Another Flip in the Wall of Rowhammer Defenses

Daniel Gruss[1], Moritz Lipp[1], Michael Schwarz[1], Daniel Genkin[2],
Jonas Juffinger[1], Sioli O'Connell[3], Wolfgang Schoechl[1], and Yuval Yarom[3,4]
[1] Graz University of Technology
[2] University of Pennsylvania and University of Maryland
[3] University of Adelaide
[4] Data61

***Abstract*—The Rowhammer bug allows unauthorized modification of bits in DRAM cells from unprivileged software, enabling powerful privilege-escalation attacks. Sophisticated Rowhammer countermeasures have been presented, aiming at mitigating the Rowhammer bug or its exploitation. However, the state of the art provides insufficient insight on the completeness of these defenses.**

In this paper, we present novel Rowhammer attack and exploitation primitives, showing that even a combination of all defenses is ineffective. Our new attack technique, *one-location hammering*, breaks previous assumptions on requirements for triggering the Rowhammer bug, i.e., we do not hammer multiple DRAM rows but only keep one DRAM row constantly open. Our new exploitation technique, *opcode flipping*, bypasses recent isolation mechanisms by flipping bits in a predictable and targeted way in userspace binaries. We replace conspicuous and memory-exhausting spraying and grooming techniques with a novel reliable technique called *memory waylaying*. Memory waylaying exploits system-level optimizations and a side channel to coax the operating system into placing target pages at attacker-chosen physical locations. Finally, we abuse Intel SGX to hide the attack entirely from the user and the operating system, making any inspection or detection of the attack infeasible. Our Rowhammer enclave can be used for coordinated denial-of-service attacks in the cloud and for privilege escalation on personal computers. We demonstrate that our attacks evade all previously proposed countermeasures for commodity systems.

I. Introduction

The Rowhammer bug is a hardware reliability issue in which an attacker repeatedly accesses (*hammers*) DRAM cells to cause unauthorized changes in physically adjacent memory locations. Since its initial discovery as a security issue [38], Rowhammer's ability to defy abstraction barriers between different security domains has been extensively used for mounting devastating attacks on various systems. Examples of previous attacks include privilege escalation, from native environments [58], from within a browser's sandbox [21], and from within virtual machines running on third-party compute clouds [63], mounting fault attacks on cryptographic primitives [9, 53], and obtaining root privileges on mobile phones [61]. Recognizing the apparent danger, these attacks have sparked interest in developing effective and efficient mitigation techniques. While existing hardware countermeasures such as using memory with error-correction codes (ECC-RAM) appear to make Rowhammer attacks harder [38], ECC-RAM is intended for server computers and is typically not supported on consumer-grade machines.

Software-based mitigations, which can be implemented on commodity systems, have also been proposed. These include ad-hoc defense techniques such as doubling the RAM refresh rates [38], removing unprivileged access to the `pagemap` interface [39, 55, 58], and prohibiting the `clflush` instruction [58]. However, recent works have already bypassed these countermeasures [6, 21, 61]. Other ad-hoc attempts, such as disabling page deduplication by default [46, 54], only prevent specific Rowhammer attacks exploiting these features [10, 53], but not all Rowhammer attacks.

The research community proposed sophisticated defenses which seemingly have solved the Rowhammer problem. Based on the underlying primitives of these defenses, we introduce a new systematic categorization into five defense classes:

- **Static Analysis.** Binary code is analyzed for specific behavior, common in side-channel attacks, e.g., using high-resolution timers or cache flush instructions [25, 32].
- **Monitoring Performance Counters.** Rowhammer relies on frequent accesses to DRAM cells, e.g., using a Flush+Reload loop. These frequent accesses are detected by monitoring CPU performance counters [6, 15, 22, 25, 32, 50, 68].
- **Monitoring Memory Access Patterns.** Rowhammer causes unusual high-frequency memory access patterns to two or more addresses in one DRAM bank. Rowhammer can be stopped by detecting such access patterns [6, 16].
- **Preventing Exhaustion-based Page Placement.** Rowhammer requires target pages to be on vulnerable memory locations. All Rowhammer privilege escalation attacks so far required memory exhaustion. Thus, preventing abuse of memory exhaustion thwarts Rowhammer attacks [21, 61].
- **Preventing Physical Proximity to Kernel Pages.** As a more complete solution, user and kernel memory cells are physically isolated through the memory allocator, thwarting all practical Rowhammer privilege-escalation attacks [11].

Notice that defenses in each class share the same assumptions, properties, and introduce the same form of protection. Defenses from different classes complement each other. Thus, given the extensive amount of research on Rowhammer countermeasures, in this paper we ask the following question:

To what extent do the approaches above actually prevent Rowhammer attacks? In particular, is it possible to successfully mount Rowhammer privilege-escalation attacks in the presence of some (or even all) of the countermeasures above?

A. Our Results and Contributions

In this paper, we show that despite numerous works on mitigating Rowhammer attacks, much remains to be done to truly understand their effectiveness and how to mitigate them. For this purpose, we introduce a new categorization for Rowhammer defenses (which we already outlined above) as a foundation for a systematic evaluation. Demonstrating the insufficiency of existing mitigation techniques, we present a novel Rowhammer attack and subsequent exploitation techniques for privilege escalation which allows defeating the underlying assumptions of all of the countermeasures mentioned above. In particular, our attack is still applicable even in the presence of *all* of the above countermeasures. We now describe the four building blocks of our attack and how each building block invalidates the assumptions of the defense classes.

Defeating Physical Kernel Isolation. The assumption of physical kernel isolation is that Rowhammer-based privilege escalation is only practical by flipping bits in kernel pages. We void this assumption by introducing *opcode flipping*, a technique for malicious and unauthorized modification of a userspace program's instructions by causing bit flips in its opcodes. By applying this technique to `sudo`, we bypass authentication checks and obtain root privileges.

Defeating Memory Access Pattern Analysis. All known Rowhammer techniques require frequent alternating accesses to *two* or more DRAM cells in the same DRAM bank. Consequently, countermeasures detect when an attacker performs such alternating accesses to *two* or more addresses in the same DRAM bank. We present *one-location hammering*, a new type of Rowhammer attack which only hammers *one* single address. Since our attack only uses *one* memory address, it does not require any knowledge of physical addresses and DRAM mappings [34, 51, 63], allowing us to perform Rowhammer attacks with even fewer requirements.

Page Placement Without Memory Exhaustion. Page deduplication is usually disabled for security reasons [46, 54, 61] as a response to page deduplication attacks [8, 19, 59], including Rowhammer attacks based on page deduplication [10, 53], Hence, attacks can only use memory exhaustion [21, 58, 61, 63] to surgically place a target page on a vulnerable physical memory location. Consequently, countermeasures aim to prevent adversarial memory exhaustion [21, 61]. We introduce *memory waylaying*, a reliable technique exploiting the Linux page cache to influence the physical location of a target page. Crucially, unlike previous techniques, memory waylaying does not exhaust the system memory and does not cause out-of-memory situations, i.e., the system remains stable and responsive.

Defeating Countermeasures based on Performance Counters and Static Analysis. Intel SGX is an x86 instruction-set extension to securely and confidentially execute programs in isolated environments, called *enclaves*, on potentially adversary-controlled systems. Enclaves run with regular user privileges and are further restricted for their own security and safety, e.g., no system calls. To protect against compromised or malicious operating systems and hardware, the memory of the enclave is encrypted to prevent any modification or inspection of the enclave's memory contents, even by the operating system's kernel and hardware components [17]. Furthermore, enclaves are excluded from the CPU performance counters [57]. Hence, this approach defeats countermeasures which rely on monitoring performance counters [6, 22, 25, 50] or on analyzing the application code or instruction stream for Rowhammer attacks [25, 32].

B. Attack Scenarios

Our attacks apply to personal computers and cloud systems. Hence, we demonstrate our attacks in both of these scenarios.

- **Native Privilege Escalation Attack.** Our Rowhammer enclave can be used on personal computers to gain root privileges on the system, even in the presence of *all* of the defenses mentioned above.
- **A Cloud Denial-of-Service Attack.** Our Rowhammer enclave can also be used in the cloud, to shut down a large number of cloud machines in a coordinated way, i.e., a "distributed" denial-of-service attack, by abusing Intel SGX security mechanisms. When SGX detects an error in the encrypted and integrity-checked memory region, it halts the entire machine until a manual power cycle is performed. By coordinating the error injection over multiple machines, an attacker can potentially take down an entire cloud provider.

C. Paper Outline

The paper is organized as follows. Section II provides background information. Section III introduces a new categorization of state-of-the-art Rowhammer defenses. Section IV defines our attacker model. Section V provides a high-level overview of our attacks and the building blocks, which are then detailed in Section VI (opcode flipping), Section VII (one-location hammering), and Section VIII (memory waylaying). Section IX summarizes and evaluates our attacks in practical scenarios. Section X discusses limitations and additional observations. We conclude our work in Section XI.

II. Background

In this section, we provide background information on the Rowhammer bug and Rowhammer defenses. We also discuss the prefetch side-channel attack which forms the primitive of our memory waylaying technique (cf. Section VIII). Finally, we provide background information on Intel SGX as well as attacks on (and from) SGX enclaves.

A. The Rowhammer Bug

The increase in density and decrease in size of DRAM cells leads to smaller capacitance of cells, allowing them to operate using lower voltages and smaller charges. While these changes have many advantages, such as an increase in DRAM capacity and lower energy consumption, they also cause DRAM cells to become more susceptible to disturbance errors and unintended physical interactions between multiple cells. Such interactions and disturbances often cause memory corruption, where the bit-value of a DRAM cell is unintentionally flipped [48].

