifying sources of allocation noise and reducing their effect on the massaging process, resulting in an improved accuracy of 23% compared to the 0.65% of prior work [73].

Summary of Contributions. This paper makes the following contributions:

- Developing the first memory-massaging technique allowing for PTE attacks on DDR4 machines other than AMD Zen processors (Section 5.4.4).
- Improving memory-massaging accuracy for DDR4 Rowhammer attacks and thus demonstrating Rowhammer's threat on DDR4. (Section 5.4.4).
- Novel techniques for overcoming memory-massaging challenges unique to DDR4 (Section 5.4.1).
- A re-evaluation of prior Rowhammer techniques while considering their efficiency and practicality for end-to-end exploits (Section 5.5).

5.2 Background

This section provides a primer on technical areas relevant to the paper, including background on Rowhammer, how Rowhammer changed for DDR4, and memory massaging as used in Rowhammer exploits.

5.2.1 Rowhammer on DDR3

The Rowhammer bug [38] showed that attackers can flip bits in memory without needing to access them directly. The bug exploits the physical nature of DRAM and its reliance on capacitors. DRAM stores values via arrays of capacitors. A charged capacitor stores a single bit value of 1 and a discharged capacitor stores a 0 (or vice versa, depending on that cell's configuration). When left on their own, capacitors will naturally leak charge over time. Thus, DRAM will periodically refresh its capacitors to combat this natural leakage and prevent any potential data loss.

However, when a user accesses a value in DRAM, the capacitors corresponding to that value are temporarily discharged, as the charge is moved into a row buffer and passed along to the calling user. Once the access is complete, the capacitors are restored back to their original charge. This means each memory access induces a momentary discharge and recharging of a selected set of capacitors in DRAM. The original Rowhammer work showed that due to the close proximity of capacitors on DDR3 DIMMs, this quick movement of charge can induce *disturbance effects* on adjacent capacitors. It was shown that these disturbance effects can slightly accelerate the leakage rate of any affected capacitors. Consequently, if an attacker rapidly accesses a particular row of memory repeatedly, she can induce enough disturbance effects on adjacent capacitors to accelerate their leakage rate and drain their charge. If the charge is drained quickly enough such that the capacitor's charge drops below the threshold value signifying a 1 versus a 0 (before the periodic charge refresh), then the capacitor's bit-value will flip from 1 to 0 (or vice versa). Thus, an attacker can flip bits in memory by rapidly accessing adjacent addresses, allowing attacker to flips bits in addresses without needing to ever access them directly.

DRAM Organization. To target specific victim addresses in memory, it is essential to have an understanding of how values and addresses are organized in DRAM. At the highest organizational level, DIMMs slotted into a motherboard are grouped into channels. The next organization level consists of "ranks" which represent the physical front and backside of the DIMM. Within each rank, addresses are sorted into "banks", where each bank is a chip of addresses packed closely together. Banks themselves are sorted into "rows", which represent physical rows of capacitors. As mentioned above, a single DRAM access loads an entire row's charge into a buffer before passing this charge along to the CPU.

Double-Sided Rowhammer. Taking advantage of DRAM's layout, [38] showed that it is possible to increase the likelihood of a bit-flip by targeting a pair of aggressors that surround a single victim row. This technique, known as *double-sided* Rowhammer was shown to drastically increase the number of bit-flips induced in victim DIMMs. It has been shown it is possible to reliably achieve bit-flips with single-sided hammering as well [23] for cases in

which double-sided Rowhammer is not feasible.

Virtual Addressing and DRAM Addressing. In order to perform double-sided Rowhammer, attackers must target multiple rows within the same bank that surrounded a specific target victim. One challenge of this requirement, however, is that DRAM addresses are not exposed to attackers. Unprivileged users can only view *virtual* memory addresses. Upon accessing a virtual address, the operating system translates the virtual address into a *physical* address containing the corresponding value. Physical addresses are, in turn, translated into *DRAM addresses* that point to a specific, corresponding channel, rank, bank, and row. This translation is handled by the CPUs memory controller, which uses a fixed hash function that maps physical addresses to DRAM.

Overcoming Hidden Addressing Functions. To work around the issue of virtual addresses mapping indirectly to DRAM addresses, researchers have developed techniques for both reverse engineering virtual-to-physical mappings [62] and physical-to-DRAM mappings [58].

For the latter, a timing side-channel technique, that takes advantage of DRAM's row buffer system, is used. Since DRAM accesses require moving charge into a row buffer, and there is only one row buffer per bank, repeatedly accessing multiple rows within the same bank will create a slight timing delay as the two addresses contend for the same row buffer. Conversely, accessing two rows in different banks will be relatively quick, since each bank can keep the same values within the row buffer, repeatedly serving each access the same row buffer values, without needing to spend time to move charge from various addresses into the row buffer and back. Thus, by timing accesses, and tracking physical address bits, attackers can learn how each physical address bit corresponds to DRAM addresses.

On DDR3-compatible CPUs, the physical-to-DRAM mapping functions used only the lower 21 physical address bits to determine channel, bank, and rank. Thus, attackers while attackers could only directly observe virtual addresses, they only needed to obtain the lower 21 bits of physical address to have full control over their DRAM accesses. Prior work [62, 45] has shown that the physical page allocator can be manipulated to provide memory-aligned, contiguous chunks of memory large enough such that the lower 21 physical address bits match the lower 21 virtual address bits, providing the exact information needed for virtual-to-DRAM address mapping.

Exploits After this new ability to flip bits in memory was discovered, a variety of exploits were developed around the bug. Exploits were demonstrated both on userspace memory [45, 23, 69] and kernelspace memory [67, 71] on DDR3. The first Rowhammer exploit [67] demonstrated how attackers can flip page table entries (PTEs) which are responsible for mapping virtual addresses to their corresponding physical addresses. By flipping bits in

PTEs, an attacker can steer a PTE to incorrectly point to another PTE, allowing attackers to overwrite this latter PTE with arbitrary data, and then use this overwritten PTE to gain access to any address in memory. The PTE exploit remains the most common exploit used to demonstrate proof-of-concept Rowhammer attacks in prior work [75, 76, 20, 24, 14, 42, 31, 32].

5.2.2 Rowhammer on DDR4

New Mitigations. Being aware of the Rowhammer flaw present in DDR3, the standards for DDR4 were designed with anti-Rowhammer protections built-in [33]. The built-in DDR4 Rowhammer defense, referred to as *target row refresh* (TRR), tracks repeated memory activations in DRAM. A counter is used to track activations of pairs of DRAM rows. If the counter detects that a pair of rows are activated repeatedly above a fixed threshold value, the addresses adjacent to these rows will be given an early refresh. This way, any charge drained from potential Rowhammer victims will be restored before their bit-values are flipped. Thus, if attackers attempt to hammer a pair of aggressors, TRR will protect the potential victims.

Multi-Sided Hammering. Soon after DDR4 was released to the public, it was quickly shown it is indeed possible to flip bits via Rowhammer on DDR4, even in the presence of TRR. As demonstrated via TRRespass [20], a fatal flaw in TRR's implementation is that it can only track a single pair of rows at a time. If an attacker accesses a pair of aggressor rows, leads the TRR counter to track those rows, and then switches to hammering a new set of aggressors, the counter will reset and begin counting the activations on the new set. Thus, TRRespass introduces a technique called "n-sided" (or *multi-sided*) hammering, in which an attacker accesses numerous rows within the same bank. By alternating accesses between a pair of "true aggressors" and numerous "dummy aggressors", the dummy accesses will cause the TRR counter to be repeatedly reset, allowing the true aggressors to always remain below the TRR activation threshold and never trigger an early refresh. TRRespass found that some DDR4 DIMMs flipped with patterns as small as 3-sided hammering, while others required up to 19-sided hammering. Furthermore, TRRespass found that after getting past the TRR mitigation, DDR4 DIMMs were even more vulnerable to Rowhammer bit-flips, due to their tighter packing of capacitors.

Hugepages. With the first DDR4 hammering technique requiring large contiguous blocks of memory to support multi-sided hammering, it became common for DDR4 Rowhammer techniques to rely on hugepages [14, 80, 31, 71, 73]. Existing repositories for DDR4 Rowhammer techniques rely on mounting a 1GB hugepage with sudo privilege [80], while unprivileged malicious attacks use transparent huge pages (THPs) [14, 71, 73], which allow for allocat-

ing 2MB blocks as an unprivileged user. Prior techniques of manipulating the allocator to provide contiguous blocks are still possible on DDR4 machines [14, 35], but have been made more challenging to execute due to Linux restricting key files that were useful for such techniques [71]. Thus, hugepages have become the more common method for obtaining contiguous memory in prior DDR4 Rowhammer works.

Multi-Banking. After TRRespass, follow-up works explored ways to improve the efficiency of bit-flips on DDR4 [31, 53, 35]. For example, Blacksmith [31] demonstrated that timing side-channels could be used to synchronize Rowhammer activations with DRAM's periodic refresh cycles. By beginning multi-sided hammering at the start of a refresh cycle, Blacksmith reported a larger number of flips than TRRespass on their tested DIMMs. Most recently, Sledgehammer [35] presented a new technique called *multibanking*. This technique stems from the observation that consecutive DRAM accesses in separate banks happen considerably faster than repeated accesses to the same bank. By distributing Rowhammer aggressor accesses across multiple banks, attackers can drastically increase their total number of aggressor row activations within a given time span, and consequently produce a larger number of bit-flips than when striking a single bank at a time. Thus, multibanking currently represents the most efficient way to produce bit-flips on DDR4.

Exploits. DDR4 Rowhammer works continued the trend established by DDR3 works [67, 75, 76, 24] of using PTE bit-flips to demonstrate the potential risks of the flips they could induce on DDR4 DIMMs [20, 31, 14, 42]. However, it is important to point out that unlike the DDR3 works, the DDR4 works have not yet reported any executions of working end-to-end PTE attacks. With the exception of ZenHammer on AMD Zen processors [32], the other works that aimed to improve the efficiency of bit-flips only reported the time needed to find a bit-flip at an offset that could be used for a PTE attack. No prior works have reported a working PTE attack on Intel processors on DDR4. More broadly, the only works to have demonstrated any end-to-end attacks on the kernel on DDR4 machines are ZenHammer [32] and GadgetHammer [73], where GadgetHammer focuses on flipping kernel stack variables. Thus, these prior works have demonstrated the possibility to generate the bit-flips needed, but not the possibility to perform the following stages of a PTE, such as "memory massaging", flipping PTE values, or gaining root over the victim machine.

5.2.3 Memory Massaging

Since Rowhammer bit-flips are largely a result of the physical nature of a given DIMM, attackers do not have direct control over exactly which bits they can flip. While they *can* target specific addresses using the techniques mentioned above, there is *no guarantee* bits in

those targeted victims will flip. Even two DIMMs of the same model, make, and manufacturer may exhibit different behavior due to process variation between the DIMM that may cause, for example, capacitors at certain addresses to be slightly closer together than they might be on other DIMMs, or slightly leakier. These minuscule differences are sufficient to cause a bit at a particular address to either be vulnerable or immune to Rowhammer bit-flips. Given these limitations, attackers have developed techniques and a multi-step approach for flipping specific victim values with high precision, despite the relative lack of control over which bits might be vulnerable to flips. These techniques are referred to as *memory templating* and, of particular interest to our work, *memory massaging*.

Memory Templating. While attackers cannot know beforehand which bits on a given DIMM might be vulnerable to flips, once a flip-vulnerable address is found, it is highly likely that the flip will be reproducible when striking the same aggressor pattern repeatedly [38]. Thus, the uncontrollable nature of Rowhammer bit-flips can be overcome by first "profiling" a DIMM and determining which addresses are vulnerable to bit-flips, and what the exact bit-offsets of those flips are. This profiling step, called *memory templating*, typically consists of allocating a large chunk of contiguous memory (either via hugepages [80] or allocator manipulation [62, 45]) and then repeatedly hammering sets of aggressors while checking their corresponding victim addresses for flips. The set that produces flips can be saved as an aggressor-victim pair to be used for the rest of the exploit, since striking those aggressors is likely to induce bit-flips in the corresponding victim.

Userspace Memory Massaging. After finding addresses with useful bit-flips, attackers must force target victims (e.g., PTEs) to reside on these flip-vulnerable addresses. This is usually done by manipulating the physical page allocator and forcing a victim allocation such that the page allocated to the victim is the flip-vulnerable page; a process known as *memory massaging*. For victims that reside in userspace (e.g., targeting a bit-flip on a secret key that resides in userspace [45]), memory massaging takes advantage of the page frame cache, also called the per-cpu (PCP) cache [9, 45]. This cache uses a first-in last-out (FILO) queue for storing recently freed pages. Each CPU core maintains its own PCP cache, meaning userspace memory massaging is as simple as freeing a flip-vulnerable page (putting it at the top of the PCP cache) followed by immediately forcing a victim page allocation from the same CPU core (pulling the flip-vulnerable page from the top of the PCP cache). Thereby, a target victim value is backed by a flip-vulnerable physical page, meaning attackers can hammer their aggressor pages to induce flips in the victim at will.

Kernelspace Massaging on DDR3. Relative to userspace massaging, forcing a kernel target to use a flip-vulnerable page is a rather complex process. Allocation requests from the kernel are typically served from a pool of pages reserved specifically for the kernel,

while userspace requests pull from an entirely different pool of free pages [71]. Since memory templating requires an attacker to allocate and hammer her own addresses, the flip-vulnerable page she identifies will likely be a userspace page. This means if this page is freed, it will join the pool of free userspace pages and not be allocated via kernel page requests, preventing kernel variables from allocating and residing on the flip-vulnerable page. However, if the pool of free kernel pages is depleted, the buddy allocator will resort to "page stealing", i.e., pulling pages from the available userspace pool and converting them into kernel pages, or vice versa. Pages belong to contiguous blocks called "orders" which are sorted by size. When stealing pages, the kernel will attempt to steal the largest order of pages (i.e. an entire 4MB contiguous block), and if none are available, will resort to stealing from the next largest available order.

Thus, the approach to memory massaging is centered around draining memory to deplete the pool of available kernel pages and higher-order userspace pages, freeing the flip-vulnerable userspace page, and then forcing a kernel allocation that will steal the flip-vulnerable page from the userspace pool. This technique was demonstrated on various DDR3 attacks for massaging both PTEs [76, 75, 67], as well as for massaging kernel stack variables, such as array inidicies [71].

Kernelspace Massaging on DDR4. Relative to DDR3, there is a lack of demonstrable PTE massaging on DDR4 machines. ZenHammer [32] reports an end-to-end PTE attack on AMD Zen platforms, but does not report any unique methodologies to PTE massaging different from what worked for DDR3 PTE massaging. Additionally, prior works that report the ability to find the flips needed for a PTE attack [31, 20], do not demonstrate PTE massaging. One work, which does achieve PTE bit-flips [42] does not use a templating-massaging approach, but rather hammers kernel memory directly in hopes that a PTE will naturally reside on a flip-vulnerable page. Other works, demonstrate successful kernel stack massaging [71, 71], but not cases of PTE massaging on DDR4.

5.3 Threat Model

We assume native code execution running locally on the victim machine and no elevated privileges. We also assume no mounted hugepages and default settings allowing transparent hugepages. Lastly, we assume the victim machine uses a DIMM vulnerable to Rowhammer attacks.

Configurations	CPU	DIMM Model	DIMM Size	Operating System	THPs Enabled?
DDR3 Machine	i7-4770	M378B5273DH0-CH9	2x 4GB	Linux 5.4.1	madvise [default]
DDR4 Machine	i9-9900K	M378A1K43BB1-CPB	1x 8GB	Linux 5.4.1	madvise [default]

Table 5.1: List of configurations used for our control experiments. Machines are configured to be under similar conditions to reduce variables in comparison of massage process.

5.4 DDR4 Memory Massaging

With the prior Rowhammer techniques [20, 31, 35], we are well equipped to flip bits on DDR4 DIMMs. However, few of them demonstrated the ability to massage kernel memory in conjunction with these techniques [32], and none have done so on Intel CPUs. The prior memory-massaging techniques that worked for DDR3 no longer work on DDR4 DIMMs paired with such CPUs. Thus, to fill this crucial gap that prevents powerful techniques from being exercised in end-to-end exploits, we present new techniques for achieving PTE massaging on DDR4 machines in conjunction with DDR4 hammering techniques.