In 2014, Kim et al. [38] showed that such bit errors can be caused in a DRAM row by rapidly accessing memory locations in adjacent DRAM rows (also known as *row hammering* [26]). To achieve these rapid DRAM accesses, data-caching mechanisms need to be bypassed, either by flushing the cache, e.g., using `clflush` [38], cache eviction [1, 6, 21], or uncached memory accesses [52]. We now describe different Rowhammer techniques to obtain bit flips in the target row.

Single-sided hammering performs frequent memory accesses (hammering) to only one row which is adjacent to the target row. In contrast, *double-sided* hammering hammers two memory rows, one on each side of the target row. As the two hammered rows must be on different sides of the target row, double-sided hammering generally requires at least partial knowledge of virtual-to-physical mappings while single-sided hammering does not. Both hammering techniques produce abnormal memory access patterns as they induce an enormous number of row conflicts. Bit flips are highly reproducible: Hammering the same offsets again yields the same bit flips.

We note that although the name single-sided hammering may suggest that only a single memory location is hammered, Seaborn and Dullien [58], who introduced this technique, described it as hammering 8 memory locations simultaneously. On their systems, two or more randomly selected addresses (i.e., no knowledge of virtual-to-physical mappings is required) are in the same DRAM bank in $61.4\,\%$ of the cases. Hence, in fact, single-sided hammering aims to hammer two memory locations in the same bank, but not necessarily neighboring the victim row.

Not a privilege-escalation attack but an escape from the NaCl sandbox was demonstrated by Seaborn and Dullien [58]. NaCl executes arbitrary generated code directly on the CPU but sanitizes it using a blacklist, e.g., disallowing system calls. To bypass the sanitizer, the attacker generates and sprays unprivileged code over the entire memory and induces an unpredictable random bit flip at an unpredictable random memory location. With a low probability, the bit flip hits the operand of an `and` instruction used to sanitize addresses used by the sandboxed code. As the code can be read and executed by the attacker, the attacker can verify whether the random bit flip modified a random code location such pointers are not fully sanitized, re-enabling traditional control-flow diversion attacks. Bhattacharya and Mukhopadhyay [9] exploited random Rowhammer bit flips in random memory locations to produce a faulty RSA signature and consequently recover the secret key.

However, as bit flips are highly reliable, more deterministic and reliable attacks have been mounted, including privilege-escalation attacks, sandbox escapes, and compromise of cryptographic keys were demonstrated using memory spraying [21, 58, 63], grooming [61], or page deduplication [10, 53].

B. Rowhammer Defenses

Rowhammer defenses can be divided into three categories based on their goal. The first category aims to *detect* Rowhammer and, after detection, stop the corresponding processes. The second category aims to *neutralize* Rowhammer bit flips to prevent their exploitation. The third category aims to *eliminate* Rowhammer bugs. We now review previous works on defending against Rowhammer attacks. We group the proposed countermeasures using the above-mentioned three categories.

Rowhammer Detection Countermeasures. Irazoqui et al. [32] proposed static code analysis to detect microarchitectural attacks in binaries in a fully automated way, e.g., when tested before loading them into an app store. Several works detect on-going attacks on commodity systems using hardware- and software-based performance counters [15, 16, 22, 25, 50, 68]. Herath and Fogh [25] proposed to use performance counters for detection of suspicious cache activity and then verifying that an attack is on-going by searching for `clflush` instructions near the instruction pointer.

Rowhammer Neutralization Countermeasures. Van der Veen et al. [61] and Gruss et al. [21] observed that the system's memory allocator only places kernel pages near userspace pages in near-out-of-memory situations. Hence, they propose to modify the allocator to prefer the out-of-memory situation over the proximate placement of kernel and userspace pages, effectively preventing memory exhaustion in turn of spraying and grooming. This prevents known Rowhammer attacks based on memory grooming or memory spraying, as the target page cannot be evicted or placed anymore, i.e., neutralizes Rowhammer bit flips. Generalizing this, Brasser et al. [11] presents G-CATT, an alternative memory allocator that isolates user and kernelspace in physical memory ensuring that the attacker cannot exploit bit flips in kernel memory, thus neutralizing Rowhammer-induced bit flips. Disabling page deduplication prevents Rowhammer attacks exploiting these features [10, 46, 53, 54].

Rowhammer Elimination Countermeasures. Aweke et al. [6] utilized performance counters to identify the locality of frequently accessed DRAM addresses and to selectively refresh nearby rows to mitigate possible Rowhammer attacks. Corbet [16] discusses a kernel module using performance counters that delays the CPU such that the DRAM module can refresh the rows if the cache-miss rate in the system exceeds a certain threshold, thus, slowing down not only Rowhammer attacks but also legitimate workloads.

In addition to Rowhammer neutralization countermeasures (e.g., G-CATT), Brasser et al. [11] also presented B-CATT, a bootloader extension scanning the DRAM and blacklisting vulnerable locations, thus, effectively reducing the amount of usable memory, but also effectively eliminating the Rowhammer bug. However, as observed by Kim et al. [38], such a countermeasure is not practical as it blocks almost the entire memory. We validated this observation on multiple systems, where on each system more than $95\,\%$ of the physical memory would be blocked by B-CATT. Seaborn and Dullien [58] suggested eliminating the Rowhammer bug by blacklisting the `clflush` instruction. This countermeasure was bypassed by mounting cache-eviction-based Rowhammer attacks [1, 6, 21].

Besides building more reliable chips or employing ECC modules, Kim et al. [38] and Kim et al. [37] proposed

probabilistic methods to eliminate bit flips in hardware. Every time a row is opened and closed, other adjacent or non-adjacent rows are opened with a low probability. Thus, if a Rowhammer attack opens and closes rows, statistically the adjacent rows are refreshed as well and, thus, bit flips are averted. The LPDDR4 standard [33] specifies two features to eliminate the Rowhammer bug: Target Row Refresh (TRR) enables the memory controller to refresh rows adjacent to a certain row; Maximum Activation Count (MAC) specifies how often a row can be activated before adjacent rows need to be refreshed. Furthermore, Ghasempour et al. [18] presented ARMOR, a cache storing frequently accessed rows in order to reduce the number of row activations in the DRAM and, thus, eliminating the Rowhammer bug.

Hence, all elimination-based defenses are either not practical or require hardware changes, making them not applicable for commodity systems. Commodity systems should instead be protected using detection- or neutralization-based approaches.

C. The Prefetch Side-Channel Attack

The prefetch side-channel attack was presented by Gruss et al. [20] as a way to defeat address-space-layout randomization. The timing difference induced by the prefetch instruction depends on the state of various caches. Prefetch instructions ignore privileges and permissions, as these checks would be performed upon a subsequent memory access anyway. Prefetch side-channel attacks also exploit the operating system design. In most operating systems, every valid memory location in a user process is mapped at least twice, once in the user process virtual memory, and once in the direct-physical mapping in the kernelspace. The *prefetch address-translation oracle* exploits this direct-physical mapping to determine whether an address in userspace maps to a specific address in the direct-physical mapping. If the guess was correct, the attacker learns the physical address of a userspace virtual address. Hence, the attacker does not have to rely on operating system interfaces to obtain physical addresses for virtual addresses.

D. Intel SGX

Intel SGX is an x86 instruction-set extension for integrity and confidentiality of code and data in untrusted environments [17]. For this purpose, SGX executes programs in so-called *secure enclaves* which use protected areas of memory that can only be accessed by the enclaves themselves. With SGX implemented in the CPU, the enclave remains protected, even if operating system, hypervisor, and hardware have been compromised. Furthermore, remote attestation allows validating the integrity of the enclave by proving its correct loading.

Intel SGX explicitly protects against DRAM-based attacks, e.g., cold-boot attacks, memory bus snooping, and memory-tampering attacks, by cryptographically ensuring confidentiality, integrity, and freshness of data stored in the main memory. Hence, it removes the DRAM from the trusted computing base. The memory containing code and data of running enclaves is a physically contiguous and encrypted block in the DRAM, called *EPC* (enclave page cache) area, which is protected from all non-enclave memory accesses using protection mechanisms implemented in the CPU. The encryption by the Memory Encryption Engine (MEE) is transparent to the processor's cores [23]. The MEE utilizes a Merkle tree to detect when the encrypted code and data stored in the DRAM have been tampered with. The MEE provides freshness to the integrity tags to mitigate replay attacks, i.e., an attacker uses an old encrypted page to replace a newer encrypted page to attack the enclave.

If an integrity or freshness error occurred, Intel SGX aborts the execution of the memory fetch immediately, and the MEE emits an error signal. Thus, the unverified data of the DRAM will never be loaded into the last-level cache [23]. Moreover, the MEE locks the memory controller, preventing any future memory operations (potentially incurring data corruption), causing the system to halt until it is rebooted.

E. Attacks on (and from) Secure Enclaves

While Intel does not claim to protect against side-channel attacks that deduce information of collected power statistics, performance statistics, branch statistics, or information on pages accessed via page tables [4], several such attacks have been demonstrated. Xu et al. [65] demonstrated a page fault side-channel attack from a malicious operating system to extract sensitive information, e.g., text documents and images. Brasser et al. [12] demonstrated a Prime+Probe cache side-channel attack, extracting 70 % of an RSA private key in an enclave. Furthermore, Schwarz et al. [57] mounted a cache side-channel attack from within an enclave to extract a full RSA private key of a co-located enclave. Xiao et al. [64] mounted control-flow inference attacks on recent SSL libraries running in secure enclaves. Moghimi et al. [47] presented CacheZoom, a tool that provides a high-resolution channel to track all memory accesses of SGX enclaves to mount key recovery attacks. Wang et al. [62] systematically analyzed side-channel threats of SGX and identified 8 potential side-channel attack vectors. However, Intel considers all of these attacks out of scope, due to their side-channel nature.

Attacks that rely on shared memory (e.g., Flush+Reload [66]) cannot be mounted, as enclave memory is inaccessible for other enclaves, processes, and the operating system. However, as enclaves use pages in the same DRAM rows, Wang et al. [62] showed that enclaves can mount DRAM row-hit side-channel attacks on other enclaves by running a cross-enclave DRAMA attack [51].

III. Categorization of State-of-the-art Defenses for Commodity Systems

Discussing Rowhammer defenses based on their goal (detection, neutralization, and elimination; cf. Section II-B), does not allow a thorough analysis and comparison, as the primitives of the different defenses in each category vary widely. As we have seen in Section II-B, none of the elimination-based defenses are practical or applicable to commodity systems. Hence, in this paper, we only focus on detection- and neutralization-based defenses. In this section, we introduce a

TABLE I: Rowhammer defenses for commodity systems.

Methodology \ Defense	MASCAT [32]	Chiappetta et al. [15]	Zhang et al. [68]	Herath and Fogh [25]	HexPADS [50]	Gruss et al. [22]	ANVIL [6]	Corbet [16]	No OOM [21, 61]	G-CATT [11]	B-CATT [11]	TRR [33]	MAC [33]	PARA/CRA/PRA [37, 38]	ARMOR [18]	ECC/Chipkill [27, 38]	Refresh Rate [38]
DETECTION																	
Static Analysis	●	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○
Performance Counters	○	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○
Memory Access Pattern	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○
NEUTRALIZATION																	
Physical Proximity	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○
Memory Footprint	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○
ELIMINATION																	
Bootloader	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
Hardware Modification	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	●	○
BIOS Update	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●

Symbols indicate whether a defense is part of defense class (●), optional aspects of the defense are part of a defense class (◐), or a defense is not part of a defense class (○).

novel systematic categorization for state-of-the-art defenses for commodity systems.

In our evaluation of defenses we identified the following 5 defense classes which can be applied to commodity systems:
D1. Detection through *static analysis*.
D2. Detection through *performance counter analysis*.
D3. Detection through analysis of *memory access patterns*.
D4. Prevention by strictly avoiding *physical proximity*.
D5. Prevention by preventing conspicuous *memory footprints*.
Other defense classes (bootloader- or BIOS-update-based) have already been shown to be ineffective (cf. Section II-B), or cannot be applied to commodity systems (hardware modifications). Table I provides an overview of Rowhammer defenses and the corresponding defense classes. We defer a discussion of effective elimination-based defenses (requiring hardware changes) to Section X-B.

In the following, we briefly describe the assumptions and implications for each of the defense classes, as well as an exhaustive list of defenses for each class.

Static Analysis. The underlying assumption of defenses based on static analysis (**D1**) is that the attack (binary) code can be accessed. This defense class is especially interesting for offline analysis, e.g., before adding software to an app store. If the detection works, the user cannot be attacked anymore. Static analysis is used by Irazoqui et al. [32] in MASCAT, an automated static code analysis tool to detect microarchitectural attacks on a large scale. Herath and Fogh [25] proposed to suspend programs with high cache miss rates and analyze instructions near the instruction pointer.

Performance Counter Monitoring. The underlying assumptions of defenses based on performance counter analysis (**D2**) are that the performance counters are available and that they include operations of the attacker program. A typical parameter for Rowhammer detection is the number of cache hits and cache misses. Detecting Rowhammer at runtime leaves a theoretical chance of missing an attack. If the detection works, attacks are stopped before they can exploit a bit flip. The use of performance counters is the basis of several defenses [22, 25, 50]. The underlying Flush+Reload loop of Rowhammer is also detected by cache attack defenses [15, 68].

Memory Access Patterns Monitoring. The underlying assumptions of defenses based on memory access patterns (**D3**) are that Rowhammer requires distinguishably abnormal memory access patterns and that these patterns can be detected. Double-sided hammering always accesses two memory locations with a distance d of $1 \leq d < 2$ rows, i.e., the two memory locations are in different rows, adjacent to the same victim row. Other Rowhammer attacks also access multiple memory locations in the same DRAM bank at a high frequency. If the underlying assumptions hold, attacks are likely stopped before they can induce a bit flip. Memory access patterns are used in different defenses [6, 16]. ANVIL [6] applies access pattern heuristics to distinguish Rowhammer attacks from legitimate work loads.

Preventing Physical Proximity. The underlying assumption of defenses based on preventing physical proximity (**D4**) is that Rowhammer attacks need to flip bits in page tables or other kernel pages to take over the system. A memory allocator can prevent physical proximity of user pages and kernel pages. G-CATT [11] is the only published defense in this class. G-CATT isolates kernel pages from user pages by leaving a gap in physical memory. If the isolation works, the user cannot take over the kernel and the system anymore.

Memory Footprints. The underlying assumptions of defenses based on prohibiting conspicuous memory footprints (**D5**) are that Rowhammer attacks need to allocate large amounts of memory to scan for bit flips and almost exhaust the entire memory to surgically place a page in a specific physical location to trigger and exploit a Rowhammer bit flip. While the memory consumption of the attacker can already raise suspicion, both spraying [21, 58] and grooming [61] easily exhaust the entire memory in a way that gets the attacker process killed by the operating system. The memory allocator by default already avoids placing kernel pages near userspace pages, and it only deviates from this behavior in near-out-of-memory situations. Not deviating from the default behavior to prevent adversarial memory exhaustion was mentioned in Rowhammer attack papers [21, 61]. If the memory allocator prevents adversarial memory exhaustion, an attacker cannot force target pages to specific memory locations anymore.

IV. ATTACKER MODEL

Our attacker model makes the following fundamental assumptions about the hardware, the operating system, installed defense mechanisms, and attacker capabilities:

Hardware. The installed DRAM modules are susceptible to Rowhammer bit flips and no dedicated hardware-based Rowhammer defense mechanisms are in place.

Operating System. The operating system is up-to-date and fully patched, and no known software vulnerabilities exist that an attacker could exploit to elevate privileges.

TABLE II: How the different defense classes are bypassed.

Bypass \ Defense Class	Static Analysis	Performance Counters	Memory Access Pattern	Physical Proximity	Memory footprint
Intel SGX	●	●	○	○	○
One-location hammering	○	○	●	○	○
Opcode flipping	○	○	○	●	○
Memory waylaying	○	○	○	○	●
Defense class defeated	●	●	●	●	●

Defenses. The system is protected with state-of-the-art Rowhammer defenses. Specifically, the system deploys at least one defense from each defense class, including static analysis [32], hardware performance counters [6, 15, 22, 25, 50, 68], memory access pattern analysis [6], physical proximity prevention [11], and prevention of near-out-of-memory situations [21, 61].

Attacker Capabilities. We assume that an attacker can start an arbitrary unprivileged user program. Furthermore, we assume that the attacker can launch an SGX enclave, which is also unprivileged.

V. High-Level View of the Attacks

In this section, we provide a high-level overview of the attack primitives we develop for our privilege-escalation attack in native environments and our denial-of-service attack in cloud environments, despite the presence of defenses from all defense classes from Section IV. Table II summarizes how we defeat every single defense class.

To defeat defense class **D1** (static analysis), we run our attack inside an SGX enclave. Code within enclaves cannot be read or inspected, as the processor prevents all accesses to the enclave memory. Hence, by encrypting the code and only decrypting it after the enclave is launched, a developer can hide arbitrary code within SGX enclaves. As a consequence, MASCAT [32] is incapable of detecting any microarchitectural or Rowhammer attack we perform inside the enclave. Furthermore, the instruction stream cannot be searched for `clflush` instructions [25].

Defense class **D2** (performance counters) is also defeated by running the attack inside an SGX enclave because the processor does not include SGX activity in process-specific performance counters for security reasons [28]. Confirming this, Schwarz et al. [57] observed that performance counters are not influenced by cache attacks running inside SGX enclaves. Hence, performance counters cannot be used to detect our attack.

One-location Hammering. To defeat defense class **D3** (memory access patterns), we introduce a new attack primitive, *one-location hammering*. As described in Section II-A, double-sided hammering and single-sided hammering have distinguishably abnormal memory access patterns. One-location hammering is based on a previously unknown Rowhammer effect. With one-location hammering, the attacker only runs a Flush+Reload loop on a single memory address at the maximum frequency. This virtually keeps the DRAM bank permanently open. We observed that one-location hammering drains enough charge from the DRAM cells to induce bit flips. As one-location hammering does not perform alternating accesses to different rows in the same bank, **D3** defenses, such as ANVIL [6] do not detect the ongoing attack. We describe one-location hammering in detail in Section VII.

Opcode Flipping. To defeat defense class **D4** (physical memory isolation), we introduce another new attack primitive, *opcode flipping*. All previous Rowhammer privilege-escalation attacks induced bit flips in carefully crafted page tables. If the page table modification is successful, the attacker gains unrestricted read and write access to the physical memory, which is equivalent to having kernel privileges [21, 58, 61, 63].

With opcode flipping, we propose a novel way of exploiting bit flips. In the x86 instruction set, bit flips in opcodes yield different but, in most cases, valid opcodes. We show that with only a single targeted bit flip in an instruction, we can alter a (setuid) binary, e.g., `sudo`, to provide an unprivileged process with root privileges. As this is a bit flip in a user page, it breaks the underlying assumption of defense class **D4**, i.e., G-CATT [11].

Previous attacks on unprivileged code [58] (cf. Section II-A for a detailed discussion) bypassed sandbox code sanitization by flipping bits in a bitmask used in a logical `and` in attacker-sprayed code. In contrast to their work, we identify potential target bit flips in any opcode in a shared binary or library, modifying opcodes and the instruction stream. Consequently, we illegitimately obtain root privileges by bypassing authentication checks. We explain opcode flipping in detail in Section VI.