The first step is to understand the differences between DDR3 and DDR4 machines that prevent the old massaging techniques from working on newer machines. Next, we work through the challenge of massaging hugepages despite their own unique allocator behavior since DDR4 techniques commonly rely on the use of hugepages. Third, we must prevent other processes' allocations from 'stealing' our target massage pages. Finally, we present the results of running all these steps together, demonstrating reliable kernel memory massaging.

5.4.1 Understanding CPU's Role in Massaging

Working DDR3 PTE massaging techniques [67] do not work when attempting to massage flip-vulnerable pages on DDR4 machines. Within the victim machine, the key structure at the center of massaging is the physical page allocator. Memory massaging consists of manipulating the physical page allocator such that a subsequent allocation request will be fed a specific flip-vulnerable page that had been placed within the pool of free pages. However, the physical page allocator's behavior is determined by software in the operating system—not the hardware. Yet a change in hardware seems to be enough to stop old techniques from working. Thus, the first step is to truly understand what components on the victim machine can affect memory massaging, if not only the allocator.

Experimental Setup. We begin by running memory massage experiments simultaneously on DDR3 and DDR4 machines in order to understand how memory massaging behaves differently on different hardware. As shown in Table 5.1, these machines both use the same operating system version (Linux version 5.4.1) which means they share the same allocator

code located in the `mm` directory of the Linux kernel source code. While this is not the latest version of the Linux kernel at the time of writing, it is the default version of our DDR4 machine, and a recent version that could run stably on our DDR3 setup. The DDR4 machine uses an Intel i7-9900K CPU and a single 8GB Samsung M378A1K43BB1-CPB DIMM, while the DDR3 machine uses an Intel i7-4770 CPU and two 4GB Samsung M378B5273DH0-CH9 DIMMs giving the same total memory capacity as the DDR4 machine. Both machines use the same default THP setting that allows for allocating THPs via the `madvise` syscall.

Controlled Experiments. Listing 5.1 shows a simplified pseudocode version of the technique we ran for this initial experiment. We first allocate memory, logging the allocated physical addresses (Lines 13 to 17), and then run a double-sided Rowhammer process to find a flip-vulnerable page (Line 19). We then perform userspace massaging on the discovered flip-vulnerable page, forcing a victim allocation via a thread spawned by the attacker (Line 24 and Lines 1 to 10). Since in these early experiments we are not concerned with running an end-to-end attack with user-level privileges, we have the victim thread print the physical addresses of its stack variables using the `pagemap` (Line 16). If the allocated physical page matches the previously unmapped flip-vulnerable page, then we have achieved a successful massage result.

We choose to use the same double-sided Rowhammer technique on both machines to eliminate any interference that may result from the hammering processes. However, since double-sided Rowhammer cannot induce bit-flips on a DDR4 machine, we simply select a fixed page among the hammered pages on both machines—whether this page is vulnerable to flips or not—and massage that page. The actual flip-vulnerability of the page cannot affect the massaging process, meaning selecting an arbitrary page in this manner will not interfere with our experimental results. We then compare the success rate of memory massaging attempts of the DDR3 machine versus that of the DDR4 machine.

Initial Results. These early experiments quickly revealed that even if same allocator and massaging techniques are used, the CPU can still create interference with the massaging process, as the DDR3 machine demonstrated a 100% success rate while the DDR4 machine a 0% success rate.

Insight: Lazy Allocation vs. Immediate Allocation. After deeper investigation, we found that the difference in CPU led to *lazy* allocations on the DDR3 machine and *immediate* allocation on the DDR4 machines. This means that on the DDR3 machine, the moment that a thread is spawned, it allocates physical pages from the pool of free pages for its local stack. However, on DDR4, the stack does not allocate physical pages for its stack when the thread is spawned. Instead, the thread will only request a physical page the moment one of its local stack variables needs to store data.

```
1   victim_proc(){
2     /*kernel stack thread allocated automatically*/
3     int local_variable;
4
5     /*get stack address*/
6     log(physical_address(local_variable));
7
8     /*maintain thread to hold kernel stack*/
9     while(1){/*do nothing*/}
10  }
11
12  attack_proc(){
13    uint64_t test_pages[NUM_PAGES];
14    for(int i = 0; i < NUM_PAGES; i++){
15      test_pages[i] = mmap(...);
16      log(physical_address(test_pages[i]));
17    }
18
19    uint64_t flip_vulnerable_page = hammer(test_pages);
20
21    munmap(flip_vulnerable_page);
22
23
24    pthread_create(..., victim_proc,...);
25
26 }
```

Listing 5.1: DDR3 Massage Code

```
1   victim_proc(){
2     /*kernel stack thread needs to be initialized with variable to
    force physical page allocation*/
3     sync(flag0);
4     sync(flag1);
5     int local_variable = INT_VAL;
6
7     /*get stack address*/
8     log(physical_address(local_variable));
9
10    /*maintain thread to hold kernel stack*/
11    while(1){/*do nothing*/}
12  }
13
14  attack_proc(){
15    uint64_t test_pages[NUM_PAGES];
16    for(int i = 0; i < NUM_PAGES; i++){
17      test_pages[i] = mmap(...);
18      log(physical_address(test_pages[i]));
19    }
20
21    uint64_t flip_vulnerable_page = hammer(test_pages);
22
23
24    pthread_create(..., victim_proc,...);
25    sync(flag0);
26    munmap(flip_vulnerable_page);
27    sync(flag1);
28
29 }
```

Listing 5.2: DDR4 Massage Code

This slight difference can drastically affect memory-massaging accuracy since the process depends on precisely placing the flip-vulnerable page at the top of the queue of free pages such that the next victim allocation uses the target page from the top of the queue. Without understanding this subtle difference, reusing prior techniques, attackers may unmap the target flip-vulnerable page, spawn the victim the thread, and continue other work required for the attack in parallel. If this extra work allocates or frees any memory, the target page may lose its position at the top of the queue, thus preventing the eventual stack allocation from using the correct page.

DDR4 CPU Modifications. Listing 5.2 shows the modifications we made to our simple massaging code to fix the issues present on our DDR4 machine. In the case of our experiments, we used `print` statements for logging and debugging, and these very prints allocated pages and interfered with the process before the victim threads had a chance to allocate memory for their local variables. We add synchronization code (Lines 3, 4, 25, and

27) to our thread to first spawn the victim thread, and then have the attacker parent thread unmap the flip-vulnerable page only immediately before the thread was ready to write data to its own local stack. Additionally, we ensure the victim allocates physical memory by forcing a write to the victim's local stack (Line 5). After this write completes, our printing and logging are performed. In a real-world attack where the victim thread cannot be altered, attackers can measure the average time delay between the thread spawning and allocating its stack in order to unmap the flip-vulnerable page with the right timing. Alternatively, attackers can target threads that utilize locks and synchronize based on those [73].

Results. After making these modifications, we achieved 100% massage accuracy on both machines in the case of this simple double-sided Rowhammer testing. These experiments revealed that while the C-level allocator code remained largely unchanged, the underlying Assembly-level code unique to the CPU hand changed in a meaningful way. After accounting for the lazy allocation possibly, we saw no further memory massaging interference from the CPU.

5.4.2 Working With Hugepages

After resolving the issue of CPU interference and achieving successful massaging in the case of simple double-sided Rowhammer, we gradually move to a scenario more closely resembling a DDR4 hammering process. The first step we take is to replace normal pages we have been working with so far with transparent huge pages (THPs). Recall that we are not yet concerned with flipping real bits and simply select an arbitrary page for our massaging experiments. Thus, up to this point, we have not been concerned with the allocation of contiguous pages (that would ensure our access remain within the same DRAM bank). Since the common standard for DDR4 Rowhammer is to rely on THPs, we move one step closer to a real attack by swapping normal pages for THPs. As expected, using the existing techniques with THPs did not work, since the allocator treats THPs differently than normal pages. Thus, we take the following steps to work toward a massaging technique that can work with THPs and allow for DDR4 Rowhammer exploits.