Memory Waylaying. To defeat defense class **D5** (memory footprints), we introduce a novel alternative to memory spraying and grooming, called *memory waylaying*. Rowhammer attacks modify pages in a predictable way by placing them in physical memory locations where a known bit flips occur. There are two techniques to achieve this: With *spraying* the attacker fills the entire memory with copies of the generated data structure; with *grooming* the attacker allocates the data structure to exploit in the exactly right moment. Both methods require exhausting the entire memory and are easily detectable by monitoring memory consumption. *Memory waylaying* performs replacement-aware page cache eviction, using only page cache pages. These pages are not visible in the system memory utilization as they can be evicted any time and hence, are considered as available memory. Consequently, memory waylaying never causes the system to run out of memory.

Our observations show that page cache pages, after being discarded from DRAM, are reloaded from the disk to a new random physical location. Through continuous eviction, the page is eventually placed on a vulnerable physical location. Memory waylaying leverages the prefetch side-channel to detect when data in virtual memory is placed on a specific physical location. By doing so, memory waylaying consumes a negligible amount of processor time and memory while

waiting for the target page to be loaded to the target physical location. Hence, it is difficult to detect. Once the data is located at the desired position, the attacker hits it with the Rowhammer bit flip and exploits the modified binary to gain root privileges. We describe memory waylaying in detail in Section VIII.

VI. Opcode Flipping

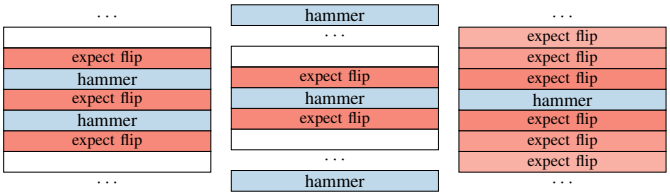
In this section, we describe *opcode flipping*, a generic technique for exploiting bit flips in cached copies of binary files. All previous generic Rowhammer privilege-escalation attacks (i.e., obtaining root privileges) flip bits in the page number field of attacker-generated page tables, to change the page that the page table entry references. Seaborn and Dullien [58] (cf. Section II-A for a detailed discussion) bypassed sandbox code sanitization by flipping bits in a bitmask used in a logical `and` in attacker-sprayed code. In contrast to their work, we identify potential target bit flips in any opcode in a shared binary or library, modifying opcodes and the instruction stream. In contrast to previous Rowhammer attacks based on memory spraying, the binary pages we attack cannot be sprayed and only exist a single time in the entire memory.

Opcode flipping exploits the fact that bit flips in opcodes can yield different, yet valid opcodes. These opcodes are often very similar to the original opcode but have different, possibly inverted, semantics. One prerequisite of opcode flipping is the ability to flip a bit of a target binary page with surgical precision. For now, we assume that the attacker can cause such a precise bit flip in a file and discuss the effect of such bit flips, before we show in Section VIII how a file can placed in memory accordingly.

Opcode Flipping Case Study. To illustrate opcode flipping we consider the example of a single bit flip in the x86 opcode `JE = 0x74` (jump if equal). A single bit flip in this opcode can yield the opcodes `JNE = 0x75` (jump if not equal), `JBE = 0x76` (jump if below or equal), `JO = 0x70` (jump if overflow), `JL = 0x7C` (jump if lower), `PUSHQ = 0x54` (push quad word), `XORB = 0x34` (xor byte), `HLT = 0xF4` (halt), and the prefix `0x64`. There are only 21 byte sequences following the prefix `0x64` which are illegal opcodes. The other 234 lead to valid opcodes.

Similarly, flips in `TEST` instructions preceding a conditional jump have the same effect. For example, with a single bit flip, the instruction `TEST EAX,EAX`, which sets the zero flag if `EAX` is zero, can be transformed to `XCHG EAX,EAX`, which never modifies the zero flag. Tests and conditional jumps are used in virtually all computer programs, and they control the decision logic of the programs. Therefore, we focus on flips in these instructions. As we show, bit flips in such instructions are sufficient to achieve our goals.

Exploitable Opcodes in Real-World Binaries. To exploit opcode flipping for privilege escalation, we target userspace applications with the `setuid` bit set, which are run as root. On Ubuntu 17.04, there are 16 `setuid` binaries owned by root, all being potential targets for privilege escalation using a bit flip. We manually analyzed one of the most prominent targets for privilege escalation, the `sudo` binary and `sudoers.so` shared library (henceforth *`sudo` binary*).



(a) Double-sided (b) Single-sided (c) One-location

Fig. 1: In contrast to double-sided and single-sided hammering, one-location hammering does not hammer multiple memory locations but only one.

We identified two regions in the `sudo` binary in which a bit flip can be exploited. First, the check whether the user is allowed to use `sudo`, i.e., if the user is in the `sudoers` file. Second, the check whether the entered password is correct. In this work, we focus on the latter.

We located 29 different offsets in the binary where a bit flip breaks the password verification logic. All identified bit flips affect the test or the conditional jump of the password-verification location. Successful attacks on the conditional jump change the condition so that it treats an incorrect password as if it was correct. Attacks on the test instruction result in different operations which ensure that the zero flag is clear, either by clearing it, e.g., `ADD AL,0xC0`, or by maintaining the previous, clear, value. We provide a list with offsets and their effect on the opcode at this position, in Appendix A.

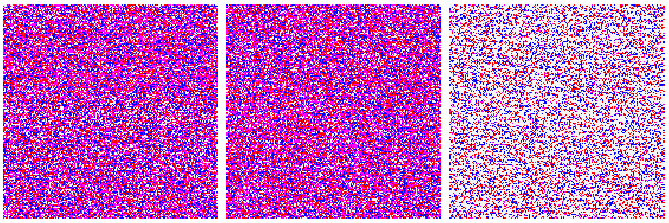
As shown in the following section, bit flip positions in memory are uniformly distributed, allowing exploitation of any of the 29 offsets in the `sudo` binary to gain root privileges.

VII. One-location Rowhammer

In this section, we describe the hammering technique we use to scan the system memory for bit flips. We assume that the attacker already knows exploitable bit offsets in binaries and only searches for memory locations where these bit offsets can be flipped through Rowhammer. We propose one-location Rowhammer as a novel alternative technique based on previously unknown Rowhammer effects. The scanning is performed from within the enclave and hence, cannot be observed through performance counters, source-code analysis or binary analysis.

Previous work described two different hammering techniques, double-sided hammering, and single-sided hammering, as described in more detail in Section II-A. Figure 1 illustrates and compares the different hammering techniques with the new one-location hammering we propose.

One-location hammering truly hammers only one memory location, i.e., the attacker does not directly induce row conflicts but only keeps one row permanently open. The core of one-location hammering is a Flush+Reload loop hammering a single randomly chosen address, as illustrated in Figure 1. One-location hammering the assumptions of defense class **D3**.



(a) Double-sided (b) Single-sided (c) One-location

Fig. 2: Flippable bit offsets over 4 kB-aligned memory regions for different hammering techniques. Bit flips from 0 to 1 (blue) and bit flips from 1 to 0 (red) may occur at any bit offset.

Both, one-location hammering and single-sided hammering are oblivious to virtual-to-physical address mappings. Hence, we can apply both hammering techniques in environments where physical address mappings are not available.

We studied the distribution of bit flips over 4 kB-aligned memory regions, i.e., pages, as this alignment can be obtained through our memory waylaying technique described in Section VIII. We performed our analysis on an Intel Skylake i7-6700K with two 8 GB Crucial DDR4-2133 DIMMs. We hammered random locations with the three techniques, in separate runs. Each test ran for 8 hours and scanned for bit flips after each hammering attempt. Figure 2 shows the distribution of bit flip offsets over 4 kB-aligned memory regions for double-sided hammering, single-sided hammering, and one-location hammering. We observe that 25 223 out of 32 768 bit offsets (77.0 %) can be flipped using double-sided hammering on at least one 4 kB-offset. 51.7 % of the bit flips were from 0 to 1.

Single-sided hammering does not induce more bit flips than double-sided hammering. However, regarding bit offsets, we observe an even slightly more uniform distribution for single-sided hammering, with 25 722 bit offsets (78.5 %). 54.1 % of the bit flips were from 0 to 1.

One-location hammering only flipped 11 969 out of 32 768 bit offsets (36.5 %) on at least one 4 kB-offset. 51.6 % of the bit flips were from 0 to 1. This is worse than double-sided hammering and single-sided hammering, which is likely the reason why this effect was not reported before. Still, our results show for the first time, that one-location hammering drains sufficient charge from the DRAM cells to induce bit flips.

We validated our results by reproducing them in a short series of tests on an Intel Haswell i7-4790 with two Kingston DDR3-1600 DIMMs, and an Intel i5-3230M with two Samsung DDR3-1600 DIMMs. On both systems, we observe bit flips for all hammering techniques, including one-location hammering. Bit flips from 0 to 1 and from 1 to 0 have approximately the same probability on all three systems.

Our data shows that the bit flips over pages generally follow a uniform distribution if a significant amount of memory is tested. As our attacker aims at finding bit flips for specific offsets on 4 kB pages, the runtime of the bit flip templating phase depends on the number of exploitable bit flip offsets. In case of the 29 bit offsets we found in `sudo`, the expected runtime on our Skylake system is less than 17 minutes per target bit flip for double-sided hammering, and less than 19 minutes for single-sided hammering. With one-location hammering the expected runtime increases to 56 minutes until a target bit flip is found. Hence, one-location hammering is 3.3 times slower in finding the target bit flip than comparable hammering methods. If evasion of defense class **D3** is a goal, a slow-down factor of 3.3 is practical.

Deciding to run the stealthy templating longer than necessary, i.e., searching for more than one bit flip, reduces the runtime of the waylaying phase (cf. Section VIII) significantly, as the attacker learns more addresses suitable for the attack.

The templating only keeps the CPU core of the enclave busy but causes no other system utilization, i.e., it does not exhaust memory, as we rely on the memory allocation of our waylaying technique, that we present in the following section.

VIII. Memory Waylaying

The attacker knows which bit offsets in pages of binaries to target to obtain root privileges, and how to hammer physical memory locations to obtain a bit flip at the right bit offset. The remaining problem is the inherent challenge of Rowhammer: Placing the target page at a physical location where the required bit flip can be induced. The known approaches to solve this challenge are spraying, i.e., filling the entire memory with copies of the page, or grooming, i.e., allocating the target page in exactly the right moment [67]. However, the page cache keeps every binary page only once in memory and prioritizes keeping binary pages in memory upon eviction. Hence, spraying is not applicable in our attack and grooming would require out-of-memory situations to force eviction of the binary page. In this section, we present *memory waylaying*, a reliable approach to solving the challenge of memory placement. It is a generic stealthy alternative to spraying and grooming, relying on a prediction oracle to determine whether a target page is at the right physical memory location.

In Section VIII-A, we show how the prefetch side-channel attack [20] can be leveraged as an oracle. In Section VIII-B, we present a technique to evict a target page from the page cache, forcing them to be relocated at the next access. Finally, Section VIII-C describes how the prefetch translation oracle and the page cache eviction are combined to the stealthy memory waylaying. We also present a fast variant, called *memory chasing*, which sacrifices stealth for speed, with no sacrifice of reliability.

A. Prefetch-based Prediction Oracle

In our memory waylaying attack, the attacker monitors page placement to detect mapping of one of the offsets in binaries and shared libraries to one of the target memory locations. We use the prefetch address-translation oracle [20] to perform this monitoring. The oracle exploits the direct-physical mapping in the Linux kernelspace. The prefetch address-translation oracle provides an attacker with the information whether two virtual

addresses map to the same physical address, even in the presence of address-space layout randomization.

The address-translation oracle consists of two steps, a sequence of prefetch instructions and a Flush+Reload attack, to measure the effect of the prefetch. While the attack is prone to false negatives due to ignored prefetch instructions, the Flush+Reload attack at its core has virtually no false positives [66], i.e., there is no cache hit if the address was not actually cached. While both steps can generally be executed in SGX enclaves, performing a Flush+Reload attack requires highly accurate timing measurements. On SGX2, `rdtsc` is available within enclaves. On SGX1, Schwarz et al. [57] demonstrated that accurate timing can be obtained by using counting threads and Wang et al. [62] mirrored `rdtsc` into the enclave. Our experiments with both approaches show that we can use either to obtain sufficiently accurate timing inside enclaves.

The address-translation oracle is first used in our attack to determine the offsets in the direct-physical map which have exploitable bit flips. It is then used a second time, to continuously monitor the set of target addresses during the memory waylaying. When an address match is detected, the next step of the attack is triggered, i.e., hitting the target page with Rowhammer.

Our prefetch address-translation oracle, which we optimized for stability, experienced no false positives over a time frame of 3737 seconds and a true positive every 4.5 seconds, i.e., the expected value for the true positive rate is 50 % when measuring for 4.5 seconds. When optimized for performance we can achieve the same performance as Gruss et al. [20], i.e., an expected measurement time of less than 50 milliseconds per address without false positives, but with a higher false negative rate. The search for the physical addresses is combined into one prefetch side-channel attack, i.e., one prefetch operation and as many Flush+Reload loops as page translations the attacker wants to find. Hence, the runtime does not increase significantly with the number of addresses, but only linearly in the amount of system memory.

B. Page Cache Eviction

Files are cached page-wise in the file page cache upon the first access to the corresponding page. Any subsequent access to a page of a file is directly served from the page cache. Thus, one prerequisite for memory waylaying is a technique to deterministically evict a page of a file from the page cache. Eviction ensures that any subsequent access to the file cannot be served from the page cache anymore, and the file is mapped to a new physical location.

Any unprivileged process could evict data from the page cache by simply allocating a large amount of memory, such that page cache pages must be evicted. This is similar to the memory exhaustion techniques in previous Rowhammer attacks and risks system crashes due to out of memory situations [21, 58, 61]. We examined the behavior of the page cache replacement algorithm to find a more reliable way to trigger eviction. While Linux provides privileged interfaces to do so, we need an approach which works without any privileges and from within enclaves, i.e., only with regular memory accesses.

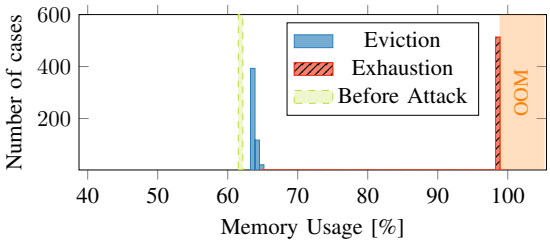


Fig. 3: Our replacement-aware page cache eviction only leads to negligible memory increase, whereas existing techniques are close to an out-of-memory situation.

A fundamental observation we made is that the replacement algorithm of the page cache prioritizes eviction of non-executable pages over executable pages. However, it does evict executable pages when filling the page cache with *read-only executable* pages. This forms a basic primitive that allows us to efficiently and reliably evict a selected page from the page cache. Because the page cache only uses otherwise unused memory pages, the technique does not result in memory pressure and avoids the unresponsiveness and out-of-memory situations that memory exhaustion causes [21, 58, 61].

For both approaches, memory exhaustion and replacement-aware page cache eviction, the amount of data which has to be accessed is at most the total amount of main memory in the system. To evaluate how much memory has to be allocated for the eviction to be successful, we use the Linux `mincore` function. The `mincore` function tells whether a given page is in the page cache. An attacker could also use this function to optimize the page cache eviction during an attack, i.e., abort the replacement-aware page cache eviction as soon as the page to be evicted is not in the page cache anymore. However, this is a trade-off between stealth and performance, as the operating system can monitor calls to the `mincore` function.

We evaluated our replacement-aware page cache eviction on an Intel Core i5-6200U with 12 GB of main memory. For the experiment, we kept the system at an typical workload, namely a browser, a mail client, and a music player were running during the experiment. Figure 3 compares traditional memory exhaustion with our replacement-aware page cache eviction to evict a specific page (in our experiment a page of the `sudo` binary) from the page cache. Our replacement-aware page cache eviction only incurs a slight increase of used memory, whereas the exhaustion-based technique is close to an out-of-memory situation. In 0.78 % of our exhaustion tests, the test program was even terminated by the operating system due to excessive memory usage. In contrast, our replacement-aware page cache eviction never leads to an out-of-memory situation. On average, for our replacement-aware page cache eviction, it was sufficient to access 5544 MB of data to evict the target page of the `sudo` binary from the page cache. The replacement-aware page cache eviction takes on average 2.68 seconds. For higher workloads, an attacker has to access even

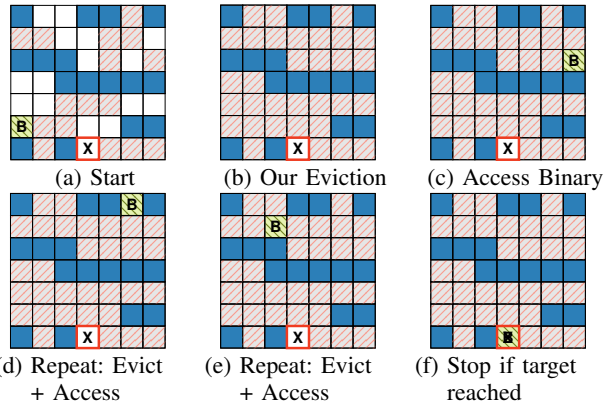


Fig. 4: Memory waylaying. In step (a) some pages are free ( ). Our eviction (b) allocates all free pages for the page cache ( ), but leaves occupied (non page cache) pages ( ) untouched. Repeating the eviction, the target page B ( ) is relocated, but the amount of occupied memory remains the same. Eventually, B is placed on the target physical location X ( ) as illustrated (f).

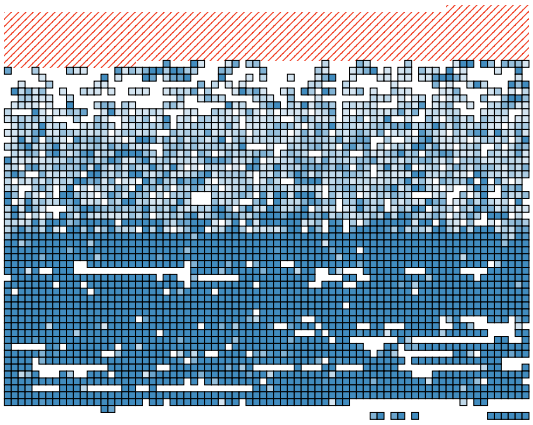


Fig. 5: Distribution of placements of a page in the physical memory. Each square represents 4 MB in the 12 GB address space of our test system. Hatched (red) areas are unavailable to the system (e.g., graphics memory, memory mapped I/O). The darker (blue) an area, the more physical pages were in this area. Even the small number of pages tested covers most of the physical memory.

less data to evict a specific page from the page cache, as the size of the page cache decreases with the memory usage of active applications.

C. Positioning Memory Pages

We combine the prefetch translation oracle (cf. Section VIII-A) and the replacement-aware page cache eviction (cf. Section VIII-B) to maneuver a target page on one of the physical locations with a bit flip (cf. Section VII). As an extension to memory waylaying, which is slow but stealthy, we also propose *memory chasing*, a faster non-stealthy variant.

Both memory waylaying and memory chasing, leverage the prefetch translation oracle to test whether our exploitable page is at the correct (i.e., vulnerable) physical page. As the physical page usually does not change often (i.e., only if there is high memory pressure or the system is rebooted), memory waylaying periodically evicts the page cache. On a subsequent access to the target page, the access cannot be served from the page cache anymore, and a new physical page is allocated and mapped. This procedure is illustrated in Figure 4.