Initial Control Experiment. We begin with the same initial experiment of testing simple double-sided Rowhammer alongside userspace massaging, aiming to massage an unmapped page into a thread's allocated stack. The key change this time is that we perform our Rowhammer and memory massaging on THPs.

Unmapping Arbitrary Hugepages. In the case of simply unmapping a single 4KB page that came from a 2MB THP, then immediately making an allocation to attempt to map the recently freed page, we did not see successful memory massaging on either the

DDR3 machine or DDR4 machine. We considered that the freed page may have potentially been consistently stolen away by another allocation, hence to counter this possibility, we added buffer pages to the process to protect the target page from any thief allocations. We first mapped many huge pages, hammered victims from among these mapped pages, freed an arbitrarily selected target page, and then freed additional "dummy" 4KB pages from our other mappings. In our victim thread, we forced many allocations (via `mmap`) in addition to the stack allocation. Upon checking the physical addresses of the additional mapped pages, we found that all the mapped pages matched with our unmapped "dummy" page or our target page. By adjusting the number of freed dummy pages, we could consistently have the stack allocate our chosen target victim page, achieving memory massaging with THPs.

Unmapping Real Hugepages. The next step was to test our THP unmapping scheme with a real, flip-vulnerable page. We began with double-sided Rowhammer on DDR3. We first mapped THPs, used Rowhammer to find a bit-flip, mapped additional THPs, unmapped the flip-vulnerable page, and then unmapped the required number of dummy pages. This was followed by spawning a thread that performed a series of allocations via `mmap`. While the `mmap` calls did allocate many of our unmapped dummy pages, neither the `mmap` calls nor the stack allocated the target flip-vulnerable page. The key issue here seemed to be tied to the pages we used as dummy pages. This time, we allocated and freed our dummy pages *after* finding our flip, and long after our flip-vulnerable THP was allocated.

Results. We made an adjustment to our procedure by allocating our dummy pages *alongside* the pages we aimed to hammer and flip. Then, after finding the flip-vulnerable page, and unmapping this page along with the previously allocated dummies, we were able to consistently massage flip-vulnerable pages into the victim, both on DDR3 and DDR4. In the case of both DDR3 and DDR4, we did not add protections to prevent the unmapping of our aggressor addresses. Such protections will be required when we perform a true end-to-end exploit, since we will need to re-hammer the aggressor and induce a bit-flip after massaging, but releasing our aggressors is acceptable for the purpose of testing massaging on its own.

Insight: Hugepage Reservation. The key insight here is that unmapping a single 4KB page that belongs to a bigger 2MB THP is not enough to return the 4KB page to the pool of free pages. We observed behavior that the allocator seems to reserve sub-pages of larger hugepages, preventing their allocation for variables from other processes, including child threads. Freeing additional pages can force this initial freed 4KB page into the pool of free pages. The entire superset 2MB hugepage does not need to be freed, and most interestingly, it is sufficient to free 4KB pages from adjacent THPs that were allocated in succession with the initial target THP. This last finding will be quite useful when we need to perform multi-sided Rowhammer and must hold onto aggressor addresses within our target

THP, preventing us from relinquishing the entire THP.

5.4.3 Preventing Rogue Allocations

The last remaining challenge is orthogonal to the CPU and THP challenges we faced previously. At this stage, we transition from userspace massaging to kernelspace massaging. Kernel space massaging requires overcoming the boundary from userspace pages to kernelspace pages. As demonstrated in [71], to overcome this boundary, we can exhaust all the available kernelspace pages such that subsequent allocations will be forced to steal from the pool of available userspace pages. Additionally, we must exhaust all high-order userspace pages such that the kernel allocations have a high probability of allocating our target page from the pool of free pages.

However, forcing many allocations to the point which exhausts all kernel pages and most user pages puts the victim machine in a state of high "memory pressure" in which there are very few remaining free pages. Any active processes on the victim machine must then contend for the few pages. Inactive processes, or processes that have not used their physical pages recently, will temporarily relinquish their physical pages for other processes to use. However, the moment they need to use memory again, they are likely to request a physical page. This means during the massaging process, the attacker will indirectly cause the victim machine to begin triggering an increased number of uncontrolled (or "rogue") allocations. By running tests with `krpobes` [36], we found that these rogue allocations can interfere with the memory massage process and reduce massage accuracy. Thus, at this stage, we seek to reduce the effect of this interference.

Unmap Timing. We find that the key technique to prevent interference from rogue allocations is to minimize activity between the unmapping of our target physical page and the victim reallocation of this page. We first remove any logging or printing used to track the massaging process, since, as confirmed in prior steps (Section 5.4.1), such activity may allocate pages and interfere with our process. Additionally, prior work [71] suggested taking additional steps beyond the unmapping of the target victim page in order to further prime the allocator to pass this freed page along for the subsequent victim allocation. This included unmapping extra pages to force an eviction from the PCP cache, followed by additional mappings to remove any additional pages that may have been inadvertently evicted from the cache alongside the target page. The `pagetypeinfo` or `buddyinfo` files could be used to help track the state of the allocator.

We find that a higher accuracy can be achieved by forgoing these additional steps beyond the initial victim unmapping. Instead of tracing the state of the PCP and gradually releasing

additional pages until the PCP is flushed, we find it more effective to release a single large block simultaneously with the target victim that will ensure PCP eviction.

Additionally, instead of making additional mapping calls to eliminate any extra pages that may have entered the queue of free pages, we simply begin forcing the victim to make its allocations. As done in prior work, we force numerous victim allocations [71, 73, 67] (either in the form of spawning threads which forces kernel stack allocations, or in the form of PTE allocations). However, in our case, we make the observation that we can immediately begin this process of forcing victim allocations, since the earlier victim allocations can consume the additional residual pages that may have entered the queue, while later victim allocation can have the chance to allocate the target victim. This process of unmapping the victim then immediately beginning victim allocations reduces the odds of a rogue allocation stealing our target victim page while time is wasted on additional userspace allocation calls.

Reducing Swapped Page Interference. In order to reduce the likelihood that a background process will force a new allocation simply due to lacking physical page backing while under high memory pressure, we add a delay immediately before unmapping our target page. During this delay, we simply spin on a do-nothing loop creating no activity on the allocator from our attacker process. This gives the chance for background process to run through their own memory accesses and re-allocate physical pages as needed. We leave these external processes to run until we detect minimal activity on the allocator. At this point, the allocator is admittedly no longer in its optimal massaged state, i.e., primed for desired behavior during the forced victim allocation. However, there is now less likely to be interference from rogue allocations during the final unmapping and remapping steps. Furthermore, we can make up for the suboptimal state of the allocator by making additional victim allocations. We test the results of our new approach in full end-to-end experiments in the next section.

5.4.4 End-to-End Massaging

We utilize the techniques developed in the immediate prior sections to run an end-to-end memory massaging technique on DDR4 with improved success rates. We begin by allocating numerous THPs and use TRRespass [80] to check these allocated pages for flips. Once we find a flip, we lock the flip-vulnerable victim page, as well as any aggressor pages, in place using `mlock`, which prevents these physical pages from being swapped out in the case of high memory pressure. We then begin simultaneously allocating userspace and kernel space pages by making `mmap` calls and requesting pages at addresses with enough distance from each other such that each `mmap` will require a new PTE allocation. The `mmap` requests are fulfilled with

userspace pages while the PTE allocations use kernelspace pages [67]. Once we have totally drained kernel memory and allocated the maximum allowable number of userspace pages, we unmap our victim page, alongside other page from the initially allocated THPs while being careful to avoid unmapping any of our required aggressors addresses, and then force our victim allocation.

We test two separate types of victim allocation. The first is kernel stack allocation. This has been achieved already in prior work, but with relatively low probability [71, 73], resulting in slow attacks that take dozens of hours due to repeatedly restarting the memory massaging procedure each time it fails. We hope to improve the accuracy of this type of massaging. The second type of victim we test is PTE allocation. As discussed previously, PTEs are a common target for Rowhammer exploits, yet no prior works have demonstrated successful DDR4 PTE massaging, with the exception of Rowhammer attacks on AMD Zen processors [32].

Kernel Stack Massaging Results. After running our experiments for 8 hours, we are able to achieve a 23% accuracy, i.e., 23% of the total number of attempted trials was able to successfully massage a kernel stack into place for a kernel stack Rowhammer attack. The average required time to successfully massage a page into place was approximately 12 minutes. This presents a drastic improvement over the 0.65% accuracy demonstrated and dozens of hours of attempts required in the prior work [73].

PTE Massaging Results. After running our experiments for 8 hours, we achieve an approximately 3% accuracy for PTE massaging. While this accuracy is notably lower than the stack massaging accuracy, it is the first instance of successful PTE massaging on a non AMD Zen CPU, filling the gap between prior work's achievement of finding pages with PTE flips and the necessary next step of massaging into PTE allocations in order to actually flip PTE bits. Additionally, achieving multiple successes within an 8 hour provides for a considerably faster attack than other DDR4 kernel attacks, which took averages on the order of 20 hours for a single successful attempt [71, 73], as well as other comparably slow Rowhammer attacks on DDR3 [12, 45].

5.5 Evaluation of Prior Techniques

Now that we can reliably massage memory on DDR4, we may pose another crucial question regarding Rowhammer and current DDR4 hammering techniques. Since as we have seen in prior sections, memory massaging can be quite sensitive to mechanisms or side-effects external to the buddy allocator implementation, it is worth verifying existing hammering technique's "compatibility" with this new memory massaging technique. While hammering

techniques focus only on generating bit-flips and should therefore be decoupled from the memory massaging process, hammering techniques have different requirements on the number of pages they must control in order to flip bits. This varied interaction with physical pages may have unintended side-effects on the massaging process. Therefore, we run both the first DDR4 Rowhammer technique, TRRespass [20], in conjunction with memory massaging in comparison with the latest hammering technique, Sledgehammer [35], also in conjunction with memory massaging and compare the results.

Experimental Setup. We run both the TRRespass and Sledgehammer on the same victim machine using the same victim DIMM in order to prevent any external factors other than the hammering techniques themselves from affecting the results. We use the same DDR4 machine as shown in Table 5.1, using an Intel i7-9900K CPU and a single 8GB Samsung M378A1K43BB1-CPB DIMM running Linux version 5.4.1.

TRRespass Evaluation. Our evaluation of TRRespass uses the same procedure outlined in Section 5.4. On all the DIMMs we tested, we found that a 10-sided pattern is most effective for hammering. We use 2MB THPs to scan for bit-flips. Upon finding our first flip-vulnerable page, we attempt to massage the page into the kernel stack using the techniques described above. We observe a success rate of 23% when performing kernel stack massaging. We also run PTE massaging and achieve a success rate of 3%.

Sledgehammer Evaluation. Our evaluation of Sledgehammer uses a similar procedure as with TRRespass. However, instead of TRRespass's multi-sided hammering, we use Sledgehammer's multi-bank hammering. This requires us to control an increased number of aggressors since we must effectively perform a multi-sided hammering technique simultaneously over multiple banks by staggering consecutive accesses over different banks. We find a hammering pattern scattered across 50 different aggressor addresses produces the best results for efficiently producing bit-flips.

After finding our first flip-vulnerable page with Sledgehammer, we perform the same procedure as above, taking care to minimize interference that may hinder the massaging process. One key difference to note is that in the case of Sledgehammer we `mlock` all 50 aggressor rows, and when we unmap additional pages surrounding the target victim page (to allow the 4KB THP subset page to rejoin the pool of free pages), we avoid unmapping any page that contains any of the 50 aggressor rows. This way, we can continue to flip bits in our victim even after completing the massaging process. We run both kernel massaging and PTE massaging techniques after finding the Sledgehammer induced bit-flip. We find that Sledgehammer achieves kernel stack massaging with an approximately 11% success rate and PTE massaging with an approximately 1% success rate.

Comparison. While memory massaging is evidently feasible with an attack such as

Sledgehammer, it presents less accurate results than TRRespass in the case of memory massaging, with TRRespass demonstrating more than 2x accuracy. Additionally, while Sledgehammer can find flips more efficiently end-to-end attack time of TRRespass is shorter than that of Sledgehammer, with TRRespass requiring only about 60% of the time needed to both find flips and perform stack massaging compared to TRRespass.

Considering the trade-off in massaging accuracy, this result of TRRespass requiring less time on average is expected, since a single massaging attempt is rather slow relative to the time needed to find a single bit-flip. Additionally, as shown in prior work [71, 73], memory massaging is often the performance bottleneck for end-to-end Rowhammer attacks, inflating attack times to up to 25 hours due to repeated massaging attempts. In the case of such Rowhammer attacks with lengthy memory massaging times, selecting a Rowhammer technique with higher-accuracy massaging can help reduce this time cost.

5.6 Discussion

The results presented thus far lead to important follow-up questions and implications regarding the development of future Rowhammer techniques. Prior works demonstrating new ways of inducing bit-flips on DDR4 [20, 35, 31, 53] are all valuable contributions toward the first necessary stage of any Rowhammer attack: flipping bits on the victim machine. However, while each work outperformed its predecessors, we find that when run in conjunction with memory massaging, the latest, most performant technique [35] is outperformed by the oldest, simplest technique [20]. We believe both techniques serve valuable use-cases and discuss the trade-offs between Rowhammer techniques in the remainder of this section.

Trade-offs Between Rowhammer Techniques. The oldest Rowhammer technique, TRRespass, yields a high massaging accuracy for kernel memory massaging attacks. In cases which an attacker decides to target a kernel victim (e.g., flipping a kernel stack pointer [73] or a PTE [67]), TRRespass is the preferred technique. Additionally, for cases in which useful bit-flips are abundant and generating a large number of bit-flips quickly is not a key requirement, TRRespass may also be useful.

However, there are numerous cases in which generating the large number of bit-flips enabled by newer techniques may prove useful. There are numerous Rowhammer exploits that do not rely on kernelspace massaging [23, 45, 71, 19, 69, 35]. Many such attacks only need to flip bits in a userspace victims to gain root or leak sensitive information, either by flipping bits in secret keys [45] or the `sudo` binary [23].

As we have seen in Section 5.4.1, when accounting for the differences in CPU-specific allocation policies and hugepage unmapping, userspace massaging is a deterministic process

with a 100% success rate. More advanced Rowhammer techniques do not create interference with such processes, thus making such techniques the clear choice since the increased flip-rate comes without drawback.

Additionally, as noted in Section 5.2, Rowhammer flips are largely dependent on the physical nature of the victim DIMM. If an attackers wishes to target a DIMM that is resilient against bit-flips, and finds it difficult to induce flips with TRRespass, more efficient techniques such as Sledgehammer present the better option.

Considerations for Future Rowhammer Techniques. While the work achieved by recent Rowhammer techniques is valuable, especially in the case of userspace massaging, we still emphasize the need for useful end-to-end techniques in the case of kernelspace attacks. Considering the continued prevalence of PTE attacks in the research field of Rowhammer exploits [67, 75, 24, 76, 20, 31, 42], it is crucial for future Rowhammer researchers to consider the feasibility of whether generated bit-flips at offsets tied to certain sensitive targets can indeed be useful for end-to-end attacks at those targets.

5.7 Mitigations

Since end-to-end Rowhammer attacks are multi-faceted exploits involving multiple levels of the computer architectural stack (i.e., hardware, software, operating system, microarchitecture), numerous methods have been proposed for defending against Rowhammer attacks. However, as we will discuss in the following section, to date there is no perfect solution, with each proposed defense already defeated by a followup attack technique. The rest of this section presents currently implemented or proposed defenses and their demonstrated or potential weaknesses.

Protecting Software Targets. One approach to defend against Rowhammer exploits is to directly protect the potential target via software. For example, in the case of targets residing in the kernel, such as PTEs, prior defenses proposed placing "buffer rows" that surround rows of memory containing sensitive data [5, 43]. Such defenses are based on the observation that Rowhammer attacks require targeting adjacent rows of memory, and, in the case of kernel attacks in particular, require flipping kernel data bits by hammering adjacent userspace aggressors. By placing buffers that either protect the entire kernel, or that protect specific targets from adjacent accesses, attackers would only be able to flip bits in these buffer rows and perform no malicious activity on the victim machine.