We evaluate the distribution of physical page numbers used for a specific binary page on one of our test systems, an Intel Core i5 with 12 GB of main memory. We repeated the memory waylaying process 57 000 times, i.e., the binary page was relocated 57 000 times. Out of these 57 000 relocations, we found 46 720 unique physical page numbers, i.e., the probability of maneuvering the binary to a physical location where it was already is only 18 % after 57 000 tries. Figure 5 visualizes the distribution of the 57 000 relocations in physical memory. We observe that even the small number of relocations we tested (i.e., 1.8 % of all pages), most of the physical memory is covered, with the exception of occupied memory regions. Thus, we conclude that eventually the target binary page is placed at a physical memory location the intended bit flip can be induced.

The advantage of memory waylaying over conventional techniques, such as grooming or spraying, is that it is stealthy, as it does not exhaust the memory. The operating system page cache is designed to occupy any unused page in the system. Most pages are rarely accessed, but it is still more efficient to keep them in memory than to reload them from the disk. Memory waylaying exploits this design, and as a consequence, it has no impact on memory utilization and only negligible impact on the overall system performance, as the page cache simply keeps a different set of pages in the otherwise unused memory. In Section IX-B, we detail the runtime of the waylaying phase in a practical example.

The disadvantage of memory waylaying is that the runtime can vary widely, from a few hours up to a few days, until the target page is placed on the correct physical location. As a faster solution, we propose memory chasing, an adaption of memory waylaying which sacrifices stealth for speed. Instead of waiting for the target page to be placed on a different physical page, we actively "chase" the binary in physical memory until it is at the correct physical page. Memory chasing runs outside of the enclave as it has a stronger interaction with userspace library functions. To change the physical page of a target binary, memory chasing exploits the copy-on-write effect of `fork` as follows:

1) `mmap` the binary as private and *writable*.
2) Fork the current process.
3) In the child process, write to the mapped binary. This ensures that the page is copied to a new physical page.
4) Kill the parent process to release the old physical page.
5) Repeat until the page is at the intended physical location (check using the prefetch translation oracle)

Although the binary content is now at the correct physical location, the page cache still holds the first version of the binary page, as the current page is *dirty* (i.e., modified). Thus, we have to trick the kernel into replacing the old binary page with the current one. We do this by evicting the page cache as described in Section VIII-B. This removes the old (cached) binary page from the page cache. After the page cache is evicted, we unmap the current binary page and immediately map it again, however, this time with read-only and execute permissions. This ensures that the freed physical page is used to cache the binary in the page cache.

Memory chasing is considerably faster than memory waylaying, as the page cache has to be evicted only once. Moving the physical page with memory chasing takes on average only $36.7\,\mu s$, whereas memory waylaying requires $2.68\,s$. However, both techniques have the advantage of not exhausting the memory in contrast to memory spraying and grooming. One disadvantage of memory chasing is the large number of fork system calls, occupying one CPU core. Therefore, depending on how stealthy the attack must be, the attacker chooses which of the two primitives to use for reliable page cache eviction. In Section IX-B, we detail the runtime of memory chasing in a practical example.

IX. Evaluation of Attacks in Native and Cloud Environments

In this section, we summarize our attacks and evaluate them in practical scenarios. We first consider a cloud scenario with a simple attack, where an attacker is able to run our attack in virtual machines on multiple cloud servers. We then consider a local scenario with our full attack, where an attacker is able to run our attack on personal computers and performs a privilege-escalation attack. We detail the procedural steps of the attacks as well as the corresponding runtime.

A. Abusing SGX for "distributed" Denial-of-Service Attacks in the Cloud

Cloud servers are typically less susceptible to Rowhammer bit flips due to the presence of ECC, double refresh rates, and slower DRAM modules [51]. In the cloud scenario, the attacker uses our attack to identify a set of vulnerable servers and take the entire set of servers down in a coordinated and distributed attack, i.e., a denial-of-service attack. In this attack, we do not aim for privilege escalation and hence, neither perform opcode flipping nor memory waylaying. The attacker runs an unprivileged SGX enclave to evade defense classes **D1** and **D2**.

If, as discussed in Section II-D, an attacker induces bit flips in the encrypted memory area (EPC) of SGX, the CPU locks the memory controller (potentially incurring data corruption), causing the system to halt until it is rebooted manually. It is important to note that only a tiny fraction of $4\,kB$ pages are adjacent to the $128\,MB$ EPC memory area. For instance, on a system with $16\,GB$ dual-channel dual-rank DDR4 memory, only 256 pages ($0.006\,\%$ of all pages) are in an adjacent DRAM row. As different allocation mechanisms are used to allocate EPC pages and normal world pages, the attacker cannot accidentally hammer EPC addresses. Hence, it is extremely unlikely to accidentally flip a bit in the EPC memory region.

Many cloud providers use KVM [24] or Xen [7] as a hypervisor to run multiple virtual machines of different tenants in parallel on the same physical hardware. To expose SGX features to virtual machines, Intel published the necessary kernel patches [29, 30, 31]. Recently, Microsoft [45] introduced *Azure confidential computing* that enables developers to use SGX in their cloud.

Our "distributed" denial-of-service attack consists of two phases, seek and destroy:

- **Seek.** The attacker launches the attack enclave on many hosts in the cloud (i.e., "distributed"), and templates the DRAM for possible bit flips. The runtime of this phase is in the range of multiple hours. As we have shown in Section VII, the position of bit flips is uniformly distributed. Thus, if an attacker finds any bit flip while templating, the DRAM very likely is also vulnerable to bit flips in the EPC region used by SGX.
- **Destroy.** The attacker shuts down every vulnerable machine found in phase 1, by simultaneously triggering a bit flip in the EPC memory area. The runtime of this phase is in the range of seconds.

Besides ethical considerations on performing this experiment on a public cloud provider, we also found that no public cloud provider offers SGX support. Microsoft's *Azure confidential computing* [45] can only be used as an early access program, that we have not been granted access to. Instead, we performed the first part of our experiment on a dual CPU server system with two Intel Haswell-EP Xeon E5-2630 v3, a setup commonly found in public clouds. We equipped the system with two Crucial DDR4-2133 DIMMs known to be susceptible to Rowhammer bit flips. Our experiments showed that due to the significantly lower clock frequency (60–76 % of the clock frequency of an Intel Skylake i7-6700K) and the by-default doubled refresh rate, bit flips are much rarer. Specifically, we observed only 3 bit flips in an 8 hour test. However, this is sufficient for our denial-of-service attack.

In the second phase, our Rowhammer enclave starts to simultaneously hammer DRAM rows in the EPC on all hosts. By triggering a bit flip within this memory region, the machine locks the memory controller (potentially incurring data corruption) and causes the system to halt until reboot.

As our Intel Haswell-EP system does not support Intel SGX, we performed the second part of our practical analysis on an Intel Skylake i7-6700K. We verified that we are able to reproducibly crash the system within 10 seconds when hammering DRAM rows used by the EPC, as Intel SGX locked down the memory controller, halting the system and forcing us to power off the system manually. We observed that occasionally, after powering on the system again, the system did not boot beyond the BIOS for several minutes. After powering the system off and on again another time, the system regularly booted again.

Our results show that SGX introduces a significant security risk for cloud providers, allowing an attacker to cause hard-to-trace denial-of-service attacks and coordinated simultaneous take-down of multiple cloud servers, e.g., in the *Azure confidential computing* cloud [45]. As the attack hurts the availability and reliability of the cloud provider, it is especially interesting for parties with conflicting economic interests.

While the same attack could also be applied to a large number of personal computers, it is unclear how an attacker would profit from denial-of-service attacks on personal computers, especially in the face of the full privilege-escalation attack we detail in the next subsection.

B. Abusing SGX to Hide Privilege-Escalation Attacks

Personal computers are more susceptible to Rowhammer bit flips, as they usually are not equipped with ECC-RAM. In this scenario, the attacker uses our full attack for privilege escalation from a regular unprivileged process to root privileges. The crucial building blocks of this attack are opcode flipping and memory waylaying. The attacker runs an unprivileged SGX enclave to evade defense classes **D1** and **D2**.

In our example attack, we apply opcode flipping as described in Section VI to exploit bit flips in opcodes in the `sudo` binary of an up-to-date Ubuntu distribution. Bit flips at some offsets in the binary (Section VI) cause a skipping of authentication checks and, thus, provide us with root privileges.

The local attack requires two preparation steps:

- **Offline Preparation.** The attacker determines which bit flip offsets in standard system executable binaries and shared libraries are exploitable. This step is repeated for a large number of binaries and shared libraries of different distributions and versions. The result of the offline preparation is a database of files, versions, and bit flip offsets (cf. Section VI). In this phase, we identified 29 exploitable bit offsets in `sudo`.
- **Online Preparation.** The attacker verifies that the binary and library versions on the target systems are in the database. This is very likely the case if the victim uses a default installation of a popular Linux distribution, e.g., Ubuntu, as all binaries and libraries are pre-compiled and hence, identical on virtually every installation.

After the preparation steps are successfully completed, the attacker continues with the main attack. The main attack consists of four phases:

- **Templating phase.** Our Rowhammer enclave templates the memory for bit flips. This is done via single-sided hammering or one-location hammering (cf. Section VII), which both are oblivious to physical addresses and hence, perfectly suited to be run in our Rowhammer enclave. To defeat defense class **D3**, the attacker can use one-location hammering. The memory is allocated via memory-mapped files (cf. Section VIII), causing no significant increase in the resident memory and, thus, avoiding out-of-memory situations.

TABLE III: Optimal parameters and runtime of the attack.