However, a follow-up work has shown that it is possible to flip buffered victims despite these defenses [91, 90, 70]. GhostKnight [91] has demonstrated attackers can run aggressor accesses within states of mis-speculation to target out-of-bound accesses, allowing attackers

to hammer victims beyond buffer boundaries. Other works [90, 70] have shown attackers can exploit the translate lookaside buffer (TLB) to strike kernel aggressors directly to flip bits within the regions protected by buffers. The TLB is effectively a cache for virtual-to-physical page mappings to allow memory accesses to skip the virtual-to-physical translation process by simply checking mappings saved to the cache. Filling the TLB requires performing a "TLB walk", reading numerous PTEs to cache their corresponding mappings. Each of these reads acts as a memory access, and the corresponding Rowhammer exploits will trigger TLB walks in such a way that they repeatedly strike a set of kernel aggressor addresses and induce bit-flips in kernel victims without ever needing to hammer user-level aggressor rows. Thus, placing a buffer between user and kernel memory is rendered ineffective.

Considering this ability to bypass buffer rows, follow-up defenses suggested placing sensitive victims, particularly PTEs, in "flip-safe" memory [84]. This work presents a defense in which the victim machine can first profile its own memory for Rowhammer bit-flips, by hammering its own addresses, to determine which physical addresses are vulnerable to bit-flips and which are totally immune. Since Rowhammer is entirely dependent on the physical nature of a victim DIMM, if an address is confirmed to be immune to bit-flips during profiling, the victim can safely assume no bit-flips can occur on that address. Thus, after running this initial profiling stage, PTEs can be allocated only to "flip-safe" addresses, preventing PTE attacks.

However, at the time of this writing, this is only one proposed defense, not implemented in real world systems. Furthermore, there are other sensitive targets in the kernel and otherwise that can lead to root access on a victim machine [23, 35, 73]. For example, [23] shows it is possible to flip bits in the `sudo` binary, leading to the `sudo` process skipping necessary permission checks and allowing a malicious attacker to freely run `sudo` on the victim machine and gain superuser privileges.

Additionally, GadgetHammer [73], shows that aside from specific sensitive targets such as `sudo` or PTEs, attackers can target general code patterns via Rowhammer as well. Any victim that contains a nested pointer dereference followed by a return of dereferenced data can lead to arbitrary read access with only a single bit-flip. Since developing adequate search tools to find such gadgets is still an open research problem [26, 55, 6, 47], it is not feasible to attempt to patch each gadget directly, since there currently is no reliable way to accurately detect all potential victim gadgets.

Protecting Against Memory Massaging. As shown throughout this paper, end-to-end Rowhammer attacks require a chain of exploits to work properly, including reverse engineering memory mappings, inducing bit-flips in victim memory, and massaging physical memory into place. Therefore, one way to prevent Rowhammer attacks is to break the

memory massaging stage. At the time of this writing, there are no known ways to prevent memory massaging in a victim machine, but future works may explore potential defenses based on the techniques discussed in this paper.

Memory massaging requires allocating a large amount of victim memory—especially in the case of kernel memory massaging. This makes kernel memory massaging rather unstealthy. While no such defense currently exists, future defenses may consider watching for a suspiciously large number of repeated memory allocation requests and taking appropriate action if a process seems to be attempting memory massaging for a Rowhammer attack. Similarly, the central mechanism that allows for kernel memory massaging is the kernel's "page stealing" which allows the kernel to convert userspace memory into kernel memory. Defenses could detect instances of page stealing and take extra precautions to prevent any additional malicious Rowhammer activity on the victim machine, such as throttling memory accesses, to prevent an exploit from running to completion after the massaging step is run.

Preventing Exploits in Software. Besides memory massaging, there are numerous steps in the Rowhammer exploit chain, including allocating contiguous memory for templating, repeatedly accessing aggressor rows to induce bit-flips in victim rows, and profiling victim memory at specific page offsets matching the offsets of target victims. Preventing any of these steps from functioning would break the exploit chain and help mitigate end-to-end Rowhammer attacks. Furthermore, while the bit-flips are inherently a flaw tied to hardware, these steps in the end-to-end exploits, including the induction of these bit-flips, are all managed entirely in software. Thus, there is room to block against Rowhammer attacks via software defenses, so that Rowhammer attacks can be mitigated swiftly on older and newer generations of machines alike via software patches without needing to wait for new hardware that is safe against Rowhammer exploits.

One key observation is that Rowhammer attacks require knowledge of victim offsets in advance. Address space layout randomization (ASLR) and kernel ASLR (KASLR) randomize memory addresses to prevent attackers from targeting specific victim addresses that may contain victim data. Having randomized victim addresses would make it difficult to know which offsets to target and massage in advance, interfering with Rowhammer attacks. However, both KASLR and ASLR can be broken via side-channel techniques [64], meaning in their current form, they are not adequate defenses. However, future defenses may consider re-randomizing addresses if Rowhammer activity is observed.

An alternate approach may be to restrict THPs to privileged users, given DDR4 technique's reliance on these easily obtainable blocks of contiguous memory. However, prior work has shown it is possible to obtain contiguous blocks of memory even without access to THPs [62, 45] and that end-to-end DDR4 attacks are possible even in such cases [35].

Preventing Bit-Flips in Hardware. Lastly, while not a viable option for older machines, building defenses into future hardware would be the surest way to defend against Rowhammer attacks. If an attacker is unable to induce bit-flips in hardware, then no Rowhammer exploits are possible.

An existing mechanism that could prevent Rowhammer bit-flips is error correction codes (ECC). Using ECC, it is possible to correct single-bit errors and detect double-bit errors. Rowhammer attacks, such as PTE exploits, typically rely on only flipping a single dangerous bit. Therefore, correcting bit-flips via ECC can prevent exploits that rely on a Rowhammer-induced bit-flip staying flipped long enough to run a multi-step exploit.

However, prior works have shown the ECC is not enough to protect against all types of Rowhammer exploits. For example, ECCsploit [12] demonstrated that it is possible to profile memory for addresses that are vulnerable to three or more bit-flips. Since ECC can only detect up to two bit-flips per word of memory, flipping three bits simultaneously prevents any ECC detection and allows bit-flips to remain permanently in DRAM. Furthermore, RAMBleed [45] has shown that ECC corrections can lead to a timing side-channel letting attackers know if bit-flips have occurred, even if they do not remain in DRAM. Since Rowhammer bit-flips are dependent on DRAM values (i.e., bits are more likely to flip if they are surrounded by opposite bit-values), learning if bits flipped via this side-channel allows attackers to learn values of data present in DRAM by hammering and watching for ECC corrections.

Lastly, the ultimate solution for preventing Rowhammer exploits is to design DIMMs invulnerable to bit-flips. DDR4 attempted this by adding target row refresh (TRR) to its standards [33], requiring all DDR4 DIMMs to track for potential Rowhammer activity and run early refreshes as needed. However, as we have seen, follow-up works quickly demonstrated it is possible to flip bits even in the presence of TRR [20, 42, 31, 53, 35, 51]. DDR5 implemented an even more advanced detection-to-early-refresh scheme called *refresh management* (RFM) [34]. However, prior work has shown Rowhammer can flip bits on DDR5 as well [32].

Academic researchers have proposed alternate detection and early refresh schemes that rely more heavily on synchronizing refreshes with Rowhammer activations [53], which may prove useful if implemented on future generations of hardware, such as a successor to DDR5.

5.8 Conclusion

In this paper we presented a DDR4-compatible kernel memory massaging technique, allowing for Rowhammer attacks on the Linux kernel on modern victim machines. It bridges the

gap between techniques that efficiently induce DDR4 bit-flips and end-to-end Rowhammer exploits. Our technique is shown to achieve the first PTE exploit on machines using DDR4 and an Intel CPU. We have also shown that older Rowhammer techniques have better compatibility with memory massaging than newer techniques, offering new insight into what considerations future work should keep in mind when developing new methods to flip bits on DDR4 for kernel attacks.

CHAPTER 6

Conclusions and Future Directions

6.1 Conclusions

This thesis has demonstrated the threat Rowhammer poses across the various levels of the computer architecture stack. In particular, we have demonstrated techniques for reverse-engineering memory controller addressing functions, attacks exploiting microarchitectural systems, exploits flipping bits in the victim's operating system, and lastly, techniques for building more reliable end-to-end attacks concerning all levels of the process.

6.1.1 DRAM Addressing

All Rowhammer attacks begin with striking specific target addresses in victim memory. Since DRAM address are not exposed to unprivileged or even privileged users, attackers must reverse-engineer the physical-to-DRAM mapping functions. However, newer CPUs changed their addressing functions in ways that prevent older techniques from working. Thus, in Chapter 2, we reverse-engineered the mapping functions in the latest Intel CPUs at the time of writing. We used these mapping functions for executing the first Rowhammer attack using such processors, and proposed techniques for obtaining the mappings on any future processors that do not use the same mapping functions.

6.1.2 Microarchitecture

While Rowhammer is an attack on physical DRAM hardware, the bit-flips produced in DRAM can be exploited at various levels of the computer architectural stack. 3 presented a microarchitectural attack in the form of SpecHammer, showing how Rowhammer bit-flips can be used to relax a key requirement of Spectre V1, making for a Spectre attack that can target a wider array of potential victims. With this exploit, we showed how Rowhammer poses a threat to microarchitectural systems, such as branch prediction, demonstrating how

a single bit-flip is enough to trigger a misspeculation that can grant an attacker arbitrary read access.

6.1.3 Operating Systems and Software

Many defenses attempt to prevent Rowhammer exploits by protecting the specific variables, code, or data structures that have been abused in prior exploits. With Chapter 4, we showed that such defenses are not enough, as we presented a Rowhammer exploit that targets general code patterns instead of any one particular sensitive software target. With the end-to-end attack on a real-world gadget found in the Linux kernel, we found that Rowhammer attacks on the operating system and software level of the computer architecture stack are more widespread than previously conceived, motivating the need to more holistic defenses to protect against such attacks.

6.1.4 End-to-End Rowhammer Attacks

Finally, in Chapter 5, instead of focusing on any one specific Rowhammer exploit or attack vector, we instead considered how we can generally make Rowhammer attacks more practical and efficient for end-to-end attacks. In particular, we identified a gap in prior work which have effectively demonstrated new ways to flip bits more efficiently on DDR4, yet never demonstrated end-to-end exploits using those bit-flips. We found the primary issued lied in a lack of effective memory massaging for DDR4 systems and developed novel techniques to drastically improve massaging performance and accuracy, leading to faster end-to-end attack times as well, while also achieving the first PTE Rowhammer attack on DDR4 while using an Intel CPU.

6.2 Future Directions

We conclude this thesis with a discussion of future research directions that can further develop the work presented in this thesis.

6.2.1 Gadget Search

Two of the chapters in this thesis, Chapter 3 and Chapter 4, presented Rowhammer attacks target specific types of code gadgets. The main concern in defending against such attacks is that there is currently no practical way to automate the process of finding and patching all vulnerable gadgets. Performing such patches manually is also impractical, since many of

these gadgets reside in large code bases such as the Linux kernel, which would be too large to manually parse. Thus, future work could seek to build gadget search tools that can identify such gadgets and remove them from potential targets until hardware defenses are developed to truly solve the issue. The key challenge to consider with such a work is balancing search accuracy with scalability for large real-world code bases. On large code, solutions that test every possible execution sequence would not be practical, with techniques such as fuzzing showing promise as potential solutions [55].

6.2.2 DDR5

The latest generation of DRAM, DDR5, comes with new defenses to prevent Rowhammer exploits. While prior work has been able to flip bits on DDR5, such flips have only been demonstrated on AMD Zen and on just one singular DIMM with an anomalously large number of flips [32]. Since this result cannot be reproduced on a wide array of DDR5 DIMMs nor on other processors, it is uncertain if this is a reliable technique for flipping bits on DDR5, or if the particular tested DIMM was especially susceptible to flips due to process variation. Therefore, it is rather important for the future of Rowhammer research to demonstrate that DDR5 is indeed vulnerable to Rowhammer bit-flips. Prior works have demonstrated that DDR5 is theoretically flippable even with its new defenses enabled [53], giving reason to believe such flips can be achieved reliably in practice. Achieving such a result would prove once again that defenses attempting to prevent Rowhammer by tracking particular activation patterns are not sufficient, motivating the need for stronger defenses.

6.2.3 Defenses

Finally, the main motivation and purpose of this thesis has been to demonstrate the threat posed by Rowhammer in order to motivate stronger, more reliable, and more holistic defenses. Rather than defenses that protect just a single layer of the computer architecture stack, or defenses that guard just one single target in memory from flips, future defenses should aim to root bit-flips at the source and prevent them from occurring in memory. Thus, it is essential for future work to consider defenses that can prevent such bit-flips. The hope would be to see less defenses emerge that patch out one particular type of exploit, and instead, defenses that call for building more stable hardware.

For example, PROTRR [53] suggests a modified refresh pattern for future DIMMs that provides proven security guarantees against Rowhammer. More defense work in this direction, testing the reliance of defenses, or the presence of flips, in simulation before the hardware design is finalized would be more likely to lead to DIMMs invulnerable to Rowhammer. It

is essential to put forth this level of effort early since once a standard is finalized and the hardware is made available to the public, there is no simple way to patch out any flaws.

While the top recommendation is to work toward defenses more in-line with PROTRR, other measures can be taken to mitigate the limitation of updated DRAM hardware releasing relatively infrequently. For example, one level up the computer architecture stack is the memory controller used to interact with the DIMMs. As we have seen in prior chapters, understanding the mappings used by these memory controllers is an essential component of Rowhammer attacks. Cutting off this capability would prevent attackers from inducing Rowhammer bit-flips. Furthermore, processors are updated and released more frequently than DRAM, with new processor generations releasing more frequently than DRAM generations. The mapping functions in DDR5 that prevent a pure software approach to reverse engineer was a step in the right direction, but not sufficient since attackers can physically probe a single CPU of a particular model, or use the page coloring approach. Randomizing the mappings between CPUs of a particular model would help to at least prevent physical probing, since it would require an attacker to have physical access to a victim's machine, making at attack considerably more difficult. Future techniques could consider potentially using dynamic mappings to prevent even the page coloring approach from working.

Lastly, the final suggestion is to consider more holistic defenses for exploits. While the best defense is to prevent bit-flips on future machines, patches for specific exploits do at least serve a purpose in protecting older machines from vulnerabilities. However, rather than patching out any one specific exploit (e.g. protecting against PTE flips), the suggestion of this thesis would be to implement defenses that provide more *general* protections from Rowhammer bit-flips.

For example, in the vein of defending addressing level, one potential defense could be randomizing mappings from virtual to physical memory when pages are shared across processes. Memory massaging a target victim page would trigger a remapping. Thus, even if an attacker has performed the reverse engineering steps and found flip-vulnerable pages and their corresponding aggressors, their aggressor-victim pairs will be broken once the memory massaging step is performed, since the pages in question will no longer be within the same bank. This defense can be implemented via an operating system update and would prevent an attacker from reliably inducing bit-flips at will.

Additionally, other defenses such as pointer sanitation or opcode checks could prevent the effects of bit-flips from propagating into any sensitive code. Indeed, rather than try to patch out individual Rowhammer gadgets or protect specific sensitive binaries from bit-flips, more extensive defenses can cut off the propagation of bit-flips into dangerous places in the victim machine and prevent future exploits before they have the chance to be discovered.

APPENDIX A

SpecHammer Appendix

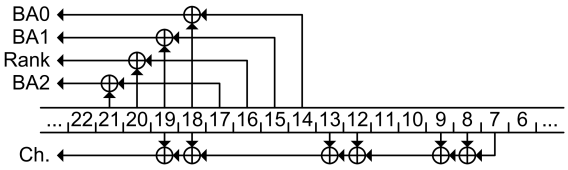

Figure A.1: **Physical to DRAM map for Ivy Bridge/Haswell (taken from [58]).**

A.1 Reverse Engineering Virtual to DRAM 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 [45]. A timing side-channel is then used to determine which physical addresses correspond to rows in the same bank [58], reverse engineering the physical to DRAM address mapping. However, these techniques relied on 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 n the next-largest freelist. 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) [21].

pagetypeinfo & buddyinfo files. The `pagetypeinfo` file shows how many free blocks are available for each order and migratetype. While previous techniques [45, 75] 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` to obtain 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 3.5, 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 [45]. 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 [58]. 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. [58] present the mapping function for numerous processors, such as the Haswell mapping (shown in Figure A.1). Therefore, for attacks on Haswell, we can use this mapping as is. For newer processors, we run Pessl's attack (as provided in [80]) 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 [15]. These newer processors are designed designed to use DDR4. DDR4 Rowhammer techniques such as TRRespass [20], 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 [15]. 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.