Method	Bitflips	Templating	Waylaying	Total
Double-sided, waylaying	91	26.1 h	69.4 h	95.5 h
Single-sided, waylaying	87	27.5 h	70.6 h	98.1 h
One-location, waylaying	50	47.3 h	90.5 h	137.8 h
Double-sided, chasing	1	0.7 h	43.7 h	44.4 h
Single-sided, chasing	1	0.7 h	43.7 h	44.4 h
One-location, chasing	1	1.3 h	44.0 h	45.4 h

  The runtime of the templating phase and the waylaying phase pose an optimization problem (see Appendix B). Table III shows the optimal solution for our scenario, e.g., the runtime with one-location hammering is 47.3 hours if followed by waylaying, and 1.3 hours if followed by memory chasing. Interruptions during this time frame are no problem, as the attacker tests independent memory locations and does not lose data over interruptions. During the templating, the enclave occupies one CPU core, which is visible to the operating system but which could also be explained by completely benign enclave operations. The result of the templating phase is a list of physical pages with bit flips matching those from the preparation phase.
- **Waylaying phase.** Our Rowhammer enclave uses a side channel to wait until one of the vulnerable target binary or library pages is placed on one of the exploitable memory locations (cf. Section VIII). The prefetch-based prediction oracle tells us when the page has been loaded at the correct position. Next, then we flip the bit in the opcode using one-location hammering in the *hammering phase*.
  The runtime of the waylaying phase depends on the number of bit flips found in the templating phase. Table III shows the optimal solution for our scenario, e.g., the runtime with one-location hammering is 90.5 hours for memory waylaying and 44.0 hours for memory chasing. The result of the waylaying phase is that a target binary page is placed on the right physical page to trigger a predictable bit flip.
- **Hammering phase.** The hammering phase only takes a few milliseconds, as it only induces the predictable bit flip on the target page using Rowhammer. The attacker can verify whether a bit was flipped by reading the content of the binary page. Thus, the result of the hammering phase is an unauthorized modification of the target binary, i.e., in our case a malicious `sudo` binary.
- **Exploitation phase.** As the binary page in memory now contains the modified opcodes, the privilege check in the target binary, i.e., `sudo`, is circumvented. Hence, the attacker simply runs the attacked binary and, thus, obtains root privileges. Consequently, the exploitation phase also has a negligible runtime.

We performed all attack steps on an i7-6700K, showing that the attack can be mounted in practice. Furthermore, we validated the templating on two other systems, an i5-3230M with Samsung DDR3-1600 memory, and an i7-4790 with Kingston DDR3-1600 memory. We also validated the waylaying phase by running it for several days as a background process on a second machine (an i5-6200U), confirming that the user

does not notice any attack activity and that it does not cause any system crashes. To eliminate traces or avoid potential instabilities due to the binary modifications, an attacker can restore the unmodified binary page by simply evicting the page cache once more. Upon the next access, the unmodified version is reloaded from the disk.

Our attack shows that existing countermeasures for commodity systems are incomplete and fundamental assumptions need to be refined to design effective countermeasures.

X. Discussion

In this section, we discuss limitations of our approach and additional observations we made while conducting our study.

A. Limitations

One limitation of our work is that an attacker in the native attack scenario likely needs to get a Rowhammer enclave signed by a signing entity, e.g., Intel or a BIOS vendor, to be able to launch the enclave. While this sounds like a solid solution to prevent Rowhammer attacks through enclaves in practice, investigations on a very similar setting show that this is not the case [14]. It is very well possible to slip malware into app stores [14]. Furthermore, most works on applications of SGX suggest that it can be used to keep the code and data secret from any third party [5, 43, 56]. Especially for secure cloud computation it is not plausible to run only signed enclaves, i.e., a cloud provider will run non-signed user enclaves. This would allow an attacker to run our attack as well. Consequently, a different solution must be found to prevent Rowhammer attacks through SGX enclaves.

Although far more stealthy than spraying and grooming, memory waylaying is still observable by the operating system. The operating system could prevent allocating too many page cache pages in a single process. However, high memory requirements could also be perfectly reasonable, e.g., trusted video processing [41], operations on large encrypted database files [13, 36, 49, 56]. Hence, it can be doubted that memory allocation alone already gives away that an attack is ongoing.

B. Rowhammer mitigations in hardware

The results of this paper clearly show that countermeasures solely implemented in software trying to mitigate Rowhammer attacks are insufficient. However, future Rowhammer defenses should also be designed with related fault attacks in mind [35, 40]. We now discuss proposed and existing countermeasures implemented that require hardware modifications.

ECC RAM can detect and correct 1-bit errors and, thus, deal with single bit flips caused by the Rowhammer attack. Furthermore, IBM's Chipkill error correction [27] allows to successfully recover from 3-bit errors. However, uncorrectable multi-bit flips can be exploitable [2, 3, 42] or can result in a denial-of-service attack similar as described in Section IX-A depending on how the operating system responds to the error. While only modern AMD Ryzen processors support ECC RAM in consumer hardware, Intel restricts its support to server CPUs, thus, making it unavailable in commodity systems.

While the LPDDR4 [33] implements TRR and MAC, van der Veen [60] still reported bit flips on a Google Pixel phone with $4\,\mathrm{GB}$ LPDDR4 memory. Doubling the refresh rate has been shown to be insufficient [6, 38] and a further increase would incur a too high performance penalty [38].

Meaney et al. [44] introduced a redundant array of independent memory (RAIM) system as a feature of IBM's zEnterprise servers, which is basically the memory-equivalent for RAID systems for hard disks. An attacker would need to induce multiple bit flips in different rows of different modules to induce an uncorrectable error, making Rowhammer attacks infeasible.

Kim et al. [38] and Kim et al. [37] proposed to eliminate bit flips in hardware by probabilistically opening adjacent or non-adjacent rows, whenever a row is opened or closed. As an ongoing Rowhammer attack would open and close a certain row repeatedly, the vulnerable adjacent rows would be refreshed before bit flips occur. We consider their approaches as a possible solutions to mitigate Rowhammer attacks in the future.

C. Design of SGX

Intel SGX aims at protecting code from untrusted third parties. Indeed, we see that it perfectly hides our attack from different defense mechanisms. While this is intentional behavior and shows that SGX works, the question arises how to cope with harmful code within SGX enclaves, which eventually will happen in the wild.

A more discerning problem of SGX is that it halts the entire system, e.g., a cloud system. This is a powerful tool for attackers regardless of whether they run in the normal world or within an SGX enclave to take down entire clouds, possibly in a coordinated and distributed way. Hence, this behavior of SGX poses a security risk. Instead of halting the system, it would be less dangerous for the provider to only stop the running enclaves and return corresponding error codes to the host application.

XI. Conclusion

In this paper, we showed that even a combination of all state-of-the-art Rowhammer defenses does not prevent Rowhammer attacks. Our novel attack and exploitation primitives systematically undermine the assumptions of all defenses. With one-location hammering, we showed that previous assumptions on how the Rowhammer bug can be triggered are invalid and keeping only one DRAM row constantly open is sufficient to induce bit flips. With a slow-down factor of only 3.3, it is still on par with previous (now mitigated) techniques. With opcode flipping, we bypass all memory layout-based defenses by flipping bits in a predictable and targeted way in the userspace `sudo` binary. We present 29 bit offsets, each allowing an attacker to obtain root privileges in practice. With memory waylaying, we present a reliable technique to replace conspicuous and unstable memory spraying and grooming techniques. Coaxing the operating system into relocating any binary page takes $2.68\,\mathrm{s}$ with our stealth-optimized variant,

and only 36.7 µs with our speed-optimized variant. Finally, we leveraged Intel SGX to hide the full privilege-escalation attack, making any inspection or detection of the attack infeasible. Consequently, our attack evades all previously proposed countermeasures for commodity systems.

Acknowledgments

We would like to thank Thomas Schuster for help with some experiments.

References

[1] M. T. Aga, Z. B. Aweke, and T. Austin, "When good protections go bad: Exploiting anti-dos measures to accelerate rowhammer attacks," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017.

[2] B. Aichinger, "DDR memory errors caused by Row Hammer," in *HPEC*, 2015.

[3] ——, "Row Hammer Failures in DDR Memory," in *memcon*, 2015.

[4] I. Anati, F. McKeen, S. Gueron, H. Huang, S. Johnson, R. Leslie-Hurd, H. Patil, C. V. Rozas, and H. Shafi, "Intel Software Guard Extensions (Intel SGX)," 2015, Tutorial Slides presented at ICSA.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx," in *OSDI*, 2016.

[6] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 743–755, 2016.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[8] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: silently breaking ASLR in the cloud," in *Usenix WOOT*, 2015.

[9] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

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

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

[12] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," in *Usenix WOOT*, 2017.

[13] H. Brekalo, R. Strackx, and F. Piessens, "Mitigating password database breaches with intel sgx," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016.

[14] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale." in *USENIX Security Symposium*, 2015.

[15] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," Cryptology ePrint Archive, Report 2015/1034, 2015.

[16] J. Corbet, "Defending against rowhammer in the kernel," Oct. 2016. [Online]. Available: https://lwn.net/Articles/704920/

[17] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016.

[18] M. Ghasempour, M. Lujan, and J. Garside, "ARMOR: A Run-time Memory Hot-Row Detector," 2015. [Online]. Available: http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer

[19] D. Gruss, D. Bidner, and S. Mangard, "Practical memory deduplication attacks in sandboxed javascript," in *ESORICS*, 2015.

[20] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

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

[22] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[23] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptology ePrint Archive, Report 2016/204, 2016.

[24] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, Feb. 2008.

[25] N. Herath and A. Fogh, "These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security," in *Black Hat Briefings*, 2015.

[26] R.-F. Huang, H.-Y. Yang, M. C.-T. Chao, and S.-C. Lin, "Alternate hammering test for application-specific DRAMs and an industrial case study," in *Proceedings of the 49th Annual Design Automation Conference (DAC)*, 2012.

[27] IBM, "IBM Chipkill Memory: Advanced ECC Memory for the IBM Netfinity 7000 M10," 2019.

[28] Intel Corporation, "Intel Software Guard Extensions (Intel SGX)," 2016, retrieved on November 7, 2016. [Online]. Available: https://software.intel.com/en-us/sgx

[29] ——, "kvm-sgx," 2017. [Online]. Available: https://github.com/01org/kvm-sgx

[30] ——, "qemu-sgx," 2017. [Online]. Available: https://github.com/01org/qemu-sgx

[31] ——, "xen-sgx," 2017. [Online]. Available: https://github.com/01org/xen-sgx

[32] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Stopping microarchitectural attacks before execution," Cryptology ePrint Archive, Report 2016/1196, 2017.

[33] Jedec Solid State Technology Association, "Low Power Double Data Rate 4," 2017. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd209-4b

[34] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse engineering of drams: Row hammer with crosshair," in *Proceedings of the Second International Symposium on Memory Systems*, 2016.

[35] N. Karimi, A. K. Kanuparthi, X. Wang, O. Sinanoglu, and R. Karri, "Magic: Malicious aging in circuits/cores," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 1, 2015.

[36] F. Kerschbaum and A.-R. Sadeghi, "Hardidx: Practical and secure index with sgx," in *Data and Applications Security and Privacy XXXI: 31st Annual IFIP WG 11.3 Conference, DBSec 2017*, vol. 10359, 2017, p. 386.

[37] D.-H. Kim, P. J. Nair, and M. K. Qureshi, "Architectural support for mitigating row hammering in dram memories," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 9–12, 2015.

[38] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.

[39] Kirill A. Shutemov, "Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace," Mar. 2015, retrieved on November 10, 2015. [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce

[40] A. Kurmus, N. Ioannou, N. Papandreou, and T. Parnell, "From random block corruption to privilege escalation: A filesystem attack vector for rowhammer-like attacks," in *Usenix WOOT*, 2017.

[41] R. Lal and P. M. Pappachan, "An architecture methodology for secure video conferencing," in *IEEE International Conference on Technologies for Homeland Security (HST)*, 2013.

[42] M. Lanteigne, "How Rowhammer Could Be Used to Exploit Weaknesses in Computer Hardware," Mar. 2016. [Online]. Available: http://www.thirdio.com/rowhammer.pdf

[43] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security Symposium*, 2017.

[44] P. J. Meaney, L. A. Lastras-Montano, V. K. Papazova, E. Stephens, J. S. Johnson, L. C. Alves, J. A. O'Connor, and W. J. Clarke, "Ibm zenterprise redundant array of independent memory subsystem," *IBM Journal of Research and Development*, vol. 56, no. 1.2, Jan 2012.

[45] Microsoft, "Introducing azure confidential computing," 2017. [Online]. Available: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing

[46] ——, "Cache and Memory Manager Improvements," Apr. 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server

[47] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," *arXiv:1703.06986*, 2017.

[48] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[49] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowo zin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *USENIX Security Symposium*, 2016.

[50] M. Payer, "HexPADS: a platform to detect "stealth" attacks," in *ESSoS*, 2016.

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

[52] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016.

[53] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *USENIX Security Symposium*, 2016.

[54] Red Hat, *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*, 2017.

[55] M. Salyzyn, "UPSTREAM: pagemap: do not leak physical addresses to non-privileged userspace," 2015. [Online]. Available: https://android-review.googlesource.com/#/c/kernel/common/+/182766

[56] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: trustworthy data analytics in the cloud using sgx," 2015.

[57] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

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

[59] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory Deduplication as a Threat to the Guest OS," in *EuroSec*, 2011.

[60] V. van der Veen, "Drammer: Deterministic rowhammer attacks on mobile platforms," 2016. [Online]. Available: http://vvdveen.com/publications/drammer.slides.pdf

[61] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *CCS*, 2016.

[62] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel

hazards in sgx," in *CCS*, 2017.
[63] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation," in *USENIX Security Symposium*, 2016.
[64] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *CCS*, 2017.
[65] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *S&P*, May 2015.
[66] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.
[67] K. S. Yim, "The rowhammer attack injection methodology," in *IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.
[68] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

APPENDIX

A. Bitflips in sudo

Table IV lists exploitable bitflip offsets that modify opcodes of `sudoers.so` (Ubuntu 17.04, `sudo` version 1.8.19p1) yielding a skip of the privilege check and, thus, elevating an unprivileged process to root privileges.

B. Computing the Optimal Runtime of our Attack

The runtime of our attack is computed as

$$\frac{P \cdot (W + n \cdot 0.05)}{2^{12} \cdot n} + \frac{n \cdot 2^{16}}{F \cdot E} + \frac{120 \cdot P}{2^{30}}$$

seconds, where P is the amount of physical memory installed in the system, W is the amount of time one waylaying relocation takes, F is the flip rate (i.e., bit flips per second), and E is the number of exploitable bit offsets within a $4\,\mathrm{kB}$ page (which depends on the target binary). $n \in \mathbb{N}$ is the optimization parameter, the number of bit flips to find in the templating phase, influencing the runtime of the templating phase and the waylaying phase. 0.05 seconds is the time the prefetch address-translation oracle consumes for one test. 120 seconds is the amount of time the prefetch side-channel attack consumes to translate a virtual to a physical address per gigabyte (2^{30} bytes) of system memory. The 2^{16} represent the 2^{15} bit offsets of a $4\,\mathrm{kB}$ page (2^{12} bytes) which can flip in both directions each.

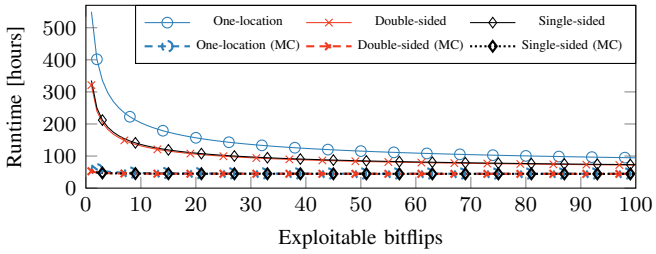


Fig. 6: Expected total runtime (templating and waylaying) until the attacker has the target page at the target physical location.

On our test system we have $P = 12$ gigabytes, $W = 2.68$ seconds for memory waylaying, $F = 0.67$, and $E = 29$. With these values we compute the runtime as

$$\frac{3 \cdot 2^{20} \cdot (2.68 + n \cdot 0.05)}{n} + n \cdot 3373.3 + 24\,\mathrm{m}$$

seconds. The minimum of this function is reached at $n = 50$.

Figure 6 shows the expected total runtime of the templating phase, and memory waylaying and chasing, depending on which hammering technique is used and how many bit offsets are exploitable.

TABLE IV: Exploitable bitflip offsets in `sudoers.so`.

#	Binary offset	Bitflip offset	Original	Flipped
1	0x8c1c	4	`lea rdi, aUser_is_exempt`	`lea rbp, aUser_is_exempt`
2	0x8c32	3	`mov eax, ebp`	`mov eax, esp`
3	0x8d4e	0	`lea rax, off_250860`	`lea rax, off_250860+1`
4	0x8d4f	0	`lea rax, off_250860`	`lea rax, unk_250760`
5	0x8d59	0	`mov eax, [rax+2C8h]`	`mov eax, [rax+2C9h]`
6	0x8d59	1	`mov eax, [rax+2C8h]`	`mov eax, [rax+2CAh]`
7	0x8d59	2	`mov eax, [rax+2C8h]`	`mov eax, [rax+2CCh]`
8	0x8d59	3	`mov eax, [rax+2C8h]`	`mov eax, [rax+2C0h]`
9	0x8d59	6	`mov eax, [rax+2C8h]`	`mov eax, [rax+288h]`
10	0x8d5a	5	`mov eax, [rax+2C8h]`	`mov eax, [rax+22C8h]`
11	0x8d5d	7	`test eax, eax`	`add eax, 485775C0h`
12	0x8d5e	0	`test eax, eax`	`test ecx, eax`
13	0x8d5f	0	`jnz short check_user_is_exempt`	`jz short check_user_is_exempt`
14	0x8dbd	3	`test al, al`	`mov eax, es`
15	0x8dbd	7	`test al, al`	`add al, 0C0h`
16	0x8dbf	0	`jnz short near ptr unk_8D61`	`jz short near ptr unk_8D61`
17	0x8dbf	3	`jnz short near ptr unk_8D61`	`jge short near ptr unk_8D61`
18	0x8dc4	3	`lea rbp, qword_252700`	`lea rbp, algn_2526F8`
19	0x8dc5	1	`lea rbp, qword_252700`	`lea rbp, dword_252900`
20	0x8dc5	2	`lea rbp, qword_252700`	`lea rbp, __imp_fflush`
21	0x8dc9	3	`mov eax, [rbp+0F0h]`	`mov ecx, [rbp+0F0h]`
22	0x8dc9	4	`mov eax, [rbp+0F0h]`	`mov edx, [rbp+0F0h]`
23	0x8dca	7	`mov eax, [rbp+0F0h]`	`mov eax, [rbp+70h]`
24	0x8dcb	3	`mov eax, [rbp+0F0h]`	`mov eax, [rbp+8F0h]`
25	0x8dcf	0	`test eax, eax`	`test ecx, eax`
26	0x8dcf	3	`test eax, eax`	`test eax, ecx`
27	0x8dd0	2	`jnz loc_8FB0`	`or eax, [rbp+1DAh]`
28	0x8dd1	0	`jnz loc_8FB0`	`jz loc_8FB0`
29	0x8e23	6	`jz loc_8FE8`	`jz near ptr algn_8FA7+1`