A.2 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 A.1 shows the changes made to Rowhammer.js's native code. The first change starting at line 530 fixes 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 561 to 576. 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 A.2 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 387. 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. Due to how TRRespass organized addresses into columns (col in the for loop) subsequent column values do not lead to subsequent address accesses. If an address being initialized in a given loop iteration happens to come before an address that has already been initialized, the cache's buddy fetcher may pull an address (that has already been initialized and flushed) into the cache. Thus, we add an additional flush (line 400) to remove buddy lines from the cache as well.

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` (line 284) 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. We therefore flush this memset data as well.

```
    .....
526  if(OFFSET2 > =0)
527  second_row_page = pages_per_row[row_index+2].at(OFFSET2);
528  if (
529  //******fixed bug***********
530  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,first_row_page)*0
    x1000))
531  !=
532  get_dram_mapping((void*)(GetPageFrameNumber(pagemap,second_row_page)
    *0x1000))
533  //************************
534  )
     {
       ....
557    #ifdef FIND_EXPLOITABLE_BITFLIPS
558    for(size_t tries = 0; tries < 2; ++tries)
559    #endif
560    {
561      //******cache flush victim***********
562      int32_t offset = 1;
563      for (; offset < 2; offset += 1)
564      for (const uint8_t* target_page8 :
565      pages_per_row[row_index+offset])
566      {
567        const uint64_t* target_page = (const uint64_t*)
568        target_page8;
569        for (uint32_t index = 0; index < (512);
570        ++index) {
571          uint64_t* victim_va = (uint64_t*)
572          &target_page[index];
573          asmvolatile("clflush(%0)"::"r"(victim_va):%"memory");
574        }
575      }
576      //*********************************
577      hammer(first_page_range, second_page_range, number_of_reads);
         ....
     }
```

Listing A.1: Rowhammer.js Modifications

```
      char *hPatt_2_str(HammerPatter * h_patt, int fields) {
134     static char patt_str[256];
135     char *dAddr_str;
136     memset(patt_str, 0x00, 256);
137     //******new cache flushes******
138     clflush(patt_str);
139     clflush(patt_str + 64);
140     clflush(patt_str + 128);
141     clflush(patt_str + 192);
142     clflush(patt_str + 256);
143     //****************************
144     .....
145     void fill_stripe(DRAMAddr d)addr, uint8_t val, ADDRMapper *
    mapper) {
146       for (size_t col=0; col<ROW_SIZE; col+=(1<<6)) {
          d_addr.col = col;
284       DRAM_pte d_pte = get_dram_pte(mapper, &d_addr);
285       memset(d_pte.v_addr, val, CL_SIZE);
286       //******new cache flushes******
287       clflush(d_pte.v_addr);
288       clflush((d_pte.v_addr) + CL_SIZE);
289       //****************************
290     }
291   }
292   .....
293   void init_stripe(HammerSuite * suite, uint8_t val){
294     .....
295     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);
387       memset(d_pte.vaddr, val, CL_SIZE);
          //******new cache flushes******
394       clflush(d_pte.v_addr);
395       clflush((d_pte.vaddr) + CL_SIZE);
396       //****************************
397 .....
398 void scan_stripe(HammerSuite * suite, HammerPattern * h_patt,
399 size_t adj_rows, uint8_t val){
400   .....
401   if(res){
402     for (int off = 0; off < CL_SIZE; off++){
403       ....
404       memset(pte.v_addr + off, t_vall, 1);
405       //******new cache flushes******
          clflush(pte.v_addr + off);
571       //****************************
        }
        memset((char *)(pte.v_addr), t_val, CL_SIZE);
597     //******new cache flushes******
598     clflush(pte.v_addr);
        clflush((pte.v_addr) + CL_SIZE);
608     //****************************
609 }
```

Listing A.2: TRRespass Modifications

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 A.1: 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 A.2: The effect of flushing victim addresses on TRRespass

A.3 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 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 [79], 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. We additionally verified that `memset` does not use non-temporal (i.e. non-caching) stores on our machines. 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.

The results are shown in Table A.1 (DDR3) and Table A.2 (DDR4). The DDR3 test was based on Rowhammer.js [22] and DDR4 on TRRespass[80] 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.

BIBLIOGRAPHY

[1] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. Anvil: Software-based protection against next-generation rowhammer attacks. *ACM SIGPLAN Notices*, 51(4):743–755, 2016.

[2] Andrew Baumann. Hardware is the new software. In *16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Chandler Carruth. Rfc: Speculative load haredning (a spectre variant #1 mitigation, 2018.

[8] 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.

[9] 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.

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[11] Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu. Are we susceptible to rowhammer? an end-to-end methodology for cloud providers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 712–728. IEEE, 2020.

[12] 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.

[13] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[14] 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.

[15] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Sec*, August 2021.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] Daniel Gruss. Program for testing for the dram "rowhammer" problem using eviction, May 2017.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[28] 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.

[29] invictus. Linux kernel heap spraying / uaf, 2017.

[30] 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/.

[31] 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.

[32] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. Zenhammer: Rowhammer attacks on amd zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 2024)*, 2024.

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

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

[35] Ingab Kang, Walter Wang, Jason Kim, Stephan van Schaik, Youssef Tobah, Daniel Genkin, Andrew Kwong, and Yuval Yarom. Sledgehammer: Amplifying rowhammer via bank-level parallelism. In *Proc. USENIX*, 2024.

[36] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel probes (kprobes). URL:https://docs.kernel.org/trace/kprobes.html.

[37] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur 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)*, pages 638–651. IEEE, 2020.

[38] 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.

[39] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

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

[41] 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.

[42] 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.

[43] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {ZebRAM}: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018.

[44] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[45] 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.

[46] Christoph Lameter and Minchan Kim. Page migration, 2016.

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

[48] Matt Linton and Pat Parseghian. More details about mitigations for the cpu speculative execution issue, 2018.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[50] 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.

[51] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–18, 2023.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] Colin Percival. Cache missing for fun and profit, 2005.

[58] 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.

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

[60] Filip Pizlo. What spectre and meltdown mean for webkit, 2018.

[61] The Chromium Projects. Site isolation, 2018.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299. Springer, 2019.

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

[68] Mark Seaborne. Program for testing for the dram "rowhammer" problem, Aug 2015.

[69] 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.

[70] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. {TLB; DR}: Enhancing {TLB-based} attacks with {TLB} desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 989–1007, 2022.

[71] 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.

[72] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Go go gadget hammer: Flipping nested pointers for arbitrary data leakage. In *33rd {USENIX} Security Symposium ({USENIX} Security 24)*, 2024.

[73] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. Go go gadget hammer: Flipping nested pointers for arbitrary data leakage. In *Proc. USENIX*, 2024.

[74] Paul Turner. Retpoline: a software contruct for preventing branch-target-injection, 2018.

[75] 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.

[76] 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.

[77] VandySec. rowhammer-armv8, Apr 2019.

[78] 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.

[79] Krishnaswamy Viswanathan. Disclosure of hardware prefetcher control on some intel processors, 2014.

[80] vusec. trresspass, Mar 2020.

[81] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2020.

[82] 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.

[83] 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.

[84] 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.

[85] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016.

[86] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[87] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise l3 cache side-channel attack. In *23rd USENIX Security Symposium*, 2014.

[88] 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.

[89] Zhenkai Zhang, Zihao Zhan, Daniel Balasubramanian, Xenofon Koutsoukos, and Gabor Karsai. Triggering rowhammer hardware faults on arm: A revisit. In *ASHES*, 2018.

[90] 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.

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

[92] 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.